Cite as:
Desolda G. (2015). Enhancing Workspace Composition by Exploiting Linked Open Data as a Polymorphic Data Source. In *Proc. of the 8th International Conference on Intelligent Interactive Multimedia Systems and Services (IIMSS 2015)*, Sorrento (Napoli), Italy, 17-19 June 2015. Vol. 40, pp. 97-108.
ISBN: 978-3-319-19829-3, DOI: 10.1007/978-3-319-19830-9_9

Enhancing Workspace Composition by Exploiting Linked Open Data as a Polymorphic Data Source

Giuseppe Desolda

Dipartimento di Informatica, Università degli Studi di Bari Aldo Moro
Via Orabona, 4 – 70125 – Bari, Italy
giuseppe.desolda@uniba.it

Abstract. In the last decade, the World Wide Web has been evolving as a data infrastructure, where a wide variety of resources is increasingly being made available as Web services. This trend is pushing the researchers to investigate approaches like composition platforms, aimed at empowering end users to access, compose and use these services. Despite the wide availability of data sources, due to the specific and diverse end users' information needs often no data source can satisfy these needs. This limits the adoption of composition platforms in real contexts and everyday use. In order to overcome this limitation, this paper presents a polymorphic data source that exploits the wide availability of information structured in the Linked Open Data cloud. To build this data source, a semi-automatic annotation algorithm is presented that creates semantic annotations for services available in a composition platform. An implementation of this approach in a mashup platform is described.

Keywords: Mashup, Linked Open Data, Semantic Web.

1 Introduction and motivations

Over the past years, we have been facing a growing amount of heterogeneous data sources available on the Web. When writing this paper, the site programmableweb.com lists more than 12000 API to retrieve data or exploit functionalities. This paper is about data retrieval APIs that can be classified into cross-domain (Wikipedia, YouTube, Google, etc.) or domain specific APIs (Government, Life Science, Music, etc.). This huge amount of information available on the Web and the opportunities offered by Web 2.0 are pushing researchers to investigate new methodologies, technologies and mechanisms to allow laypeople, i.e., end users without expertise in programming, to access and manipulate data sources by exploiting visual mechanisms. In the last 10 years, different platforms with various composition paradigms have been proposed [1, 2]. Typically, they implement visual mechanisms to access, create, compose, modify and use data sources usually available through the APIs. They are often known as mashup platforms.

According to [3], different features affect the mashup quality, for example, the *data quality* dimension, characterized by *accuracy, timeliness, completeness* and *availabil-*

ity. Regarding this dimension and in particular the completeness, the data sources available nowadays describe a portion of a domain and often do not include many details. It is sometimes possible to overcome this limitation by composing different data sources, but in some cases, when the end users' information need is more specific, no data source could provide the useful information. This is a vast limitation in exploiting mashup platforms in real contexts. In fact, although the current platforms allow laypeople to easily use and compose data sources, often they cannot benefit from a composition platform due to a lack of data sources if used in real contexts.

To overcome this lack of information and better satisfy the end users' information needs, this paper presents a new *polymorphic* data source built upon the Linked Open Data cloud. It is called polymorphic because it provides mutable information with respect to the data sources of which it is composed.

The remainder of this paper is structured as follows. Section 2 describes the polymorphic data source, the use of Linked Open Data to build this data source and the integration of the polymorphic data source in a mashup platform. Section 3 describes the annotation algorithm and its performance evaluation. Section 4 reports related works. Section 5 concludes the paper and also outlines future work.

2 Polymorphic data source: a source for many purposes

To explain the idea of a polymorphic data source, let us consider the following scenario that refers to a typical situation when laypeople want to exploit a platform to mashup services according to their needs, but at a certain point they leave it because they do not find useful data sources available through the platform. "John is using a mashup platform. He adds the SongKick service to his workspace to find upcoming musical events in his city. He also needs to retrieve, for each event artist, a list of related videos. For this purpose, John composes SongKick artist attribute with YouTube. Now, John has two widgets in his workspace: SongKick and YouTube; the first allows him to search upcoming musical events and the second automatically performs a search (with the artists' name) each time John clicks on a specific musical artist in the list of upcoming events. Afterwards, he wants to know, for each artist, details such as genre, starting year of activity and artist photo. *Searching for useful services on the composition platform, John does not find any service that satisfies his needs. Thus, John is not supported anymore by the platform and has to go to the Web for a usual (manual) search for the specific information".

