Undefined 2 (2011) 1–8
IOS Press

The Alignment API 4.0 [1]

Editor(s): Sören Auer, Universität Leipzig, Germany
Solicited review(s): Jens Lehmann, Universität Leipzig, Germany; Jun Zhao, University of Oxford, UK
Open review(s): Prateek Jain, Wright State University, USA

Jérôme David [a], Jérôme Euzenat [a,*], François Scharffe [a], and Cássia Trojahn dos Santos [a]
[a] *INRIA & LIG, 655 avenue de l'Europe, 38330 Montbonnot Saint-Martin, France*

Abstract. Alignments represent correspondences between entities of two ontologies. They are produced from the ontologies by ontology matchers. In order for matchers to exchange alignments and for applications to manipulate matchers and alignments, a minimal agreement is necessary. The Alignment API provides abstractions for the notions of network of ontologies, alignments and correspondences as well as building blocks for manipulating them, such as matchers, evaluators, renderers and parsers. We recall the building blocks of this API and present here the version 4 of the Alignment API through some of its new features: ontology proxys, the expressive alignment language EDOAL and evaluation primitives.

Keywords: Alignment API, Ontology matching, Ontology alignment, Alignment management, Alignment evaluation, Alignment language, EDOAL, Ontowrap, OntoSim

1. Motivation

Using ontologies is the privileged way to achieve interoperability among heterogeneous systems within the semantic web. However, as ontologies are not necessarily compatible, they may in turn need to be reconciled. Ontology reconciliation often requires to find the correspondences between entities, e.g., classes, objects, properties, occurring in these ontologies.

Alignments are sets of correspondences between elements of two ontologies [13]. They are generated by hand or by ontology matchers and they can be used for merging ontologies, transforming queries or linking data sets. During their life cycle, alignments may be written in files and further read, applied various thresholds, merged together and finally transformed into a more operational format.

Considering ontology alignments as first class citizens has several benefits:

- from a semantic web point of view, as it is possible to dynamically find and reuse existing alignments;
- from a software engineering point of view, as alignments can be passed from a program to another;
- from an ontology engineering and management point of view, as they will evolve together with the ontology life cycle.

In order for alignments produced by any means to be treated uniformly and for matchers and applications to consume and produce such alignments, we designed a Java API for alignments: the Alignment API [9].

The Alignment API is a set of abstractions for expressing, accessing and sharing ontology alignments. Its reference implementation provides a minimal implementation of this interface. It aims at facilitating the development of tools manipulating alignments and calling matchers. It is also a way for matcher developers to deliver alignments in a well-supported framework.

We describe in this paper the version 4.0 of the Alignment API which has been deeply enhanced since its first version in 2003. We emphasise those new fea-

[1] Partial support provided by European Network of Excellence Knowledge web (FP6-IST-507482).
[*] Corresponding author. E-mail: Jerome.Euzenat@inria.fr

0000-0000/11/$00.00 © 2011 – IOS Press and the authors. All rights reserved

tures which were not available for [9]. After describing the API itself (§2), we describe more precisely areas of major enhancement, such as the abstraction from ontologies and concrete measures (§3), the development of an expressive alignment language (§4) and the evaluation of alignments (§5).

2. The Alignment API in a nutshell

The Alignment API itself is a reduced set of Java interfaces. The most important parts of the API are the representation interfaces which provide access to information about the API elements. These interfaces are first presented before considering the process interface and the reference implementation of the API.

2.1. Representation classes

The Alignment API offers four main representational classes whose structure is represented in Figure 1.

`OntologyNetwork` is a container for a set of ontologies and a set of alignments. It makes it easy to retrieve alignments tied to an ontology as well as to manipulate them as a network, i.e., traversing them, closing them, etc.

`Alignment` is the main class of the API. An `Alignment` is mostly made of a set of `Cell`s and metadata about the alignment, such as the aligned ontologies, the alignment arity, provenance metadata, and any other metadata that can be tied to an alignment.

`Cell` represents a correspondence: it relates two entities with a `Relation`. The entities may be any identified element of an `Ontology` (see §3) or any construct from the expressive EDOAL language (see §4). In addition, `Cell` supports any type of additional metadata (including confidence values).

`Relation` represents the relation between two entities. The set and type of relations are extensible in the Alignment API and its implementation.

