DORIS: Discovering Ontological Relations in Services

Maria Koutraki
University of Versailles
France
kom@prism.uvsq.fr

Dan Vodislav
University of Cergy-Pontoise
France
dan.vodislav@u-cergy.fr

Nicoleta Preda
University of Versailles
France
preda@prism.uvsq.fr

ABSTRACT

In this paper we demonstrate DORIS, a system that automatically discovers the schema of a given Web service, and translates it into the schema of a given knowledge base (KB). More precisely, the system abstracts the Web service as a view with binding patterns [5], expressed according to the schema of the KB. In addition, it provides functions that transform the results of Web service calls into the schema of the KB. Users can interact with the system, align different real Web services to the KB, discover dependencies, and execute Web service calls.

1. INTRODUCTION

Recently, more and more data providers, including *Amazon*, *musicbrainz*, *IMDB*, *geonames*, *Google*, and *Twitter* choose to export their data through Web services. Today, these services cover a large variety of domains: books, music, movies, geographic databases, transportation networks, social media, and even personal data.

The wealth of data exported by Web services is an opportunity for the development of new intelligent applications. For instance, it is possible to develop an application that computes vacation plans. This application could check the user's calendar through a Web service from Google Calendar, find books about the destination from the Web service of Amazon, gather travel recommendations from a social Web service such as *Twitter*'s, and retrieve ticket offers from a travel Web service. For this purpose, the application would need to join data across different services. Unfortunately, each Web service typically uses its own schema. This poses a major challenge for the interoperability of the data.

Data Integration Model. A Web service is essentially a parametrized query over a remote source. The results of the calls are usually encoded in XML or JSON. They conform to a schema that is local to the Web service. In order to make the service interoperable with other services, the state of art solution [5] is to assume a common global schema. Then, the Web services are represented as parametrized conjunctive queries (i.e., views with binding patterns) over this global schema. Consider for example a Web service that takes as input the name of an author and returns as result the birthdate, gender, and books with publishing date for this author. Now, assume that the global schema contains relations such as *created* (for the author of a book) or *hasName* (relation between an entity and its name). Then the Web service can be expressed in this global schema as the parametrized query of Figure 1. The adornment of the head indicates which arguments of the head atom are input and which are output parameters. Here, l_1 is the only input.

The results of a Web service are typically provided in XML or JSON. Figure 2 (bottom) shows the result of a call to the service with Dario Fo as input. The XML tree contains two books for this author. Since the application expects the results to be answers to the parametrized query of Figure 1, a *transformation function* is needed to translate the call results into tuples. Figure 2 (top) shows the result of this translation. This way, applications can reason about Web services in terms of the views defined for them. This has the advantage of separating the problem of orchestrating Web services (according to the logic of the application) from the problem of transforming the data. While the first problem has attracted a lot of attention lately, the work on defining the schema mapping and designing the transformation function is often done manually. This approach, however, does not scale to the large number of Web services.

Problem Description. The central challenge is: *Given a Web service, how can we automatically compute its view definition and the transformation function?* Our idea is to build upon existing knowledge bases (KBs). These provide not only a suitable global schema, but also facts in this schema. These facts can be used to probe Web services, to guess the definition of the service in terms of the global schema, and to create a view for the service.

To use a service, we must know the type of the values it accepts as input (*input domain*) e.g. authors' names. Hence, an additional challenge is to *automatically detect the input domain of a given Web service.* The standard approach is to probe the Web service with nonsense data, in order to trigger an error message. Then, the service is probed again with input values from different classes (e.g. authors, books, etc.), and the results are compared to the error message. If we get something else than the error message, then we assume we found the input domain. Furthermore, some Web services expect inputs that are local to the source. For example, if a IMDB service expects as input IMDB ids of movies. It is difficult to discover this by probing. Our idea is to use

$$getBooksByAuthor^{iooooo}(l_1, l_2, l_3, l_4, l_5, l_6) \leftarrow hasName(x, l_1), birthdate(x, l_2), deathdate(x, l_3), gender(x, y), label(y, l_4)$$
$$created(x, z), hasName(z, l_5), date(z, l_6)$$

Figure 1: View with binding patterns for the Web service getBooksByAuthor.

