An API for OntoLex LIME datasets

Manuel Fiorelli, Maria Teresa Pazienza, Armando Stellato

ART Research Group, Dept. of Enterprise Engineering (DII),
University of Rome Tor Vergata
Via del Politecnico, 1, 00133 Rome, Italy
{fiorelli, pazienza}@info.uniroma2.it
stellato@uniroma2.it

Abstract. The OntoLex W3C Community Group published its final report on the Lexicon Model for Ontologies (*lemon*) in May 2016, specifying a suite of vocabularies for the linguistic grounding of ontologies and RDF datasets in general. The Linguistic Metadata (LIME) vocabulary is the *lemon* module describing coarse-grained metadata about datasets as a whole, to represent resuming information at the level of ontology-lexicon interface. The purpose of this metadata is to support the understanding and exploitation of available lexical material, and in the first place to facilitate the discovery of datasets that may be of interest. Towards the realization of that vision, we propose an API that supports the manipulation of LIME metadata, as well as its automatic generation by means of a profiler. We discuss the architecture of the API, as well as the main design decisions, which can inform the development of APIs for other vocabularies.

Keywords: *lemon*, Ontolex, LIME, Metadata, API

1 Introduction

The OntoLex W3C Community Group published its final report [1] in May 2016 specifying the Lexicon Model for Ontologies (*lemon*), a suite of vocabularies (or modules) that support the grounding of ontologies and RDF datasets in the natural language. The core module (ontolex) establishes the structure of the ontology-lexicon interface, upon which the other modules (synsem, decomp, vartrans, lime) provide support for different linguistic aspects. Our focus is on the LIME [2,3] module for the representation of dataset-level metadata about the ontology-lexicon interface. This metadata provides resuming information about a dataset, allowing humans and machines to understand which lexical material is available, and how it can be used to achieve given goals. Additionally, LIME metadata can be used to discover relevant datasets. This second aspect is particularly relevant in the more distributed scenarios foreseen by the OntoLex community group, such as the one in which the ontology, the lexicon and the lexicalization set establishing the connection between them are published as separate datasets. In such a case, LIME metadata ties together these disparate datasets, supporting crawling, indexing and then searching this network of lexicalized datasets.

The realization of that grand vision clearly presupposes the adoption and consistent use of the LIME vocabulary. As a first step in that direction, we propose an API for the manipulation and generation of LIME metadata. This API is implemented in Java on top of the RDF middleware RDF4J [4] (formerly, Sesame). Our work is open source and made available free of charge on its repository: `https://bitbucket.org/art-uniroma2/lime-api/` (hereafter, we will refer to commit `c1e7492`). The API is still under development, but already in a mature stage and fully usable.

The paper is structured as follows: Section 2 motivates the need for a LIME API, which is then discussed thoroughly in Section 3. Section 4 evaluates our work. Section 5 discusses related works. Section 6 concludes the paper.

2 Motivation

The OntoLex *lemon* specification is implemented as a collection of OWL ontologies, therefore RDF tools and libraries may be sufficient to manage OntoLex *lemon* data. However, they represent quite a low-level approach. While these RDF editors can be required to accomplish some tasks, tools dedicated to *lemon* can ease common tasks, and offer a simplified view even for non RDF-experts.

Our LIME API aims at achieving that simplification for the management of LIME metadata. On the one hand, the use of our API allows for detection of common errors at compile-time. On the other hand, composition allows expressing complex queries via the API in a type-safe manner. Conversely, writing a SPARQL query as a string in a computer program does not benefit from any check by the compiler (misspellings are quite common) nor from the possibility to compose and reuse existing fragments.

Finally, one of the greatest advantages of a widespread API for a given resource is the consistency it brings in accessing it, providing further semantics by means of common use. This aspect is mostly relevant in the LIME profiling component, which generates metadata about a dataset by inspecting it. An agreed implementation of the standard can thus guarantee that everybody computes the metadata in the same manner.

