Semantic Matching: Algorithms and Implementation*

Fausto Giunchiglia, Mikalai Yatskevich, and Pavel Shvaiko

Department of Information and Communication Technology,
University of Trento,
38050, Povo, Trento, Italy
{fausto, yatskevi, pavel}@dit.unitn.it

Abstract. We view *match* as an operator that takes two graph-like structures (e.g., classifications, XML schemas) and produces a mapping between the nodes of these graphs that correspond semantically to each other. *Semantic matching* is based on 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 matching, and we discuss their implementation within the S-Match system. We evaluate S-Match against three state of the art matching systems, thereby justifying empirically the strength of our approach.

1. Introduction

Match is a critical operator in many well-known metadata intensive applications, such as schema/ontology integration, data warehouses, data integration, e-commerce, 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, see [43,12,40,42] for recent surveys, while some examples of individual approaches addressing the matching problem can be found in [1,2,5,6,10,11,13,16,30,32,33,35,39][1].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* [20]. This approach is based on two key ideas. The first is that we calculate mappings between schema elements by computing *semantic relations* (e.g., equivalence, more general, disjointness), instead of computing coefficients rating match quality in the [0,1] range, as it is the case in most previous approaches, see, for example, [11,13,32,39,35]. 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 automatically translated into propositional formulas which explicitly codify the labels' intended meaning. This allows us to translate the matching problem into a pro-

* This article is an expanded and updated version of an earlier conference paper [23].
[1] See www.OntologyMatching.org for a complete information on the topic.

positional validity problem, which can then be efficiently resolved using (sound and complete) state of the art propositional satisfiability (SAT) deciders, e.g., [31].

A vision of the semantic matching approach and some of its implementation were reported in [20,21,25]. In contrast to these works, this paper 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) an evaluation of the performance and quality of the implemented system, called S-Match, against other state of the art systems, which proves empirically the benefits of our approach. This article is an expanded and updated version of an earlier conference paper [23]. Therefore, three contributions mentioned above were originally claimed and substantiated in [23]. The most important extensions over [23] include a technical account of: (i) word sense disambiguation techniques, (ii) management of the inconsistencies in the matching tasks, and (iii) an in-depth discussion of the optimization techniques that improve the efficiency of the matching algorithm.

The rest of the paper is organized as follows. Section 2 introduces the semantic matching approach. It also provides an overview of four main steps of the semantic matching algorithm, while Sections 3,4,5,6 are devoted to the technical details of those steps. Section 7 discusses semantic matching with attributes. Section 8 introduces the optimizations that allow improving efficiency of the basic version of the algorithm. The evaluation results are presented in Section 9. Section 10 overviews the related work. Section 11 provides some conclusions and discusses future work.

2. Semantic Matching

In our approach, we assume that all the data and conceptual models (e.g., classifications, database schemas, ontologies) can be generally represented as graphs (see [20] for a detailed discussion). This allows for the statement and solution of a *generic (semantic) matching problem* independently of specific conceptual or data models, very much along the lines of what is done in Cupid [32] and COMA [11]. We focus on tree-like structures, e.g., classifications, and 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 [7]. From now on we assume that a graph-to-tree transformation can be done by using existing systems, and therefore, we focus on other issues instead.

The semantic matching approach is based on two key notions, namely:

- *Concept of a label*, which denotes the set of documents (data instances) that one would classify under a label it encodes;
- *Concept at a node,* which denotes the set of documents (data instances) that one would classify under a node, given that it has a certain label and that it is in a certain position in a tree.

Our approach can discover the following semantic relations between the *concepts at nodes* of two schemas: *equivalence* (=); *more general* (⊒); *less general* (⊑); *disjointness* (⊥). When none of the relations holds, the special *idk* (I do not know) [2] 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. Notice that the strongest semantic relation always exists since, when holding together, more general and less general relations are equivalent to equivalence. The semantics of the above relations are the obvious set-theoretic semantics.

A *mapping element* is a 4-tuple $\langle ID_{ij}, a_i, b_j, R \rangle$, $i = 1,...,N_A$; $j = 1,...,N_B$ where ID_{ij} is a unique identifier of the given mapping element; a_i is the i-th node of the first tree, N_A is the number of nodes in the first tree; b_j is the j-th node of the second tree, N_B is the number of nodes in the second tree; and R specifies a semantic relation which may hold between the *concepts at nodes* a_i and b_j. *Semantic matching* can then be defined as the following problem: given two trees T_A and T_B compute the $N_A \times N_B$ mapping elements $\langle ID_{ij}, a_i, b_j, R' \rangle$, with $a_i \in T_A$, $i=1,..., N_A$; $b_j \in T_B$, $j = 1,..., N_B$; and R' is the strongest semantic relation holding between the *concepts at nodes* a_i and b_j. Since we look for the $N_A \times N_B$ correspondences, the cardinality of mapping elements we are able to determine is 1:N. Also, these, if necessary, can be decomposed straightforwardly into mapping elements with the 1:1 cardinality.

Let us summarize the algorithm for semantic matching via a running example. We consider small academic courses classifications shown in Figure 1.

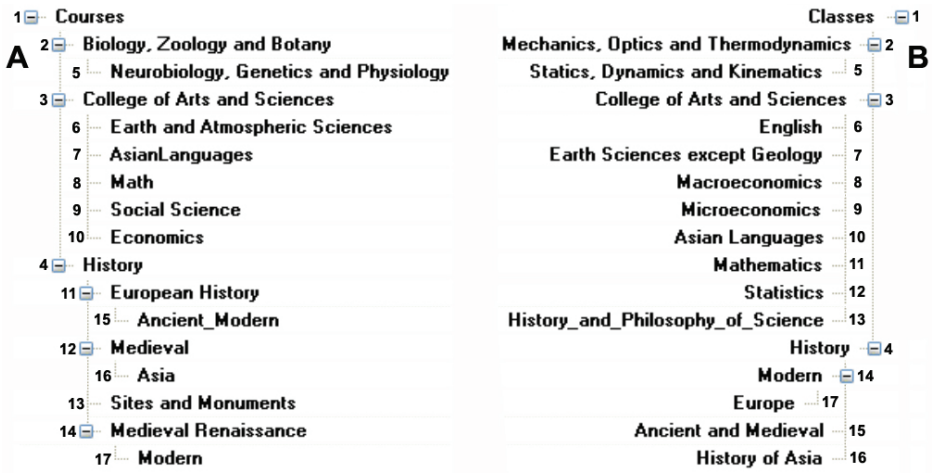

Fig. 1. Parts of two classifications devoted to academic courses

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. Thus, for ex-

[2] Notice *idk* is an explicit statement that the system is unable to compute any of the declared (four) relations. This should be interpreted as either there is not enough background knowledge, and therefore, the system cannot explicitly compute any of the declared relations or, indeed, none of those relations hold according to an application.

ample, in the tree A, $C_{History}$ and C_4 are, respectively, the concept of the label *History* and the concept at node 4. Also, to simplify the presentation, whenever it is clear from the context we assume that the concept of a label can be represented by the label itself. In this case, for example, $C_{History}$ becomes denoted as *History*. Finally, we sometimes use subscripts to distinguish between trees in which the given concept of a label occurs. For instance, $History_A$, means that the concept of the label *History* belongs to the tree A.

The algorithm takes as input two schemas and computes as output a set of mapping elements in four macro steps:

- *Step 1*: for all labels L in two trees, compute concepts of labels, C_L.
- *Step 2*: for all nodes N in two trees, compute concepts at nodes, C_N.
- *Step 3*: for all pairs of labels in two trees, compute relations among C_L*'s*.
- *Step 4*: for all pairs of nodes in two trees, compute relations among C_N*'s*.

The first two steps represent the preprocessing phase, while the third and the fourth steps are the element level and structure level matching respectively[3]. It is important to notice that *Step 1* and *Step 2* can be done once, independently of the specific matching problem. *Step 3* and *Step 4* can only be done at run time, once the two trees which must be matched have been chosen. We also refer in the remainder of the paper to the element level matching (*Step 3*) as *label matching* and to the structure level matching (*Step 4*) as *node matching*.

We view labels of nodes as concise descriptions of the data that is stored under the nodes. During *Step 1*, we compute the meaning of a *label* at a node (in isolation) by taking as input a label, by analyzing its real-world semantics (e.g., using WordNet [37] [4]), and by returning as output a *concept of the label*. Thus, for example, by writing $C_{History}$ we move from the natural language ambiguous label *History* to the concept $C_{History}$, which codifies explicitly its intended meaning, namely the data (documents) which are about history.

During *Step 2* we analyze the meaning of the *positions* that the labels of nodes have in a tree. By doing this we *extend* concepts of labels to *concepts at nodes*. This is required to capture the knowledge residing in the structure of a tree, namely the context in which the given concept of label occurs [17]. Thus, for example, in the tree A, when we write C_4 we mean the concept describing all the documents of the (academic) courses, which are about history.

Step 3 is concerned with acquisition of "world" knowledge. Relations between concepts of labels are computed with the help of a library of element level semantic matchers. These matchers take as input two concepts of labels and produce as output a semantic relation (e.g., equivalence, more/less general) between them. For example, from WordNet [37] we can derive that *course* and *class* are synonyms, and therefore, $C_{Courses} = C_{Classes}$.

---

[3] Element level matching (techniques) compute mapping elements by analyzing schema entities in isolation, ignoring their relations with other entities. Structure-level techniques compute mapping elements by analyzing how schema entities appear together in a structure, see for more details [42,43].

[4] WordNet is a lexical database for English. It is based on *synsets* (or senses), namely structures containing sets of terms with synonymous meanings.

Step 4 is concerned with the computation of the relations between concepts at nodes. This problem cannot be resolved by exploiting static knowledge sources only. We have (from *Step 3*) background knowledge, codified as a set of relations between concepts of labels occurring in two trees. This knowledge constitutes the background theory (axioms) within which we reason. We need to find a semantic relation (e.g., equivalence, more/less general) between the concepts at any two nodes in two trees. However, these are usually complex concepts obtained by suitably combining the corresponding concepts of labels. For example, suppose we want to find a relation between C_4 in the tree A (which, intuitively, stands for the concept of courses of history) and C_4 in the tree B (which, intuitively, stands for the concept of classes of history). In this case, we should realize that they have the same extension, and therefore, that they are equivalent.

3. Step 1: Concepts of Labels Computation

Technically, the main goal of *Step 1* is to automatically translate ambiguous natural language labels taken from the schema elements' names into an internal logical language. We use a propositional description logic language[5] (L^C) for several reasons. First, given its set-theoretic interpretation, it "maps" naturally to the real world semantics. Second, natural language labels, e.g., in classifications, are usually short expressions or phrases having simple structure. These phrases can often be converted into a formula in L^C with no or little loss of meaning [18]. Third, a formula in L^C can be converted into an equivalent formula in a propositional logic language with boolean semantics. Apart from the atomic propositions, the language L^C includes logical operators, such as *conjunction* ($\sqcup$), *disjunction* ($\sqcap$), and *negation* ($\neg$). There are also comparison operators, namely *more general* ($\sqsupseteq$), *less general* ($\sqsubseteq$), and *equivalence* (=). The interpretation of these operators is the standard set-theoretic interpretation.

We compute concepts of labels according to the following four logical phases, being inspired by the work in [34].

1. *Tokenization.* Labels of nodes are parsed into tokens by a tokenizer which recognizes punctuation, cases, digits, stop characters, etc. Thus, for instance, the label *History and Philosophy of Science* becomes ⟨*history*, *and*, *philosophy*, *of*, *science*⟩. The multiword concepts are then recognized. At the moment the list of all multiword concepts in WordNet [37] is exploited here together with a heuristic which takes into account the natural language connectives, such as *and*, *or*, etc. For example, *Earth and Atmospheric Sciences* becomes ⟨*earth sciences*, *and*, *atmospheric*, *sciences*⟩ since WordNet contains senses for *earth sciences*, but not for *atmospheric sciences*.

[5] A propositional description logic language (L^C) we use here is the description logic ALC language without the role constructor, see for more details [4]. Note, since we do not use roles, in practice we straightforwardly translate the natural language labels into propositional logic formulas.

2. *Lemmatization.* Tokens of labels are further lemmatized, namely they are morphologically analyzed in order to find all their possible basic forms. Thus, for instance, *sciences* is associated with its singular form, *science.* Also here we discard from further considerations some pre-defined meaningless (in the sense of being useful for matching) words, articles, numbers, and so on.

3. *Building atomic concepts.* WordNet is queried to obtain the senses of lemmas identified during the previous phase. For example, the label *Sciences* has the only one token *sciences*, and one lemma *science.* From WordNet we find out that *science* has two senses as a noun.

