Semantic Web 1 (2010) 1–9
IOS Press

S-Match: an open source framework for matching lightweight ontologies

Editor(s): Jie Tang, Tsinghua University Beijing, China
Solicited review(s): Ming Mao, SAP Research, Palo Alto, CA, U.S.A.; Wei Hu, Nanjing University, China; Shenghui Wang, Vrije Universiteit Amsterdam, The Netherlands
Open review(s): Prateek Jain, Kno.e.sis Center, Wright State University, Dayton, OH, U.S.A.

Fausto Giunchiglia [a], Aliaksandr Autayeu [a] and Juan Pane [a]

[a] *DISI, via Sommarive, 14, 38123 Trento, Italy*
E-mail: {name.surname}@disi.unitn.it

Abstract.
Achieving automatic interoperability among systems with diverse data structures and languages expressing different viewpoints is a goal that has been difficult to accomplish. This paper describes S-Match, an open source semantic matching framework that tackles the semantic interoperability problem by transforming several data structures such as business catalogs, web directories, conceptual models and web services descriptions into lightweight ontologies and establishing semantic correspondences between them. The framework is the first open source semantic matching project that includes three different algorithms tailored for specific domains and provides an extensible API for developing new algorithms, including possibility to plug-in specific background knowledge according to the characteristics of each application domain.

Keywords: data integration, semantic matching, lightweight ontologies, open source framework

1. Introduction

Interoperability among different viewpoints and languages which use different terminology and where knowledge can be expressed in diverse forms is a difficult problem. With the advent of the Web and the consequential information explosion, the problem seems to be emphasized. People face these concrete problems when retrieving, disambiguating and integrating information coming from a wide variety of sources. Many of these sources of information can be represented using lightweight ontologies, which provide the formal representation upon which it is possible to reason automatically about hierarchical structures such as classifications, database schemas, business catalogs, and file system directories, among others.

Semantic matching constitutes a fundamental technique which applies in many areas such as resource discovery, data integration, data migration, query translation, peer to peer networks, agent communication, schema and ontology merging. Semantic matching is a type of ontology matching technique that relies on semantic information encoded in lightweight ontologies to identify nodes that are semantically related. It operates on graph-like structures and has been proposed as a valid solution to the semantic heterogeneity problem, namely managing the diversity in knowledge [1].

S-Match[1] is an open source semantic matching framework that provides several semantic matching algorithms and facilities for the development of new ones. It includes components for transforming tree-like structures into lightweight ontologies, where each node label in the tree is translated into propositional description logic (DL) formula, which univocally codifies the meaning of the node.

S-Match contains the implementation of the basic semantic matching, the minimal semantic matching, and the structure preserving semantic matching (SPSM) algorithms. The basic semantic matching algorithm is a general purpose matching algorithm, very customizable and suitable for many applications. Minimal semantic matching algorithm exploits additional knowledge encoded in the structure of the input and

[1] `http://s-match.org/`

1570-0844/10/$27.50 © 2010 – IOS Press and the authors. All rights reserved

is capable of producing minimal mapping and maximal mapping. SPSM is a type of semantic matching producing a similarity score and a mapping preserving structural properties: (i) one-to-one correspondences between semantically related nodes; (ii) functions are matched to functions and variables to variables.

The key contributions of the S-Match framework are:

i) working open source implementation of semantic matching algorithms;
ii) several interfaces ranging from easy to use Graphical User Interface (GUI) and Command Line Interface (CLI) to Application Program Interface (API). They suit different purposes varying from running a quick and easy experiment to embedding S-Match in other projects;
iii) the implementation of three different versions of semantic matching algorithms, each suitable for different purposes, and the flexibility for integrating new algorithms and linguistic oracles tailored of other domains;
iv) an open architecture extensible to work with different data formats with a ready-to-run implementation of the basic formats.

The rest of the paper is organized as follows: Section 2 introduces lightweight ontologies and enumerates several data structures that can be transformed into them; Section 3 gives an overview of the semantic matching algorithm and the different versions that are supported by S-Match; Section 4 presents the general architecture of the framework and the macro-level components; Section 5 gives an introduction to different interfaces available in the framework. Finally, Section 6 provides a summary of the open source project hosting the open source framework.

2. Lightweight ontologies

Classification structures such as taxonomies, business catalogs[2], web directories[3] and user directories in the file system, among others, are perhaps the most natural tools used by humans to organize information content. The information is hierarchically arranged under topic nodes moving from general ones to more specific ones as we go deeper in the hierarchy. This attitude is known in Knowledge Organization as the *get-specific* principle [2].