3 LIME API

The LIME API consists of different layers providing assistance at progressively higher levels of abstraction. At the lowest level, a set of *vocabulary classes* define constants for the IRI defined by the OntoLex *lemon* specification, which can be used in a program in place of hardcoded strings (mitigating the risk of misspellings). Then a *repository (connection) wrapper* decorates the RDF4J API with operations tailored to the manipulation of LIME objects. At the highest level, the *profiler* disburdens the user from writing LIME metadata, by generating it automatically.

3.1 Vocabulary Classes

A first basic contribution of our API is a collection of vocabulary classes that provide constants for the IRIs defined by the OntoLex *lemon* specification. These vocabulary classes are alike the ones shipped by RDF4J for popular vocabularies.

The code in Fig. 1 provides an excerpt of the vocabulary class for the LIME module. These classes can be used in place of hand-written IRIs to manipulate RDF statements via RDF4J APIs such as the *RepositoryConnection* and *Model* APIs. In the fragment below we use the LIME vocabulary class for retrieving instances of the class `lime:Lexicon`:

```
Set<Resource> lexicons = QueryResults.asModel(
  conn.getStatements(null, RDF.TYPE, LIME.LEXICON)).subjects();
```

These vocabulary classes benefit developers with code completions and, to a limited extent, with static code checking: indeed, a misspelling in a constant identifier (e.g. `LIME.LEXICON`) is recognized as a compile-time error. Conversely, an error in a string literal (e.g. "`http://www.w3.org/ns/lemon/lime#Lexicon`") does not produce a compile-time error, and usually manifests itself in the form of runtime failures (which have to be traced back to the causing defect in the source code). Regarding the last aspect, it is worth to notice that our vocabulary classes were generated automatically directly form the ontologies accompanying the OntoLex specification, by means of a small utility we developed for that purpose.

3.2 Repository (Connection) Wrappers

While discussing the *vocabulary classes*, we presented a small fragment of code for retrieving all lexicons described in a metadata file. Fragments such as that one clearly express some information need that is generic enough to be turned into a first-class operation of our API.

To that end we developed a *connection wrapper* that decorates a plain RDF4J connection to an RDF Repository with specific operations for the LIME vocabulary.

```
public class LIME {

  public static final String NAMESPACE =
                      "http://www.w3.org/ns/lemon/lime#";

  public static final String PREFIX = "lime";

  public static final Namespace NS =
                  new SimpleNamespace(PREFIX, NAMESPACE);

  public static final IRI CONCEPTUALIZATION_SET;

  // ...
}
```

Fig. 1. An excerpt of code of the static fields describing the LIME vocabulary

Additionally, we provided a *repository wrapper* that overrides the operation `Repository#getConnection()` to return a wrapper around a connection created by the delegate repository. The code fragment below demonstrates the creation of a *repository wrapper*, which is then used to obtain a *connection wrapper*.

```
LIMERepositoryWrapper repoWrapper =
                          new LIMERepositoryWrapper(repository);

LIMERepositoryConnectionWrapper connWrapper =
                                     repoWrapper.getConnection();
```

The connection wrapper exposes an operation to retrieve all lexicons:

```
Collection<Resource> lexicons =
                 QueryResults.asList(connWrapper.getLexicons());
```

Homogeneously with similar operators in RDF4J, the method `getLexicons()` returns an iterator-like object. The class `QueryResults` can be used to fetch all values from this iterator, and put them into an ordinary Java collection. We defined similar operations for retrieving *lexicalization sets*, *concept sets*, *conceptualization sets* and *lexical linksets*.

One advantage of APIs over the direct of use a query language such as SPARQL is their composability, in the sense that the result of an API invocation can be processed through other APIs. While useful for rapid prototyping, this approach has the known downside of performing badly as the size of the data increases. The reason is that the composition is performed by the client program, losing the optimizations implemented by a query engine.