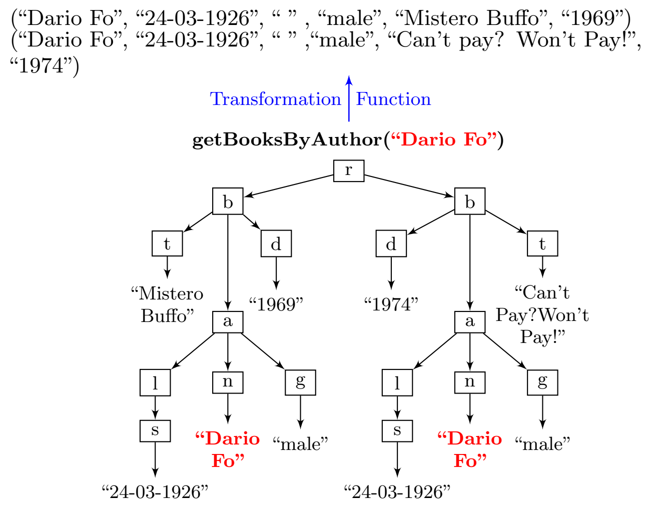
the results of one Web service in order to discover the input domain of another one.

Our approach is fully implemented in a system called DORIS. The system can automatically discover the schema of a Web service and build the transformation function into a global schema. Also, it automatically detects local input domains.

2. PRELIMINARIES

Call Results. A Web service API provides several related *Web service operations* (WS). A WS is called by accessing a parameterized URL and by retrieving the result document from the server. The document is typically in XML or JSON. For uniformity, we assume that call results in JSON are transformed to XML. This can be done with standard tools. In line with the XML standard, we represent an XML document as a rooted, ordered, labeled tree. We call *text nodes* the nodes that represent XML attribute values or contents of XML elements. Figure 2 shows a call result. Text nodes are shown in quotation marks.

RDF Knowledge Bases. RDF is the standard of knowledge representation on the Semantic Web. The RDF model assumes fixed global sets of entities, literals and binary relations. An *entity* is an identifier of a real-world object or an abstract concept. A *literal* is a string, a date, or a number. A *relation* holds between two entities or between an entity and a literal. An element of a relation is called *fact*, and a *knowledge base* (KB) is a collection of facts. Usually, a KB is depicted as a labeled directed graph, where the nodes are entities or literals and the edges denote facts. In Figure 3, the entities are framed and the literals are shown in quotation marks.



(“Dario Fo”, “24-03-1926”, “ ” , “male”, “Mistero Buffo”, “1969”)
(“Dario Fo”, “24-03-1926”, “ ” ,“male”, “Can’t pay? Won’t Pay!”, “1974”)
Transformation
Function
getBooksByAuthor(“Dario Fo”)
r
b
b
t
d
d
t
“Mistero Buffo”
“1969”
“1974”
“Can’t Pay?Won’t Pay!”
a
a
l
n
g
l
n
g
s
“Dario Fo”
“male”
s
“Dario Fo”
“male”
“24-03-1926”
“24-03-1926”

Figure 2: The transformation of a call result.

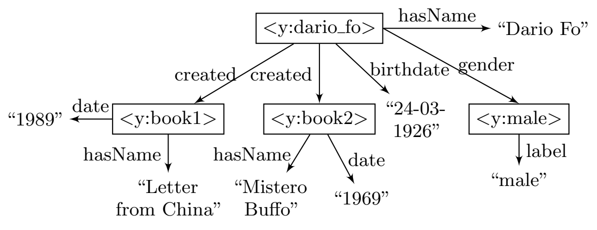


Figure 3: KB fragment.