Let us now look at a scenario that is the same as the previous one until the asterisk, but it goes on with the following. "To retrieve the desired information, John decides to expand the SongKick artist attribute with the polymorphic data source. When he chooses the polymorphic data source, the platform shows a list of new properties related to the concept of musical artist. Thus, John decides to create the new data source with the genre, the starting year of activity and the artist photo properties. Henceforward, John can find a list of upcoming events on SongKick and can visualize the additional artist's information on the polymorphic data source by clicking on a specific artist on SongKick (Fig. 1)".

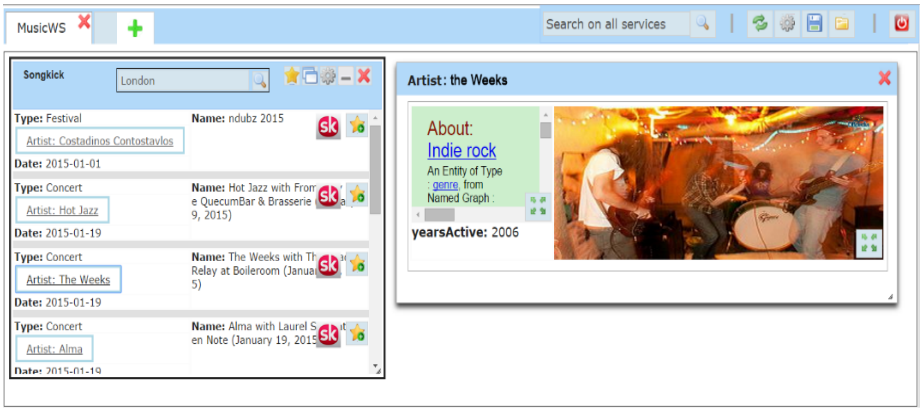

Fig. 1. Composition of DBpedia polymorphic data source with SongKick artist attribute.

In the previous scenario, John could continue to compose SongKick with the same polymorphic data source starting from other SongKick attributes. For each attribute that John decides to expand, the polymorphic data source provides different properties related to the semantics of the starting attribute (for example, for the SongKick place attribute, properties like borough, census, year and demographics should be shown). Thus, this type of data source is considered polymorphic because it can provide different information (properties) according to the data source attribute that is selected. On the contrary, the classic data sources (YouTube, Wikipedia, etc.) provide the same properties independently of the selected attribute.

2.1 Linked Open Data as a basis for the polymorphic data source

The polymorphic data source is built by exploiting the huge amount of information available in the Linked Open Data (LOD) cloud. In 2009, Tim Berners-Lee defined Linked Data as "a set of best practices for publishing and connecting structured data on the Web" [4]. The goal of the Linked Data project is to publish data so that they are readable by a human and an automatic agent. The LOD are Linked Data distributed under an open license that allows its reuse for free. At the time of this paper, there are more than 1000 KB datasets published in the LOD cloud.

Nowadays, one of the biggest KBs in the LOD cloud is DBpedia (the structured version of Wikipedia). The DBpedia English version describes 4.58 million things, out of which 4.22 million are classified in its ontology. In the DBpedia ontology, there are 685 classes. Thanks to the availability of this huge amount of information and its semantics structured in the DBpedia ontology, DBpedia has been chosen as the starting point to create the polymorphic data source.

An annotation algorithm has been developed to annotate each attribute of each service (available in a platform) with a class of the DBpedia ontology. Each class has to be semantically similar to the attribute.

2.2 An implementation in a mashup platform

The algorithm and the polymorphic data source have been implemented in the mashup platform described in [5]. This platform provides a composition paradigm elicited during users' studies [6] and evaluated during field studies in cultural heritage and technology enhanced learning domains [7]. The platform is implemented by using Primefaces, an open source User Interface (UI) component library for JavaServer Faces (JSF) based applications. It is deployed on a remote Apache Web server and a Mysql database is used for the user account management. It provides different mashup mechanisms to access, compose and use services. In particular, the end user can add services to his own workspaces and can compose services by using two mashup mechanisms called *join* and *union* [6].