4. *Building complex concepts.* When existing, all tokens that are prepositions, punctuation marks, conjunctions (or strings with similar roles) are translated into logical connectives and used to build complex concepts out of the atomic concepts constructed in the previous phase. Thus, for instance, commas and conjunctions are translated into logical disjunctions, prepositions, such as *of* and *in,* are translated into logical conjunctions, and words like *except*, *without* are translated into negations. Thus, for example, the concept of label *History and Philosophy of Science* is computed as $C_{History\ and\ Philosophy\ of\ Science} = (C_{History} \sqcup C_{Philosophy}) \sqcap C_{Science}$, where $C_{Science} = \langle science, \{senses_{WN} \#2\} \rangle$ is taken to be the union of two WordNet senses, and similarly for *history* and *philosophy*. Notice that natural language *and* is converted into logical disjunction, rather than into conjunction (see [34] for detailed discussion and justification for this choice).

The result of *Step 1* is the logical formula for concept of label. It is computed as a full propositional formula were literals stand for atomic concepts of labels.

In Figure 2 we present the pseudo-code which provides an algorithmic account of how concepts of labels are built. In particular, the **`buildCLab`** function takes the tree of nodes `context` and constructs concepts of labels for each node in the tree. The nodes are preprocessed in the main loop in lines 220-350. Within this loop, first, the node label is obtained in line 240. Then, it is tokenized and lemmatized in lines 250 and 260, respectively. The (internal) loop on the lemmas of the node (lines 270-340) starts from stop words test in line 280. Then, WordNet is queried. If the lemma is in WordNet, its senses are extracted. In line 300, atomic concept of label is created and attached to the node by the **`addACOLtoNode`** function. In the case when WordNet returns no senses for the lemma, the special identifier `SENSES_NOT_FOUND` is attached to the atomic concept of label[6]. The propositional formula for the concept of label is iteratively constructed by **`constructcLabFormula`** (line 340). Finally, the logical formula is attached to the concept at label (line 350) and some sense filtering is performed by **`elementLevelSenseFiltering`**[7].

[6] This identifier is further used by element level semantic matchers in *Step 3* of the matching algorithm in order to determine the fact that the label (lemma) under consideration is not contained in WordNet, and therefore, there are no senses in WordNet for a given concept.

[7] The sense filtering problem is also known under the name of word sense disambiguation (WSD), see, e.g., [29].

```
Node struct of
              int nodeId;
              String label;
              String cLabel;
              String cNode;
              AtomicConceptAtLabel[] ACOLs;
AtomicConceptOfLabel struct of
              int id;
              String token;
              String[] wnSenses;
200. void buildCLab(Tree of Nodes context)
210.  String[] wnSenses;
220.  For each node in context
230.    String cLabFormula="";
240.    String nodeLabel=getLabel(node);
250.    String[] tokens=tokenize(nodeLabel);
260.    String[] lemmas=lematize(tokens);
270.    For each lemma in lemmas
280.      if (isMeaningful(lemma))
290.        if (!isInWordnet(lemma))
300.         addACOLtoNode(node, lemma, SENSES_NOT_FOUND);
310.       else
320.         wnSenses= getWNSenses(token);
330.         addACOLtoNode(node, lemma, wnSenses);
340.     cLabFormula=constructcLabFormula(cLabFormula, lemma);
350.   setcLabFormula(node, cLabFormula);
360.   elementLevelSenseFiltering(node);
```

Fig. 2. Concept of label construction pseudo code

The pseudo code in Figure 3 illustrates the sense filtering technique. It is used in order to filter out the irrelevant (for the given matching task) senses from concepts of labels. In particular, we look whether the senses of atomic concepts of labels within each concept of a label are connected by any relation in WordNet. If so, we discard all other senses from atomic concept of label. Otherwise we keep all the senses. For example, for the concept of label *Sites and Monuments* before the sense filtering step we have $\langle Sites, \{senses_{WN}\#4\}\rangle \sqcup \langle Monuments, \{senses_{WN}\#3\}\rangle$. Since the second sense of *monument* is a hyponym of the first sense of *site*, notationally $Monument\#2 \sqsubseteq Site\#1$, all the other senses are discarded. Therefore, as a result of this sense filtering step we have $\langle Sites, \{senses_{WN}\#1\}\rangle \sqcup \langle Monuments, \{senses_{WN}\#1\}\rangle$.

elementLevelSenseFiltering takes the node structure as input and discards the irrelevant senses from atomic concepts of labels within the node. In particular, it executes two loops on atomic concept of labels (lines 30-120 and 50-120). WordNet senses for the concepts are acquired in lines 40 and 70. Then two loops on the WordNet senses are executed in lines 80-120 and 90-120. Afterwards, checking whether the senses are connected by a WordNet relation is performed in line 100. If so, the senses are added to a special set, called *refined senses* set (lines 110, 120). Finally, the WordNet senses are replaced with the refined senses by **saveRefinedSenses**.

```
10.void elementLevelSenseFiltering(Node node)
20. AtomicConceptOfLabel[] nodeACOLs=getACOLs(node);
30. for each nodeACOL in nodeACOLs
40.  String[] nodeWNSenses=getWNSenses(nodeACOL);
50.  for each ACOL in nodeACOLs
60.   if (ACOL!=nodeACOL)
70.    String[] wnSenses=getWNSenses(ACOL);
80.     for each nodeWNSense in nodeWNSenses
90.      for each wnSense in wnSenses
100.       if (isConnectedbyWN(nodeWNSense, focusNodeWNSense))
110.        addToRefinedSenses(nodeACOL,nodeWNSense);
120.        addToRefinedSenses(focusNodeACOL, focusNodeWNSense);
130.   saveRefinedSenses(context);

140. void saveRefinedSenses(context)
150.   for each node in context
160.     AtomicConceptOfLabel[] nodeACOLs=getACOLs(node);
170.    for each nodeACOL in NodeACOLs
180.      if (hasRefinedSenses(nodeACOL))
190.        //replace original senses with refined
```

Fig. 3. The pseudo code of element level sense filtering technique

4. Step 2: Concepts at Nodes Computation

Concepts at nodes are written in the same propositional description logic language as concepts of labels. Classifications and 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 [26], 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, in the tree A, the concept at node four is computed as follows: $C_4 = C_{Courses} \sqcap C_{History}$.

Further in the paper we require the concepts at nodes to be consistent (satisfiable). The reasons for their inconsistency are negations in atomic concepts of labels. For example, natural language label *except_geology* is translated into the following logical formula $C_{except\_geology} = \neg C_{geology}$. Therefore, there can be a concept at node represented by a formula of the following type $C_{geology} \sqcap ... \sqcap \neg C_{geology}$, which is inconsistent. In this case the user is notified that the concept at node formula is unsatisfiable and asked to decide a more important branch, i.e., (s)he can choose what to delete from the tree, namely $C_{geology}$ or $C_{except\_geology}$. Notice that this does not sacrifice the system performance since this check is made within the preprocessing (i.e., off-line, when the tree is edited)[8]. Let us consider the following example: $C_N = ... \sqcap C_{Medieval} \sqcap C_{Modern.}$ Here, concept at node formula contains two concepts of labels, which are as from

[8] In general case the reasoning is as costly as in the case of propositional logic (i.e., deciding unsatisfiability of the concept is co-NP hard). In many real world cases (see [25] for more details) the corresponding formula is *Horn*. Thus, its satisfiability can be decided in linear time.

WordNet disjoint. Intuitively, this means that the context talks about either *Medieval* or *Modern* (or there is implicit disjunction in the concept at node formula). Therefore, in such cases, the formula for concept at node is rewritten in the following way: $C_N = (C_{Medieval} \sqcup C_{Modern}) \sqcap ...$

The pseudo code of the second step is presented in Figure 4. The **buildCNode** function takes as an input the tree of nodes with precomputed concepts of labels and computes as output the concept at node for each node in the tree. The sense filtering (line 620) is performed by **structureLevelSenseFiltering** in the way similar to the sense filtering approach used at the element level (as discussed in Figure 3). Then, the formula for the concept at node is constructed within **buildcNodeFormula** as conjunction of concepts of labels attached to the nodes in the path to the root. Finally, the formula is checked for unsatisfiability (line 640). If so, user is asked about the possible modifications in the tree structure or they are applied automatically, specifically implicit disjunctions are added between disjoint concepts (line 650).

```
600. void buildCNode(Tree of Node context)
610.  for each node in context
620.    structureLevelSenseFiltering (node,context);
630.    String cNodeFormula= buildcNodeFormula (node, context);
640.    if (isUnsatisifiable(cNodeFormula))
650.      updateFormula(cNodeFormula);
```

Fig. 4. Concepts at nodes construction pseudo code

Let us discuss how the structure level sense filtering operates. As noticed before, this technique is similar to the one described in Figure 3. The major difference is that the senses now are filtered not within the node label but within the tree structure. For all concepts of labels we collect all their ancestors and descendants. We call them a *focus* set. Then, all WordNet senses of atomic concepts of labels from the focus set are compared with the senses of the atomic concepts of labels of the concept. If a sense of atomic concept of label is connected by a WordNet relation with the sense taken from the focus set, then all other senses of these atomic concepts of labels are discarded. Therefore, as a result of sense filtering step we have (i) the WordNet senses which are connected with any other WordNet senses in the focus set or (ii) all the WordNet senses otherwise. After this step the meaning of concept of labels is reconciled with respect to the knowledge residing in the tree structure. The pseudo code in Figure 5 provides an algorithmic account of the structure level sense filtering procedure.

The **structureLevelSenseFiltering** function takes a node and a tree of nodes as input and refines the WordNet senses within atomic concepts of labels in the node with respect to the tree structure. First, atomic concepts at labels from the ancestor and descendant nodes are gathered into the focus set (line 420). Then, a search for pairwise relations between the senses attached to the atomic concepts of labels is performed (lines 440-520). These senses are added to the refined senses set (lines 530-540) and further **saveRefinedSenses** from Figure 3 is applied (line 550) in order to save the refined senses.

```
400.void structureLevelSenseFiltering (Node node, Tree of Nodes context)
410.  AtomicConceptOfLabel[] focusNodeACOLs;
420.  Node[] focusNodes=getFocusNodes(node, context);
430.  AtomicConceptOfLabel[] nodeACOLs=getACOLs(node);
440.  for each nodeACOL in nodeACOLss
450.    String[] nodeWNSenses=getWNSenses(nodeACOL);
460.    for each nodeWNSense in nodeWNSenses
470.      for each focusNode in focusNodes
480.        focusNodeACOLs=getACOLs(focusNode);
490.        for each focusNodeACOL in focusNodeACOLs
500.          String[] fNodeWNSenses=getWNSenses(focusNodeACOL);
510.          for each fNodeWNSense in nodeWNSenses
520.            if (isConnectedbyWN(nodeWNSense, fNodeWNSense))
530.              addToRefinedSenses(nodeACOL,nodeWNSense);
540.              addToRefinedSenses(focusNodeACOL, focusNodeWNSense);
550.  saveRefinedSenses(context);
```

Fig. 5. The pseudo code of structure level sense filtering technique

5. Step 3: Label Matching

5.1 A library of label matchers

Relations between concepts of labels are computed with the help of a library of element level semantic matchers [24]. These matchers take as input two atomic concepts of labels and produce as output a semantic relation between them. Some of them are re-implementations of well-known matchers used in Cupid [32] and COMA [11]. The most important difference is that our matchers ultimately return a semantic relation, rather than an affinity level in the [0,1] range, although sometimes using customizable thresholds.

Our label 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 matchers' approximation level. The relations produced by a matcher with the first approximation level are always correct. For example, *name* $\sqsupseteq$ *brand* as returned by *WordNet*. In fact, according to WordNet *name* is a hypernym (superordinate word) of *brand*. Notice that *name* has 15 senses and *brand* has 9 senses in WordNet. We use sense filtering techniques to discard the irrelevant senses, see Sections 3 and 4 for details. The relations produced by a matcher with the second approximation level are likely to be correct (e.g., *net* = *network*, but *hot* = *hotel* by *Prefix*). 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*). Note, matchers by default are executed following the order of increasing approximation level. The fourth column reports the matchers' type. The fifth column describes the matchers' input.

Table 1. Element level semantic matchers implemented so far.

Matcher name	Execution Order	Approximation level	Matcher type	Schema info
Prefix	2	2	String-based	Labels
Suffix	3	2		
Edit distance	4	2		
N-gram	5	2		
Text Corpus	13	3		Labels + Corpus
WordNet	1	1	Sense-based	WordNet senses
Hierarchy distance	6	3		
WordNet Gloss	7	3	Gloss-based	WordNet senses
Extended WordNet Gloss	8	3		
Gloss Comparison	9	3		
Extended Gloss Comparison	10	3		
Semantic Gloss Comparison	11	3		
Extended semantic gloss comparison	12	3		

We have three main categories of matchers: *string-*, *sense-* and *gloss- based* matchers. String-based matchers exploit string comparison techniques in order to produce the semantic relation, while sense-based exploit the structural properties of the WordNet hierarchies and gloss-based compare two textual descriptions (*glosses*) of WordNet senses. Below, we discuss in detail some matchers from each of these categories.

