OntoGenesis: An Architecture for Automatic Semantic Enhancement of Data Services

Bruno C. N. Oliveira[1], Alexis Huf[1], Ivan Salvadori[1], and Frank Siqueira[1]

[1]**Graduate Program in Computer Science (PPGCC), Department of Informatics and Statistics (INE), Federal University of Santa Catarina (UFSC), Florianópolis/SC, Brazil**

Corresponding author: Bruno C. N. Oliveira[1]
Email address: brunocn.oliveira@gmail.com

DOI: 10.1108/IJWIS-04-2018-0020
Published in: International Journal of Web Information Systems Volume 15, Issue 1 pp. 2-27, ISSN: 1744-0084

ABSTRACT

Purpose – This paper describes a software architecture that automatically adds semantic capabilities to data services. The proposed architecture, called OntoGenesis, is able to semantically enrich data services, so that they can dynamically provide both semantic descriptions and data representations.
Design/methodology/approach – The enrichment approach is designed to intercept the requests from data services. Therefore, a domain ontology is constructed and evolved in accordance with the syntactic representations provided by such services in order to define the data concepts. In addition, a property matching mechanism is proposed to exploit the potential data intersection observed in data service representations and external data sources so as to enhance the domain ontology with new equivalences triples. Finally, the enrichment approach is capable of deriving on demand a semantic description and data representations that link to the domain ontology concepts.
Findings – Experiments were performed using real-world datasets, such as DBpedia, GeoNames as well as open government data. The obtained results show the applicability of the proposed architecture and that it can boost the development of semantic data services. Moreover, the matching approach achieved better performance when compared with other existing approaches found in the literature.
Research limitations/implications – This work only considers services designed as data providers, i.e., services that provide an interface for accessing data sources. In addition, our approach assumes that both data services and external sources – used to enhance the domain ontology – have some potential of data intersection. Such assumption only requires that services and external sources share particular property values.
Originality/value – Unlike most of the approaches found in the literature, the architecture proposed in this paper is meant to semantically enrich data services in such way that human intervention is minimal. Furthermore, an automata-based index is also presented as a novel method that significantly improves the performance of the property matching mechanism.
Keywords – Data Service, Semantic Web Service, Semantic Web, Ontology Alignment, Ontology Construction, Property Matching, Service Description, Semantic Representation
Paper type – Research paper

DOI: 10.1108/IJWIS-04-2018-0020
Published in: International Journal of Web Information Systems Volume 15, Issue 1 pp. 2-27, ISSN: 1744-0084

1 INTRODUCTION

The number of Web services, scattered in public and private networks, that are meant to provide data already stored in some data source has been constantly increasing. Such Web services are also referred to as data services and are useful for providing access interfaces to data sources that cannot be completely disclosed (Carey et al., 2012). Additionally, such services can also employ Semantic Web technologies (Berners-Lee et al., 2001) in order to provide data in a sophisticated machine-readable format and, therefore, be easily reused by third parties and integrated to complex Web applications. Several researchers have been discussing the benefits of employing Semantic Web services; for instance, enhance data interoperability (Salvadori et al., 2017) and assist in automating tasks such as service discovery, selection, composition, etc. (McIlraith et al., 2001; Sycara et al., 2003).

However, the implementation and adoption of Semantic data services in real-world applications are limited mainly due to the format of stored data. Such data is usually stored in a syntactic form, i.e., only the structure of the data is specified, but not the semantics. In addition, various major challenges contribute to this lack of adoption. Some common issues include the time and effort demanded in the construction of domain ontologies and the semantic annotation of data services. These are complex tasks that require domain expert knowledge. Concerns over agreement in semantic modeling must also be taken into account. In practice, assuming that data provided by services will always be defined by a universal ontology is not realistic (Fellah et al., 2016). As a result, given the existing heterogeneous ontologies describing the same real-world entity, developing and integrating Semantic data services become challenging tasks.

Since building an ontology for a data source is a difficult and time consuming task by its nature, some support tools (Cimiano and Völker, 2005; Salem and AbdelRahman, 2010; Nguyen and Lu, 2016) have been developed to help users in the ontology construction process. These tools, though, often require availability of data dumps for generating a domain ontology. This limitation hinders the adoption of such tools in Service-Oriented Architecture (SOA), in which availability of data depends on the service interface. Extensional ontology matching techniques (Euzenat and Shvaiko, 2007) attempt to solve the problem of ontology heterogeneity using instance data to infer equivalences at the schema level. On the other hand, data services are susceptible to changes, and ontologies that describe data are required to evolve in parallel, otherwise they become inconsistent. Moreover, in order to perform extensional matching, ontology matchers generally assume that the two ontologies that are to be matched have already been created and associated with a large set of instances. This becomes a challenge when such data is partially available (due to the service interface) and when instance data of both ontologies are unrelated. Yao et al. (2014) proposed a mechanism to create a unified ontology based on a set of JSON documents provided by a Web service. Nevertheless, the generated ontology does not leverage semantic concepts defined by external sources, aiming to reuse existing concepts and minimize heterogeneity issues. In addition, previous works have proposed approaches to enrich services with semantics, most of them focusing on service descriptions (Bravo et al., 2014; Luo et al., 2016). In contrast, few proposals address the enrichment of data provided by services. For instance, the method proposed by Salvadori et al. (2017) aims to enrich representations of data-based microservices with `owl:sameAs` and `rdfs:seeAlso` links. A framework aimed to identify alignments between heterogeneous ontologies is also proposed. A drawback of such approach is that microservices must already employ a domain ontology and provide semantic data.