In this paper only the join composition is considered. As reported in the scenarios, with the join function the user composes a service A with a service B, in order to expand the results of service A with details provided by service B. This composition is assisted by a wizard procedure that the user activates by clicking on a gearwheel button in the upper right corner of the service widget. By clicking on this icon, the user 1) selects the attribute that he wants to expand, 2) selects the data source from which to gather information (DBpedia in our case), 3) chooses a visual template to visualize the new results and finally 4) uses drag&drop to map a subset of service attributes into the visual template. The fourth step is the most interesting for the aim of this paper. In fact, while composing a service with another 'traditional' service the list of attributes in step 4 is always the same, by choosing the polymorphic data source the list of attributes is different in relation to the semantics of the selected attribute during step 1.

In the current implementation, the service that provides details is shown as a window only when the user clicks on a specific item (e.g., click on "the Weeks" artist in Fig. 1). This visualization emerged as requirement during the users' study described in [6].

3 An algorithm for data source annotation

This section describes a semi-automatic algorithm that creates semantic annotations for services available in a composition platform. Its performance evaluation is also reported.

3.1 Generation of a set of candidate classes to annotate attributes

In order to create this polymorphic behaviour, a mapping step is required between all data source attributes registered in the platform and the DBpedia ontology classes. In general, this problem falls in the ontology matching area. In order to start from consolidated approaches, the methodologies surveyed in [8, 9] were investigated with the aim of creating an ad-hoc solution based on the literature, without the pretension of building a new ontology matching methodology. Furthermore, a great deal of specific literature in the semantic web area has already been produced for the problems of semantic annotation of a service [10]. These approaches have been taken into account to design the proposed algorithm.

The proposed solution can be classified as a semi-automatic and instance-based annotation algorithm. As is described in the following, it is defined as semi-automatic because a user has to provide a set of example queries (about 10) when a data source is registered in the platform. Furthermore, it is defined as instance-based because it infers a set of candidate classes to annotate the attributes starting from the results (instances) of the queries. The main goal of the algorithm is to annotate each attribute of each service with a DBpedia class that is semantically similar to the attribute. The algorithm is reported in Table 1. The criterion for choosing the most important class, as specified in line 12 of Table 1, is illustrated in Section 3.2.

Table 1. The instance-based semi-automatic annotation algorithm

	Input: Set *T* of Triples *t*=(*s,A,Q*), *s* is a data source, *A* is a set of attributes *a* for s, and *Q* is a set of queries on *s* **Output:** Set *R* of results *r=(s, L)*, *s* is a data source, *L* is a set of <a_i,l_i> where *a* is an attribute of *s* and *l* is its label
1:	**for** each $t \in T$ **do**
2:	**create** *I* as empty set of instance results for queries and an empty set M
3:	**for** each $q \in Q$ **do**
4:	**query** *s* by using *q* and collect instance results into *I*
5:	**end for**
6:	**for** each attribute *a* of *s* **do**
7:	**create** *L_temp* as empty set of labels for *a*
8:	**for** each $i \in I_a$ **do**
9:	**query** DBpedia by using value of *i* and obtain a set *C* of classes
10:	**put** values of *C* into *L_temp*
11:	**end for**
12:	**calculate** most important class *l* in *L_temp* and put <a,l> into L
13:	**end for**
14:	**end for**

In order to understand the algorithm, an example of its execution on a real data source is here reported. Let us consider SongKick data source *s* and a set *Q* of queries on it (e.g., London, Liverpool, Rome). This set Q is manually provided only once at the time of service registration in the platform. Each instance of the SongKick results is characterized by the set of attributes *A= {artist, place, event_type, event_name, date}*. The algorithm starts by executing all the queries in Q on SongKick and by collecting all the results in set *I*. For each attribute a_i in A, the algorithm considers all the instances selecting only the a_i attribute values. For example, after having queried SongKick with queries in Q, for the artist attribute the algorithm creates set I_{artist}*= {Ligabue, U2, One Direction, Taylor Swift,...},* that is a set of musical artists who will perform at places stored in Q. The algorithm uses each instance of I_{artist} to query DBpedia. The aim is to find the same instance of each element in I_{artist} as a DBpedia thing and add its classes in set *L_temp*. Obviously, although the instances of each attribute have the same meaning (for example, all artist instances are singers), not all the retrieved DBpedia things are instances of the same class. Thus, at the end of the execution on all the attributes, the results appear as shown in Table 2, where the *Class* column indicates the DBpedia classes inferred for each attribute and the *%* column indicates the frequency of the classes at the end of DBpedia queries. Furthermore, when the algorithm queries DBpedia, sometimes the retrieved things are wrongly classified. For example, when DBpedia is