3. ASSUMPTIONS

After manually inspecting more than a hundred WSs, we have made several observations, which we will use as assumptions for our approach:

I. REST Web services. We assume that the WS uses the REST architecture, and that the user provides the URL of the WS, as well as the position of its inputs. Most WS nowadays use REST[1], and examples of calls are usually available in the documentations.

II. The KB and the WS overlap. Our approach requires a KB whose entities overlap with the entities of the WS. For example, we assume that if we call the WS with an author from the KB, we get books that are also in the KB.

III. Entities correspond to XML nodes. Whenever an XML tree contains an entity, this entity is usually represented as a structured node with sub-nodes. Furthermore, two entities of the same class are represented by two different nodes with the same label. For example, in Figure 2, every book is represented by a subtree that is rooted at a b-node, and every author is rooted at an a-node.

[1]`http://www.programmableweb.com/`

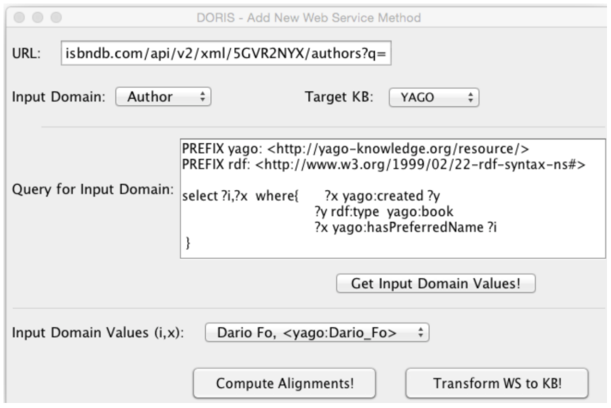
IV. The call result is connected to the input. We expect that the entities contained in the call result appear in the KB and are connected to the input entity by paths. We observed that these paths do not exceed a length of 4.

V. A relation corresponds to a linear path query. For instance, we can map the path */r/b(x,y)* in the XML result in Figure 2 to the relation *created(x,y)* that the KB defines for authors and their books. To make this correspondence explicit, we will use the same variable to denote both the node in the XML result and the corresponding entity in the KB. In the example, we will use x to denote the author node in the XML tree and the author entity in the KB.

VI. Text nodes map to literals in the KB. We have noticed that text nodes typically correspond to literals in the KB. Using a simple string similarity function, we can map a text node to an equivalent literal in the KB in the majority of cases.

VII. Single input domain. For now, our approach is limited to WSs with one input. We have found that the majority of real WSs are covered by this assumption. This is because typically only one input is mandatory.

4. DORIS FUNCTIONALITIES



URL: isbndb.com/api/v2/xml/5GVR2NYX/authors?q=

Input Domain: Author

Target KB: YAGO

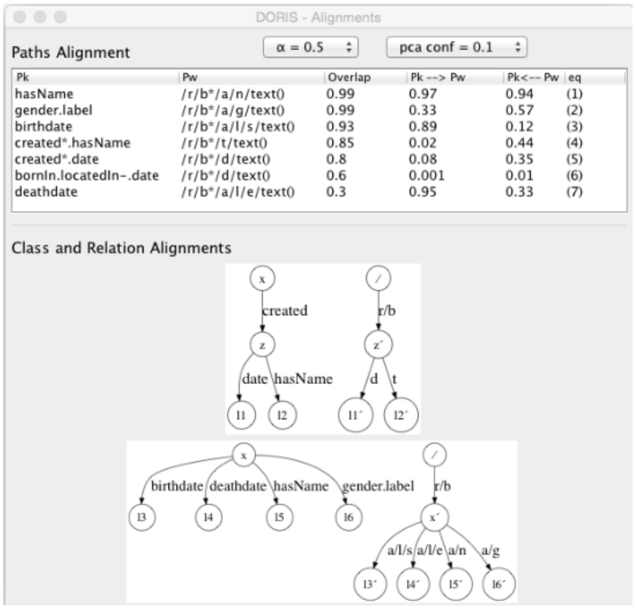
Query for Input Domain:

```
PREFIX yago: <http://yago-knowledge.org/resource/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

select ?i,?x  where{     ?x yago:created ?y
                        ?y rdf:type  yago:book
                        ?x yago:hasPreferredName ?i
 }
```

Input Domain Values (i,x): Dario Fo, <yago:Dario_Fo>

Figure 4: Adding a Web Service.