Let us consider how the operations above can be used to retrieve *lexicalization sets* for a given *reference dataset* and *language*. Firstly, one would obtain an iterator over all *lexicalization sets*, and then for each one check (with a separate API invocation) its *language* and *reference dataset*. This algorithm clearly requires a number of API calls proportional to the number of available *lexicalization sets*. Moreover, when the repository connection is accessing a remote dataset, each invocation entails a network round-trip-time and the transmission of potentially irrelevant data (i.e. every *lexicalization set* not matching the user criterion).

Alternatively, one could write just one SPARQL query and move the entire computation to the query engine, which would return only the relevant data. However, this approach is not compositional, and does not benefit from compile-time checks. Actually, if an information need is sufficiently recurring, it can be reasonable to promote the associated SPARQL query to a new operation in the API.

Nonetheless, in our API we reconcile compositionality and efficiency, by allowing the user to specify as Java code complex matching conditions that are then translated into a SPARQL query for efficient execution.

Our design was inspired by the library Hamcrest [5], which supports the declarative definition of *intents* for unit tests. Following its approach, we implemented an *embedded domain-specific language*, which is supported by a number of *class methods* that

instantiate matchers. In the examples below, we assume that the user has statically imported such methods, so that they can be used without mentioning the defining class (i.e. `LIMEMatchers`)

The fragment below shows how to retrieve *lexicalization sets* matching the aforementioned conditions on reference dataset and language:

```
Collection<Resource> lexicalizationSets =
  QueryResults.asList(conn.getLexicalizationSets(
    suchThat(
      language("en"),
      referenceDataset("http://example.org/referenceDataset1")
  )));
```

The API invocation above produces the following SPARQL query, which corresponds to the user's expressed information need:

```
SELECT ?v0 WHERE {
  ?v0 a lime:LexicalizationSet .
  ?v0 lime:language "en" .
  ?v0 lime:referenceDataset
                      <http://example.org/referenceDataset1> .
}
```

The method `getLexicalizationSets` expects a `Matcher<? super LexicalizationSet>` producing the SPARQL fragment expressing the desired selection criterion over *lexicalization sets*. `Matcher` is a generic interface, which can be instantiated with a Java type representing the class of resources over which the criterion is defined. The interface `LexicalizationSet` in the example is just one of the interfaces we defined to represent the classes in the LIME vocabulary. Following the example of the Guava library [6] by Google, the operation `getLexicalizationSets` uses the `super` keyword to express a lower bound on the type parameter in the instantiation of `Matcher`: specifically, it accepts a matcher over *lexicalization sets* and any of its super classes. To illustrate the rationale, consider matchers like functions taking an instance of a given type, say *Dataset* (i.e. in the sense specified by VoID [7]). In our example, we have a stream of *lexicalization sets*, which are a kind of *dataset*, and thus we can apply a matcher for datasets. Conversely, given a stream of *datasets*, we are not guaranteed that they are *lexicalization sets* and thus that a matcher over *lexicalization sets* is sensible. It is worth to notice that these constraints are not strictly necessary, as a matcher could simply reject an illegal argument. However, they are useful in practice, forcing the user to write conditions that conform to the domain and range restrictions expressed in the ontology. The methods `language` and `referenceDataset` instantiate matchers that test whether the resource under evaluation (a *lexicalization set* in our case) has the given value for the eponym property (in the LIME namespace). Since the domain of these properties is the union of different classes, we introduced a number of super interfaces, corresponding to different class unions. For example, `language` produces a `Matcher<LexicalizationSetOrLexicon>`, where `Lex-`

icalizationSetOrLexicon subsumes both LexicalizationSet and Lexicon. Finally, the method suchThat produces a matcher that is the conjunction of the matchers passed as arguments.

Our information need can be made even more complex, for example by requesting *conceptualization sets* with an *average ambiguity* less than 0.5 for an English lexicon the subject of which is *agriculture*, as for the example in Fig. 2

The fact that we push the comparison down into the SPARQL query, allows us to take advantage of the optimizations that some triple stores implement (e.g. GraphDB: http://graphdb.ontotext.com/documentation/free/storage.html#literal-index).