5.1.1 Sense-based matchers

We have two sense-based matchers. Let us discuss how the *WordNet* matcher works. As it was already mentioned, WordNet [37] is based on *synsets* (or senses), namely structures containing sets of terms with synonymous meanings. For example, the words *night, nighttime* and *dark* constitute a single synset. Synsets are connected to one another through explicit (lexical) semantic relations. Some of these relations (hypernymy, hyponymy for nouns and hypernymy and troponymy for verbs) constitute *kind-of* and *part-of* (holonymy and meronymy for nouns) hierarchies. For instance, *tree* is a kind of *plant.* Thus, *tree* is hyponym of *plant* and *plant* is hypernym of *tree*. Analogously, from *trunk* being a part of *tree* we have that *trunk* is meronym of *tree* and *tree* is holonym of *trunk*.

The *WordNet* matcher translates the relations provided by WordNet to semantic relations according to the following rules:

- A $\sqsubseteq$ B, if A is a hyponym, meronym or troponym of B;
- A $\sqsupseteq$ B, if A is a hypernym or holonym of B;
- A = B, if they are connected by synonymy relation or they belong to one synset (*night* and *nighttime* from the example above);
- A $\perp$ B, if they are connected by antonymy relation or they are the siblings in the *part of* hierarchy.

5.1.2 String-based matchers

We have five string-based matchers. Let us discuss how the *Edit distance* matcher works. It calculates the number of simple editing operations (delete, insert and re-

place) over the label's characters needed to transform one string into another, normalized by the length of the longest string. The result is a value in [0,1]. If the value exceeds a given threshold (0.6 by default) the equivalence relation is returned, otherwise, *Idk* is produced.

5.1.3 Gloss-based matchers

We have six gloss-based matchers. Let us discuss how the *Gloss comparison* matcher works. The basic idea behind this matcher is that the number of the same words occurring in the two WordNet glosses increases the similarity value. The equivalence relation is returned if the number of shared words exceeds a given threshold (e.g., 3). *Idk* is produced otherwise. For example, suppose we want to match *Afghan hound* and *Maltese dog* using the gloss comparison strategy. Notice, although these two concepts are breeds of dog, WordNet does not have a direct lexical relation between them, thus the *WordNet* matcher would fail in this case. However, the glosses of both concepts are very similar. *Maltese dog* is defined as a breed of toy dogs having a long straight silky white coat. *Afghan hound* is defined as a tall graceful breed of hound with a long silky coat; native to the Near East. There are 4 shared words in both glosses, namely breed, long, silky, coat. Hence, the two concepts are taken to be equivalent.

5.2 The label matching algorithm

The pseudo code implementing *Step 3* is presented in Figure 6. The label matching algorithm produces (with the help of matchers of Table 1) a matrix of relations between all the pairs of atomic concepts of labels from both trees.

```
700. String[][] fillCLabMatrix(Tree of Nodes source,target);
710. String[][]cLabsMatrix;
720. String[] matchers;
730. int i,j;
740. matchers=getMatchers();
750. for each sourceAtomicConceptOfLabel in source
760.  i=getACoLID(sourceAtomicConceptOfLabel);
770.  for each targetAtomicConceptOfLabel in target
780.   j= getACoLID(targetAtomicConceptOfLabel);
790.   cLabsMatrix[i][j]=getRelation(matchers,
              sourceAtomicConceptOfLabel,targetAtomicConceptOfLabel);
795. return cLabsMatrix
800. String getRelation(String[] matchers,
                          AtomicConceptOfLabel source, target)
810.  String matcher;
820.  String relation="Idk";
830.  int i=0;
840.  while ((i<sizeof(matchers))&&(relation=="Idk"))
850.   matcher= matchers[i];
860.   relation=executeMatcher(matcher,source,target);
870.   i++;
880.  return relation;
```

Fig. 6. Label matching pseudo code

fillCLabMatrix takes as input two trees of nodes. It produces as output the matrix of semantic relations holding between the atomic concepts of labels in both trees. First, the element level matchers of Table 1, which are to be executed (based on the configuration settings), are acquired in line 740. Then, for each pair of atomic concepts of labels in both trees, semantic relations holding between them are determined by using the **getRelation** function (line 790).

getRelation takes as input an array of `matchers` and two atomic concepts of labels. It returns the semantic relation holding between this pair of atomic concepts of labels according to the element level matchers. These label matchers are executed (line 860) until the semantic relation different from *Idk* is produced. Notice that execution order is defined by the `matchers` array.

The result of *Step 3* is a matrix of the relations holding between atomic concepts of labels. A part of this matrix for the example in Figure 1 is shown in Table 2.

Table 2. `cLabsMatrix:` matrix of relations among the atomic concepts of labels.

A \ B	*Classes*	*History*	*Modern*	*Europe*
Courses	=	*idk*	*idk*	*idk*
History	*idk*	=	*idk*	*idk*
Medieval	*idk*	*idk*	⊥	*idk*
Asia	*idk*	*idk*	*idk*	⊥

6. Step 4: Node Matching

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 first discuss in detail the tree matching algorithm. Then, we consider the node matching algorithm.

6.1 The tree matching algorithm

The tree matching algorithm is concerned with decomposition of the tree matching task into a set of node matching tasks. It takes as input two preprocessed trees obtained as a result of *Steps 1,2* and a matrix of semantic relations holding between the atomic concepts of labels in both trees obtained as a result of *Step 3*. It produces as output the matrix of semantic relations holding between concepts at nodes in both trees. The pseudo code in Figure 7 illustrates the tree matching algorithm.

```
900.String[][] treeMatch(Tree of Nodes source, target, String[][]
cLabsMatrix)
910. Node sourceNode,targetNode;
920. String[][]cNodesMatrix, relMatrix;
930. String axioms, context_A, context_B;
940. int i,j;
960. For each sourceNode in source
970.  i=getNodeId(sourceNode);
980.  context_A=getCnodeFormula (sourceNode);
990.  For each targetNode in target
1000.    j=getNodeId(targetNode);
1010.    context_B=getCnodeFormula (targetNode);
1020.    relMatrix=extractRelMatrix(cLabsMatrix, sourceNode,
                                                  targetNode);
1030.    axioms=mkAxioms(relMatrix);
1040.    cNodesMatrix[i][j]=nodeMatch(axioms,context_A,context_B);
1050. return cNodesMatrix;
```

Fig. 7. The pseudo code of the tree matching algorithm

treeMatch takes two trees of *Nodes* (*source* and *target*) and the matrix of relations holding between atomic concepts of labels (*cLabsMatrix*) as input. It starts from two loops over all the nodes of source and target trees in lines 960-1040 and 990-1040. The node matching problems are constructed within these loops. For each node matching problem we take a pair of propositional formulas encoding concepts at nodes and relevant relations holding between the atomic concepts of labels using the **getCnodeFormula** and **extractRelMatrix** functions respectively. The former are memorized as $context_A$ and $context_B$ in lines 980 and 1010. The latter are memorized in *relMatrix* in line 1020. In order to reason about relations between concepts at nodes, we build the premises (*axioms*) in line 1030. These are a conjunction of the concepts of labels which are related in relMatrix. For example, the semantic relations in Table 2, which are considered when we match C_4 in the tree A and C_4 in the tree B are $Classes_B = Courses_A$ and $History_B = History_A$. In this case *axioms* is $(Classes_B \leftrightarrow Courses_A) \wedge (History_B \leftrightarrow History_A)$. Finally, in line 1040, the semantic relations holding between the concepts at nodes are calculated by **nodeMatch** and are reported as a bidimensional array (*cNodesMatrix*). A part of this matrix for the example in Figure 1 is shown in Table 3.

Table 3. *cNodesMatrix*: matrix of relations among the concepts at nodes (matching result).

A \ B	C_1	C_4	C_{14}	C_{17}
C_1	=	$\sqsupseteq$	$\sqsupseteq$	$\sqsupseteq$
C_4	$\sqsubseteq$	=	$\sqsupseteq$	$\sqsupseteq$
C_{12}	$\sqsubseteq$	$\sqsubseteq$	$\bot$	$\bot$
C_{16}	$\sqsubseteq$	$\sqsubseteq$	$\bot$	$\bot$

6.2 The node matching algorithm

Each node matching problem is converted into a propositional validity problem. Semantic relations are translated into propositional connectives using the rules described in Table 4 (second column).

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

rel(a,b)	Translation of *rel(a, b)* into propositional logic	Translation of Eq. 2 into Conjunctive Normal Form
$a=b$	$a \leftrightarrow b$	*N/A*
$a \sqsubseteq b$	$a \rightarrow b$	$axioms \wedge context_A \wedge \neg context_B$
$a \sqsupseteq b$	$b \rightarrow a$	$axioms \wedge context_B \wedge \neg context_A$
$a \perp b$	$\neg(a \wedge b)$	$axioms \wedge context_A \wedge context_B$

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:

$$(axioms) \rightarrow rel(context_A, context_B), \quad (1)$$

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

$$axioms \wedge \neg\, rel(context_A, context_B) \quad (2)$$

Table 4 (third column) describes how Eq. 2 is translated before testing each semantic relation. Notice that Eq. 2 is in Conjunctive Normal Form (CNF), namely it is a conjunction of disjunctions of atomic formulas. The check for equivalence is omitted in Table 4, since $A=B$ holds if and only if $A \sqsubseteq B$ and $A \sqsupseteq B$ hold, i.e., both $axioms \wedge context_A \wedge \neg context_B$ and $axioms \wedge context_B \wedge \neg context_A$ are unsatisfiable formulas.

We assume the labels of nodes and the knowledge derived from element level semantic matchers to be all globally consistent. Under this assumption the only reason why we get an unsatisfiable formula is because we have found a match between two nodes. In fact, *axioms* cannot be inconsistent by construction. Consistency of $context_A$ and $context_B$ is checked in the preprocessing phase (see, Section 4 for details). However, *axioms* and *contexts* (for example, $axioms \wedge context_A$) can be mutually inconsistent. The situation occurs, for example, when *axioms* entails negation of the variable occurring in the *context*. In this case, the concepts at nodes are disjoint. In order to guarantee the correct behavior of the algorithm we perform the disjointness test first. It does not influence the algorithm correctness in general but allow us to obtain the correct result in this special case.

Let us consider the pseudo code of a basic node matching algorithm, see Figure 8. In line 1110, **`nodeMatch`** constructs the formula for testing disjointness. In line 1120, it converts the formula into CNF, while in line 1130 it checks the CNF formula for unsatisfiability. If the formula is unsatisfiable the disjointness relation is returned.

Then, the process is repeated for the less and more general relations. If both relations hold, then the equivalence relation is returned (line 1220). If all the tests fail, the

idk relation is returned (line 1280). 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 [31].

```
1100.String nodeMatch(String axioms, context_A, context_B)
1110. formula= And(axioms, context_A, context_B);
1120. formulaInCNF=convertToCNF(formula);
1130. boolean isOpposite= isUnsatisfiable(formulaInCNF);
1140. if (isOpposite)
1150.   return "⊥";
1160. String formula=And(axioms,context_A,Not(context_B));
1170. String formulaInCNF=convertToCNF(formula);
1180. boolean isLG=isUnsatisfiable(formulaInCNF)
1190. formula=And(axioms, Not(context_A), context_B);
1200. formulaInCNF=convertToCNF(formula);
1210. boolean isMG= isUnsatisfiable(formulaInCNF);
1220. if (isMG && isLG)
1230.   return "=";
1240. if (isLG)
1250.   return "⊑";
1260. if (isMG)
1270.   return "⊒";
1280. return "Idk";
```

Fig. 8. The pseudo code of the node matching algorithm

From the example in Figure 1, trying to prove that C_4 in the tree B is less general than C_4 in the tree A, requires constructing the following formula:

$$((Classes_B \leftrightarrow Courses_A) \wedge (History_B \leftrightarrow History_A)) \wedge$$
$$(Classes_B \wedge History_B) \wedge \neg (Courses_A \wedge History_A)$$

The above formula turns out to be unsatisfiable, and therefore, the less general relation holds. Notice, if we test for the more general relation between the same pair of concepts at nodes, the corresponding formula would be also unsatisfiable. Thus, the final relation retuned by the **NodeMatch** algorithm for the given pair of concepts at nodes is the equivalence.

7. Semantic Matching with Attributes

So far we have focused on classifications, which are simple class hierarchies. If we deal with, e.g., XML schemas, their elements may have attributes, see Figure 9.