These classes provide access to the information in instances. They also provide local methods for manipulating this information: adding correspondences to alignments, cutting correspondences under a confidence threshold, etc.

An RDF vocabulary and an RDF/XML format, corresponding to the `Alignment` class, can be used for serialising alignments.

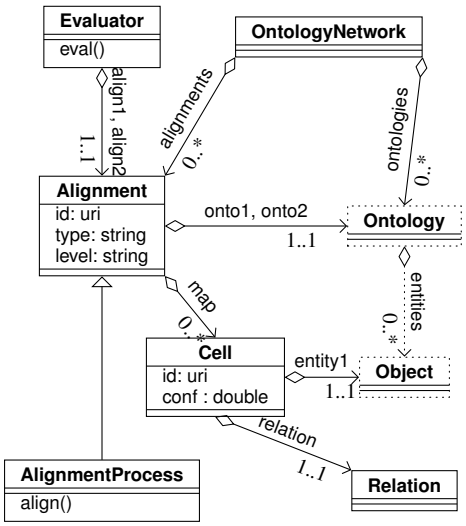

Fig. 1. Relations between the main classes of the Alignment API and the classes of Ontowrap (`Ontology`). Only the main attributes and methods are presented.

2.2. Processing

In addition to being a storage structure, the Alignment API provides a minimal processing structure, which can be used by applications for manipulating and consuming alignments. We present its most important classes:

`AlignmentProcess` is the interface for all matchers. Matching two ontologies is achieved in two steps: creating an instance of an `Alignment` implementing `AlignmentProcess` and initialising it with two `Ontology` instances, then calling the `align()` method. This method takes two arguments: an initial `Alignment` which may be considered by the matcher and a `Property` object providing parameters for the matcher.

`Evaluator` is the interface for alignment evaluators which compare a first `Alignment` which may be taken as the reference and a second `Alignment` (see §5).

`AlignmentVisitor` or `Renderer` is the interface for defining `Alignment` visitors which can output alignments in different formats, when calling the `render()` method of `Alignment`.

These operations are presented in Figure 2.

2.3. API implementation

Together with the API definition, a reference implementation of the Alignment API is available. It pro-

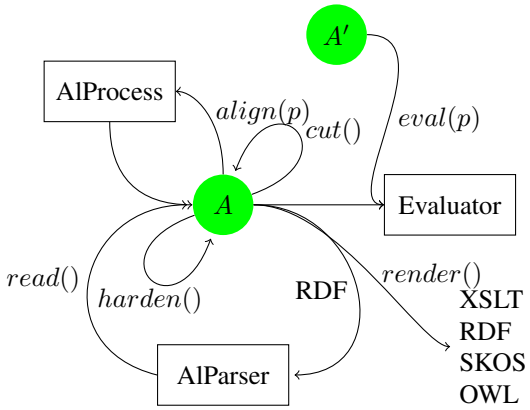

Fig. 2. The main operations of the Alignment API and associated processing classes: matchers (process), evaluators and parser.

vides, for each class of the API, a basic implementation and offers the following services:

- Storing, finding, and sharing alignments;
- Piping matching algorithms (improving an existing alignment);
- Manipulating alignments (trimming under a threshold, merging, inverting, and hardening);
- Generating processing output (transformations, axioms, rules);
- Comparing alignments (finding differences).

Thus, to quickly instantiate the API, it is sufficient to refine this basic implementation. This will take advantage of all the services already implemented in the base implementation. It is also possible to define completely new implementations of the API. The benefit of this is that such an implementation could still be used by the tools which rely on the Alignment API.

The API implementation also provides more precise implementations. They must be considered as examples, such as simple concrete matching methods (e.g., `StringDistAlignment` and all derivates of `DistanceAlignment`), or as fully implemented utilities, such as the provided `Renderer`s or `Evaluator`s. More precisely, the Alignment API implementation comes with:

- a base implementation of the interfaces with many useful facilities;
- a library of sample matchers;
- a library of renderers (for RDF, XSLT, SWRL, OWL, C-OWL, SEKT Mapping Language, SKOS, HTML);
- a library of evaluators (see §5);
- a library of wrappers for several ontology APIs (see §3);

Finally, the API implementation provides tools for manipulating alignments, such as the batch utilities described in §5 or the `AlignmentParser` which can parse an RDF file in the Alignment format, or the EDOAL extension of this format (see §4), and return the corresponding `Alignment`.