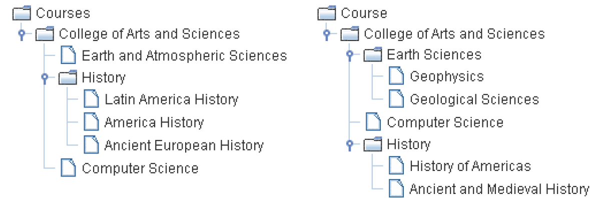

Fig. 1. Two example course catalogs to be matched

The information in the classification is normally described using natural language labels (see Figure 1), which has proven to be very effective in manual tasks (for example, for manual indexing and manually navigating the tree). However, these natural language labels present limitations when one tries to automate reasoning over them, for instance for automatic indexing, search and semantic matching or when dealing with multiple languages.

Translating the classifications containing natural language labels into their formal counterpart, i.e., lightweight ontologies, is a fundamental step toward being able to automatically work with them. Following the approach described in [2] and exploiting dedicated Natural Language Processing (NLP) techniques tuned to short phrases [3], each node label can be translated into an unambiguous formal expression, i.e., into a propositional Description Logic (DL) expression. As a result, lightweight ontologies, or formal classifications, are tree-like structures where each node label is a language-independent propositional DL formula codifying the meaning of the node. Taking into account its context (namely the path from the root node), each node formula is subsumed by the formula of the node above [4]. As a consequence, the backbone structure of a lightweight ontology is represented by subsumption relations between nodes, i.e., *"the extension of a concept of a child node is a subset of the extension of the concept of the parent node"* [5].

Giunchiglia et al. show in [6], [4] and [7] how lightweight ontologies can be used to automate important tasks, in particular to favor interoperability among different knowledge organization systems. For example, [6] shows how data and conceptual models such as database schemes, object oriented schemes, XML schemes and concept hierarchies can be converted into graph-like structures that can be used as input of the Semantic Matching. In [7] it has been demonstrated how lightweight ontologies can also be used for representing web services and therefore automate the web

[2]`unspsc.org`, `eclass-online.com`
[3]`dmoz.org`, `dir.yahoo.com`

service composition task. This shows that Lightweight Ontologies, while being simple structures, are powerful enough to encode several types of models ranging from data and classification models to service descriptions, reducing the complexity of the semantic interoperability problem (in many cases) to that of matching two lightweight ontologies.

3. Semantic matching

Semantic matching is a type of ontology matching [8] technique that relies on semantic information encoded in lightweight ontologies [2] to identify nodes that are semantically related. A considerable amount of research has been done in this field, which can be seen in extensive surveys [8,9,10], papers, to cite a few [11,12], and in systems, such as Falcon[4], COMA++[5], Similarity Flooding[6], HMatch[7] and others.

S-Match algorithm is an example of semantic matching operator. Given any two graph-like structures, like classifications, database or XML schemas and ontologies, matching is an operator that identifies those nodes in the two structures which semantically correspond to one another. For example, applied to file systems it can identify that a folder labeled "car" is semantically equivalent to another folder "automobile" because they are synonyms in English. This information can be taken from a background knowledge, e.g., a linguistic resource such as WordNet [13].

Figure 1 shows two University course catalogs. This is a typical example of data integration where we need to match these course catalogs in case of a transfer of a student from one University to another, where the receiving university has to decide which courses to recognize from the former University. There are different ways the set of mapping elements can be returned, according to the specifics of the problem where the semantic matching approach is being applied. The following sub-sections introduce different versions of the semantic matching algorithm while Section 4.8 provides more details on how these versions are implemented in S-Match.

[4]ws.nju.edu.cn/falcon-ao
[5]dbs.uni-leipzig.de/Research/coma.html
[6]www-db.stanford.edu/~melnik/mm/sfa/
[7]islab.dico.unimi.it/hmatch/

3.1. The basic algorithm

The basic semantic matching algorithm was introduced in [6] and later extended in [1]. The key intuition of Semantic Matching is to find semantic relations, in the form of *equivalence* ($=$), *less general* ($\sqsubseteq$), *more general* ($\sqsupseteq$) and *disjointness* ($\perp$), between the meanings (concepts) that the nodes of the *lightweight ontologies* represent, and not only the labels [1]. This is done using a four steps approach, namely:

- *Step 1*: Compute the concepts of label, C_Ls;
- *Step 2*: Compute the concepts at node, C_Ns;
- *Step 3*: Compute relations between the concepts of label;
- *Step 4*: Compute relations between the concepts at node;

In the *first step*, the natural language labels of the nodes are analyzed with the aid of a linguistic oracle (e.g., WordNet) in order to extract the intended meaning. The intended meaning is a *complex concept*, made of *atomic concepts*, which roughly correspond to individual words. The output of this step is a language independent Description Logic (DL) formula that encodes the meaning of the label. The main objective of this step is to unequivocally encode the intended meaning of the label, thus eliminating possible ambiguities introduced by the natural language such as, but not only, homonyms and synonyms. If we consider the example in Figure 1, we need to disambiguate the word "History". This is a trivial task for a human, which would know that "History" is being used in the sense of *humanistic discipline* as opposed to the *past times* sense, but this task has proven to be difficult to automate. As SENSEVAL competitions show, this problem is noted for particularly difficult to beat simple baseline approaches. For example, a 4% improvement over a baseline is considered good [14].