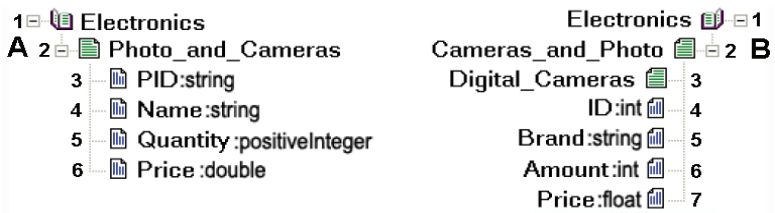

Fig.9. Two simple XML schemas

Attributes are ⟨*attribute−name, type*⟩ 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 the tree A, the attributes *PID* and *Name* are defined on the same domain *string*, but their intended use are the internal (unique) product identification and representation of the official products' 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 the tree A, *PIDs* are encoded as *string*s, while in the tree B, *IDs* are encoded as *int*s. However, both attributes serve for the same purpose of the unique products' 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.

7.1 Exploiting datatypes

In order to reason with datatypes we have created a *datatype ontology*, O_D, specified in OWL [45]. It describes the most often used XML schema built-in datatypes and relations between them. The backbone taxonomy of O_D is based on the following rule: *the is-a relationship holds between two datatypes if and only if their value spaces are related by set inclusion.* Some examples of axioms of O_D are: float $\sqsubseteq$ double, int $\sqsubseteq$ string, anyURI $\sqsubseteq$ 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 description logics with the quantification construct and datatypes. Thus, we compute concepts of labels and concepts at nodes as formulas in the description logics *ALC*(*D*) language [38]. For example, in the tree A in Figure 9, C_4, namely, the concept at node describing all the string data instances which are the names of electronic photography products is encoded as follows: $Electronics_A \sqcap (Photo_A \sqcap Cameras_A) \sqcap \exists Name_A.string$.

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 O_D and retrieves a semantic relation between them. For example, from axioms of O_D, the *Datatype* matcher can learn that float $\sqsubseteq$ double, and so on.

Step 4. Compute relations among concepts at nodes. In the case of attributes, the node matching problem is translated into an *ALC*(*D*) 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 [27]. From the example in Figure 9, trying to prove that C_7 in the tree B is less general than C_6 in the tree A, requires constructing the following formula:

$$((Electronics_A=Electronics_B) \sqcap (Photo_A=Photo_B) \sqcap (Cameras_A=Cameras_B) \sqcap (Price_A=Price_B) \sqcap (float \sqsubseteq double)) \sqcap (Electronics_B \sqcap (Cameras_B \sqcup Photo_B) \sqcap \exists Price_B.float) \sqcap \neg (Electronics_A \sqcap (Photo_A \sqcup Cameras_A) \sqcap \exists Price_A.double)$$

It turns out that the above formula is unsatisfiable. Therefore, C_7 in the tree B is less general than C_6 in the tree A. However, this result is not what the user expects. In fact, both C_6 in the tree A and C_7 in the tree B describe prices of electronic products, which are photo cameras. The storage format of *prices* in A and B (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. Technically, in order to implement this assumption, we should add corresponding axioms (e.g., float = double) to the premises of Eq. 1. On the one hand, with respect to the case of not considering datatypes (see, Section 7.2), 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.

7.2 Ignoring datatypes

In this case, information about datatypes is discarded. For example, ⟨*Name, string*⟩ becomes *Name*. Then, the semantic matching algorithm builds concepts of labels out of attributes' names in the same way as it does in the case of elements' names, and so on. Finally, it computes mapping elements using the algorithm of Section 6. A part of the cNodesMatrix with relations holding between attributes for the example in Figure 9 is presented in Table 5. Notice that this solution allows a mappings' 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) for Figure 9.

A \ B	C_4	C_5	C_6	C_7
C_3	=	*idk*	*idk*	*idk*
C_4	*idk*	⊒	*idk*	*idk*
C_5	*idk*	*idk*	=	*idk*
C_6	*idk*	*idk*	*idk*	=

The task of determining mappings typically represents a first step towards the ultimate goal of, for example, data translation, query mediation, 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.

8. Efficient Semantic Matching

The node matching problem in semantic matching is a CO-NP hard problem, since it is reduced to the validity problem for the propositional calculus. 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 at node is a conjunction (e.g., C_7 in the tree A in Figure 1 is defined as $Asian_A \sqcap Languages_A$), the node matching tasks can be solved in linear time. When we have *disjunctive concepts at nodes*, i.e., the concept at node contains both conjunctions and disjunctions in any order (e.g., C_3 in the tree B in Figure 1 is defined as $College_B \sqcap (Arts_B \sqcup Sciences_B)$), we use techniques allowing us to avoid 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.

8.1 Conjunctive concepts at nodes

Let us make some observations with respect to Table 4 (Section 6.2). The first observation is that the *axioms* part remains the same for all the tests, and it contains only clauses with two variables. In the worst case, it contains $2 \times n_A \times n_B$ clauses, where n_A and n_B are the number of atomic concepts of labels occurred in $context_A$ and $context_B$, respectively. The second observation is that the formulas for testing less and more general relations are very similar and they differ only in the negated context formula (e.g., in the test for less general relation $context_B$ is negated). This means that Eq. 2 contains one clause with n_B variables plus n_A clauses with one variable. In the case of disjointness test $context_A$ and $context_B$ are not negated. Therefore, formula Eq. 2 contains $n_A + n_B$ clauses with one variable.

8.1.1 The node matching problem by an example

Let us suppose that we want to match C_{16} in the tree A and C_{17} in the tree B in Figure 1. The relevant semantic relations between atomic concepts of labels are presented in Table 2. Thus, *axioms* is as follows:

$$\begin{aligned}&(course_A \leftrightarrow class_B) \wedge (history_A \leftrightarrow history_B) \wedge \\ &\neg(medieval_A \wedge modern_B) \wedge \neg(asia_A \wedge europe_B)\end{aligned} \quad (3)$$

which, when translated in CNF, becomes:

$$(\neg course_A \vee class_B) \wedge (course_A \vee \neg class_B) \wedge (\neg history_A \vee history_B) \wedge (history_A \vee \neg history_B) \wedge (\neg medieval_A \vee \neg modern_B) \wedge (\neg asia_A \vee \neg europe_B) \quad (4)$$

As from *Step 2*, $context_A$ and $context_B$ are constructed by taking the conjunction of the concepts of labels in the path from the node under consideration to the root. Therefore, $context_A$ and $context_B$ are:

$$course_A \wedge history_A \wedge medieval_A \wedge asia_A \quad (5)$$

$$class_B \wedge history_B \wedge modern_B \wedge europe_B \quad (6)$$

while their negations are:

$$\neg course_A \vee \neg history_A \vee \neg medieval_A \vee \neg asia_A \quad (7)$$

$$\neg class_B \vee \neg history_B \vee \neg modern_B \vee \neg europe_B \quad (8)$$

So far we have concentrated on atomic concepts of labels. The propositional formulas remain structurally the same if we move to conjunctive concepts at labels. Let consider the following example:

Fig. 10. Two simple classifications (obtained by modifying, pruning the example in Figure 1)

Suppose we want to match C_2 in the tree A and C_2 in the tree B in Figure 10. Axioms required for this matching task are as follows: $(course_A \leftrightarrow class_B) \wedge (history_A \leftrightarrow history_B) \wedge (medieval_A \sqsubseteq \bot modern_B) \wedge (asia_A \sqsubseteq \bot europe_B)$. If we compare them with those of Eq. 3 and Eq.4, which represent axioms for the above considered example in Figure 1, we find out that they are the same. Furthermore, as from *Step 2*, the propositional formulas for $context_A$ and $context_B$ are the same for atomic and for conjunctive concepts of labels as long as they "globally" contain the same formulas. In fact, concepts at nodes are constructed by taking the conjunction of concepts at labels. Splitting a concept of a label with two conjuncts into two atomic concepts has no effect on the resulting matching formula. The matching result for the matching tasks in Figure 10 is presented in Table 6.

Table 6. The matrix of relations between concepts at nodes (matching result) for Figure 10.

A \ B	C_1	C_2
C_1	=	$\sqsupseteq$
C_2	$\sqsubseteq$	$\bot$

8.1.2 Optimizations

Tests for less and more general relations. Using the observations in the beginning of Section 8.1 concerning Table 4, Eq. 2, with respect to the tests for less/more general relations, can be represented as follows:

$$\overbrace{\bigwedge_{q=0}^{n*m}(\neg A_s \vee B_t) \wedge \bigwedge_{w=0}^{n*m}(A_k \vee \neg B_l) \wedge \bigwedge_{v=0}^{n*m}(\neg A_p \vee \neg B_r)}^{Axioms} \wedge \overbrace{\bigwedge_{i=1}^{n} A_i}^{Context_A} \wedge \overbrace{\bigvee_{j=1}^{m} \neg B_j}^{\neg Context_B} \quad \textbf{(9)}$$

where n is the number of variables in $context_A$, m is the number of variables in $context_B$. The A_i's belong to $context_A$, and the B_j's belong to $context_B$. s, k, p are in the [0..n] range, while t, l, r are in the [0..m] range. q, w and v define the number of particular clauses. *Axioms* can be empty. Eq. 9 is composed of clauses with one or two variables plus one clause with possibly more variables (the clause corresponding to the negated context). The key observation is that the formula in Eq. 9 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* [9]. Notice, that DPLL-based SAT solvers require quadratic time in this case [47].

In order to understand how the linear time algorithm works, let us prove the unsatisfiability of Eq. 9 in the case of matching C_{16} in the tree A and C_{17} in the tree B in Figure 1. In this case, Eq. 9 is as follows:

$$\begin{gathered}(\neg \mathbf{course}_A \vee class_B) \wedge (\mathbf{course}_A \vee \neg class_B) \wedge (\neg \mathbf{history}_A \vee history_B) \wedge \\ (\mathbf{history}_A \vee \neg history_B) \wedge (\neg \mathbf{medieval}_A \vee \neg modern_B) \wedge (\neg \mathbf{asia}_A \vee \neg europe_B) \wedge \\ \mathbf{course}_A \wedge \mathbf{history}_A \wedge \mathbf{medieval}_A \wedge \mathbf{asia}_A \wedge \\ (\neg class_B \vee \neg history_B \vee \neg modern_B \vee \neg europe_B)\end{gathered} \quad \textbf{(10)}$$

In Eq.10, the variables from $context_A$ are written in bold face. First, we assign *true* to all unit clauses occurring in Eq. 10 positively. Notice these are all and only the clauses in $context_A$. This allows us to discard the clauses where $context_A$ variables occur positively (in this case: $\mathbf{course}_A \vee \neg class_B$, $\mathbf{history}_A \vee \neg history_B$). The resulting formula is as follows:

$$\begin{gathered}class_B \wedge history_B \wedge \neg modern_B \wedge \neg europe_B \wedge \\ (\neg class_B \vee \neg history_B \vee \neg modern_B \vee \neg europe_B)\end{gathered} \quad \textbf{(11)}$$

Eq. 11 does not contain any variable derived from $context_A$. Notice that, by assigning *true* to $class_B$, $history_B$ and *false* to $modern_B$, $europe_B$ we do not derive a contradiction. Therefore, Eq. 10 is satisfiable. In fact, a (Horn) formula is unsatisfiable if and only if the empty clause is derived (and it is satisfiable otherwise).

Let us consider again Eq. 11. For this formula to be unsatisfiable, all the variables occurring in the negation of $context_B$ ($\neg class_B \vee \neg history_B \vee \neg modern_B \vee \neg europe_B$ in our example) should occur positively in the unit clauses obtained after resolving *axioms* with the unit clauses in $context_A$ ($class_B$ and $history_B$ in our example). For this to happen, for any B_j in $context_B$ there must be a clause of form $\neg A_i \vee B_j$ in *axioms*, where A_i is a formula of $context_A$. Formulas of the form $\neg A_i \vee B_j$ occur in Eq. 9 if and only if

we have the axioms of form $A = B_j$ and $A_i \sqsubseteq B_j$. 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 B_j in Eq. 9.
- Step 2. For each axiom of type $A_i = B_j$ and $A_i \sqsubseteq B_j$ mark the corresponding B_j.
- Step 3. If all the B_j's are marked, then the formula is unsatisfiable.