3.3 Profiler

The peak of the contribution of the LIME API is a *profiler*, which automates the computation of LIME metadata for different scenarios. As already said, the necessity of this component is determined by the need for a reference implementation of the metrics defined by the OntoLex specification.

Under the hood, the profiler executes some SPARQL queries on the provided dataset, recognizing the main types of dataset defined by LIME. Currently, the profiler is able to recognize lexicalizations of ontologies and thesauri expressed as RDFS and SKOS(-XL) labels, as well as using *lemon* lexical entries.

The behavior of the profiler can be controlled via configuration parameters, as well as an initial set of LIME metadata, e.g. telling the name and void:uriSpace of distributed *reference datasets* not contained in the profiled data. Work is underway to

```
Collection<Resource> conceptualizationSets = QueryResults
  .asList(conn.getConceptualizationSets(
    suchThat(
      avgAmbiguity(lessThan(0.5)), lexiconDataset(
        suchThat(language("en"),hasProperty(DCTERMS.SUBJECT,
          vf.createIRI(
            "http://dbpedia.org/resource/Agriculture")
        ))
      )
    )
   ));
```

```
SELECT ?v0 {
 ?v0 rdf:type lime:ConceptualizationSet .
 ?v0 lime:avgAmbiguity ?v1 .
 FILTER( ?v1 < "0.5"^^xsd:decimal)
 ?v0 lime:lexiconDataset ?v2 .
 ?v2 lime:language "en" .
 ?v2 dct:subject <http://dbpedia.org/resource/Agriculture> .
}
```

Fig. 2. A complex query expressed in the LIME API and its equivalent SPARQL code

complete the support for other dataset types defined in *lemon*, e.g. *concept sets*, *conceptualization sets* and *lexical linksets*.

Fig. 3 reports an excerpt of the metadata generated from a dump of the EuroVoc[1] thesaurus. In this scenario, the main dataset coincides with the *reference dataset*, and a number of `void:subsets` correspond to lexicalizations in different natural languages: for conciseness, we only show the details of the Italian lexicalization set. Currently, we assume that `void:entities` in the reference dataset gives the number of potentially lexicalizable resources, and thus that it can be used to compute `lime:percentage` and `lime:avgNumOfLexicalizations`. In case of thesauri, we count as entities only *concepts*, *collections* and *concept schemes*. So the fact that the number of lexicalized references is higher than the number of concepts is explained by the labels attached to *concept schemes*. The current implementation of the profiler does not collapse country-specific variants, and counts lexical entries by canonical form (altough the specification in some cases seems to suggest to count the lexical entry resources).

The AGROVOC [8] metadata is already published in the form of an associated VoID file (linked by all of its resources through the `void:inDataset` property) that includes LIME metadata[2] produced through (an earlier version of) this API, while EUROVOC will follow the same publication approach with the next release of its data.

```
@prefix meta: <http://eurovoc.example.org/void.ttl#> .
@prefix base: <http://eurovoc.example.org/void.ttl> .
<> a void:DatasetDescription ;
   foaf:primaryTopic meta:EuroVoc .

meta:EuroVoc a void:Dataset ;
   void:triples 2505003 ;
   dct:conformsTo <http://www.w3.org/2004/02/skos/core> ;
   void:classPartition [void:class skos:Concept ;
                        void:entities 7172 ], … ;
   void:entities 7302 ;
   void:subset meta:EuroVoc_mt_lexicalization_set, … .

meta:EuroVoc_mt_lexicalization_set a lime:LexicalizationSet;
   dct:language <http://id.loc.gov/vocabulary/iso639-1/it> ,
                <http://lexvo.org/id/iso639-3/ita> ;
  lime:avgNumOfLexicalizations 2.537 ;
  lime:language "it" ;
  lime:lexicalizationModel <http://www.w3.org/2008/05/skos-
xl> ;
  lime:lexicalizations 18521;
  lime:percentage 0.997 ;
  lime:referenceDataset meta:EuroVoc ;
  lime:references 7273 .
```

Fig. 3. An excerpt of the Lime metadata describing a dump of the EuroVoc thesaurus

[1] `http://eurovoc.europa.eu/`
[2] `http://aims.fao.org/aos/agrovoc/void.ttl`

Table 1. Results of the empirical evaluation on two thesauri. Timings are based on a single run.