In the *second step*, we compute the meaning of the node considering its position in the tree, namely, the path of the particular node to the root. For example, in the node "History" of Figure 1 by having computed the C_L in the previous step, we know exactly which "History" we are referring to. In this step we restrict the extension of the concept to that of only "Courses" about "History", and not everything about "History".

During the *third step*, we compute the relations between the concepts of label of two input trees. This is done with the aid of two kinds of element level matchers: semantic and syntactic. Semantic element level matchers rely on linguistic oracles to find seman-

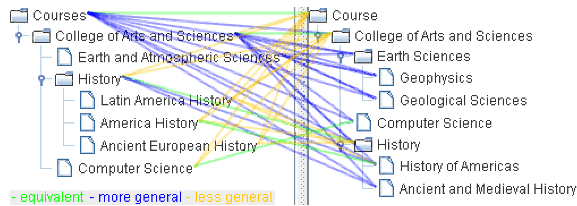

Fig. 2. Result of the basic semantic matching algorithm.

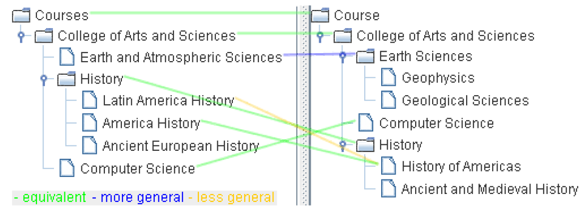

Fig. 3. Result of the minimal semantic matching algorithm.

tic correspondences between the concepts of label. For example, it would discover that the concepts for "car" and "automobile" are synonyms. Instead, syntactic element level matchers, such as *N-Gram* and *Edit Distance* are used if no semantic relation could be found by the semantic element level matcher. The output of this step is a matrix of relations between all atomic concepts encountered in the nodes of both *lightweight ontologies* given as inputs. This constitutes the theory and axioms which will be used in the following step (see [1] for complete details).

In the *fourth step*, we compute the semantic relations $(=, \sqsupseteq, \sqsubseteq, \perp)$ between the concepts at node (C_Ns). By relying on the axioms built on the previous step, the problem is reformulated as a propositional satisfiability (SAT) problem between each pair of nodes of the two input lightweight ontologies. This problem is then solved by a sound and complete SAT engine.

Steps 1 and 2 need to be done only once (preferably off-line) for each input tree. Then, each time the tree needs to be matched, these steps can be skipped and the resulting lightweight ontologies can be used directly. Steps 3 and 4 can only be done at runtime, since they require both input lightweight ontologies in order to compute the mapping elements.

The output of the Semantic Matching is a set of *mapping elements* in the form $\langle N_1^i, N_2^j, R \rangle$, namely a set of semantic correspondences between the nodes in the two lightweight ontologies given as input. Where N_1^i is the *i-th* node of the first lightweight ontology, N_2^j is the *j-th* node of the second lightweight ontology, and R is a semantic relation in $(=, \sqsupseteq, \sqsubseteq, \perp)$. Figure 2 shows a graphical representation of a set of mapping elements.

3.2. *Minimal Semantic Matching*

Considering the hierarchical nature of inputs (the *lightweight ontologies*) and taking into account the subsumption relation existing between the set-theoretic interpretations of input labels [4], we outline here the Minimal Semantic Matching algorithm. This algorithm exploits the above mentioned properties of the inputs to reduce matching time, the algorithm dependency on the background knowledge and to output the minimal set of possible mapping elements. The minimality property is especially desirable if the output mapping is to be further processed by humans.

The minimal set of possible mapping elements between two Lightweight Ontologies is such that:

i) all the (redundant) mapping elements can be computed from the minimal set,
ii) none of the mapping elements can be dropped without loosing property i).

This minimal set of possible mapping elements always exists and it is unique [4]. Another important property of the minimal mappings algorithm is that it is possible to compute the maximum set of mapping elements based on the minimal mappings set. This computation can be done by inferring all *redundant* mapping elements from the minimal set [4], therefore, significantly reducing the number of node matching operations to be performed (see step *four* of the basic algorithm in Section 3.1), which otherwise consume considerable time for solving the propositional satisfiable (SAT) problems.

A graphical representation of the minimal mappings set can be seen in Figure 3. Note how the set of mappings is drastically reduced in comparison to the set of mappings returned by the "default" semantic matcher and shown in Figure 2. This reduced set is more human-readable and in general corresponds to what a person expects to see as the result of the semantic matcher.