To complete the analysis, let us now suppose that we have not "*europe*", but "*except europe*" as a node of the tree depicted in Figure 1. This means that $context_B$ contains the negated variable $\neg europe_B$. Eq. 10 in this case is rewritten as follows:

$$\begin{gathered}(\neg \boldsymbol{course_A} \vee class_B) \wedge (\boldsymbol{course_A} \vee \neg class_B) \wedge (\neg \boldsymbol{history_A} \vee history_B) \wedge \\ (\boldsymbol{history_A} \vee \neg history_B) \wedge (\neg \boldsymbol{medieval_A} \vee \neg modern_B) \wedge (\neg \boldsymbol{asia_A} \vee \neg europe_B) \wedge \\ \boldsymbol{course_A} \wedge \boldsymbol{history_A} \wedge \boldsymbol{medieval_A} \wedge \boldsymbol{asia_A} \wedge \\ (\neg class_B \vee \neg history_B \vee \neg modern_B \vee europe_B)\end{gathered} \quad (\mathbf{12})$$

Suppose that we have replaced all the occurrences of $\neg europe_B$ and $europe_B$ in the formula with $europe_{nB}$ and $\neg europe_{nB}$ respectively. In fact, we replace the variable with the new one which represents its negation. Notice that this replacement does not change the satisfiability properties of the formula. Truth assignment satisfying the new formula will satisfy the original formula after inverting the truth value of the new variable ($europe_{nB}$ in our example). Notice also that the replacement changed the clause with $europe_B$ variable in *axioms* ($\neg \boldsymbol{asia_A} \vee europe_{nB}$ in Eq. 13).

$$\begin{gathered}(\neg \boldsymbol{course_A} \vee class_B) \wedge (\boldsymbol{course_A} \vee \neg class_B) \wedge (\neg \boldsymbol{history_A} \vee history_B) \wedge \\ (\boldsymbol{history_A} \vee \neg history_B) \wedge (\neg \boldsymbol{medieval_A} \vee \neg modern_B) \wedge (\neg \boldsymbol{asia_A} \vee europe_{nB}) \wedge \\ \boldsymbol{course_A} \wedge \boldsymbol{history_A} \wedge \boldsymbol{medieval_A} \wedge \boldsymbol{asia_A} \wedge \\ (\neg class_B \vee \neg history_B \vee \neg modern_B \vee \neg europe_{nB})\end{gathered} \quad (\mathbf{13})$$

Let us assign to *true* the unit clauses occurring in Eq. 13 positively. This allows us to discard a number of clauses. A simplified formula is depicted as Eq. 14.

$$\begin{gathered}class_B \wedge history_B \wedge \neg modern_B \wedge europe_B \wedge \\ (\neg class_B \vee \neg history_B \vee \neg modern_B \vee \neg europe_{nB})\end{gathered} \quad (\mathbf{14})$$

This formula is satisfiable by assigning $class_B$, $history_B$, $europe_B$ to *true* and $modern_B$ to *false*. Therefore, less general relation does not hold between the concept at node *Asia* and the concept at node *Except Europe*.

In order to construct an optimized algorithm for determining satisfiability of Eq. 13 let us compare Eq. 10 and Eq. 13. The parts of the formula representing contexts are the same. The differences are in axioms part of the formula and they are introduced by a variable replacement. Let us analyze how the replacement of the variable with its negations influences various classes of clauses in axioms, see Table 7.

Table 7. The correspondence between axioms and clauses.

Axioms	$A_i \sqsubseteq B_j$ $A_i = B_i$	$B_j \sqsubseteq A_i$ $A_i = B_i$	$A_i \sqcap B_j$
The classes of propositional clauses With two variables	$\neg A_i \vee B_j$	$A_i \vee \neg B_j$	$\neg A_i \vee \neg B_j$
The classes of clauses after replacement of A_i with its negation A_{ni}	$A_{ni} \vee B_j$	$\neg A_{ni} \vee \neg B_j$	$A_{ni} \vee \neg B_j$
The classes of clauses after replacement of B_i with its negation B_{ni}	$\neg A_i \vee \neg B_{nj}$	$A_i \vee B_{nj}$	$\neg A_i \vee B_{nj}$
The classes of clauses after replacement of A_i and B_j with their negations A_{ni} and B_{nj} respectively	$A_{ni} \vee \neg B_j$	$\neg A_{ni} \vee B_j$	$A_{ni} \vee B_j$

Let us concentrate on three classes of propositional clauses depicted in the second row of Table 7. As from Eq. 9, we have only these classes of clauses in *axioms*. The axioms from which the particular class of clauses can be derived are described in the first column. Rows 2-5 demonstrate how the replacement of variables with its negation influences the clause. The first observation from Table 7 is that the new class of clauses ($A_i \vee B_j$) is introduced in *axioms*. The variables derived from both $context_A$ and $context_B$ occur in these clauses positively. This means that the clauses of form $A_i \vee B_j$ are discarded from the formula after unit propagation and cannot influence its satisfiability properties. The second observation is that all other clauses in Eq. 13 belong to the same classes as ones in Eq. 10. Therefore, the general observation made for Eq. 10 (namely, the formula is satisfiable if and only if there are clauses $\neg A_i \vee B_j$ in *axioms* for any B_j in $context_B$) holds for Eq. 13. As from Table 7, we have the clauses $\neg A_i \vee B_j$ in Eq. 13 in three cases:

- There are axioms $A_i = B_j$ and $A_i \sqsubseteq B_j$, where A_i and B_j occur in *contexts* of the original formula positively.
- There are axioms $A_i = B_j$ and $B_j \sqsubseteq A_i$, where A_i and B_j occur in *contexts* of the original formula negatively.
- There are axioms $A_i \sqcap B_j$, where A_i occurs in $context_A$ of the original formula positively and B_j occurs in $context_B$ of the original formula negatively.

These considerations suggest the following algorithm for testing the satisfiability (notice Step1 and Step 3 remain the same as in the previous version):

- Step 1. Create an array of size m. Each entry in the array stands for one B_j in Eq. 9.
- Step 2a. If A_i and B_j occur positively in $context_A$ and $context_B$ respectively, for each axiom $A_i = B_j$ and $A_i \sqsubseteq B_j$ mark the corresponding B_j.
- Step 2b. If A_i and B_j occur negatively in $context_A$ and $context_B$ respectively, for each axiom $A_i = B_j$ and $B_j \sqsubseteq A_i$ mark the corresponding B_j.
- Step 2c. If B_j occurs negatively in $context_B$ and A_i occurs positively in $context_A$ for each axiom $A_i \sqcap B_j$ mark the corresponding B_j.
- Step 3. If all the B_j*'s* are marked, then the formula is unsatisfiable.

The pseudo code of the optimized algorithm is presented in Figure 11.

```
1155.  if (context_A and context_B are conjunctive)
1156.    isLG=fastHornUnsatCheck (context_A, context_B, axioms, "⊑","⊒");
1157.    isMG=fastHornUnsatCheck (context_B, context_A, axioms, "⊒","⊑");
1158.  else
```

```
  1500.boolean fastHornUnsatCheck(String context, neg_context, axioms,
rel, neg_rel)
  1510. int m=getNumOfVar(String neg_context);
  1520. boolean array[m];
  1530. for each axiom in axioms
  1540.  String A_i= getFirstVariable(axiom);
  1550.  String B_j= getSecondVariable(axiom);
  1560.  int j=getNumberInContext(B_j);
  1570.  if((occurs_positevely (A_i, context))&&
                              (occurs_positevely (B_j, neg_context)))
  1580.   if((getAType(axiom)="=")||(getAType(axiom)=rel))
  1590.    array[j]=true;
  1600.  if ((occurs_negatively (A_i, context))&&
                              (occurs_negatively (B_j, neg_context)))
  1610.   if((getAType(axiom)="=")||(getAType(axiom)=neg_rel))
  1620.    array[j]=true;
  1630.  if ((occurs_positevely (A_i, context))&&
                              (occurs_negatively (B_j, neg_context)))
  1640.   if(getAType(axiom)="⊥")
  1650.    array[j]=true;
  1660. for (i=0; i<m; i++)
  1670.  if (!array[i])
  1680.   return false;
  1690. return true;
```

Fig. 11. Optimization pseudo code of tests for less and more general relations

Thus, **nodeMatch** can be modified as in Figure 11 (the numbers on the left indicate where the new code must be positioned). **fastHornUnsatCheck** implements the three steps above. Step 1 is performed in lines (1510-1520). Then, a loop on *axioms* (lines 1530-1650) implements Step 2. The final loop (lines 1660-1690) implements Step 3.

Disjointness test. Using the same notation as before in this section, Eq. 2 with respect to the disjointness test can be represented as follows:

$$\overbrace{\bigwedge_{q=0}^{n*m}(\neg A_s \vee B_t)\wedge\bigwedge_{w=0}^{n*m}(A_k \vee \neg B_l)\wedge\bigwedge_{v=0}^{n*m}(\neg A_p \vee \neg B_r)}^{Axioms}\wedge\overbrace{\bigwedge_{i=1}^{n}A_i}^{Context_A}\wedge\overbrace{\bigwedge_{j=1}^{m}B_j}^{Context_B} \quad \textbf{(15)}$$

For example, the formula for testing disjointness between C_{16} in the tree A and C_{17} in the tree B in Figure 1 is as follows:

$$(\neg \mathbf{course}_A \vee class_B)\wedge(\mathbf{course}_A \vee \neg class_B)\wedge(\neg \mathbf{history}_A \vee history_B)\wedge (\mathbf{history}_A \vee \neg history_B)\wedge(\neg \mathbf{medieval}_A \vee \neg modern_B)\wedge(\neg \mathbf{asia}_A \vee \neg europe_B)\wedge \mathbf{course}_A \wedge \mathbf{history}_A \wedge \mathbf{medieval}_A \wedge \mathbf{asia}_A \wedge class_B \wedge history_B \wedge modern_B \wedge europe_B \quad \textbf{(16)}$$

Eq. 16 is Horn, and thus, similarly to Eq. 10, the satisfiability of this formula can be decided by the unit propagation rule. After assigning *true* to all the variables in $context_A$ and propagating the results we obtain the following formula:

$$class_B \wedge history_B \wedge \neg modern_B \wedge \neg europe_B \wedge class_B \wedge history_B \wedge modern_B \wedge europe_B \quad \textbf{(17)}$$

If we further unit propagate $class_B$ and $history_B$ (this means that we assign them to *true*), then we obtain the contradiction $modern_B \wedge \neg modern_B \wedge europe_B \wedge \neg europe_B$. Therefore, the formula is unsatisfiable. This contradiction arises because $(\neg \mathbf{medieval_A} \vee \neg modern_B)$ and $(\neg \mathbf{asia_A} \vee \neg europe_B)$ occur in Eq. 16, which, in turn, are derived (as from Table 4) from the disjointness axioms $modern_B \bot medieval_A$ and $asia_A \bot europe_B$. In fact, all the clauses in Eq. 15 contain one positive literal except for the clauses in *axioms* corresponding to disjointness relations. Thus, the key intuition here is that if there are no disjointness axioms, then Eq. 15 is satisfiable. However, if there is a disjointness axiom, atoms occurring there are also ensured to be either in $context_A$ or in $context_B$, hence, Eq. 15 is unsatisfiable. Therefore, the optimization consists of just checking the presence/absence of disjointness axioms in *axioms*.

To complete the analysis suppose that we have negated variable in $context_B$ in the same fashion as described in the example with negations given before in this section. Then, Eq. 16 can be rewritten as follows:

$$\begin{gathered}(\neg \mathbf{course_A} \vee class_B) \wedge (\mathbf{course_A} \vee \neg class_B) \wedge (\neg \mathbf{history_A} \vee history_B) \wedge \\ (\mathbf{history_A} \vee \neg history_B) \wedge (\neg \mathbf{medieval_A} \vee \neg modern_B) \wedge (\neg \mathbf{asia_A} \vee \neg europe_B) \wedge \\ \mathbf{course_A} \wedge \mathbf{history_A} \wedge \mathbf{medieval_A} \wedge \mathbf{asia_A} \wedge class_B \wedge history_B \wedge modern_B \wedge \neg europe_B\end{gathered} \quad \textbf{(18)}$$

As in the case of less general relation all the occurrences of the negated variable are replaced with a new variable representing its negation (i.e., $\neg europe_B$ and $europe_B$ are replaced by $europe_{nB}$ and $\neg europe_{nB}$ respectively), see Eq. 19.

$$\begin{gathered}(\neg \mathbf{course_A} \vee class_B) \wedge (\mathbf{course_A} \vee \neg class_B) \wedge (\neg \mathbf{history_A} \vee history_B) \wedge \\ (\mathbf{history_A} \vee \neg history_B) \wedge (\neg \mathbf{medieval_A} \vee \neg modern_B) \wedge (\neg \mathbf{asia_A} \vee europe_{nB}) \wedge \\ \mathbf{course_A} \wedge \mathbf{history_A} \wedge \mathbf{medieval_A} \wedge \mathbf{asia_A} \wedge class_B \wedge history_B \wedge modern_B \wedge europe_{nB}\end{gathered} \quad \textbf{(19)}$$

After the unit propagation of the variables derived from $context_A$ we obtain

$$class_B \wedge history_B \wedge \neg modern_B \wedge europe_{nB} \wedge class_B \wedge history_B \wedge modern_B \wedge europe_{nB} \quad \textbf{(20)}$$

Eq. 20 is satisfiable. This means that the concept at node *Asia* is not disjoint with the concept at node *Except Europe*. The replacement introduces the new class of clauses $A_i \vee B_j$. However, such clauses are discarded after the unit propagation, and therefore, do not influence the satisfiability of the formula. As from Table 7, all other clauses introduced after the replacement belong to the same classes as ones in Eq. 16. This means that the major observation made in this section, namely the fact that the satisfiability of Eq. 16 can be decided by checking the presence/absence of the clauses of form $\neg A_i \vee \neg B_j$ holds for Eq. 19. As from Table 7, we have the clauses of form $\neg A_i \vee \neg B_j$ in Eq. 19 in the following three cases:

– There are axioms of form $A_i \bot B_j$, where both A_i and B_j occur in *contexts* of the original formula positively.