	AGROVOC	EuroVoc
Time to load data (ms)	68044	42667
Time to profile (ms)	3622	2354
Total time (ms)	71666	45021
Triples	4800064	2505003
Schemes / Concepts / Collections	2 / 32721 / 0	130 / 7172 / 0
Lexicalizations	649498	389539
Languages	32	30

4 Evaluation

Our primary means for evaluating the LIME API was a set of unit tests accompanying its source code. Concerning the *repository (connection) wrapper*, the unit tests allowed us to spot a number of programming errors, which would have prevented some combination of conditions. The relevance of unit tests for the *profiler* is instead determined by their ability to function as an *executable specification* of what is the expected metadata for different dataset types. These tests *verify* that the profiler meets its specification, while their small size allows manual computation and verification of the statistics, so that we can ascertain the correctness of the specification, and ultimately *validate* our software. We hope that a few people knowledgeable about *lemon* will verify by hand these test cases, thus strengthening our confidence in the correctness of the profiles, especially when the profiler is applied to datasets sufficiently large that verification by hand is almost impossible. We believe that the efficiency gain of our approach to API composition has already been supported convincingly, while the scalability of the *profiler* required some empirical verification. To that end, we considered two very large thesauri encoded in SKOS: AGROVOC and EuroVoc. We performed the experiments on a Notebook equipped with an Intel(R) Core(TM) i7-3630QM CPU @ 2.40GHz, 24 GB of RAM DDR3 1600 MHz, Windows Home 10 64bit operating system and Oracle JDK 8 (build 1.8.0_121-b13). **Table 1** reports the total required time, as well as its break out into time to load the data and time to produce the metadata.

5 Related Works

Our LIME API is similar in purpose to existing APIs for *lemon*, such the one [9] developed in the context of the Monnet Project and the one [10] developed as part of a PhD thesis on the automatic lexicon generation system M-ATOLL [11].

Both the M-ATOLL and the Monnet *lemon* API define an object-oriented model corresponding to the (Monnet) *lemon* ontology. However, they differ in how they implement that model. The M-ATOLL API uses the RDF middleware Apache Jena [12,13] as a means to read/write an in-memory representation of the ontology lexicon.

Crucially, they adopt a white-listing or pull approach, so that only known properties from the RDF input are loaded into memory and thus can be accessed through the API. In a certain sense, this approach misses the opportunity offered by RDF to use and combine different vocabularies together. Conversely, our *repository (connection) wrapper* is an extension of the RDF4J API: therefore, our convenience methods can be combined freely with low-level RDF operations, and it is even possible to use SPARQL as a query language. With this regard, the Monnet API is more similar to our LIME API, since the object-model can be implemented as a view over a (remote) RDF dataset. Moreover, they allow to retrieve resources of interest based on the evaluation of arbitrary SPARQL queries. To our knowledge, however, the API does not offer any facility – as our LIME API does – to compositionally assemble these queries, which must be represented as strings.

The profiling module for the automatic generation of LIME metadata is related to works on dataset analytics. With this regard, LODStats [14] is an extensible framework for dataset analytics, which supports the efficient computation of different statistics. They designed a formalism for the explicit representation of these statistics, aiming at *understandability* and *semantic clarity*. We pursued these goals, by implementing the profiler as a collection of SPARQL queries. Conversely, LODStats traded the expressivity of full-graph patterns for a more lightweight formalism that was better suited to stream processing. LODStats needs this level of scalability, since it has been deployed as a publicly-accessible service providing statistics on an as-big-as-possible portion of the Linked Open Data cloud. Currently, we target smaller-scale deployments, related to a few datasets under the control of the same publisher. In this scenario, the publisher is likely to have already loaded these datasets into a triple store capable of executing SPARQL queries. Indeed, our evaluation on two real-world thesauri showed that profiling is a heavy but doable activity.