If we analyze the mappings in Figure 2 for the "Courses" root node in the left tree, we can see that "Courses" is equal to "Course" (the root node of the right tree), but is more general than all the children nodes of the "Course" root node. By comparing these mappings to the mapping in Figure 3 (only the $=$ relation between the roots) we can see how the minimal version collapses all the other mapping elements, be-

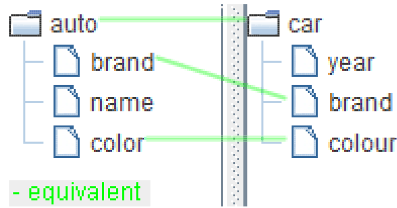

Fig. 4. Result of Structure Preserving Semantic Matching (SPSM).

cause they can be inferred from the equivalence relation. In general, if two nodes are semantically equal, all the children nodes will be more specific compared to the parent (see [4] for the theoretical basis).

3.3. Structure Preserving Semantic Matching (SPSM)

In many cases it is desirable to match structurally identical elements of both the source and the target parts of the input. This is specially the case when comparing signatures of functions such as web service descriptions or APIs or database schemas. Structure Preserving Semantic Matching (SPSM) [7] is a variant of the basic semantic matching algorithm. SPSM can be useful for facilitating the process of automatic web services composition, returning a set of possible mappings between the functions and their parameters.

SPSM computes the set of mapping elements preserving the following structural properties of the inputs:

i) *only one mapping element per node is returned*. This is required in order to match only one parameter (or function) in the first input tree, to only one parameter (or function) in the second input tree.
ii) *leaf nodes are matched to leaf nodes and internal nodes are matched to internal nodes*. The rationale behind this property is that a leaf node represents a parameter of a function, and an internal node corresponds to a function. This way a parameter which contains a value will not be confused with a function.

Figure 4 shows the output of SPSM when matching two simple database schemas, consisting of one table each: table “auto” with columns “brand”, “name” and “color” on the left and table “car” with columns “year”, “brand” and “colour” on the right. Observing the results of SPSM in this example we can see that the set of structural properties is preserved:

i) The root node “auto” in the left tree has only one mapping to the node “car” in the right tree on the same level, that is, root.
ii) The leaf node “brand” in the left tree is mapped to the leaf node “brand” in the right tree, and similarly with the leaf node “color” in the left tree and the node “colour” in the right tree - the leaf node is mapped to the node on the same level, that is, to a leaf node.

4. The Framework

4.1. Algorithms

Currently S-Match contains implementations of the basic semantic matching algorithm [1], as well as minimal semantic matching algorithm [4] and structure preserving semantic matching algorithm [7]. The basic algorithm is a general purpose matching algorithm, very customizable and suitable for many applications. The minimal semantic matching algorithm produces minimal and maximal mapping elements sets. The minimal set is well suited for manual evaluations, as it contains “compressed” information and saves experts’ time. The maximum mapping elements set contains all possible links and is well suited for consumption by applications which are not aware of semantics of lightweight ontologies. The structure preserving semantic matching (SPSM) algorithm is an algorithm well suited for matching API and database schemas. It matches the inputs distinguishing between structural elements such as functions and variables.

4.2. Architecture

S-Match is developed to have a modular architecture that allows easy extension and plug-in of ad-hoc components for specific tasks. Figure 5 shows the main components of the S-Match architecture, and a reference to the four steps of the Semantic Matching algorithm outlined in Section 3.

4.3. Loaders

The *Loaders* package contains loaders from various formats. This component *loads* the files containing the tree-like structures from tab-indented formats and XML formats. The provided `IContextLoader` interface allows extra loaders to be added for formats such as RDF and OWL, which alternatively can be accessed using S-Match-AlignAPI integration.

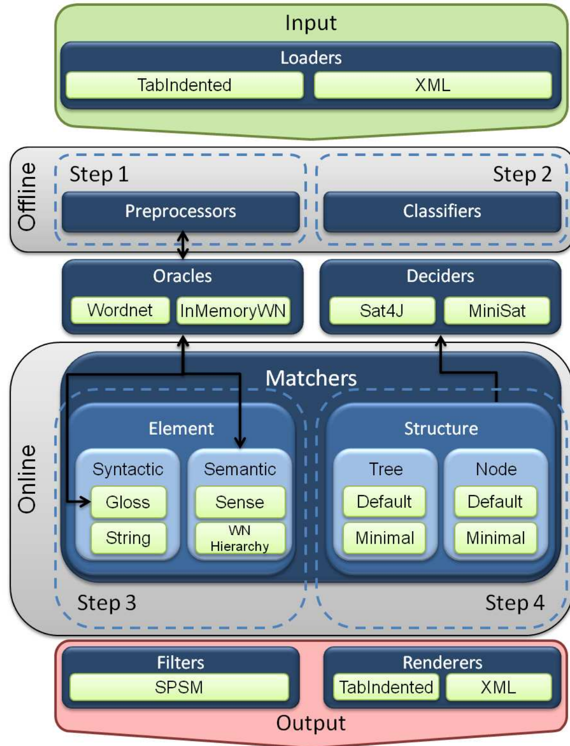

Fig. 5. S-Match architecture

4.4. *Preprocessors*

The *Preprocessors* package contains components, providing translation of natural language metadata, such as classification labels, ontology class names and library subject headings into lightweight ontologies. These components use linguistic information provided by the components from the *Oracles* package to extract atomic concepts from labels of node. They also use the knowledge of the language, such as peculiarities of the syntax of subject headings or features of the structure of classification labels to construct complex concepts out of atomic ones. These components output the concepts of label (C_Ls).