– There are axioms of form $B_j \sqsubseteq A_i$ and $A_i \equiv B_j$, where A_i occurs negatively in $context_A$ of the original formula and B_j occurs positively in $context_B$ of the original formula.
– There are axioms of form $A_i \sqsubseteq B_j$ and $A_i \equiv B_j$, where A_i occurs positively in $context_A$ of the original formula and B_j occurs negatively in $context_B$ of the original formula.

Thus, the pseudo code of **nodeMatch** should be modified as shown in Figure 12.

```
  1105.if (contextA and contextB are conjunctive)
  1106. isOpposite=optimizedUnsatTestForDisjointness (axioms, contextA,
contextB);
  1107.else

  1300. optimizedUnsatTestForDisjointness (axioms, contextA, contextB);
  1310. for each axiom in axioms
  1320.  String Ai= getFirstVariable(axiom);
  1330.  String Bj= getSecondVariable(axiom);
  1340.  if ((occurs_positively (Ai, contextA))&&
                                    (occurs_positively (Bj, contextB)))
  1350.   if (getAType(axiom)="⊥")
  1360.    return true;
  1370.  if ((occurs_negatively (Ai, contextA))&&
                                    (occurs_positively (Bj, contextB)))
  1380.  if((getAType(axiom)="=")||(getAType(axiom)="⊒"))
  1390.    return true;
  1400.  if ((occurs_positevely (Ai, contextA))&&
                                    (occurs_negatively (Bj, contextB)))
  1410.  if((getAType(axiom)="=")||(getAType(axiom)="⊑"))
  1420.    return true;
  1430. return false;
```

Fig. 12. Disjointness test optimization pseudo code

optimizedUnsatTestForDisjointness check three conditions listed above. The first condition is checked in lines 1340-1360. In lines 1370-1390 the second condition is checked. Finally, the third condition is checked in lines 1400-1420.

8.2. Disjunctive concepts at nodes

8.2.1 The node matching problem by an example

Now, we allow for the concepts at nodes to contain conjunctions and disjunctions in any order. Suppose, we want to match C_5 in the tree A and C_5 in the tree B in Figure 1. The relevant part of `cLabsMatrix` is shown in Table 8.

Table 8. `cLabsMatrix:` matrix of relations among the atomic concepts of labels.

A \ B	*Classes*	*Mechanics*	*Optics*	*Statistics*	*Dynamics*	*Kinematics*
Courses	=					
Biology						
Zoology						
Botany						
Neurobiology						
Genetics						
Physiology						

As from Table 4, the *axioms* is as follows:

$$(course_A \leftrightarrow class_B) \tag{21}$$

Eq.21 in CNF then becomes:

$$(\neg course_A \vee class_B) \wedge (course_A \vee \neg class_B) \tag{22}$$

As from *Step 2*, $context_A$ and $context_B$ are:

$$class_B \wedge (mechanics_B \vee optics_B \vee thermodynamics_B) \wedge (statics_B \vee dynamics_B \vee kinematics_B) \tag{23}$$

$$course_A \wedge (biology_A \vee zoology_A \vee botany_A) \wedge (neurobiology_A \vee genetics_A \vee physiology_A) \tag{24}$$

The negations of $context_A$ and $context_B$, in turn, are:

$$\neg class_B \vee (\neg mechanics_B \wedge \neg optics_B \wedge \neg thermodynamics_B) \vee (\neg statics_B \wedge \neg dynamics_B \wedge \neg kinematics_B) \tag{25}$$

$$\neg course_A \vee (\neg biology_A \wedge \neg zoology_A \wedge \neg botany_A) \vee (\neg neurobiology_A \wedge \neg genetics_A \wedge \neg physiology_A) \tag{26}$$

The matching result for this task is presented in Table 9.

Table 9. `cNodesMatrix`: matrix of relations among the concepts at nodes (matching result).

A \ B	C_1	C_2	C_5
C_1	=	*idk*	*idk*
C_2	*idk*	*idk*	*idk*
C_5	*idk*	*idk*	*idk*

8.2.2 Optimizations

As from Table 4, *axioms* is the same for all the tests. However, $context_A$ and $context_B$ may contain any number of disjunctions. Some of them are coming from the concepts of labels, while others may appear from the negated $context_A$ or $context_B$ (e.g., see tests for less/more general relations). Thus, for instance, as from Table 4 in case of test for less general relation we obtain the following formula:

$$(\neg course_A \vee class_B) \wedge (course_A \vee \neg class_B) \wedge (mechanics_B \vee optics_B \vee thermodynamics_B) \wedge (statics_B \vee dynamics_B \vee kinematics_B) \wedge ((\neg biology_A \wedge \neg zoology_A \wedge \neg botany_A) \vee (\neg neurobiology_A \wedge \neg genetics_A \wedge \neg physiology_A)) \quad (27)$$

With disjunctive concepts at nodes, Eq. 2 is a full propositional formula and no hypothesis can be made on its structure. As a consequence, its satisfiability must be tested using a standard DPLL SAT solver. Thus, for instance, CNF conversion of Eq. 27 is as follows:

$$(\neg course_A \vee class_B) \wedge (course_A \vee \neg class_B) \wedge (mechanics_B \vee optics_B \vee thermodynamics_B) \wedge (statics_B \vee dynamics_B \vee kinematics_B) \wedge ((\neg course_A \vee \neg biology_A \vee \neg neurobiology_A) \wedge (\neg course_A \vee \neg biology_A \vee \neg genetics_A) \wedge (\neg course_A \vee \neg biology_A \vee \neg physiology_A) \wedge (\neg course_A \vee \neg zoology_A \vee \neg neurobiology_A) \wedge (\neg course_A \vee \neg zoology_A \vee \neg genetics_A) \wedge (\neg course_A \vee \neg zoology_A \vee \neg physiology_A) \wedge (\neg course_A \vee \neg botany_A \vee \neg neurobiology_A) \wedge (\neg course_A \vee \neg botany_A \vee \neg genetics_A) \wedge (\neg course_A \vee \neg botany_A \vee \neg physiology_A)) \quad (28)$$

In order to avoid the space explosion, which may arise when converting a formula into CNF (see for instance Eq. 28), we apply a set of structure preserving transformations [41,19]. The main idea is to replace disjunctions occurring in the original formula with newly introduced variables and explicitly state that these variables imply the subformulas they substitute. Consider for instance Eq. 27. We obtain:

$$(\neg course_A \vee class_B) \wedge (course_A \vee \neg class_B) \wedge (mechanics_B \vee optics_B \vee thermodynamics_B) \wedge (statics_B \vee dynamics_B \vee kinematics_B) \wedge new_1 \wedge new_2 \wedge (new_1 \rightarrow \neg biology_A \vee \neg zoology_A \vee \neg car_A) \wedge (new_2 \rightarrow \neg neurobiology_A \vee \neg genetics_A \vee \neg physiology_A) \quad (29)$$

where new_1 and new_2 stand for newly introduced variables. Eq. 29 is converted into CNF as follows:

$$(\neg course_A \vee class_B) \wedge (course_A \vee \neg class_B) \wedge (mechanics_B \vee optics_B \vee thermodynamics_B) \wedge (statics_B \vee dynamics_B \vee kinematics_B) \wedge new_1 \wedge new_2 \wedge (\neg new_1 \vee \neg biology_A \vee \neg zoology_A \vee \neg car_A) \wedge (\neg new_2 \vee \neg neurobiology_A \vee \neg genetics_A \vee \neg physiology_A) \quad (30)$$

Notice that the size of the propositional formula in CNF grows linearly with respect to number of disjunctions in original formula. To account for this optimization in **nodeMatch** all calls to **convertToCNF** are replaced with calls to **optimizedConvertToCNF**, (see Figure 13):

```
1120. formulaInCNF=optimizedConvertToCNF(formula);
...
1170. formulaInCNF=optimizedConvertToCNF(formula);
...
1200. formulaInCNF=optimizedConvertToCNF(formula);
```

Fig. 13. The CNF conversion optimization pseudo code

9. Evaluation

In this section, we present the performance and quality evaluation of the matching system we have implemented, called S-Match. In particular, we evaluate basic and optimized versions of our system, called (S-Match$_B$) and (S-Match) respectively, against three state of the art matchers, namely Cupid [32], COMA [11][9], and SF [35] as implemented in Rondo [36]. 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 the way they compute mappings.

9.1. Evaluation set-up

The evaluation was performed on seven matching tasks from different application domains, see Table 10. There are three matching tasks from a business domain (#1,3,5). The first business example (#1) describes two company profiles: a Standard one (mini) and Yahoo Finance (mini), while, #5, represents their full versions. The third business example (#3) deals with BizTalk[10] purchase order schemas. There is one matching task from an academy domain (#2). It describes courses taught at Cornell University (mini) and at the University of Washington (mini). Finally, there are three matching tasks on general topics (#4,6,7) as represented by the well-known web directories, such as Google[11], Yahoo[12], and Looksmart[13]. Table 10 provides some indicators of the complexity of these test cases[14].

Table 10. Some indicators of the complexity of the test cases.

#	Matching task	Max. depth	# nodes	# labels	Concepts at nodes
1	***Yahoo(mini)-Standard(mini)***	2/2	10/16	22/45	Conjunctive Disjunctive
2	***Cornell-Washington***	3/3	34/39	62/64	Conjunctive Disjunctive
3	***CIDX – Excel***	3/3	34/39	56/58	Conjunctive Disjunctive
4	***Looksmart-Yahoo***	10/8	140/74	222/101	Conjunctive Disjunctive
5	***Yahoo-Standard***	3/3	333/115	965/242	Conjunctive Disjunctive
6	***Google-Yahoo***	11/11	561/665	722/945	Conjunctive Disjunctive
7	***Google-Looksmart***	11/16	706/1081	1048/1715	Conjunctive Disjunctive

[9] We thank to 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 [11]. A newer version of the system COMA++ exists but we do not have it.

[10] http://www.microsoft.com/biztalk/

[11] http://www.google.com/Top/

[12] http://dir.yahoo.com/

[13] http://www.looksmart.com/

[14] Source files and description of the schemas tested can be found at our project web-site, experiments section: http://www.dit.unitn.it/~accord/

The reference mappings (also called expert mappings) for some of these problems (namely for the tasks #1,2,3) were established manually. Then, the results computed by the systems have been compared with expert mappings. It is worth noticing that the task of creation of expert mappings is an error-prone and a time consuming one. Even if for the moment of writing this paper we have created expert mappings for the biggest matching tasks (#6,7) of Table 10, we do not report these findings in this paper. Addressing in full detail the emerged issues along that process as well as the matching results achieved is out of scope of this paper, see for some details [3,22]. Thus, in this evaluation study we focus mostly on the performance characteristics of S-Match, involving large matching tasks, namely schemas with hundreds and thousands of nodes. Notice, scalability properties of matching systems is among the most important problems of schema matching (in general) these days, see e.g., [7,12]. Quality characteristics of the S-Match results which are presented here address only medium size schemas. We acknowledge that a large-scale quality evaluation is also of high importance. However, we view it as a separate direction, requiring (beyond some preliminary results of [3,22]) further in-depth investigations. Thus, we pose it as future work.

There are three further observations that ensure a fair (qualitative) 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 others, discovers the disjointness relation which can be interpreted as strong dissimilarity in terms of other systems under consideration. Therefore, we did not take into account the disjointness relations when specifying the expert mappings. The second observation is that, since S-Match returns a matrix of relations, while all 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* by the core relations. 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.

As match quality measures we have used the following indicators: *precision*, *recall*, *overall*, and *F-measure*. *Precision* varies in the [0,1] range; the higher the value, the smaller 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 the set of correct mappings (true positives) which have not 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. Time is very important for us, since it shows the ability of matching systems to scale up.

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). We report the best results obtained with respect to directional-

ity, 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 14).

All the tests have been performed on a P4-1700, with 512 MB of RAM, with the Windows XP operating system, and with no applications running but a single matching system. The systems were limited to allocate no more than 512 MB of memory. All the tuning parameters (e.g., thresholds, combination 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. S-Match was also used in default configuration, e.g., threshold for string-based matchers was 0.6. This threshold has been defined after experimentation on several schema matching tasks (see for details the URL in footnote 14). Finally, all the element level matchers of the third approximation level (e.g., gloss-based matchers) were not involved in the evaluation since all the matching tasks under consideration were successfully resolved by the matchers of Table 1 which belong to the first and the second approximation levels; see [22] for the preliminary evaluation results of matchers belonging to the third approximation level as well as for the tasks where they are useful.