This work proposes an architecture, called OntoGenesis, aimed to generate domain ontologies and automatically enriching data services with semantic concepts defined in such ontologies. The benefits

of our proposal are two-fold. Firstly, OntoGenesis provides a way to gradually build domain ontologies from syntactic representations provided by data services and to reuse well-known concepts by identifying data intersection with external sources. Secondly, the architecture enables the migration of syntactically defined data services toward Semantic data services. Therefore, legacy data services are able to serve both semantic description and representations to their consumers. This paper is an extended version of (Oliveira et al., 2017), that includes the generation of semantic service descriptions, contains an expanded exposition of the mechanism responsible for high performance results and provides further evaluations of the architecture. Results show that our approach can achieve suitable F-Measure scores and reasonable performance in terms of processing time and memory consumption.

The remainder of this paper is organized as follows: Section 2 summarizes the main concepts that are required for understanding this work. Section 3 presents the semantic enrichment approach for data services, while the OntoGenesis architecture and its main components are detailed in Section 4. Section 5 presents the evaluation scenario and methodology along with the obtained results. Related research efforts found in the literature are discussed in Section 6. Finally, Section 7 draws the conclusions and presents some perspectives for future work on this research subject.

2 BACKGROUND

2.1 Data Services

A Web data source is usually organized in accordance with a data model or schema, which can be unknown to consumers who access data through a Web interface. In general, these data sources are wrapped as Web Services, i.e., they expose data through one or more Web data services (hereafter, called data services or just services) (Bianchini et al., 2015).

Data services aim to provide a Web interface for handling data in the sense that they act as data providers, allowing abstraction of access to data sources. Bianchini et al. (2015) define a data service s as an operation, method or query to access data from a given data source. These services are modeled as a set of: i) service inputs s_i, which consist in parameters that are needed to invoke the service and access data; and ii) outputs s_o, representing data that is accessed through service s. Data access is usually resource (i.e., entity) oriented, that is, a consumer requests a resource to a service, providing s_i, and receives a representation of such resource, s_o. The output representation can be seen as a snapshot of the state of a resource at a given time, available in different formats, such as XML, JSON, HTML, etc. It is worth noting that data services are not tied to any particular technology; they can be implemented, for example, using SOAP or REST technology stacks.

A crucial factor that hinders service integration occurs in the conceptual level, since data services often employ different terminologies (for the attribute names, for instance), even though providing information about the same real-world concept. To overcome this issue, Semantic Web technologies (Berners-Lee et al., 2001) may be applied to data services, resulting in Semantic data services. Such approach is aimed at providing machine-readable descriptions of data, therefore facilitating its integration and reuse. According to McIlraith et al. (2001), Semantic Web services should expose information about available services, their properties, execution interfaces, pre- and post-conditions, in a sophisticated machine-readable format. Regarding Semantic data services, managed resources, as well as their properties and relationships, should also be semantically enriched. This means that, besides the service description, representations provided by such services should be associated with semantic concepts.

On the other hand, Lira et al. (2014) consider Semantic data services as access points to data that is natively stored as RDF (Resource Description Framework) triples in a particular data source. In contrast,

we argue that Semantic data services can provide both semantic description and semantic representations, regardless of how data is stored and maintained. Thus, enriching data service representations means to provide semantic data taking advantage of RDF formats, such as JSON-LD.

2.2 Ontology Construction and Matching

Ontologies aim to provide common vocabularies for different domains of knowledge. According to Guarino (1997), ontologies can be classified based on two dimensions: level of detail and dependency level. At the first level, a very detailed ontology aims to specify the meaning of a vocabulary in order to establish a consensus about certain concepts, whilst in a scenario where users already have a consensus, a simpler and less detailed ontology (operational) can be developed and shared, taking into account specific operations of inference.

At the dependency level, ontologies can be classified in four types according to their generality level: top-level, domain, task and application. Domain ontologies are the most common and are employed in many application fields (Guizzardi, 2007). Since the use of ontologies in the present research is aimed at the integration of data and the enrichment of services with semantic concepts related to a particular application domain, the approach introduced in this work aims to build operational domain ontologies. Therefore, most of the subsequent discussions apply to this type of ontology, although it may be extended, with some restrictions, to other types.

Most effort involved in semantically enriching services is in the construction of ontologies as well as in adapting and evolving them in accordance with demanded changes. This is due to the naturally time-consuming task involved in the domain ontology development. In general, ontology engineers and domain experts manually develop domain ontologies to provide a domain-specific model suitable for describing the semantics of a service. Ontology Learning (OL) (Maedche and Staab, 2001) is a topic interested in automating and thereby facilitating the creation of ontologies by domain experts and ontology engineers. In this way, ontological elements, such as concepts and relations, are extracted from different resources. Some researchers, such as Alfaries (2010), examine existing techniques and tools available for (semi-)automatically learning domain ontologies from Web service resources.

Although OL offers mechanisms to automate the ontology construction process, it is essential to reuse semantic concepts from existing ontologies. This allows further integrations and logic-based reasoning to be performed by software agents. In this sense, ontology matching techniques emerge to solve ontology heterogeneity issues by identifying alignments – usually expressed by OWL equivalences – between different ontologies. To yield an alignment, the following inputs may be used in addition to the ontologies: i) a known alignment A_0, ii) matching parameters (such as weights or thresholds), and iii) external resources. The alignment contains a set of correspondences between classes and properties of such ontologies. Each correspondence denotes a relation of equivalence, generalization or disjointness between two elements of O_1 and O_2 (Pavel and Euzenat, 2013).