4.5. *Classifiers*

The *Classifiers* package constructs the concepts at node (C_Ns) using the information stored in the tree plus the concepts of the labels of each node. The output of this step is a set of DL formulas representing the concepts of each node. These formulas represent the Lightweight Ontology (see Section 2) for the input trees.

4.6. *Oracles*

The *Oracles* package provides access to linguistic knowledge and background knowledge. This component contains linguistic oracles, which provide access to linguistic knowledge, such as base forms and senses and sense matchers, which find relations between word senses handed out by linguistic oracle. On one hand, linguistic knowledge (e.g. the knowledge one needs to translate natural language into propositional description logics) and background knowledge (e.g. the knowledge one needs to match "car" to "automobile" or to know that "apple" is less general than "fruit") are separate and can be provided by different components. On the other hand, in practice, they are often provided by a single component handling out of a dictionary, such as WordNet [13], base forms as well as word senses and relations between those senses.

4.7. *Deciders*

The *Deciders* package provides access to sound and complete satisfiability reasoners, as well as other logic-related services, such as conversion of an arbitrary logical formula into its conjunctive normal form. The services of the package are used by classifiers during the construction of the concepts at node and by the structure level matchers during the matching process.

4.8. *Matchers*

The *Matchers* package is organized as two sub packages, one for the *Element* level matchers, and one for the *Structure* level matchers.

The *Element* level matchers are used to compute the relations between the labels of node (the *third* step of the Semantic Matching algorithm). The package is divided in two sub-packages, one for semantic matchers, which use the linguistic *Oracles* to compute semantic relations between concepts, and the syntactic matchers which are used if no semantic relation could be extracted by the semantic matcher. Currently S-Match includes Wordnet-based syntactic gloss matchers, and a set of eight string-based matchers (see Table 1).

The *Structure* level matchers in Figure 5 are used to compute semantic relations between the concepts of node. For this purpose, after transforming the problem into a propositional satisfiability (SAT) problem, they rely on the *Deciders* package that provides sound and complete SAT engines. The output of this step is a set of *mapping elements* that contains semantic relation

Table 1
S-Match element level matchers

Type	Name
Sense-based	WordnetMatcher, WNHierarchy
String-based	Prefix, Suffix, EditDistance, NGram
Gloss-based	WNGloss, WNExtendedGLoss

between the nodes. The core parts of matching algorithms are implemented by structure level matchers.

The `DefaultTreeMatcher` implements the basic semantic matching algorithm by matching two trees in a very simple manner. It takes all nodes of the source tree and sequentially matches them to all nodes of the target tree.

The `OptimizedStageTreeMatcher` implements the minimal semantic matching algorithm by splitting the matching task into four steps, according to the relations and the partial order between them. First, it walks simultaneously two trees and searches for disjoint relation between the nodes. If found, the subtrees of the two nodes in question could be skipped. Then it repeats the operation searching for less and more generality, skipping appropriate subtrees in case a relation is found. Finally, it searches for the presence of both less and more general relations between the same nodes and establishes an equivalence relation between them instead.

Alternatively, the minimal mapping can be obtained from a normal mapping by using a minimal filter, as explained in the Section 4.9.

The `SPSMTreeMatcher` uses the results of the `DefaultTreeMatcher` to building a graph to compute the minimal number of edit distance required to transform one lightweight ontology into the other. This number of operations is then used to compute a semantic similarity score as defined in [7]. Once the score is computed, the resulting mapping elements that comply with the structural properties presented in Section 3.3 is computed by applying the SPSM filter, as explained in the Section 4.9.

4.9. Filters

The *Filters* package provides components which filter the mapping. For example, the `RedundantMappingFilterEQ` filters what can be considered redundant mappings between nodes [4] to return a minimal set of mapping results (for example, to be shown in a User Interface). The `SPSMMappingFilter` filters a normal mapping into a mapping where the structure is preserved. This package also contains implementation of the utility filters such `RandomSampleMappingFilter` which obtains a random sample from the mapping, for example, for evaluation, or `RetainRelationsMappingFilter`, which retains only the relations of interest from a complete mapping.

4.10. Renderers

Finally, the *output step* is concerned with the formatting of the mapping elements to be suitable for each particular case. The *Renderers* package takes care of saving the results (and the computed Lightweight Ontology) in the appropriate format for future use. Currently, the Lightweight Ontology can be saved in an XML format, which can be loaded by the *Loaders* package, to avoid repeating the first two steps more than once. The set of mapping elements can currently be saved in plain text file (see Section 5.4 for more details).