3. Abstract support for ontologies and distances

The implementation of the Alignment API relies on two external abstractions that we describe below:

Ontowrap provides uniform access to part of ontology APIs useful for matchers;

OntoSim provides various distance and similarity measures. It relies on Ontowrap.

3.1. Ontowrap

There are many different APIs for ontologies. Even if the Alignment API is independent from these APIs, it is often convenient to interact with them, in particular when one wants to match ontologies. For that purpose, we have designed the Ontowrap API which encapsulates interactions with ontology APIs. The architecture of Ontowrap has been designed to be easily extensible to other ontology APIs.

The Ontowrap package defines the abstract factory `OntologyFactory` which is used for generating ontologies. Depending on the level of interaction needed with the ontologies, the factory may provide three levels of ontology interfaces, each one extending the previous one:

`Ontology` simply provides information about an ontology: its URI, its URL and eventually its knowledge representation formalism (OWL, SKOS, etc.). Since, this interface does not assume that the ontology has been loaded, it does not offer access to the information about entities (classes, properties, individuals).

`LoadedOntology` provides access to an ontology that has been loaded in main memory. However, the interaction with the API is still limited. It allows for retrieving the entities and provides their type (class, property, individual), URI, labels and comments. This interface is easy to implement and applies to ontologies as well as to "lightweight" ontologies, such as unstructured thesauri.

`HeavyLoadedOntology` offers a broader access to an ontology by obtaining the relations between entities (subsumption and other properties), the classes of individuals, and the domain and range of properties.

For each major ontology API, a concrete factory and at least one `Ontology`, `LoadedOntology` or `HeavyLoadedOntology` implementation is provided.

Ontowrap offers implementations for the major ontology APIs: OWL-API [1], Jena OWL API [3], the SKOS API [14], and our own SKOSLite loader.

3.1.1. The HeavyLoadedOntology interface

One problem with `HeavyLoadedOntology` implementations is that the various APIs do not return the same set of entities to a seemingly same question. For instance, consider the `getSuperClasses(aClass)` method. It is a simple method which returns the superclasses of a class. But there are many ways of defining these superclasses:

- Direct superclasses, i.e., those superclasses which have no other superclass as subclass (classes part of the transitive reduction of the superclass relation).
- Named superclasses, i.e., only those which are identified by a URI (in the OWL model, any restriction is a class and can be a superclass).
- Asserted superclasses, i.e., those classes which have been explicitly asserted to be superclasses in the ontology description.

As Figure 3 shows, these three dimensions are orthogonal, i.e., one can pick up any combination of these three modalities and provide a different answer to the question. In addition, they can be refined:

- Named superclasses can be restricted to Local superclasses, i.e., those which are defined in the same name space as the ontology.
- Unasserted superclasses can be taken within those superclasses that can be obtained by limited reasoning methods like Inherited superclasses obtained only by transitive closure computation.
- Unnamed superclasses may be restricted to those Mentioned in the ontology (a property restriction appearing explicitly in the ontology) or to any of such restrictions, their number can be infinite.

To deal with this heterogeneity, the methods querying relations between entities take these 3 dimensions as parameters and Ontowrap will try to satisfy them as far as possible. It also includes the

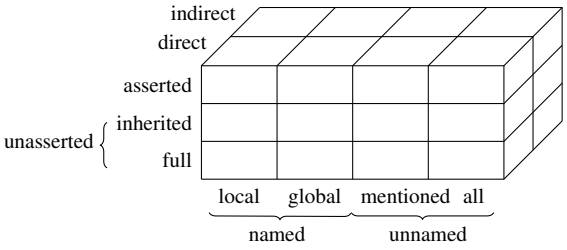

Fig. 3. The dimensions considered by `HeavyLoadedOntology` provide for 24 different ways to query sub-/super-classes.

`getCapabilities(direct, asserted, named)` method to check beforehand if the API meets the requirements of the application and to raise an exception if this is not the case.

3.2. OntoSim

OntoSim is an API dedicated to the computation of similarities between ontologies and ontology entities. It provides measures used for matching ontologies and supports the development of new measures. In particular, it provides methods for aggregating similarity matrices:

- aggregation schemes allowing for summarising a set of values in a single one;
- extractors for selecting a subset of values in a matrix according to several strategies: maximum weight graph matching, Hausdorff, average linkage, etc.