Four basic techniques for ontology matching are identified by Euzenat and Shvaiko (2007). The name-based technique considers only the name of ontology elements (e.g., labels of properties and classes). The structured-based technique considers the structure of ontology elements (e.g., subclass relations). Semantic-based techniques, on the other hand, usually leverage reasoner tasks in order to infer equivalences between different ontologies. Lastly, the extensional techniques use the ontology instances to identify similar individuals and thereby match classes and properties.

Extensional matching techniques can be adapted to SOA in such way that information provided by data services can be added as instances of the service ontology. Thus, besides the construction of domain ontologies for data services, this work is concerned with the extensional matching technique to find out

alignments between service ontologies and external ontologies, specifically focusing on their properties. Such alignments are often expressed by the `owl:equivalentProperty` axiom.

3 SEMANTIC ENHANCEMENT OF DATA SERVICES

This work focuses on dynamically enhancing data services with semantic features, so as they can provide both semantic description and semantic representations. This goal is achieved by associating semantic concepts defined in domain ontologies with service representations provided by such service, in order to provide Linked Data. To this end, it is necessary to include a semantic adapter in the data service aiming to perform such associations and create the new semantic description and representations.

Figure 1 presents an overview of the workflow for the data service semantic enrichment. Firstly, a consumer sends a request to a data service. After processing the request, the service sends to an Enricher a syntactic representation of the response (serialized, for instance, in XML or JSON). The Enricher extracts all elements from such representation and constructs a domain ontology for the service, including classes, data type and object properties, as well as identified equivalent property links to external concepts. Such equivalences can be found out by the Enricher through external data sources that link to existing semantic concepts.

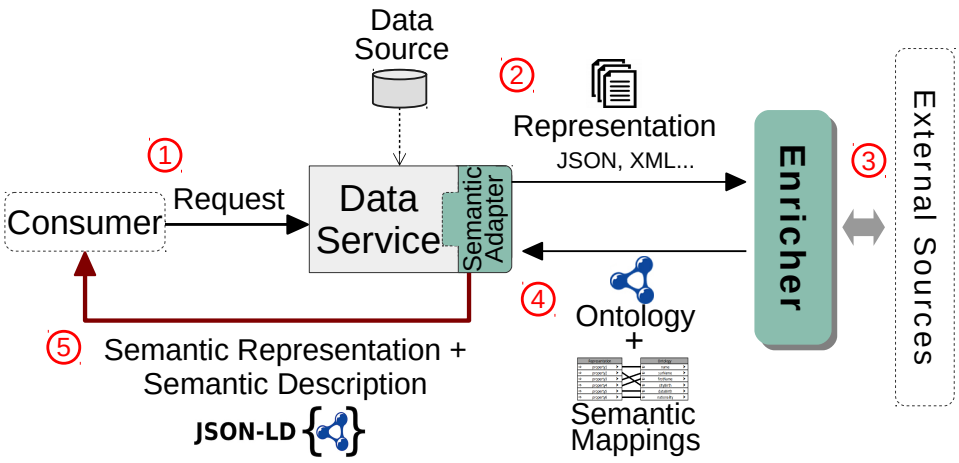

Figure 1. Schema of Semantic Enrichment of a Data Service.

The Enricher should output the domain ontology along with semantic associations, known as Semantic Mappings (SM), between the syntactic attributes from service representations and the new ontology concepts. A semantic mapping can be defined as a 3-tuple $SM = \{a, c, t\}$, where a is the attribute of a representation, c is the concept represented in the ontology, and t is its type (a class or an object data/property). As an example, suppose that it is created a datatype property c, where $c =$ `"http://data-service/ontology#name"`, for the attribute $a =$ `"name"` of a certain representation. The semantic mapping generated shall be $SM = \{$`"name"`, `"http://data-service/ontology#name"`, `"Datatype property"`$\}$. Semantic Mappings are useful for generating semantic representations as well as a semantic description for the data service. In this way, according to the set of semantic mappings yielded by the Enricher, the syntactic representation is automatically converted to JSON-LD by a semantic adapter, which in turn is sent back to the consumer. Likewise, a semantic description for the service is generated according to the input and output parameters, and it is published though an endpoint so that consumers can obtain it.

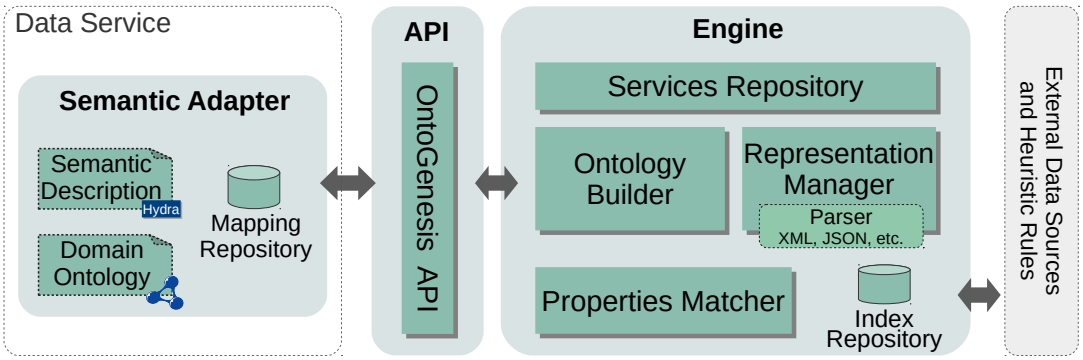

Figure 2. OntoGenesis Architecture.

As shown in Figure 1, the Semantic Adapter is a component attached to a data service, which intercepts the responses and returns JSON-LD to consumers. Such JSON-LD is used to serialize not only the representations provided by the data service, but also the semantic description, which defines the entities a service is able to manage and how to obtain them. The Semantic Adapter can also be seen as a connector responsible for the communication between the data service and the Enricher. The next section presents a detailed discussion concerning the Semantic Adapter as well as the other components designed for the Enricher.