5. The interfaces

S-Match provides 3 different interfaces for executing the matching algorithms, managing the inputs, the outputs and configuring the framework. These interfaces are:

1. Java API: useful for extending the functionality of the framework by, for example, implementing specific *Loaders* and *Renderers*, replacing the linguistic *Oracles* and implementing specific element and structure level *Matchers*.
2. Command Line Interface: useful for managing S-Match as a configurable black box algorithm, which parameters can be fine tuned via command line parameters and configuration files.
3. Graphical User Interface: useful for testing purposes where human experts need to quickly assess the resulting mapping elements.

5.1. Java API

The Java API provided by S-Match can be particularly useful for extending the basic matchers to include, for example, geospatial matchers (to check that a street is part of a city) or temporal matchers (to check that December 31st is equal to New Year's eve). Many commonly used S-Match services are exposed via the *IMatchManager* interface. The `MatchManager`

```
...
1:  IMatchManager mm = MatchManager.getInstance();
2:  Properties config = new Properties();
3:  config.load(new FileInputStream("s-match-Tab2XML.properties"));
4:  mm.setProperties(config);
5:  IContext s = mm.loadContext("../test-data/cw/c.txt");
6:  IContext t = mm.loadContext("../test-data/cw/w.txt");
7:  mm.offline(s);
8:  mm.offline(t);
9:  IContextMapping<INode> result = mm.online(s, t);
10: mm.renderMapping(result, "../test-data/cw/result.txt");
...
```

Fig. 6. S-Match API example.

class is an implementation of this interface and can be used as shown in Figure 6.

Once the `MatchManager` is instantiated (*line 1*), the following step is to load the configuration file (*lines 2-4*). Once the framework is configured, the source and target trees need to be loaded (*lines 5-6*) and converted into lightweight ontologies by preprocessing them (*lines 7-8*) before matching (*line 9*). Finally, the mappings can be rendered into an output file (*line 10*). The example code shown in Figure 6 is included in the open source distribution as `SMatchAPIDemo` [8].

5.2. *Command line interface*

S-Match command line interface provides facilities for configuring the framework by specifying several components such as the configuration files, commands to run and their arguments and options. Currently the following commands are available in the S-Match CLI:

- `offline`: transforms the input tree-like structures into lightweight ontologies by performing the offline preprocessing (*first* and *second* steps of Section 3.1) and rendering the results for future reuse. This is useful for transforming the trees only once, and then avoiding this overhead each subsequent time the same tree needs to be matched.
- `online`: computes the mapping elements (*third* and *fourth* steps of Section 3.1) between lightweight ontologies and writes the results into an output file. This performs the *online* and *output* steps of Figure 5. The command is decoupled from the previous offline step in order to avoid preprocessing overheads and allowing better performance when running experiments.
- `filter`: filters the mapping elements by loading a specific mapping element set, filtering it and then saving it into an output file. This can be used to further process the results from a particular version of semantic matching algorithm (Section 3). For example, to expand the minimal set of mapping elements into a maximum set by computing the redundant mapping elements.
- `convert`: transforms between *I/O* formats by using different implementations of *Loaders* and *Renderers*. This is useful, for example, to transform tab-indented trees into AlignAPI format and vice versa.
- `wntoflat`: converts the WordNet dictionary into internal binary format. This format can then be loaded into memory to speed up element level matching process.

Each of these commands can be customized by specifying a properties file which configures the components and the run-time environment. When this property file is not specified, a default configuration file[9] available in the open source distribution is used. Furthermore, a tuple of <property_key,value> can be specified multiple times in the command line overriding the loaded value from the properties file. This is particularly useful at times when one or two out of many options in the configuration file need to be changed, for example, when conducting a series of experiments where several threshold values need to be tested.

5.3. *Graphical User Interface*

S-Match can also be used with a simple Graphical User Interface (GUI). Figure 7 shows the GUI with two example course catalogs and the resulting Minimal Mapping loaded. The set of mapping elements computed by the framework can be easily analyzed by a human using the GUI in Figure 7 in contrast to the set of mapping elements rendered as output using any format as shown in Figure 9.

5.4. *I/O formats*

S-Match provides generic interfaces for dealing with the input and output of tree-like structures, Lightweight Ontologies and *mapping elements*. While currently implementing a set of *Loaders* and *Render-*