It also comes with a set of distances (string, objects, collections) and a wrapper for the secondstring package [4].

4. EDOAL: an expressive alignment language

The basic Alignment API implementation [9] originally linked only named entities. This is not sufficient when one wants to express relations between groups of such entities. This is especially a problem if the ontology language is not very expressive, e.g., thesauri languages.

The EDOAL language (Expressive and Declarative Ontology Alignment Language) extends the Alignment format in order to capture more precisely correspondences between heterogeneous ontological entities [12]. To achieve this, EDOAL includes a set of

constructors and operators to be used for expressing the entities in the Alignment format.

For example, let us consider two ontologies: v(in) having a concept Bordeaux and w(ine) having a concept Wine. While a simple subsumption correspondence between the two concepts, Bordeaux $\leq$ Wine, is valid and can be expressed in the initial Alignment format, a more precise correspondence would consider only those wines whose terroir is located in the "Aquitaine" region: Wine $\wedge$ hasTerroir·locatedIn = "Aquitaine" $\geq$ Bordeaux (see Figure 4).

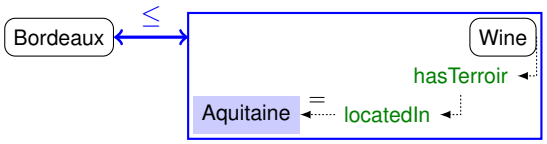

Fig. 4. Restriction on an attribute path value.

This language seems to be heavily related to OWL and indeed it has many features of it. One of the main reason for not simply adopting OWL was to not commit to a particular ontology language. Moreover, there are various differences which made us preserve an OWL-like syntax while not simply adopting OWL itself:

- EDOAL is less than OWL: it does not allow for defining new named entities, which is arguably one of the main features of OWL.
- EDOAL is more related to a rule language than to OWL by introducing variables within expressions, which are used for defining constraints and transformations. Moreover, the relation in correspondences has to be independent from the ontology language, although it is possible to render it with OWL relations, such as `owl:subClassOf`.
- EDOAL contains additional constructions for expressing data transformation.

However, we tried to keep it as close as possible to OWL. Indeed, EDOAL is largely inspired by constraints provided in description logics or in the OWL language. It allows for grouping the basic named entities found in ontologies (classes and properties) with classical boolean operators (disjunction, conjunction, complement) or property construction operators (inverse, composition, reflexive, transitive and symmetric closures) and constraints (through domain, range, cardinality and value restrictions). The syntax corresponding to the example of Figure 4 is:

```
<al:Cell rdf:about="bordeauxMap">
 <al:entity1>
  <ed:Class rdf:about="&v;Bordeaux"/>
 </al:entity1>
 <al:entity2>
  <ed:Class>
   <ed:and rdf:parseType="Collection">
    <ed:Class rdf:about="&w;Wine"/>
    <ed:AttributeValueRestriction>
     <ed:onAttribute>
      <ed:Relation>
       <ed:compose rdf:parseType="Collection">
        <ed:Relation rdf:about="&w;hasTerroir"/>
        <ed:Relation rdf:about="&pr;locatedIn"/>
       </ed:compose>
      </ed:Relation>
     </ed:onAttribute>
     <ed:comparator rdf:resource="&xsd;equals"/>
     <ed:value>
      <ed:Instance rdf:about="&w;Aquitaine"/>
     </ed:value>
    </ed:AttributeValueRestriction>
   </ed:and>
  </ed:Class>
 </al:entity2>
 <al:measure
    rdf:datatype='&xsd;float'>1.</al:measure>
 <al:relation>SubsumedBy</al:relation>
</al:Cell>
```

Moreover, EDOAL is also able to express transformations, e.g., unit transformations, between entities. For that purpose, correspondences may contain transformation elements and EDOAL allows for defining variables used to denote parts of the matched entities in transformations.

Although, there are few matchers able to generate such elaborate alignments, the need to represent them for the purpose of complex manipulation, e.g., linked data link generation, is increasing. First attempts towards complex ontology matching [16] use a refinement step to go from equivalence or subsumption matches to more complex correspondences. The work on alignment patterns [17] also uses EDOAL to solve ontology mismatches.