4 THE ONTOGENESIS ARCHITECTURE

This section presents OntoGenesis, an architecture for semantically enriching data services. The OntoGenesis architecture is divided in three major components, as depicted in Figure 2. The first one, the OntoGenesis Engine, is responsible for constructing an ontology for the data service and for yielding semantic mappings in accordance with the syntactic attributes of the data service representations. The second component, called OntoGenesis API, is a Web API that provides a communication interface for accessing functionalities provided by the OntoGenesis Engine. Finally, the Semantic Adapter is a lightweight library for accessing the OntoGenesis API and for assisting the data service in providing its semantic description and semantic representations. It is important to notice that the Enricher depicted in Figure 1 comprises both the API and the Engine components shown in Figure 2.

4.1 OntoGenesis Engine

As shown in Figure 2, the OntoGenesis Engine comprises five main components: *Services Repository*, *Representation Manager*, *Ontology Builder*, *Index Repository* and *Properties Matcher*.

The *Services Repository* manages information about the registered data services. The information stored includes the service name, its URI address, and the semantic features created by OntoGenesis (i.e, the domain ontology and the semantic mappings). Furthermore, this component assists in the operation of the other components providing the necessary information about the data services registered in OntoGenesis.

The *Representation Manager* aims to extract the elements from a data service representation, such as attributes and their values, useful for the ontology construction process. To this end, it provides a common abstraction to any data format, so that specific parsers can be seamlessly encapsulated in this component, allowing the OntoGenesis Engine to properly deal with different data formats, such as JSON, XML, CSV, among others. Since the results of the Representation Manager are used by other components, format-specific details are discarded using the common abstraction. We employ an object-inspired abstraction, in

which a representation consists of a set of objects, each consisting of a map from attribute names to a set of attribute values. As for attribute values, they are divided into object values (creating a tree structure) and primitive values such as numbers, strings and booleans. JSON arrays are considered as multiple values for the same attribute name, i.e., the array ordering is discarded.

The *Ontology Builder* analyzes the syntactic elements extracted by the Representation Manager to construct a domain ontology for the service producing the data. If a domain ontology has already been constructed from a previous representation sent by the service, the Ontology Builder updates the domain ontology with the new identified elements. Therefore, the ontology evolves as new representations are provided to OntoGenesis by the data service. The ontology construction process is further detailed in section 4.1.1.

The *Index Repository* is a sub-component of the Engine that stores, in a key-value database, indexes of literal data gathered from external data sources and data services for which OntoGenesis construct ontologies. Considering that both service data and external data consist in triples, both indexes use property as key and object as values. Only triples whose object is a literal have the lexical form of the object (i.e., discarding the datatype) included in these indexes. Despite the aforementioned similarities, service data and external data employ different index structures and procedures. The index for service data is implemented as a simple hash-based map of properties to hash-based sets of values. This implementation strategy allows for fast updates on the index. The second index, built from external data sources, is a hash-based map from properties to a Deterministic Finite Automaton (DFA). This index allows fast searches on the index for similar terms. Details concerning this index, such as the automata construction process and the similarity-based query are exposed in Section 4.1.2.

Finally, the *Properties Matcher* is an important component of OntoGenesis Engine that aims to properly match the set of terms of each property provided by a data service, with the set of terms of each property existing in external data sources. Thus, it uses the Index Repository to obtain all the terms associated to properties. In other words, it uses both indexes to perform the matching mechanism so as to figure out the overlap between the objects of properties in different datasets. The aim of the *Properties Matcher* is to find out equivalences between the constructed ontology and well-known ontologies used by external sources. Section 4.1.3 describes in detail the matching process as well as the algorithm applied to match the properties and establish equivalences between ontologies.

4.1.1 Ontology Construction

Listing 2 presents a sample of a primary OWL ontology constructed by the *Ontology Builder* based on a given JSON representation, shown in Listing 1). The values contained in this JSON represent real data of a police report, published by the Public Security Secretariat of the state of São Paulo (SSP/SP), with information regarding a person involved in the reported event (e.g., a victim, witness or perpetrator).

Attribute names of a representation are mapped to a property instance in the ontology. The type of this property is determined as follows:

1. `owl:ObjectProperty` if used exclusively with object values of a representation;
2. `owl:DatatypeProperty` if used exclusively with primitive values;
3. only `rdf:Property` otherwise.

The URI of the property instance is determined by a simple concatenation of the attribute name with the ontology prefix, which is defined in accordance with the service URI provided by the Services Repository component.

Listing 1 Sample of a JSON representation of a Police Report.

```
{
  "PoliceReport": {
     "reportID": "2015-10004-794",
     "location": "Train Station ...",
      ...
     "personInvolved": {
        "name": "CARLOS ALBERTO DOS SANTOS",
        "docID": "015****18",
        "birthDate": "12-21-1966",
        "nationality": "Brazilian",
        "placeOfBirth": "Sao Paulo-SP",
        "gender" : "Male",
         ...
    }
  }
}
```