Paths Alignment α = 0.5 pca conf = 0.1

Pk	Pw	Overlap	Pk --> Pw	Pk<-- Pw	eq
hasName	/r/b*/a/n/text()	0.99	0.97	0.94	(1)
gender.label	/r/b*/a/g/text()	0.99	0.33	0.57	(2)
birthdate	/r/b*/a/l/s/text()	0.93	0.89	0.12	(3)
created*.hasName	/r/b*/t/text()	0.85	0.02	0.44	(4)
created*.date	/r/b*/d/text()	0.8	0.08	0.35	(5)
bornIn.locatedIn-.date	/r/b*/d/text()	0.6	0.001	0.01	(6)
deathdate	/r/b*/a/l/e/text()	0.3	0.95	0.33	(7)

Figure 5: Paths and Class/Relation Alignments.

DORIS is a software library for automatic discovery of WSs schema. We describe here how DORIS discovers the alignments to a given KB schema and how it computes the transformation function for the WS results. We first consider a WS for which the input domain is explicitly given. Then (paragraph 4) will illustrate how local input domains can be discovered automatically.

Path alignment. Given an KB, a WS URL, and a number of instances form the input domain of the WS, our system first probes the WS by calling it until 100 valid samples are obtained. This relatively small number of calls is sufficient to perform the automated schema mapping. The key challenge is now to identify in the call results the nodes that represent entities (as per Assumption III). This is not an easy task, because the nodes in the XML tree usually do not have meaningful labels. The entity node for Mistero Buffo in Figure 2, e.g., is not called "Mistero Buffo", but "b". Therefore, we resort to the following approach: We consider a linear path from the root node of the XML document to a text node (in the example: *r/b/t/text()*). Then, we match the text node to a literal in the KB (as per Assumption VI). This literal will be connected in the KB to the input entity (Assumption IV). In the example, the KB literal "Mistero Buffo" is indeed connected to the input entity Dario Fo, by the path *created/hasName*. Therefore, we align the path of the XML document to the path in the KB. This yields the alignments shown in Figure 5 at the top.

Our approach considers two metrics for measuring the confidence of the alignments: The *overlap* measures the proportion of samples for which the two paths have at least one literal in common. Generally, this metric produces good results, although spurious matches may introduce false positives (such as (6)) and false negatives due to incomplete relations (such as (7)). The other metric is adapted from [4], a rule mining approach for KBs. Let $p_K(x, y)$ and $p_W(x, y)$ be the two linear path queries over the KB and over the WS result, respectively, where x denotes the input entity (and the root node in the WS result), and y denotes a literal (and the text node in the WS result). We can say that the relation denoted by p_K is subsumed by the relation denoted by p_W, if $\forall\ x,\ p_K(x, y) \rightarrow p_W(x, y)$. The PCA confidence [4] was developed to measure the degree of truth of this implication, if the data source of the succedent of this rule is incomplete. Since both the WS and the KB can be incomplete in our scenario, we construct the implication both ways, and measure the degree of truth by the PCA confidence. The results for both the overlapping metric and the PCA confidence are shown in Figure 5. We remark that the PCA confidence has a very good precision for functional relations such as hasName (1), birthday (3), or deathdate (7). Also, incomplete relations such as deathdate (7) rank high according to this metric. Due to a pagination mechanism, a WS result may contain only a subset of the result entities (books in our case). This explains the good values in (4) and (5).

For the next step, we only consider alignments that have an overlapping confidence higher than a threshold $\alpha = 0.5$, and a PCA confidence greater than 0.1 for at least one of the implications. The user can play with different other thresholds. This filters out spurious mappings such as (6).