[8] demos/smatchapi/SMatchAPIDemo.java

[9] ../conf/s-match.properties

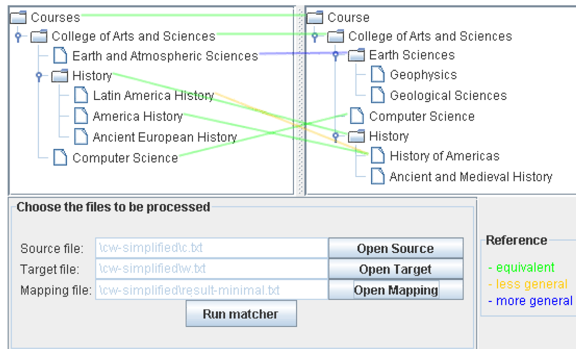

Fig. 7. S-Match GUI with course catalogs and a mapping loaded

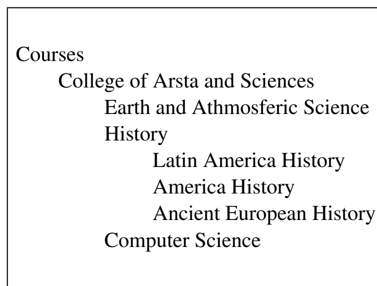

Fig. 8. Example input in tab-indented format.

ers, the framework can be easily extended by implementing the `IContextLoader` and `IContextRenderer` interfaces.

The *Loaders* package includes the implementation of loaders for the tree-like structures and *mapping elements*. Currently, S-Match supports loading tree-like structures in *tab indented* format. Figure 8 shows an example tree from Figure 1 in tab indented format.

Given that steps 1 and 2 need to be performed only once per tree, S-Match also provides the facilities (in the *Renderers* package) for saving the resulting lightweight ontology from the second step in XML format. Consequently, one can load the Lightweight Ontology directly from the XML file (using the *Loaders* package) and save time and resources that are critical for testing purposes and when working with big Lightweight Ontologies.

Current *Renderers* and *Loaders* for the *mappings elements* consists of the implementation of a plain text file format, containing one line per *mapping element* in the form $\langle N_1^i \to R \to N_2^j, \rangle$. Where N_1^i is the path from the root to the particular node in one tree, $\to$ is a tab character, R is the semantic relation in (= *equivalence*, > *more general*, < *less general*, ! *disjointness*) and N_2^j is the path from the root to the node in the second tree.

\Courses = \Course
\Courses\College of Arts and Sciences = \Course\College of Arts and Sciences
...
\Courses\College of Arts and Sciences\History = \Course\College of Arts and Sciences\History
...
\Courses\College of Arts and Sciences\History\Latin America History < \Course\College of Arts and Sciences\History\History of Americas
...

Fig. 9. Example output in the plain text format.

Figure 9 shows an extract (4 out of 43) from the set of mapping elements shown in Figure 2.

S-Match integration into the *Alignment API (AlignAPI)*[10] allows S-Match users to access more input and output formats and allows AlignAPI users to use S-Match matching capabilities. AlignAPI is being widely used for benchmarking purposes for the last 6 years in the Ontology Alignment Evaluation Initiative (OAEI)[11].

6. The open source distribution

S-Match[12] is the open source implementation of the Semantic Matching framework outlined in Section 3 and defined and evaluated in [1] at the date of writing this paper, and it has been released under GNU Library or Lesser General Public License (LGPL). S-Match provides implementations of the basic semantic matching algorithm [1], the minimal semantic matching algorithm [4], and the Structure Preserving Semantic Matching (SPSM) [7]. It provides necessary implementation for transforming tree-like structures, such as Classifications, Web and file system directories and Web services descriptions, among others, into Lightweight Ontologies (see Section 2). It also provides a Graphical User Interface for selecting the inputs, running the matching and visualizing the results, as well as the command line API (Section 5). It provides a Java library[13] that enables other projects to exploit semantic matching capabilities.

The latest version of S-Match can be found at the download page[14] of the site and the complete documentation including the "Getting started" guide, the manual, the publications and the presentations can be found at the documentation page[15].

[10] `http://alignapi.gforge.inria.fr/`
[11] `http://oaei.ontologymatching.org/`
[12] `http://s-match.org/`
[13] `s-match.jar`
[14] `http://s-match.org/download.html`
[15] `http://s-match.org/documentation.html`

The datasets used for testing the performance of the framework have also been released. The TaxMe2 [15] dataset with annotations and reference mapping elements can be found at the datasets page[16]. This dataset has been used since 2005 in the yearly *Directory Track* of the Ontology Alignment Evaluation Initiative (see [16,17] for latest editions) associated with the ISWC Ontology Matching Workshop, where it has shown over the past years to be robust with a good discrimination ability, i.e., different sets of correspondences are still hard to find for different systems. For example, S-Match achieves precision, recall and f-measure of 46%, 30% and 36% respectively, and compares well with 8 other tested systems [15].

7. Conclusion

This paper presents S-Match, an open source semantic matching framework. S-Match includes the implementation of three versions of semantic matching algorithms, designed for different application domains. The framework allows new algorithms and background knowledge to be included when specific matchers or linguistic information is required. The graphical user interface allows users to easily interpret the results, while the programmatic API provides great flexibility for exploiting the matching algorithms from other systems.

We have outlined how lightweight ontologies, while being simple and intuitive structures, can be used to hold many knowledge organization systems. This shows that S-Match, while being designed to work with lightweight ontologies, is of much importance because it covers a great number of information structures.