queried with the 'Ligabue' string, three thing instances of different classes are retrieved: one instance of Artist (Luciano Ligabue, singer), one instance of Agent (Antonio Ligabue, painter) and one instance of Italian_opera_singer (Ilva Ligabue, opera singer). Obviously, the second and third things are false positives that create noise in the set L_temp. However, it is empirically observed that this noise represents only a tiny percentage (typically less than 4%). For this reason, no classes less than 4% are considered in the rest of the algorithm.

Table 2.Candidate classes for annotable attributes of the SongKick data source; the *Class* column indicates the DBpedia class associates; the % coloumn indicates the frequency of each class

Artist		**EventType**		**Location**	
Class	***%***	***Class***	***%***	***Class***	***%***
Agent	14	Event	25	Place	17
Work	13	FilmFestival	25	Settlement	11
MusicaWork	12	Organization	13	PopulatedPlace	11
Organization	7	Television	13	Work	6
Album	7	Agent	12	Album	6
Person	7	TelevisionShow	12	MusicalWork	6
Band	6			Agent	6
Artist	6			Organization	6
Single	5			Building	6
MusicalArtist	5			Architectural	6
…	…			…	…

Until now, the algorithm has collected a set of promising (candidate) classes to annotate attribute data sources. The easiest annotation solution could be to select the most frequent class (In Table 2 *Agent* for Artist, *Event* for EventType and *Place* for Location), but the performance of this solution is improved by the second step of the proposed algorithm that takes into account both the class frequency and the ontology tree structure.

3.2 Choosing the best class from the set of candidates

The starting point for choosing the best class for each data source attribute is the set of promising classes that the algorithm has built in the previous step. The goal of the next step is to assign to each class in Table 2 a rating that takes into account both the class frequency and the ontology tree structure. In the end, the class with the highest score is used to annotate the service attribute. To explain why the tree structure is important, consider the DBpedia sub-tree in Fig. 2. In that figure the sub-tree of the DBpedia ontology is depicted; it has been built by considering, for an easy explanation, the first ten candidate classes of the attribute Artist in Table 2. If the algorithm annotates the Artist attribute with the most frequent class, then the Agent class (14%) will be chosen. However, by looking inside the semantics and the properties that characterize this class, it is evident that the Agent class is too general for the concept of the musical artist of the SongKick Artist attribute. We need to choose a more specific class. There are two aspects to consider, in order to annotate data sources with the best classes: the *class coverage* and the *number of properties*. The class coverage is the percentage of retrieved

DBpedia instances covered by each class (node percentages in Fig. 2) and it is calculated as sum of class percentage with all its sub-class percentages. This value is higher in the ontology top level classes and vice versa, because each class also cover the sub-class instances. On the other hand, the more specific is the class, the more properties the user can choose when creating the polymorphic data source. For this reason, the proposed algorithm tries to find a trade-off between the class coverage and the number of properties.

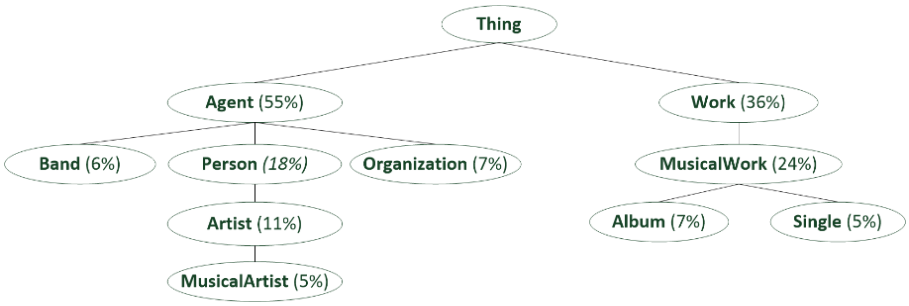

Fig. 2. Sub-tree of the DBpedia ontology built by using the SongKick artist attributes. For each node, the percentage indicates the class coverage