Class and Relation Alignment. We have now aligned XML paths to text nodes with KB paths to literals. There can be one or several entity nodes in the XML path, and these correspond to entities in the KB path. In the example, the XML path */r/b/t/text()* walks through the entity node */r/b* (Mistero Buffo). In the corresponding KB path *created/hasName*, Mistero Buffo comes midway in the path, directly after the *created* edge. The challenge is now to find out which positions in the XML path correspond to which positions in the KB path.

If the tree encodes several entities that are in the same relationship with the input entity, these are represented by sibling nodes labeled with the same name (per Assumption III). Furthermore, the sub-trees rooted at those nodes should have an isomorphic structure. We call such nodes *branching points*, and we annotate them with $*$ in the XML path. In the example, we get */r/b*/t/text()*. For the KB path, a branching point is a relation that is not functional[2] and that does not lead to a literal. In the example, we get *created*.hasName*. The clou of our algorithm is the insight that if we have a path of functional relations on the KB side, then

[2]To deal with noisy KBs, we consider here that a relation is functional if its functionality [8] is between 0.9 and 1.

Table 1: **Average performance of alignments.**

	Precision		Recall	
	Classes	Relations	Classes	Relations
YAGO	0.92	0.91	0.96	0.93
DBPedia	0.89	0.88	0.98	0.95
BNF	1	1	1	1

this path has to be aligned to an XML (sub-)path with no branching points. This means that when we aim to align an XML path and a KB path, the right-most branching point of the XML path must be aligned to the right-most branching point in the KB path. Thus, we deduce that the nodes selected by the XML path */r/b* correspond to the path *created*. Thereby, we have discovered the KB relation *created* in the XML result.

The mapping can be biased to the sample of inputs. In our example, the authors and the books (the domain and the range of *created*) map to the same positions in the XML. Our system detects such inconsistencies and re-probes the WS with a book from the KB that has several authors.

Once we have mapped positions in the XML path to entities in the KB, we can annotate the nodes in the XML path with the classes of which the entities are instances. This gives us a complete mapping of the XML document to the KB: paths are mapped to KB relations, and nodes are mapped to KB classes. Our demo shows the alignments at the bottom of Figure 5.

View & Transformation Function. After the alignment process DORIS is able to construct the view with binding patterns for the WS. The body of the view definition in Figure 1 is exactly the conjunction of the atoms that have been discovered by the alignment algorithm in Figure 5.

The transformation function takes the form of an XSLT script. The script is generated automatically based on the alignments that we discovered. Basically, every entity node gives rise to a *xsl:for-each* block.

New Input Domains. DORIS can also automatically discover the input domain of a WS in case that the input values correspond to outputs of other WSs of the same API, which is often the case. This is practically useful when the input values are local to the API (e.g. ids). In order to accomplish that we use an already aligned WS (W_1) of the same API. Our goal is to see whether the input domain of W_2 (new WS) can be defined in terms of the outputs of W_1. We cannot simply try all outputs of W_1 as inputs of W_2, because W_2 may simply always reply with some results, even for nonsensical input. Our key observation is that if we hit the right input domain, then W_2 should return properties of the input entity that we used to probe W_1 (Assumption IV). We can detect this by checking back with the KB. We actually compute the path alignment described above. This allows us to determine which nodes in the result of W_1 are correct inputs for W_2 and so provide input samples for it.

Evaluation. DORIS is evaluated on 50 real WSs, covering the domains of music, books, and geolocations. We report the average precision and the recall of our class and relation alignment in Table 1, for 3 KBs. The results for BNF(KB of books from the French National Library) are so good because it covers only books, is virtually noise-free, and has a high coverage. The results on the other KBs are more than respectable. All evaluation results and details can be found on our Web site `http://oasis.prism.uvsq.fr/`.

5. DEMONSTRATION

At the demonstration the users will be able to interact with DORIS's user interface in order to discover ontological relations in a given WS and to transform it in terms of a global KB. Moreover, they will be able to discover the local input domains for WSs that accept ids. A list of WS URLs will be provided in order to use the system, however users are free to test WS of their own choice. DBpedia [1], YAGO [9] and BNF (bnf.fr) are the available target KBs in this demonstration.