We are currently working on making more datasets available and extending the programmatic API to include a web service layer that can be used to execute matching online at the S-Match project site or as enterprise semantic matching services. We are also working on enriching the supported input formats by including other standards such as the Ontology Alignment API, among others.

References

[1] Fausto Giunchiglia, Mikalai Yatskevich, and Pavel Shvaiko. Semantic matching: Algorithms and implementation. In *Journal on Data Semantics IX*, pages 1–38. 2007.

[2] Fausto Giunchiglia, Maurizio Marchese, and Ilya Zaihrayeu. Encoding classifications into lightweight ontologies. In *Journal on Data Semantics VIII*, pages 57–81. 2007.

[3] Ilya Zaihrayeu, Lei Sun, Fausto Giunchiglia, Wei Pan, Qi Ju, Mingmin Chi, and Xuanjing Huang. From web directories to ontologies: Natural language processing challenges. In *The Semantic Web*, pages 623–636. 2007.

[4] Fausto Giunchiglia, Vincenzo Maltese, and Aliaksandr Autayeu. Computing minimal mappings. In *Proceedings of the 4th Workshop on Ontology matching at ISWC*. 2009.

[5] Fausto Giunchiglia and Ilya Zaihrayeu. *Encyclopedia of Database Systems*, chapter Lightweight Ontologies. Springer, June 2009.

[6] Fausto Giunchiglia and Pavel Shvaiko. Semantic matching. *Knowl. Eng. Rev.*, 18(3):265–280, 2003.

[7] Fausto Giunchiglia, Fiona McNeill, Mikalai Yatskevich, Juan Pane, Paolo Besana, and Pavel Shvaiko. Approximate structure-preserving semantic matching. In *Proceedings of the OTM 2008 Confederated International Conferences. ODBASE 2008*, pages 1217–1234, Moterrey, Mexico, 2008. Springer-Verlag.

[8] Jérôme Euzenat and Pavel Shvaiko. *Ontology Matching*. Springer-Verlag New York, Inc., 2007.

[9] Pavel Shvaiko and Jérôme Euzenat. Ten challenges for ontology matching. In Robert Meersman and Zahir Tari, editors, *OTM Conferences (2)*, volume 5332 of *Lecture Notes in Computer Science*, pages 1164–1182. Springer, 2008.

[10] Namyoun Choi, Il-Yeol Song, and Hyoil Han. A survey on ontology mapping. *SIGMOD Record*, 35(3):34–41, 2006.

[11] David Aumueller, Hong Hai Do, Sabine Massmann, and Erhard Rahm. Schema and ontology matching with COMA++. In Fatma Özcan, editor, *SIGMOD Conference*, pages 906–908. ACM, 2005.

[12] Wei Hu and Yuzhong Qu. Falcon-AO: A practical ontology matching system. *J. Web Sem.*, 6(3):237–239, 2008.

[13] George A. Miller. WordNet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[14] Rada Mihalcea and Andras Csomai. SenseLearner: Word Sense Disambiguation for All Words in Unrestricted Text. In *43rd Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference, 25–30 June 2005, University of Michigan, USA*. The Association for Computational Linguistics, 2005.

[15] Fausto Giunchiglia, Mikalai Yatskevich, Paolo Avesani, and Pavel Shvaiko. A large dataset for the evaluation of ontology matching systems. *The Knowledge Engineering Review Journal*, 24:137–157, 2008.

[16] Caterina Caracciolo, Jérôme Euzenat, Laura Hollink, Ryutaro Ichise, Antoine Isaac, Véronique Malaisé, Christian Meilicke, Juan Pane, Pavel Shvaiko, Heiner Stuckenschmidt, Ondrej Sváb-Zamazal, and Vojtech Svátek. Results of the ontology alignment evaluation initiative 2008. In Pavel Shvaiko, Jérôme Euzenat, Fausto Giunchiglia, and Heiner Stuckenschmidt, editors, *Proceedings of the 3rd ISWC international workshop on Ontology Matching, Karlsruhe (DE)*, 2008.

[17] Jérôme Euzenat, Alfio Ferrara, Laura Hollink, Antoine Isaac, Cliff Joslyn, Véronique Malaisé, Christian Meilicke, Andriy Nikolov, Juan Pane, Marta Sabou, François Scharffe, Pavel Shvaiko, Vassilis Spiliopoulos, Heiner Stuckenschmidt, Ondrej Sváb-Zamazal, Vojtech Svátek, Cássia Trojahn dos Santos, George Vouros, and Shenghui Wang. Results of the on-

[16] `http://s-match.org/datasets.html`

tology alignment evaluation initiative 2009. In Pavel Shvaiko, Jérôme Euzenat, Fausto Giunchiglia, Heiner Stuckenschmidt, Natasha Noy, and Arnon Rosenthal, editors, *In Proc. 4th International Workshop on Ontology Matching (OM-2009), collocated with ISWC-2009*, pages 73–126, Fairfax (VA US), 2009.