9.2. Evaluation results

We present the time performance results for all the tasks of Table 10, while quality results, as from the previous discussion are possible to estimate only for some of the matching tasks (#1,2,3). The evaluation results for the matching problems #1,2,3 are shown in Figure 14.

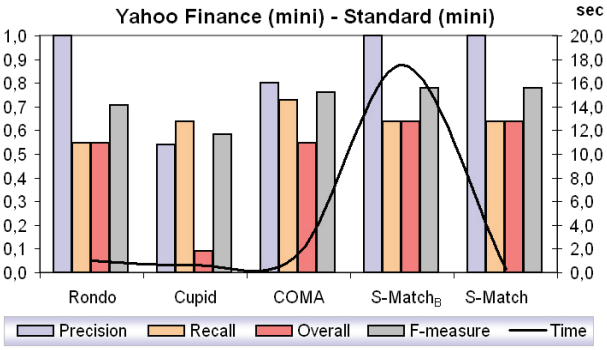

Fig.14.1 Evaluation results: Yahoo Finance (mini) vs. Standard (mini), test case #1

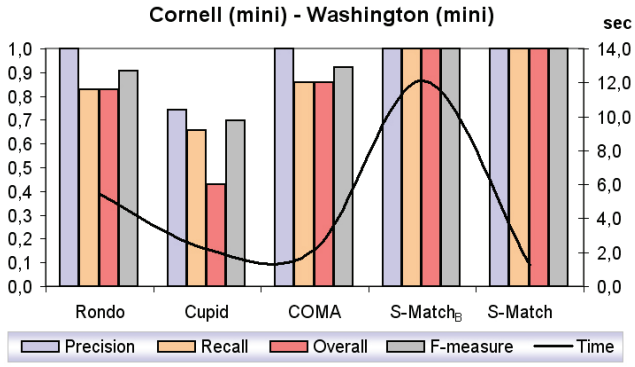

Fig. 14.2 Evaluation results: Cornell (mini) vs. Washington (mini), test case #2

For example, in Figure 14.2, 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.

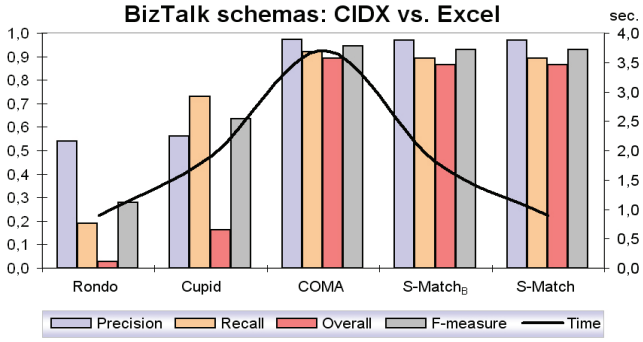

Fig. 14.3 Evaluation results: CIDX vs. Excel, test case #3

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

The time performance results obtained for the matching tasks #4,5,6,7 of Table 10 are presented in Figure 15. Cupid went out of memory on all the tasks. Therefore, we present the results for other systems.

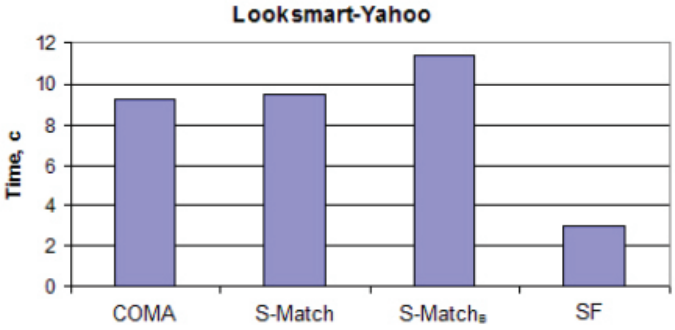

Fig. 15.1. Execution times: Looksmart vs. Yahoo, test case #4

In the case of Looksmart-Yahoo matching problem the trees contain about hundred nodes each. S-Match works about 18% faster than S-Match$_B$ and about 2% slower than COMA. SF, in turn, works about 3 times faster than S-Match. The relatively poor improvement (18%) occurs because our optimizations are implemented in a straightforward way. More precisely, on small trees (e.g., test case #4) a big constant factor[15] dominates the growth of all other components in S-Match computational complexity formula.

On Yahoo-Standard matching problem S-Match works about 40% faster than S-Match$_B$. It performs 1% faster than COMA and about 5 times slower than SF. The relatively small improvement in this case can be explained by noticing that the maximum depth in both trees is 3 and that the average number of labels at nodes is about 2. The optimizations cannot significantly influence the system performance.

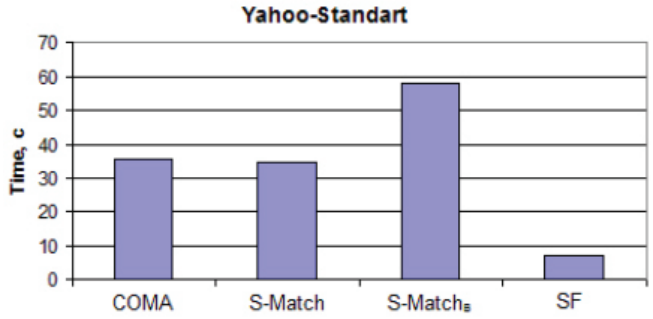

Fig. 15.2. Execution times: Yahoo vs. Standard, test case #5

The next two matching problems are much bigger than the previous ones. They contain hundreds and thousands of nodes. On these trees SF went out of memory. Therefore, we provide the results only for the other systems. In the case of Google-Yahoo matching task S-Match is more than 6 times faster than S-Match$_B$. COMA performs about 5 times slower than the optimized version. These results suggest that the optimizations described in this paper are better suited for big trees. In the case of the

[15] This is also known in the literature as an implementational constant.

biggest matching problem, involving Google-Looksmart, S-Match performs about 9 times faster than COMA, and about 7 times faster than S-Match$_B$.

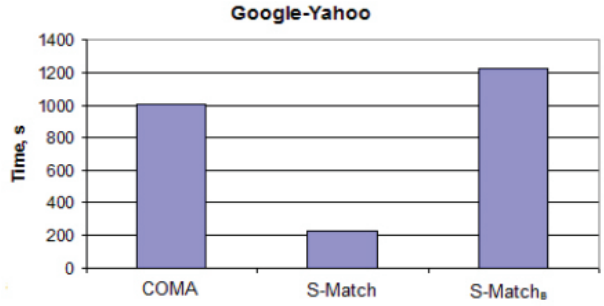

Fig.15.3. Execution times: Google vs. Yahoo, test case #6

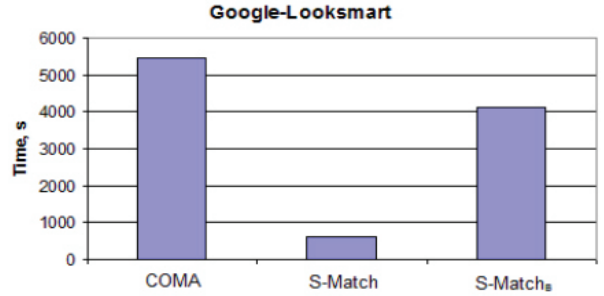

Fig. 15.4. Execution times: Google vs. Looksmart, test case #7

Having considered matching tasks of Table 10, we conclude that S-Match performs (in terms of execution time) slightly slower than COMA and SF on the schemas with one up to three hundred of nodes (see, Figures 15.1-15.2). At the same time, S-Match is considerably faster on the schemas with more than five hundreds nodes (see, Figures 15.3-15.4), thereby indicating system scalability.

9.3. 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.

We pose a large-scale qualitative evaluation of the system as future work. Thus, in our evaluation we have focused only on the overall qualitative system results, hence, not discussing exhaustively element level matchers, e.g., by showing impact of each of them on the matching results (see, for some preliminary results [22]). Also, it is worth mentioning that, e.g., string-based matchers, have already been extensively evaluated in [11,44].

Performance measures. *Time* is an 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 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).

10. Related Work

At present, there exists a line of semi-automated schema matching systems, see, for instance [5,10,13,15,32,30,35,39,49,28,46]. A good survey and a classification of matching approaches up to 2001 is provided in [42], an extension of its schema-based part and a user-centric classification of matching systems is provided in [43], while the work in [14] considers both [42, 43] as well as some other classifications.

In particular, for individual matchers, [43] introduces the following criteria which allow for detailing further (with respect to [42]), 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 [37]), 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 [42] is useful from an architectural perspective. [43] 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., [13,35]); *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., [32,11]).

Let us consider the closest to S-Match schema-based state of the art systems in light of the above criteria.

Rondo. The Similarity Flooding (SF) [35] approach, as implemented in Rondo [36], 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 nodes' 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 [32] 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 [11] 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.

Reduction of semantic heterogeneity is typically performed in two steps. So far, we have concentrated on the first step, namely on determining correspondences between semantically related entities. The second step is the ultimate goal of the matching exercise, which can be *data translation, query answering,* and so on. Here, mappings are taken as input and are analyzed in order to generate, e.g., query expressions that automatically translate/exchange data instances between the information sources, see, for example, [16,48]. Notice that taking as input semantic relations, instead of coefficients in the [0,1] range, potentially enables, e.g., data translation systems to produce better results, since, for example, in such systems as Clio [16], the fist step is to interpret the correspondences by giving them a clear semantics.

11. Conclusions

We have presented a new semantic schema matching algorithm and its optimizations. Our solution builds on 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 strength of our approach.

Future work includes development of an *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. Also, we are planning to extend the semantic matching approach by computing the *overlapping relation* (with the intersection semantics). This relation might be useful when, e.g., input schemas encode a domain of interest at different levels of details. 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 [3]. Here, the key issue is that in these cases, specifying expert mappings manually is neither desirable nor feasible task, thus a semi-automatic approach is needed. Comparison of matching algorithms on large real-world schemas from different application domains will also be performed *extensively*.

References

1. P. Atzeni, P. Cappellari, and P. Bernstein. Model-independent schema and data translation. In *Proceedings of EDBT*, pages 368–385, 2006.
2. P. Atzeni, P. Cappellari, and P. Bernstein. Modelgen: model independent schema translation. In *Proceedings of ICDE*, pages 1111–1112, 2005.
3. P. Avesani, F. Giunchiglia, and M. Yatskevich. A large scale taxonomy mapping evaluation. In *Proceedings of ISWC*, pages 67-81, 2005.
4. F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider (eds.). *The Description Logic Handbook*. Cambridge University Press, 2002.
5. S. Bergamaschi, S. Castano, and M. Vincini. Semantic integration of semistructured and structured data sources. *SIGMOD Record*, pages 54–59, 1999.
6. P. Bouquet, L. Serafini, and S. Zanobini. Semantic coordination: A new approach and an application. In *Proceedings of ISWC*, pages 130–145, 2003.
7. P. Bernstein, S. Melnik, M. Petropoulos, and C. Quix. Industrial-strength schema matching. *SIGMOD Record*, 33(4):38 – 43, 2004.
8. M. Davis, G. Longemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, (5(7)), 1962.
9. M. Davis and H. Putnam. A computing procedure for quantification theory. In *Journal of the ACM*, number 7, pages 201–215, 1960.
10. 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.
11. H. H. Do and E. Rahm. COMA - a system for flexible combination of schema matching approaches. In *Proceedings of VLDB*, pages 610–621, 2002.
12. A. Doan and A. Halevy. Semantic integration research in the database community: A brief survey. *AI Magazine, Special Issue on Semantic Integration*, 2005.
13. J. Euzenat and P. Valtchev. Similarity-based ontology alignment in OWL-lite. In *Proceedings of ECAI*, pages 333–337, 2004.
14. J. Euzenat and P. Shvaiko. *Ontology matching*. Springer, 2007 (to appear).
15. A. Gal, A. Anaby-Tavor, A. Trombetta, and D. Montesi. A framework for modeling and evaluating automatic semantic reconciliation. *The VLDB Journal*, (14(1)): 50-67, 2005.
16. L. M. Haas, M. A. Hernandez, H. Ho, L. Popa, and M. Roth. Clio grows up: from research prototype to industrial tool. In *Proceedings of SIGMOD,* pages 805–810, 2005.
17. F. Giunchiglia: Contextual reasoning. *Epistemologia, special issue on "I Linguaggi e le Macchine"*, vol. XVI, (1993) 345-364.
18. F. Giunchiglia, M. Marchese, and I. Zaihrayeu. Encoding Classifications into Lightweight Ontologies. In *Proceedings of ESWC*, pages 80-94, 2006.
19. E. Giunchiglia, R. Sebastiani. Applying the Davis-Putnam procedure to non-clausal formulas. In *Proceedings of AI*IA*, pages 84-94, 1999.
20. F. Giunchiglia and P. Shvaiko. Semantic matching. *The Knowledge Engineering Review Journal*, (18(3)):265-280, 2003.
21. 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.
22. F. Giunchiglia, P. Shvaiko, and M. Yatskevich. Discovering Missing Background Knowledge in Ontology Matching. In *Proceedings of ECAI*, pages 382-386, 2006.
23. F. Giunchiglia, P. Shvaiko, and M. Yatskevich. Semantic schema matching. In *Proceedings of CoopIS*, pages 347-365, 2005.
24. F. Giunchiglia and M. Yatskevich. Element level semantic matching. In *Proceedings of Meaning Coordination and Negotiation workshop at ISWC, 2004*.
25. F. Giunchiglia, M. Yatskevich, and E. Giunchiglia. Efficient semantic matching. In *Proceedings of ESWC*, pages 272-289, 2005.