Setup. DORIS is implemented in Java (JVM 1.7). We use standard Java libraries to parse the XML and JSON call results, Jena 2.11 library to store the KBs, and the Jena SPARQL functionality to query them.

Scenario 1: Align a new WS. In the first scenario the user wants to align a WS which provides the books of a given author. First of all, the URI of the WS is given to the system as input (Figure 4). The user chooses the Target KB (YAGO in this case) and also defines the input domain which is *Author*. Since the input domain is known in this scenario, a prefabricated conjunctive query is submitted over the KB to select the input values. The query selects pairs of the form (i, x), where x is the YAGO entity for the author, and i is the label of that entity (which will be used for querying the Web service). Then the user will have the results of the alignments (Figure 5). At the top of the figure are presented the results of paths alignment. In this scenario the thresholds are set, but users can select other values from the drop-down menus. In the bottom there is a graphical representation of the class and relation alignment.

After the alignment, the user can visualize the WS results in terms of the KB using the view and the transformation function that are produced by pushing the button "Transform WS to KB!" (Figure 4) .

Scenario 2: Discover a new Input Domain. In this scenario the user wants to align a WS that has a local input domain. He selects the already aligned WS of the previous scenario (W_1) and a new one which returns information about a book with a specific id (W_2). The user will be informed about W_1 output nodes that are possible inputs values for W_2. The candidates will be ranked based on the number of common properties between W_1 and W_2.

User Interaction. Apart from the two proposed scenarios the users will be free to interact with the system and align other WSs or play with different threshold values. We provide a list of WS URIs from several domains. Nevertheless, if the audience is familiar with other WSs is welcome to align them. More details and screen shots about DORIS can be found in our Web page.

6. RELATED WORK

WS discovery. Previous works [3, 7] focus on annotating a WS with concepts. However, they rely on the existence of schema descriptions – unavailable for REST services.

Schema Matching. Numerous approaches are concerned with mapping an existing schema to another schema [2]. REST WSs however, do not publish any specifications for their schemas. Closer to our approach are instance-based solutions [8] developed for aligning two KBs. However, in our case, we do not have two KBs, but only one KB and an XML tree. The XML tree does not give away which of its nodes correspond to entities and which nodes do not.

Query discovery. Query discovery is concerned with the actual translation of data from one source to another. The state of art solution [6] exploits schema constraints. These, however, are not available for WSs. Closest to our work, [10] addresses the problem of transforming XML data to RDF triples. However, their model is *global as view*. Hence, they cannot map the WS to a global schema.

7. REFERENCES

[1] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives. DBpedia: A nucleus for a Web of open data. In *ISWC*, 2007.

[2] P. A. Bernstein, J. Madhavan, and E. Rahm. Generic schema matching, ten years later. In *VLDB*, 2011.

[3] X. Dong, A. Y. Halevy, J. Madhavan, et al. Similarity search for web services. In *VLDB*, 2004.

[4] L. Galárraga, C. Teflioudi, K. Hose, and F. M. Suchanek. AMIE: association rule mining under incomplete evidence. In *WWW*, 2013.

[5] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 2001.

[6] R. J. Miller, L. M. Haas, and M. A. Hernández. Schema mapping as query discovery. In *VLDB*, 2000.

[7] S. Quarteroni, M. Brambilla, and S. Ceri. A bottom-up, knowledge-aware approach to integrating and querying web data services. *TWEB*, 7(4):19, 2013.

[8] F. M. Suchanek, S. Abiteboul, and P. Senellart. Paris: Probabilistic alignment of relations, instances, and schema. *PVLDB*, 5(3), 2011.

[9] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *WWW*, 2007.

[10] H. Xiao and I. F. Cruz. Integrating and exchanging xml data using ontologies. *J. Data Semantics*, 2006.