In order to take into account these aspects, the *x-value* is introduced. It is an index that quantifies the semantic similarity between a class and an attribute considering both the frequency and the ontology structure. In particular, to consider the ontology structure, the algorithm starts by generating all the combinations of the sub-trees built with the classes in the list of candidates. The length of these groups ranges from 2 up N (where N is the number of candidate classes of an attribute). For each class in these sub-trees, the algorithm calculates the x-value. At the end of the computation, each class has many x-values, but only the highest value of each class is considered for the final ranking. The generation of these groups is performed to explore all possible paths in the ontology tree, as also performed in [11]. In particular, when the algorithm generates all sub-trees, new classes could be added to the candidate list. For example, let us consider the Fig. 2 and suppose that the MusicalWork class is not generated in the candidate list of Table 2. During the generation of all sub-trees, when the Single and Album classes are used to create a sub-tree, also MusicalWork is considered because it is their common ancestor. Thus, MusicalWork is added in the candidate list and its x-value is calculated. Choosing the MusicalWork, or an ancestor in general, the coverage could be higher than its sub-class coverage, maintaining a good number of properties that describe the semantics of the considered data source attribute.

The formula of an x-value that estimates the power of a class into each sub-tree is:

$$x - value = (classAncestorRatio + parentPower + nodePower)\%class.$$

Let us analyse in detail each component. The first one is *classAncestorRatio*.

$$classAncestorRatio = \frac{\%class * nLevelScore}{\%commonFather * nLevelScore}$$

This value takes into account the coverage of the current class with respect to the coverage of the first common ancestor in the sub-tree. To penalize classes in the higher levels (such as the ancestors), the numerator and denominator are multiplied by the *nLevelScore*, a number that ranges from 1 to 100 and it is calculated as:

$$nLevelScore = (100/OntologyDepth) * classLevel$$

In this formula, the *OntologyDepth* indicates the maximum depth of the ontology (7 in DBpedia), while *classLevel* specifies the depth of the considered class. It is evident that nLevelScore is high in deeper levels and low in higher levels. In this way, in the classAncestorRatio, the classes at higher levels are penalized.

The second component in the x-value is the *parentPower*. It quantifies the impact of the common ancestor with respect to all the classes at the same level.

$$parentPower = \frac{\%TotRoot}{\%TotRootLevel} \qquad classPower = \frac{\%class}{\%allClasses}$$

In this component, *%TotRootLevel* is the sum of the coverage rate of all classes at the same level of the sub-tree root. The *%TotRoot* is the coverage rate of the sub-tree root. This component is introduced in the x-value to solve the problem of the class sparsity in a tree. Let us consider in the Fig. 2 the Work and Agent classes. When sub-trees with Work or Agent as ancestors are generated, this component has a high value in Agent sub-classes. In this way, the x-value rewards sub-classes in the Agent branch instead of those in the Work branch.

The third component is the *classPower*, which indicates the weight of the considered class in the sub-tree. In classPower, the *%class* is the percentage of the class considered in the sub-tree and *%allClasses* is the sum of the percentage of all classes in the considered tree.

At the end of the generation of all trees and the calculation of the x-values, the list of candidate classes is expanded with all the new ancestor classes of the generated sub-tree. Each class has several ratings, one for each group in which it appears. The class with the highest score is selected to annotate the service attributes.

3.3 Performance evaluation of the annotation algorithm

To the best of our knowledge, no datasets with data sources attributes exist annotated with DBpedia class. Thus, to establish the performance of the algorithm, two experts created and manually annotated a set of 7 services (for a total of 18 annotable attributes) by using DBpedia classes. In fact, not all the services attributes can be annotated with a DBpedia class (i.e. URL attributes). Furthermore, due to the nature of the algorithm, the numerical attributes (ticket price, temperature, humidity, height, weight, etc.) cannot be annotated because it is impossible to infer the classes from numerical values. As described in the Future Work Section, this limit can be overcome by combining the proposed approach with natural language processing of the attribute name [12, 13].