EDOAL support is still under development in the Alignment API 4.0: there is no specific evaluator and only RDF and OWL renderers are available.

5. Evaluating alignments

The Alignment API provides support for alignment evaluation. Such an evaluation is usually carried out by comparing the alignments provided by matchers with a

reference alignment. The quality of the alignment can be assessed with the help of some measure.

The most commonly used and understood measures are precision (true positive/retrieved) and recall (true positive/expected) which have been adapted for ontology matching by considering alignments as mere sets of correspondences.

The Alignment API defines the `Evaluator` interface for evaluation purposes. This interface specifies the `eval()` method that runs the evaluation between two alignments. It is then possible to access the results of this evaluation through specific accessors or by printing results.

The implementation of the API provides a `BasicEvaluator`, which can be used as a basis for specific evaluators and a `GraphEvaluator` whose result is not a number but a curve. Several `Evaluator` implementations are available with the API:

`PRecEvaluator` implements the classical precision/recall evaluation as well as the derived measures (F-measure, fallout, overall) [5].

`SymMeanEvaluator` implements a weighted symmetric difference between the entities that are in one alignment and those common to both alignments (missing correspondences count for 0., others weight 1. complemented by the difference between their strengths). This is a measure of the similarity of two alignments. The result is split between the kinds of entity considered (class/property/individual).

`ExtPRecEvaluator` implements relaxed precision and recall as defined in [8]. They replace the count of true positive by a similarity measure between the two alignments which may count a correspondence as not fully incorrect when it is close to an expected correspondence but not identical. This provides higher values for precision and recall based on the proximity of obtained results with expected results. This class provides three flavours of relaxed measures: symmetric, effort-based and oriented. The symmetric measure considers how far two matched classes are in the hierarchy; the effort-based measure tries to weight the effort required to correct an erroneous correspondence; and the oriented measure considers the way incorrect correspondences affect the correctness of a particular task to be carried out with the alignment.

`SemPRecEvaluator` implements semantic precision and recall as defined in [10] based on the Pellet reasoner [18] (or any `OWLReasoner` implementation). Instead of syntactically comparing correspondences, this measure checks if each correspondence of one alignment is entailed by the other one. This is useful because returned alignments may have a different form than the reference alignment and yet be equivalent to it.

The Alignment API implementation offers a set of services for carrying out evaluation based on the `AlignmentProcess` and `Evaluator` classes. The `GroupAlign` class allows for batch matching of pairs of ontologies. From a directory containing a set of subdirectories, each one containing a pair of ontologies, it will run a particular matcher and deliver an alignment in each directory. This is useful when matching many pairs of ontologies for evaluation purpose.

A corresponding utility (`GroupEval`) implements batch evaluations. When the reference alignment is in the same directory structure, `GroupEval` iterates each subdirectory and evaluates the alignments using an `Evaluator` implementation. It can do this in parallel for the results of several matchers. The output can be provided as comma-separated value, HTML or LaTeX tables.

For plotting, the `GenPlot` class generates various outputs from such a directory structure. It can output precision/recall graphs (see Figure 5) as gnu-plot files and generates a corresponding Latex file or Google chart API HTML. It can also display other types of graphs, such as Receiver Operating Characteristic (ROC) curves.

These evaluation facilities are largely used within the Ontology Alignment Evaluation Initiative (OAEI)[1] yearly evaluation campaigns both by organisers and participants. The Alignment format is used as the result format for OAEI.

6. Related systems

The Alignment API aims at providing only the basic support for matching and alignment manipulation. It is not a matcher nor an alignment editor. But, we welcome, and would facilitate as much as possible, interface implementation as well as any other embeddings. We have developed a NeOn Alignment plug-in based on the API and it is being integrated within Mondeca's Intelligent topic manager (ITM) tool.