Listing 2 Sample of an Ontology in Turtle built from the JSON Representation.

```
@prefix :      <http://service-example/ontology#> .
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix owl:   <http://www.w3.org/2002/07/owl#> .
@prefix xsd:   <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .
<http://service-example/ontology> a owl:Ontology .
:PoliceReport       a owl:Class .
:PersonInvolved     a owl:Class .
:hasPersonInvolved a owl:ObjectProperty;  rdfs:domain  :PoliceReport;
                                          rdfs:range   :PersonInvolved .
:reportID      a owl:DatatypeProperty ;   rdfs:domain  :PoliceReport;
                                          rdfs:range   xsd:string .
:location      a owl:DatatypeProperty ;   rdfs:domain  :PoliceReport;
                                          rdfs:range   xsd:string .
:name          a owl:DatatypeProperty ;   rdfs:domain  :PersonInvolved;
                                          rdfs:range   xsd:string .
:docID         a owl:DatatypeProperty ;   rdfs:domain  :PersonInvolved;
                                          rdfs:range   xsd:string .
:birthDate     a owl:DatatypeProperty ;   rdfs:domain  :PersonInvolved;
                                          rdfs:range   xsd:date .
:nationality   a owl:DatatypeProperty ;   rdfs:domain  :PersonInvolved;
                                          rdfs:range   xsd:string .
:placeOfBirth a owl:DatatypeProperty ;    rdfs:domain  :PersonInvolved;
                                          rdfs:range   xsd:string .
:gender        a owl:DatatypeProperty ;   rdfs:domain  :PersonInvolved;
                                          rdfs:range   xsd:string .
# ...
```

Classes are generated from two sources. First, any `owl:ObjectProperty` instance p originates a new class C (e.g., line 07 of Listing 2), as well as the triple $\langle p$ `rdfs:range` $C\rangle$ (line 10). Any property q extracted from object values of the attribute name corresponding to p will also take the triple $\langle q$ `rdfs:domain` $C\rangle$ (lines 11-26). The second source is the name of the endpoint from where the representation originated. Such class R generated in this manner will be the `rdfs:domain` of all properties that correspond to attribute names found in the root objects of the representations sharing the same endpoint name (e.g, line 11). The rationale for this is that, within a typical Web service, endpoints (for instance, a resource method in JAX-RS[1]) will serve entities of the same type. Furthermore, such endpoint name can be inferred by a tool automating registry and representation submission to OntoGenesis.

In parallel with the ontology construction process, the Ontology Builder also yields the Semantic Mappings (SM), as discussed in Section 3. For each ontology property p of a type t created in accordance with an attribute a of the service representation, it generates the 3-tuple $SM = \{a, p, t\}$, which is useful for converting such representation to JSON-LD.

Despite the fact that the primary ontology produced by the Ontology Builder supplies semantic

[1] JSR 339: `https://jcp.org/en/jsr/detail?id=339`.

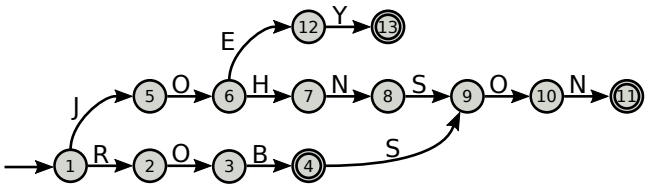

Figure 3. Sample of minimized DFA accepting some first names.

concepts related to the data provided by the service, such concepts are only known by the data service. In order to allow richer integration with other existing Semantic Web applications or services, it is necessary that the constructed ontology reuses (or aligns to) concepts defined by well-known and open ontologies/vocabularies. To this end, OntoGenesis aims to find out equivalent concepts (specifically properties) between the constructed ontology and external sources in order to expand the ontology and thereafter afford further reasoning tasks.

4.1.2 Similarity-based Index Construction and Querying

As previously described, the second type of index managed by the *Index Repository* assumes the form of $p \rightarrow M$, where p is the RDF predicate (more specifically a property observed in external data source) and M is a Deterministic Finite Automaton (DFA) (Hopcroft and Ullman, 1990). These automata are built so that the language accepted by the automata M for property p, denoted $L(M)$, is the set of all lexical forms of literal objects observed for property p in the external data sources. For example, Figure 3 shows one such DFA that accepts the strings "JOEY", "JOHNSON", "ROB" and "ROBSON". The algorithm for adding a new literal value w to M consists in attempting to recognize w, and when recognition fails, inserting the transitions and states required so that w is accepted. For example, adding the string "JOHN" to the DFA in Figure 3 only requires marking state 8 as final, while adding "ROBERT" requires adding the transitions and states for "ERT", after "ROB" has been already recognized. Once all triples in the external sources are added in this manner, the DFA M is minimized, for better performance during querying.

Queries against the external data index are not queries for $w \in L(M)$, as would be done in a normal index. Instead, what is required of this index are Similarity-Based Queries (SBQ), where given w, the answer should be whether there is an $w' \in L(M)$, sufficiently similar to w. In this paper, the similarity criteria is the Levenshtein distance with a threshold of 1 ($d_L(w,w') \leq 1$). Formally, the SBQ can be expressed as evaluating whether $L(M) \cap L(LEV_1(w)) \neq \emptyset$, where LEV_1 is an automata that accepts any string w' such that $d_L(w,w') \leq 1$ (Schulz and Mihov, 2002). Since the intersection automaton is an automaton that models concurrent execution of its component automata, the query algorithm materializes neither it nor $LEV_1(w)$, and computes them on-demand while evaluating if the intersection is empty.

The algorithm for SBQ is shown in Algorithm 1. The algorithm concurrently explores states in M and in $LEV_1(w)$, which are combined in a joint state (js). Joint states contain the state in M and encode the state in LEV_1 by the unrecognized portion of w and the number of edits applied. We denote access to these components, respectively by $js[1]$, $js[2]$ and $js[3]$. Exploration starts from the initial state of M, against the whole w string and with zero edits. Transitions in M (lines 9 – 12) and transitions in $LEV_1(w)$ (lines 13 — 15) are explored until a final state in M is reached (lines 6–8). Transitions for M are obtained from the transition function δ_M, while transitions from LEV_1 (insert, delete or replace a character) are generated by the *editStates* function. Transitions in LEV_1 change the joint state by increasing the number of edits ($js[3]$) and by changing the unrecognized string accordingly (deletions and insertions retain the same unrecognized string, while replacements recognize its first character).