To evaluate the performance of the automatic annotation algorithm, a new metric called *Accuracy* is introduced. First, a score has been associated with each attribute

comparing its automatic annotation to the manual one (AAA stands for Attribute Automatically Annotated; AMA stands for Attribute Manually Annotated). In particular:

- 10 points if AAA = AMA;
- 8 points if AAA is at the same level of AMA (not the same, but very similar semantic);
- 7 points if AAA is 1 level up/down from AMA as a sub-class or a super-class;
- 5 points if AAA is 2 levels up/down from AMA as a sub-class or a super-class;
- 0 points in all other cases.

The *Accuracy* of the overall automatic annotation is calculated as:

$$Accuracy = \frac{\sum_i^N A_i}{\sum_i^N MAXscore}$$

In the Accuracy formula the numerator is the sum of the accuracy of all the attributes, instead the denominator is the sum of the MAXscore that is the maximum accuracy that an attribute can have (10 in our case). The final accuracy ranges from 0 to 100. The Accuracy is calculated both for the annotation performed by associating the most frequent classes in Table 2 (baseline) and for the annotation performed by associating the classes with the proposed algorithm.

Table 3. Accuracy comparison between the baseline and the algorithm

	Accuracy
Baseline	56%
Algorithm	91%

As shown in Table 3, it is evident how important it is to consider the ontology structure during the automatic annotation procedure. In fact, in the latter (91%) the accuracy is clearly improved.

This metric is quite different with respect to the ones such as precision and recall used for the service semantic annotations [13]. In fact, the classic precision and recall consider true and false values, if the automatic annotation matches the manual one. However, in our case, we can also consider as good annotations classes like super-classes/subclasses, but penalizing them because they do not match exactly the manual annotation. The penalizing factor has been empirically established.

4 Related work

The core of the entire paper is the new polymorphic data source, introduced in the composition platforms to overcome the problem of lack of data source. To build this type of data source, the services available in composition platforms need to be enriched with semantic annotations. To the best of our knowledge, no previous works have tried to solve this problem with similar data sources. However, much effort has been dedicated to enriching Web services with semantic annotations, as described in [10]. In general, the goal of semantic annotations is to improve the mashup platforms with mechanisms

like service recommendation, to assist the users during composition [14] or service discovery [15]. This is a hard problem because the Web APIs lack explicit and sufficient semantic information. In fact, API providers usually offer details like input and output parameters in the form of unstructured text in their Web page. A system that tries to exploit this HTML information is SWEET that assists the user in manually annotating services with hRESTS and MicroWSMO formats with Web page information [16]. However, the weakness of this approach is the heterogeneity of the provider Web pages, the lack of information and the manual end user effort that limits the large-scale service annotation.

To overcome the limits of manual methods, different semi-automatic approaches have been proposed. For example, two different solutions reported in [12, 13] annotate services with DBpedia classes and their attributes with DBpedia properties based on syntactical matching and other natural language processing techniques. Although the fact that these approaches seem promising, they still require the intervention of an expert to set-up the system by adding low-level APIs details [12] and, mostly, the presence of WSDL service descriptors that are not available for all Web APIs and that still require a heavy manual effort.

In order to overcome the limitations identified in the literature, the annotation algorithm proposed in this paper has been designed to be: 1) usable by laypeople (no expertise is required to provide example queries), 2) fully automatic (except for the typing of a set of queries) and 3) not constrained by the presence of an HTML or a WSDL service description. It is also reusable, in order to annotate services with other ontology classes (e.g., Freebase or Yago).

5 Conclusions and future work

This paper describes a polymorphic data source, a solution that aims to address an important limitation that affects the use of a composition platform in real contexts. In order to build this new polymorphic data source, an annotation algorithm has been developed. The initial results reveal that the algorithm creates good annotations that reflect a good quality of the polymorphic data source.

Although the algorithm performance appears encouraging, the proposed algorithm does not aim to solve ontology matching problems. It is only based on the literature and is an ad-hoc solution for the specific problem. Thus, one aspect that could be addressed in the future is the improvement of the algorithm by investigating techniques as, for example, NLP approaches [12, 13], to improve the annotation accuracy and to annotate non-annotable attributes (e.g., the numerical attributes). Finally, user studies are planned to evaluate the benefits of a polymorphic data source.