[1] `http://oaei.ontologymatching.org`

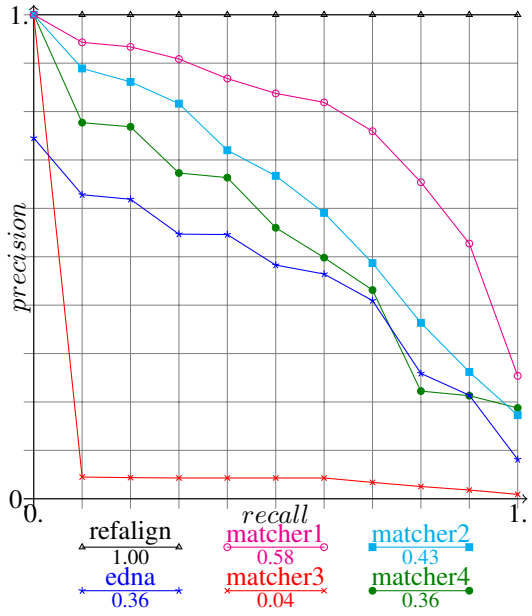

Fig. 5. Precision/recall graphs for benchmarks. The results given by the participants are cut under a threshold necessary for achieving $n\%$ recall and the corresponding precision is computed (values under the bar are mean average precision).

There exist tools similar to the Alignment API, but they usually have a different focus.

Several systems have been created as toolboxes for matchers. Their emphasis is rather on supporting matching in an integrated environment rather than on the alignment representation and its sharing over tools. There are "model management" tools like COMA++ [6] or ontology managers, such as Protégé in which the Prompt suite [15] is available. These systems generally come with a rich library of techniques and graphical interfaces for assembling matchers out of these techniques. This is typically what the Alignment API does not offer. These systems do not put the emphasis on sharing the alignments out of their environments like the Alignment API.

A system closer to the Alignment API is FOAM [7] whose development has been discontinued. For sharing and evaluating alignments, FOAM was relying on the Alignment API. Recently, [19] has taken inspiration from the Alignment API in order to share alignments between terminologies.

Finally, Silk [2] is a system similar in spirit but with a different target. Its goal is to express instance matchers in a declarative way. This is not that far from what we would like to do with EDOAL transformations, but SILK goes further in providing distance computation mechanisms within its language.

So, the main advantage of the Alignment API over these systems is to be a standalone and portable library for sharing alignments.

7. Availability, limitations and impact

The Alignment API is distributed under the LGPL license since December 2003. It can be found at `http://alignapi.gforge.inria.fr` together with ample online documentation. It is bundled with all required libraries for being immediately usable and tutorials are available for a quick start.

The Alignment API is constantly maintained (20 stable releases since 2003 and continuous source access in our svn server). Over the years, many individuals have contributed to the improvement of the API and are credited from our Credit page.

The Alignment API has encountered a wide diffusion testified by numerous tools using it (we counted more than 30 tools with published papers mentioning the use of the API[2]). Several tools are based directly on the API since it offers a wide range of implemented constructs to start with; other tools simply use the Alignment format as input/output. The Alignment API is used in the Ontology Alignment Evaluation Initiative (OAEI) for data and result processing.

We have embedded the Alignment API within several external tools, such as the NeOn toolkit and our Alignment server (bundled with the API). The Alignment server offers access to the Alignment API through a variety of means including browser interface and web service interface (REST and SOAP).

As far as scalability is concerned, our experience shows that the API implementation itself carries little overhead: it is able to handle alignments of tens of thousands of terms (from large thesauri) with no slow down. We have not investigated more scalability issues because there are hardly larger ontologies at the moment. However, the API is used as well for dealing with instances. Although this is possible, this has never been the primary purpose of this API. It may be more adequate to use a specific tool based directly on secondary memory storage and indexing for dealing with instances and dropping that support from the API.

[2]See `http://alignapi.gforge.inria.fr/impl.html` for reference.

The interface between instance matching and ontology matching is certainly one of the major limitation of the Alignment API because it has not been designed for dealing with instances. In the near future, we would like to define a clean link with data link generators, like Silk, without giving up our specificity as ontology alignment manager.

Among the planned developments, the issue that will most likely lead to version 5 is the better integration of theoretical developments genericising the relations and confidence to algebra of relations [11] and ordered structures. It would also be fruitful to better support reasoning with alignments and ontologies.

8. Conclusion

Alignment representation and manipulation in standard ways are needed in several important areas of the semantic web. We have designed and implemented the Alignment API for addressing this need.

The Alignment API is both an API for representing alignments and for developing, integrating and composing matchers. It comes with a Java implementation which has been used in many developments and provides examples and basic tools for manipulating alignments. It is used by several teams around the world which testify for its usability and usefulness.

The API is gradually improved incorporating bug fixes, extensions (like new standard evaluators and renderers), new features and new components, such as the abstract support for ontology and distances or the EDOAL alignment language which have been presented here.