A related use case is the integration of the profiler into an RDF/ontology/thesaurus editor as a means to better understand the dataset being edited from the perspective of available lexical material. Current support for statistics inside editors is limited to lower-level structural characteristics. TopBraid Composer [15] reports the number of triples and the number of instances per meta-class (i.e. classes defined by the ontology vocabulary, such as `owl:Class`) and per domain class. Furthermore, it differentiates between local, imported and inferred entities. VoID defines the property `void:triples` to represent the number of triples in a dataset, while instances of different meta-classes can be represented via different `void:classPartition`s. Unfortunately, the distinction between local, imported and inferred is not easy to capture. The number of local classes and properties can be represented more succinctly via the VOAF [16] vocabulary, utilized by the Linked Open Vocabularies [17] (LOV) project. However, there is a semantic mismatch, because VOAF ignores deprecated classes and properties, whereas TopBraid does not. It is interesting to observe that similarly to what we did, the documentation of VOAF associates its metadata vocabulary with SPARQL queries, as a sort of operational definition. Protégé [18] (in the specific, version 5.1.0) computes a number of *ontology metrics*, which are essentially counts over different types of ontology axioms. An interesting feature is the computation of the DL expressivity, i.e. the specific language in the DL family used in the ontology.

The aforementioned LODStats and LOV already report languages used in a dataset. However, they do not fulfill the needs of the OntoLex community for a mechanism to discover related but independently published lexical material. In fact, LOV supports this interlinking at the level of vocabularies, by making explicit the relationships between vocabularies and their use in datasets.

6 Conclusions and Future Works

As we envisaged [19] in the context of ontology mediation, lexical assets should be properly published and made findable on the web, in order to be properly exploited for accomplishing diverse tasks (e.g. discovery, alignment, etc..).

Towards the realization of such a scenario, our work on LIME API intends to foster the creation of LIME metadata describing lexical assets published on the Semantic Web. In order to reach end-users, we also aim plan to seamlessly support the use of the *lemon* vocabulary in the third version of our thesaurus/ontology editor VocBench [20]. In [21], we have already shown how *custom forms* support the instantiation of the Design patterns for the ontology-lexicon interface [22] within VocBench. Thanks to the work presented in this paper, we will also support the production of LIME metadata by integrating the LIME profiler into VocBench 3.

Concerning the lower levels of the API for data manipulation, we will extend them with additional operations (in particular, write operations), borrowing and integrating ideas from previous *lemon* APIs. Depending on the feedback of the community, we will also consider to support other *lemon* modules (starting from the *core*). Similarly, we will investigate the usefulness of an extension of the LIME module to support other *lemon* modules in addition to the core one.

Acknowledgments. This work has been partially funded by the ISA^2 Work Programme action for the realization of the VocBench 3 collaborative editing platform for ontologies and thesauri: `https://ec.europa.eu/isa2/solutions/vocbench3`

References

1. Cimiano, P., McCrae, J.P., Buitelaar, P.: Lexicon Model for Ontologies: Community Report, 10 May 2016. Community Report, W3C (2016) https://www.w3.org/2016/05/ontolex/.
2. Fiorelli, M., Pazienza, M.T., Stellato, A.: LIME: Towards a Metadata Module for Ontolex. In : 2nd Workshop on Linked Data in Linguistics: Representing and Linking lexicons, terminologies and other language data, Pisa, Italy (2013)
3. Fiorelli, M., Stellato, A., McCrae, J.P., Cimiano, P., Pazienza, M.T.: LIME: the Metadata Module for OntoLex. In Gandon, F., Sabou, M., Sack, H., d'Amato, C., Cudré-Mauroux, P., Zimmermann, A., eds. : The Semantic Web. Latest Advances and New Domains (Lecture Notes in Computer Science). Proceedings of the 12th Extended Semantic Web Conference (ESWC 2015), Portoroz, Slovenia, May 31 - 4 June, 2015. 9088. Springer International Publishing (2015), pp.321-336