Heuristic rules are also employed as sources of information for the alignment process. Such rules are

Algorithm 1 Similarity Based Querying with Levenshtein

```
1:  procedure FINDSIMILAR(M, w = w_1 ... w_n)
2:      stack ← ∅                                   ▷ Joint states of M and LEV_1(w)
3:      PUSH(stack, ⟨initialState(M), w_1 ... w_n, 0⟩)
4:      while stack ≠ ∅ do
5:          js ← POP(stack)
6:          if isFinal(s[1]) then
7:              return true
8:          end if
9:          next ← δ(M, js[1])                      ▷ Next state in M, if defined
10:         if next ≠ null then
11:             PUSH(stack, ⟨next, js[2]_2 ... js[2]_n, js[3]⟩)
12:         end if
13:         if js[3] < 1 then
14:             PUSH(stack, editStates(js))         ▷ Explore transitions from LEV_1(w)
15:         end if
16:     end while
17:     return false
18: end procedure
```

patterns that can be expressed as regular expressions, bound to a property p. One simple example of a rule $\mathscr{R}$ is `dbo:date` $\rightarrow$ `[0-9]{4}-[0-9]{2}-[0-9]{2}`. When the terms of a property p_1 (from a data service) match, for instance, with such $\mathscr{R}$, then p_1 can be considered as an equivalent property of `dbo:date`. Since the index for external sources consists of a DFA M, instead of a tree, heuristic rules can be merged directly into M, causing any string that matches a rule to be considered part of the index.

4.1.3 Property Matching

In order to calculate the degree of overlapping data between two properties (existing in both indexes) and find out more accurate equivalent properties, we define a strength value for each performed property matching. We provide an equation to compute the equivalent property strength for two properties p_1 and p_2 as follows:

$$Strength(p_1, p_2) = \frac{|\mathscr{V}p_1 \bigcap_s \mathscr{V}p_2|}{|\mathscr{V}p_1|} \tag{1}$$

where $\mathscr{V}p_1$ is the set of terms provided by a data service pertaining to a given property p_1, and $\mathscr{V}p_2$ is the set of values of a property p_2 provided by an external source, or recognized by any heuristic rule bound to p_2. The intersection $\bigcap_s$ uses a similarity-based algorithm to check whether a term $t_1 \in \mathscr{V}p_1$ is similar to a term $t_2 \in \mathscr{V}p_2$ and thereby can be considered as a co-occurrence. Besides Levenshtein, other similarity measures could be used, possibly requiring a change in the external sources index for efficient similarity-based query. Based on the overlap strength, we identify the degree of correspondence between two properties, so the higher the overlapping strength, the most likely that the properties are equivalent.

Figure 4 illustrates the equation (1) in a scenario with three properties. `service:prop1` is a property from an ontology automatically constructed for a data service, while the other two properties come from external data sources. The rounded hatched rectangles represent $\mathscr{V}p_1 \bigcap_{d_L \leq 1} \mathscr{V}p_2$. Although `geo:name` has an intersection with the generated property, we can observe that the strength of this relation (0.33) is smaller than that between `service:prop1` and `dbo:givenName` (0.83). In view of that, we can define a threshold for the strength so that property pairs without significant support for their equivalence can be discarded.

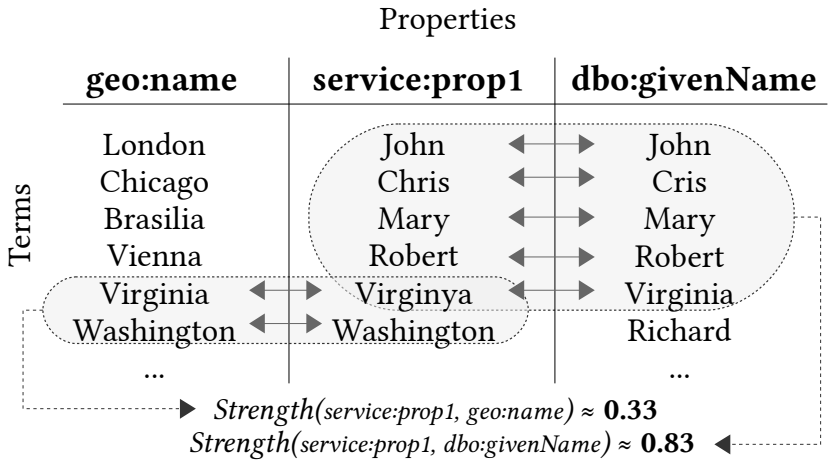

Figure 4. Example of overlapping property values.

Algorithm 2 presents the main steps for determining property equivalences from comparing property values provided by data services with external data sources. The procedure receives as input the ontology $\mathcal{O}$ constructed for the data service; the indexes $\mathcal{I}$ and $\mathcal{I}'$ containing, respectively, data from service representations and external sources; and the strength threshold represented by α. The algorithm starts matching all properties from the data services with properties obtained from external sources (lines 2-4). For each P_1, P_2 pair, the algorithm counts how many of the values enumerated in $\mathcal{V}p_1$ (line 3) are also present in $\mathcal{V}p_2$ (lines 5-10). This counting employs the FINDSIMILAR procedure (Algorithm 1) on the DFA for the objects of P_2 (lines 5 and 8). The strength is computed in line 11, in accordance to equation (1). In what follows, a triple stating that P_1 and P_2 are equivalent is created (line 12) and it is added to the ontology if strength satisfies the threshold α, or removed from the ontology otherwise (lines 14-16).