References

[1] S. Bechhofer, R. Volz, P. Lord, Cooking the semantic web with the OWL API, in: Proc. 2nd International Semantic Web Conference (ISWC), vol. 2870 of Lecture notes in computer science, Sanibel Island (FL US), 2003.

[2] C. Bizer, J. Volz, G. Kobilarov, M. Gaedke, Silk - a link discovery framework for the web of data, in: Proc. 2nd WWW Workshop on Linked Data on the Web, Madrid (ES), 2009.

[3] J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, K. Wilkinson, Jena: implementing the semantic web recommendations, in: Proc. 13th international World Wide Web conference alternate track papers & posters, ACM, New York, NY, USA, 2004.

[4] W. Cohen, P. Ravikumar, S. Fienberg, A comparison of string metrics for matching names and records, in: Proc. KDD Workshop on Data Cleaning and Object Consolidation, Washington (DC US), 2003.

[5] H.-H. Do, S. Melnik, E. Rahm, Comparison of schema matching evaluations, in: Proc. Workshop on Web, Web-Services, and Database Systems, vol. 2593 of Lecture notes in computer science, Erfurt (DE), 2002.
URL http://dol.uni-leipzig.de/pub/2002-28

[6] H.-H. Do, E. Rahm, COMA – a system for flexible combination of schema matching approaches, in: Proc. 28th International Conference on Very Large Data Bases (VLDB), Hong Kong (CN), 2002.

[7] M. Ehrig, Ontology alignment: bridging the semantic gap, Semantic web and beyond: computing for human experience, Springer, New-York (NY US), 2007.

[8] M. Ehrig, J. Euzenat, Relaxed precision and recall for ontology matching, in: Proc. K-CAP Workshop on Integrating Ontologies, Banff (CA), 2005.
URL http://ceur-ws.org/Vol-156/paper5.pdf

[9] J. Euzenat, An API for ontology alignment, in: Proc. 3rd International Semantic Web Conference (ISWC), vol. 3298 of Lecture notes in computer science, Hiroshima (JP), 2004.

[10] J. Euzenat, Semantic precision and recall for ontology alignment evaluation, in: Proc. 20th International Joint Conference on Artificial Intelligence (IJCAI), Hyderabad (IN), 2007.

[11] J. Euzenat, Algebras of ontology alignment relations, in: Proc. 7th conference on international semantic web conference (ISWC), Karlsruhe (DE), vol. 5318 of Lecture notes in computer science, 2008.
URL http://www.springerlink.com/index/DY8Y9F31A9GT9762

[12] J. Euzenat, F. Scharffe, A. Zimmermann, Expressive alignment language and implementation, deliverable 2.2.10, Knowledge web (2007).

[13] J. Euzenat, P. Shvaiko, Ontology matching, Springer, Heidelberg (DE), 2007.

[14] S. Jupp, S. Bechhofer, R. Stevens, A flexible API and editor for SKOS, in: Proc. 6th European Semantic Web Conference (ESWC), Heraklion (GR), vol. 5554 of Lecture Notes in Computer Science, 2009.

[15] N. Noy, M. Musen, The PROMPT suite: interactive tools for ontology merging and mapping, International Journal of Human-Computer Studies 59 (6) (2003) 983–1024.

[16] D. Ritze, C. Meilicke, O. Šváb Zamazal, H. Stuckenschmidt, A pattern-based ontology matching approach for detecting complex correspondences, in: CEUR (ed.), Proc. ontology matching workshop (OM2009), vol. 551, 2009.
URL http://ceur-ws.org/Vol-551/om2009_Tpaper3.pdf

[17] F. Scharffe, D. Fensel, Correspondence patterns for ontology mediation, in: Springer (ed.), Proc. 16th International Conference on Knowledge Engineering and Knowledge Management (EKAW), 2008.

[18] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, Y. Katz, Pellet: a practical OWL-DL reasoner, Journal of Web Semantics 5 (2) (2007) 51–53.

[19] L. van der Meij, A. Isaac, C. Zinn, A web-based repository service for vocabularies and alignments in the cultural heritage domain, in: Proc. 7th European semantic web conference (ESWC), Hersonissos (GR), vol. 6088 of Lecture notes in computer science, 2010.