Acknowledgments. This work is partially supported by the Italian Ministry of University and Research (MIUR) under grant PON 02_00563_3470993 "VINCENTE" and by the Italian Ministry of Economic Development (MISE) under grant PON Industria 2015 MI01_00294 "LOGIN". We also thank the student Vincenzo Lucente for contributing to system implementation.

References

1. Cappiello C., Matera M., Picozzi M., Sprega G., Barbagallo D., and Francalanci C. (2011). DashMash: A Mashup Environment for End User Development. In *Web Engineering*. Auer S., Díaz O., and Papadopoulos G.A. (ed.), Vol. 6757, pp. 152-66. Springer. Berlin / Heidelberg.
2. Ardito C., Costabile M. F., Desolda G., Lanzilotti R., Matera M., Piccinno A., and Picozzi M. (2014). User-Driven Visual Composition of Service-Based Interactive Spaces. In *Journal of Visual Languages & Computing*. 25, 4, 278-96.
3. Yahoo! Inc. YahooPipes. Retrived February 22 from http://pipes.yahoo.com/pipes/
4. Cappiello C., Daniel F., and Matera M. (2009). A Quality Model for Mashup Components. In *Web Engineering*. Gaedke M., Grossniklaus M., and Díaz O. (ed.), Vol. 5648, pp. 236-50. Springer Berlin Heidelberg.
5. Bizer C., Heath T., and Berners-Lee T. (2009). Linked data-the story so far. In *International journal on semantic web and information systems*. 5, 3, 1-22.
6. Ardito C., Costabile M. F., Desolda G., Lanzilotti R., Matera M., and Picozzi M. (2014). Visual Composition of Data Sources by End-Users. In *Proceedings of the International Working Conference on Advanced Visual Interfaces*, Como, Italy. pp. 257-60.
7. Ardito C., Bottoni P., Costabile M. F., Desolda G., Matera M., and Picozzi M. (2014). Creation and Use of Service-based Distributed Interactive Workspaces. In *Journal of Visual Languages & Computing* 25, 6, 717-26.
8. Hooi Y., Hassan M. F., and Shariff A. (2014). A Survey on Ontology Mapping Techniques. In *Advances in Computer Science and its Applications*. Jeong H.Y., S. Obaidat M., Yen N.Y., and Park J.J. (ed.), Vol. 279, pp. 829-36. Springer Berlin Heidelberg.
9. Shvaiko P. and Euzenat J. (2013). Ontology Matching: State of the Art and Future Challenges. In *Knowledge and Data Engineering*, IEEE Transactions on. 25, 1, 158-76.
10. Reeve L. and Han H. (2005). Survey of semantic annotation platforms. In *Proceedings of the 2005 ACM symposium on Applied computing*, Santa Fe, New Mexico. pp. 1634-8.
11. Jain P., Hitzler P., Sheth A. P., Verma K., and Yeh P. Z. (2010). Ontology alignment for linked open data. In *Proceedings of the 9th international semantic web conference on The semantic web - Volume Part I*, Shanghai, China. pp. 402-17.
12. Saquicela V., Vilches-Blázquez L. M., and Corcho Ó. (2010). Semantic Annotation of RESTful Services Using External Resources. In *Current Trends in Web Engineering*. Daniel F. and Facca F. (ed.), Vol. 6385, pp. 266-76. Springer Berlin Heidelberg.
13. Zhang Z., Chen S., and Feng Z. (2013). Semantic Annotation for Web Services Based on DBpedia. In *Proceedings of the IEEE 7th International Symposium on Service Oriented System Engineering (SOSE)*, pp. 280-5.
14. Bianchini D., De Antonellis V., and Melchiori M. (2010). A recommendation system for semantic mashup design. In *Proceedings of the Workshop on Database and Expert Systems Applications (DEXA)*, pp. 159-63.
15. Talantikite H. N., Aissani D., and Boudjlida N. (2009). Semantic annotations for web services discovery and composition. In *Computer Standards & Interfaces*. 31, 6, 1108-17.
16. Maleshkova M., Kopecký J., and Pedrinaci C. (2009). Adapting SAWSDL for Semantic Annotations of RESTful Services. In *On the Move to Meaningful Internet Systems: OTM 2009 Workshops*. Meersman R., Herrero P., and Dillon T. (ed.), Vol. 5872, pp. 917-26. Springer Berlin Heidelberg.