In order to improve efficiency, the implementation of the algorithm in OntoGenesis Engine was done in such a way that the strength result is stored in memory and updated to each new set of values coming from the data service. Thus, it is not necessary to recalculate the intersection of the whole set, but only of the new terms received, to update the strength.

4.2 OntoGenesis API

The OntoGenesis API is a RESTful API meant to be accessible for all data services that need to be semantically enriched. This API exposes two main features: i) register the data service in the Services Repository, and ii) invoke the OntoGenesis Engine to semantically enrich received representations from registered data services. It is worth noting that the OntoGenesis API is an intermediate layer of communication between services and the main components of the OntoGenesis Engine.

When a given consumer interacts with a registered service, the latter sends the resource requested by the user to the OntoGenesis API for semantic enrichment. The OntoGenesis API invokes the OntoGenesis Engine and returns to the data service its new ontology, along with the semantic mappings. Therefore, legacy data services that provide purely syntactic representations are able to register with OntoGenesis and be dynamically enriched with semantic data.

The OntoGenesis API supports custom configurations, such as a threshold, for the equivalent properties strength, and the size of the representation buffer. The former must be configured with values from 0 to 1 and it is used during execution of the property matching algorithm (Algorithm 2). The representation buffer size defines how many data service representations will be sent to the OntoGenesis Engine to be

Algorithm 2 Property Matching

```
1:  procedure MATCHPROPERTIES(𝒪, ℐ, ℐ′, α)
2:      for each P1 ∈ GETPROPERTIES(ℐ) do
3:          𝒱p1 ← GETVALUES(ℐ, P1)
4:          for each P2 ∈ GETPROPERTIES(ℐ′) do
5:              M ← GETDFAFOR(ℐ′, P2)
6:              overlap ← 0
7:              for each v ∈ 𝒱p1 do
8:                  if FINDSIMILAR(M, v) then
9:                      overlap ← overlap + 1
10:                 end if
11:             end for
12:             strength ← overlap / |𝒱p1|
13:             equivProp ← ⟨P1 owl:equivalentProperty P2⟩
14:             if strength ≥ α then
15:                 add equivProp to 𝒪
16:             else if equivProp ∈ 𝒪 then
17:                 remove equivProp from 𝒪
18:             end if
19:         end for
20:     end for
21: end procedure
```

processed. When it is customized with a value greater than one, representations from each service are stored in a buffer before they are sent to the Engine. Thus, instead of forwarding representations one by one, it sends a batch of representations to the Engine in order to reduce the calls and therefore improve the performance of the semantic matching process. It is worth noting, however, that the larger the buffer size, the longer it will take for the service to be enriched. Experimental results concerning the representation buffer size are presented in Section 5.

Finally, the OntoGenesis API offers an interface that allows loading new external data sources into the Index Repository. Accordingly, users can send new datasets to be loaded, so that their content will be considered at runtime by the Properties Matcher in forthcoming service requests.

4.3 Semantic Adapter

The Semantic Adapter is a component that must be attached to a data service that will be semantically enriched and thereby provide both semantic description and representations. As shown in Figure 1, it is responsible for the interaction between the data service and the OntoGenesis API. In addition to being imported into the data service implementation, its only required configuration is the URI of the OntoGenesis API. Therefore, by using the Semantic Adapter, the service registration in OntoGenesis is performed automatically when the service is deployed.

It manages three elements: the Semantic Description, the Domain Ontology generated by the Engine and the Mapping Repository. The Mapping Repository has the purpose of storing the associations of the elements extracted from the representations and the service description with the concepts defined in the ontology. It is accessed whenever the service responds to any request. Both the semantic description and the ontology are published by the Semantic Adapter through a service endpoint, so as to make them accessible to external consumers.