26. N. Guarino. The role of ontologies for the Semantic Web (and beyond). Technical report, Laboratory for Applied Ontology, ISTC-CNR, 2004.
27. V. Haarslev, R. Moller, and M. Wessel. RACER: Semantic middleware for industrial projects based on RDF/OWL, http://www.sts.tu-harburg.de/~r.f.moeller/racer/.
28. B. He and K. C.-C. Chang. Automatic Complex Schema Matching across Web Query Interfaces: A Correlation Mining Approach. *ACM Transactions on Database Systems*, 31(1): 346 – 395, 2006.
29. N. Ide and J. Veronis. Word Sense Disambiguation: the state of the art. *Computational linguistics*, 24(1):1–40, 1998.
30. J. Kang and J.F. Naughton. On schema matching with opaque column names and data values. In *Proceedings of SIGMOD*, pages 205–216, 2003.
31. D. Le Berre SAT4J: A satisfiability library for Java. http://www.sat4j.org/.
32. J. Madhavan, P. Bernstein, and E. Rahm. Generic schema matching with Cupid. In *Proceedings of VLDB*, pages 49–58, 2001.
33. J. Madhavan, P. Bernstein, A. Doan, and A. Halevy. Corpus-based schema matching. In *Proceedings of ICDE*, pages 57–68, 2005.
34. B. Magnini, L. Serafini, and M. Speranza. Making explicit the semantics hidden in schema models. In *Proceedings of the workshop on Human Language Technology for the Semantic Web and Web Services at ISWC*, 2003.
35. S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity Flooding: A versatile graph matching algorithm. In *Proceedings of ICDE*, pages 117–128, 2002.
36. S. Melnik, E. Rahm, and P. Bernstein. Rondo: A programming platform for generic model management. In *Proceedings of SIGMOD*, pages 193–204, 2003.
37. A.G. Miller. WordNet: A lexical database for English. *Communications of the ACM*, 38(11), pages 39–41, 1995.
38. J. Z. Pan. *Description Logics: reasoning support for the Semantic Web*. PhD thesis, School of Computer Science, The University of Manchester, 2004.
39. G. A. Modica, A. Gal, and H. M. Jamil. The use of machine-generated ontologies in dynamic information seeking. In *Proceedings of CoopIS*, pages 433–448, 2001.
40. N. Noy. Semantic Integration: A survey of ontology-based approaches. *SIGMOD Record*, 33(4):65–70, 2004.
41. D. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, (2):293–304, 1986.
42. E. Rahm and P. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, (10(4)):334–350, 2001.
43. P. Shvaiko and J. Euzenat. A survey of schema-based matching approaches. *Journal on Data Semantics*, IV:146-171, 2005.
44. G. Stoilos, G. B. Stamou, S. D. Kollias. A String Metric for Ontology Alignment. In *Proceedings of ISWC*, pages 624-637, 2005.
45. 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.
46. W. Su, J. Wang, and F. Lochovsky. Holistic Schema Matching for Web Query Interface. In *Proceedings of EDBT*, pages 77-94, 2006.
47. G. Tsetin. On the complexity proofs in propositional logics. *Seminars in Mathematics*, 8, 1970.
48. Y. Velegrakis, J. Miller, and Lucian Popa. Preserving mapping consistency under schema changes. *The VLDB Journal*, 13(3):274–293, 2004.
49. P. Ziegler, C. Kiefer, C. Sturm, K. Dittrich, and A. Bernstein. Detecting Similarities in Ontologies with the SOQA-SimPack Toolkit. In *Proceedings of EDBT*, pages 59-76, 2006.

Appendix A. Pseudo code of the optimized algorithm

```
Node struct of
              int nodeId;
              String label;
              String cLabel;
              String cNode;
              AtomicConceptAtLabel[] ACOLs;

AtomicConceptOfLabel struct of
              int id;
              String token;
              String[] wnSenses;



10.void elementLevelSenseFiltering(Node node)
20. AtomicConceptOfLabel[] nodeACOLs=getACOLs(node);
30. for each nodeACOL in nodeACOLs
40.  String[] nodeWNSenses=getWNSenses(nodeACOL);
50.  for each ACOL in nodeACOLs
60.   if (ACOL!=nodeACOL)
70.    String[] wnSenses=getWNSenses(ACOL);
80.     for each nodeWNSense in nodeWNSenses
90.      for each wnSense in wnSenses
100.       if (isConnectedbyWN(nodeWNSense, focusNodeWNSense))
110.        addToRefinedSenses(nodeACOL,nodeWNSense);
120.        addToRefinedSenses(focusNodeACOL, focusNodeWNSense);
131.   saveRefinedSenses(context);



140. void saveRefinedSenses(context)
150.   for each node in context
160.     AtomicConceptOfLabel[] nodeACOLs=getACOLs(node);
170.    for each nodeACOL in NodeACOLs
180.      if (hasRefinedSenses(nodeACOL))
190.        //replace original senses with refined



200. void buildCLab(Tree of Nodes context)
210.  String[] wnSenses;
220.  For each node in context
230.    String cLabFormula="";
240.    String nodeLabel=getLabel(node);
250.    String[] tokens=tokenize(nodeLabel);
260.    String[] lemmas=lematize(tokens);
270.    For each lemma in lemmas
280.      if (isMeningful(lemma))
290.        if (!isInWordnet(lemma))
300.         addACOLtoNode(node, lemma,SENSES_NOT_FOUND);
310.       else
320.         wnSenses= getWNSenses(token);
330.         addACOLtoNode(node, lemma, wnSenses);
340.     cLabFormula=constructcLabFormula(cLabFormula, lemma);
350.   setcLabFormula(node, cLabFormula);
360.   elementLevelSenseFiltering(node);
```

```
400.void structureLevelSenseFiltering (Node node, Tree of Nodes context)
410.  AtomicConceptOfLabel[] focusNodeACOLs;
420.  Node[] focusNodes=getFocusNodes(node, context);
430.  AtomicConceptOfLabel[] nodeACOLs=getACOLs(node);
440.  for each nodeACOL in nodeACOLs
450.    String[] nodeWNSenses=getWNSenses(nodeACOL);
460.    for each nodeWNSense in nodeWNSenses
470.      for each focusNode in focusNodes
480.        focusNodeACOLs=getACOLs(focusNode);
490.        for each focusNodeACOL in focusNodeACOLs
500.          String[] fNodeWNSenses=getWNSenses(focusNodeACOL);
510.          for each fNodeWNSense in nodeWNSenses
520.            if (isConnectedbyWN(nodeWNSense, fNodeWNSense))
530.              addToRefinedSenses(nodeACOL,nodeWNSense);
540.              addToRefinedSenses(focusNodeACOL, focusNodeWNSense);
550.  saveRefinedSenses(context);



600. buildCNode(Tree of Node context)
610.  for each node in context
620.    String cNodeFormula= buildcNodeFormula (node, context);
630.    structureLevelSenseFiltering (node,context);
640.    if (isUnsatisifiable(cNodeFormula))
650.      updateFormula(cNodeFormula);
      //error please reconcile the context



700. String[][] fillCLabMatrix(Tree of Nodes source,target);
710. String[][]cLabsMatrix;
720. String[] matchers;
730. int i,j;
740. matchers=getMatchers();
750. for each sourceAtomicConceptOfLabel in source
760.  i=getACoLID(sourceAtomicConceptOfLabel);
770.  for each targetAtomicConceptOfLabel in target
780.   j= getACoLID(targetAtomicConceptOfLabel);
790.   cLabsMatrix[i][j]=getRelation(matchers,
               sourceAtomicConceptOfLabel,targetAtomicConceptOfLabel);

800. String getRelation(String[] matchers,
                          AtomicConceptOfLabel source, target)
810.  String matcher;
820.  String relation=”Idk”;
830.  int i=0;
840.  while ((i<sizeof(matchers))&&(relation==”Idk”))
850.   matcher= matchers[i];
860.   relation=executeMatcher(matcher,source,target);
870.   i++;
880.  return relation;
```

```
900.String[][] treeMatch(Tree of Nodes source, target,
                                    String[][] cLabsMatrix)
910. Node sourceNode,targetNode;
920. String[][]cNodesMatrix, relMatrix;
930. String axioms, context_A, context_B;
940. int i,j;
950. cLabsMatrix=fillCLabMatrix(source,target);
960. For each sourceNode in source
970.  i=getNodeId(sourceNode);
980.  context_A=getCnodeFormula (sourceNode);
990.  For each targetNode in target
1000.    j=getNodeId(targetNode);
1010.    context_B=getCnodeFormula (targetNode);
1020.    relMatrix=extractRelMatrix(cLabsMatrix, sourceNode,
                                    targetNode);
1030.    axioms=mkAxioms(relMatrix);
1040.    cNodesMatrix[i][j]=nodeMatch(axioms,context_A,context_B);
1050. return cNodesMatrix;



1100.String nodeMatch(String axioms, context_A, context_B)
1105. if (context_A and context_B are conjunctive)
1106.  isOpposite=optimizedUnsatTestForDisjointness (axioms, context_A,
                                                        context_B);
1107. else
1110.  formula= And(axioms, context_A, context_B);
1120. formulaInCNF=optimizedConvertToCNF(formula);
1130.  boolean isOpposite= isUnsatisfiable(formulaInCNF);
1140. if (isOpposite)
1150.   return "⊥";
1155.  if (context_A and context_B are conjunctive)
1156.    isLG=fastHornUnsatCheck (context_A, context_B, axioms, "⊑","⊒");
1157.    isMG=fastHornUnsatCheck (context_B, context_A, axioms, "⊒","⊑");
1158.  else
1160.  String formula=And(axioms,context_A,Not(context_B));
1170. formulaInCNF=optimizedConvertToCNF(formula);
1180.  boolean isLG=isUnsatisfiable(formulaInCNF)
1190.  formula=And(axioms, Not(context_A), context_B);
1200. formulaInCNF=optimizedConvertToCNF(formula);
1210.  boolean isMG= isUnsatisfiable(formulaInCNF);
1220. if (isMG && isLG)
1230.   return "=";
1240. if (isLG)
1250.   return "⊑";
1260. if (isMG)
1270.   return "⊒";
1280. return "Idk";
```

```
1300. optimizedUnsatTestForDisjointness (axioms, context_A, context_B);
1310. for each axiom in axioms
1320.  String A_i= getFirstVariable(axiom);
1330.  String B_j= getSecondVariable(axiom);
1340.  if ((occurs_positively (A_i, context_A))&&(occurs_positively (B_j,
                                                          context_B)))
1350.   if (getAType(axiom)="⊥")
1360.    return true;
1370.  if ((occurs_negatively (A_i, context_A))&&(occurs_positively (B_j,
                                                          context_B)))
1380.  if((getAType(axiom)="=")||(getAType(axiom)="⊒"))
1390.    return true;
1400.  if ((occurs_positevely (A_i, context_A))&&(occurs_negatively (B_j,
                                                          context_B)))
1410.  if((getAType(axiom)="=")||(getAType(axiom)="⊑"))
1420.    return true;
1430. return false;



1500.boolean fastHornUnsatCheck(String context, neg_context, axioms,
                                                    rel, neg_rel)
1510. int m=getNumOfVar(String neg_context);
1520. boolean array[m];
1530. for each axiom in axioms
1540.  String A_i= getFirstVariable(axiom);
1550.  String B_j= getSecondVariable(axiom);
1560.  int j=getNumberInContext(B_j);
1570.  if((occurs_positevely (A_i, context))&&(occurs_positevely (B_j,
                                                    neg_context)))
1580.   if((getAType(axiom)="=")||(getAType(axiom)=rel))
1590.    array[j]=true;
1600.  if ((occurs_negatively (A_i, context))&&(occurs_negatively (B_j,
                                                    neg_context)))
1610.   if((getAType(axiom)="=")||(getAType(axiom)=neg_rel))
1620.    array[j]=true;
1630.  if ((occurs_positevely (A_i, context))&&(occurs_negatively (B_j,
                                                    neg_context)))
1640.   if(getAType(axiom)="⊥")
1650.    array[j]=true;
1660. for (i=0; i<m; i++)
1670.  if (!array[i])
1680.   return false;
1690. return true;
```