4. RDF4J. Available at: http://rdf4j.org/
5. Hamcrest. Available at: http://hamcrest.org/
6. Guava: Google Core Libraries for Java. Available at: https://github.com/google/guava
7. Alexander, K., Cyganiak, R., Hausenblas, M., Zhao, J.: Describing Linked Datasets with the VoID Vocabulary (W3C Interest Group Note). In: World Wide Web Consortium (W3C). (Accessed March 3, 2011) Available at: http://www.w3.org/TR/void/
8. Caracciolo, C., Stellato, A., Morshed, A., Johannsen, G., Rajbhandari, S., Jaques, Y., Keizer, J.: The AGROVOC Linked Dataset. Semantic Web Journal 4(3), 341–348 (2013)
9. The Lemon API. In: Monnet Project. Available at: https://github.com/monnetproject/lemon.api
10. Lemon API. Available at: https://github.com/ag-sc/lemon.api
11. Walter, S.J.: Generation of Multilingual Ontology Lexica with M-ATOLL. Ph.D. dissertation (2016)
12. Apache Jena. Available at: https://jena.apache.org/
13. McBride, B.: Jena: Implementing the RDF Model and Syntax Specification. In : Semantic Web Workshop, WWW2001 (2001)
14. Auer, S., Demter, J., Martin, M., Lehmann, J.: LODStats – An Extensible Framework for High-Performance Dataset Analytics. In ten Teije, A., Völker, J., Handschuh, S., Stuckenschmidt, H., d'Acquin, M., Nikolov, A., Aussenac-Gilles, N., Hernandez, N., eds. : Knowledge Engineering and Knowledge Management. Springer, Berlin, Heidelberg (2012), pp.353-362
15. TopBraid Composer. Available at: http://www.topquadrant.com/tools/ide-topbraid-composer-maestro-edition/
16. Vocabulary of a Friend (VOAF). Available at: http://purl.org/vocommons/voaf
17. Vandenbussche, P.-Y., Atemezing, G.A., Poveda-Villalón, M., Vatant, B.: Linked Open Vocabularies (LOV): a gateway to reusable semantic vocabularies on the Web. Semantic Web 8(3), 437--452 (2017)
18. Gennari, J., Musen, M., Fergerson, R., Grosso, W., Crubézy, M., Eriksson, H., Noy, N., Tu, S.: The evolution of Protégé-2000: An environment for knowledge-based systems development. International Journal of Human-Computer Studies 58(1), 89–123 (2003) Protege.
19. Fiorelli, M., Pazienza, M.T., Stellato, A.: A Meta-data Driven Platform for Semi-automatic Configuration of Ontology Mediators. In Calzolari, N., Choukri, K., Declerck, T., Loftsson, H., Maegaard, B., Mariani, J., Moreno, A., Odijk, J., Piperidis, S., eds. : Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14), Reykjavik, Iceland (May 2014)
20. Stellato, A., Rajbhandari, S., Turbati, A., Fiorelli, M., Caracciolo, C., Lorenzetti, T., Keizer, J., Pazienza, M.T.: VocBench: a Web Application for Collaborative Development of Multilingual Thesauri. In : The Semantic Web. Latest Advances and New Domains (Lecture Notes in Computer Science) 9088. Springer International Publishing (2015), pp.38-53
21. Fiorelli, M., Lorenzetti, T., Pazienza, M.T., Stellato, A.: Assessing VocBench Custom Forms in Supporting Editing of Lemon Datasets. In Gracia, J., Bond, F., McCrae, J.P., Buitelaar, P., Chiarcos, C., Hellmann, S., eds. : Language, Data, and Knowledge (Lecture Notes in Artificial Intelligence) 10318. Springer International Publishing (2017) (in press).
22. McCrae, J.P., Unger, C.: Design Patterns for Engineering the Ontology-Lexicon Interface. In Buitelaar, P., Cimiano, P., eds. : Towards the Multilingual Semantic Web. Principles, Methods and Applications. Springer Berlin Heidelberg (2014), pp.15-30