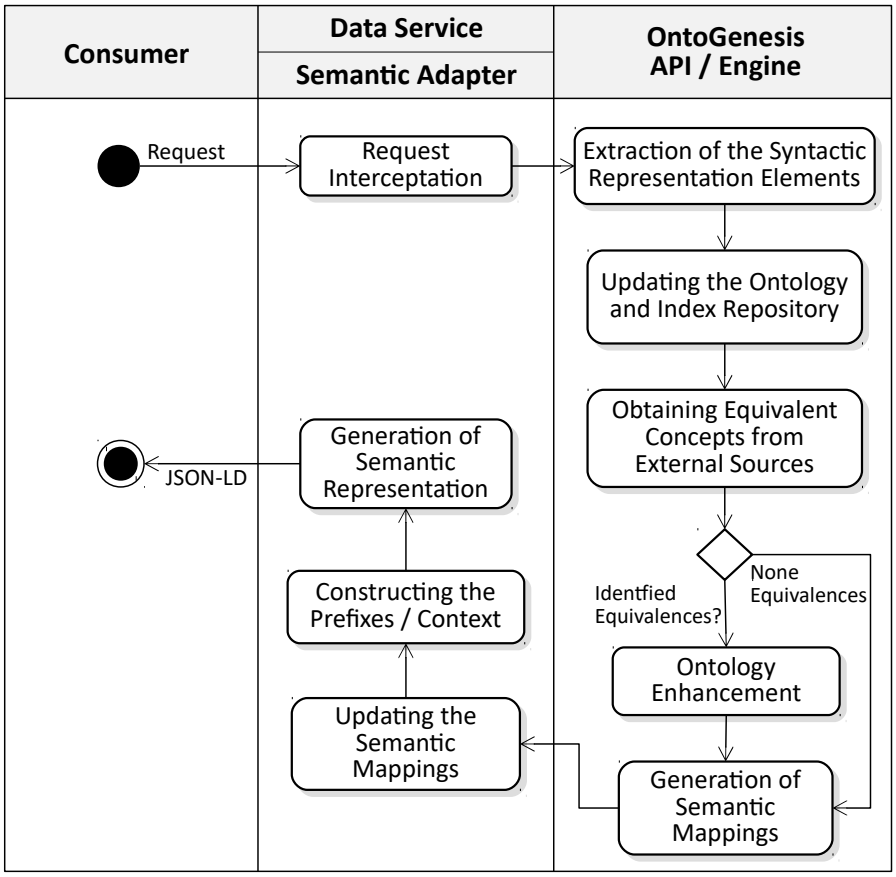

Figure 5. Processing Flow of the Representations Enrichment.

4.3.1 Generating Semantic Representations

Figure 5 shows the activity diagram in which a consumer sends a request to the data service and illustrates the processing flow performed by the main components. When a given consumer sends a request to the data service, the Semantic Adapter intercepts the request and transparently invokes the OntoGenesis API, which delegates to the Engine the construction of a domain ontology and semantic mappings. Based on these artifacts, a new semantic representation serialized in JSON-LD is generated by the adapter and returned to service consumers. Therefore, instead of providing syntactic data, the data service is able to serve linked data to its clients.

The Semantic Adapter also takes into consideration the `@id`, `@type` and `@context` syntax tokens (Lanthaler et al., 2014) to create JSON-LD documents. The `@id` uniquely identifies the resource data (specifically, the requested URI) that is being described. The `@type` describes the resource described by the JSON-LD with a concept in the ontology, and the `@context` is used to map terms that expand to URIs, also defined in the ontology. These terms are kept the same as declared in the syntactic representation, to avoid removing any field that would otherwise be present in the original representation. A key benefit of adopting this feature is to allow both semantic-capable and syntactic consumers (i.e., those unable to process semantic data) to process the new outputs of the data service, since the attribute labels do not change. Listing 3 shows a sample of JSON-LD context mapping the JSON representation and ontology illustrated in the previous example of Listings 1 and 2.

Listing 3 Excerpt of a JSON-LD context.

```
{   "@context": {
       "PoliceReport": "http://service-example/ontology/PoliceReport",
       "reportID": "http://service-example/ontology/reportID",
        ...
       "personInvolved": "http://service-example/ontology/hasPersonInvolved",
       "name": "http://service-example/ontology/name",
       "docID": "http://service-example/ontology/docID",
       "birthDate": "http://service-example/ontology/birthDate",
       "nationality": "http://service-example/ontology/nationality",
        ...
    },
    "@id": "http://service-example/policeReport?reportID=2015-10004-794",
    "@type": "PoliceReport",
}
```

Listing 4 TemplatedLink of a Semantic Description using Hydra.

```
1  {
2    "@context":"http://www.w3.org/ns/hydra/context.jsonld",
3    "@id":"http://service-example/doc",
4    "@type":"hydra:ApiDocumentation",
5    "hydra:supportedClass":[
6      {
7        "@id":"http://service-example/ontology/EntryPoint",
8        "@type":"hydra:Class",
9        "hydra:supportedProperty": [
10         {
11           "@id":"findPerson"
12           "rdfs:range":"http://service-example/ontology/PersonInvolved",
13           "@type":["hydra:TemplatedLink","hydra:IriTemplate"],
14           "hydra:template":"http://service-example/person/rg/{?idPerson}",
15           "mapping":[
16             {
17               "@type":"IriTemplateMapping",
18               "variable":"idPerson",
19               "property":"http:/service-example/ontology/docID",
20               "required":true
21             }
22           ],
23           "supportedOperation":[
24             {
25               "@type":"Operation",
26               "method":"GET",
27               "returns":"http://service-example/ontology/PersonInvolved"
28             }  ]
29         }  ]
30     }  ]
31 }
```

4.3.2 Generating Semantic Description

The service description is also enriched with semantic features during a request. The Semantic Adapter stores a semantic service description containing all the endpoints available. It adopts Hydra vocabulary (Lanthaler and Guetl, 2013) for representing data service descriptions, defining the entities that a data service is able to manage and how to obtain them. Hydra employs the concept of classes to describe the resources (identified by a URI), containing their supported properties and operations. Each operation also informs the HTTP method that must be used. The code of Listing 4 describes an Hydra TemplatedLink from a service description using Hydra. Line 23 shows the supported operations (with a GET method). The type of resource to be returned is indicated by the element `returns` (line 27).

Hydra also provides support for the consumer to build the URL at run time, through the *IriTemplate* class. This class specifies a URI Template (Gregorio et al., 2012) and maps its variables to properties described in Hydra. In the example of Listing 4, `idPerson` must be replaced by a value associated with the Person class, in order to access the desired resource.

Algorithm 3 shows the basic steps performed by the Semantic Adapter to enrich the mappings. It is called after OntoGenesis returns the ontology and Semantic Mappings. The procedure receives as

Algorithm 3 Mappings enrichment.

```
1:  procedure EnrichMapping(URI, httpMethod)
2:      uriTemp ← GetOrCreateUriTemplate(URI, httpMethod)
3:      P_uri ← GetParametersOf(uriTemp)
4:      P_¬SM ← ∅                                          ▷ Parameters not in SM
5:      json ← ∅
6:      for each param ∈ P_uri do
7:          if param ∈ SM then
8:              op ← SM.getOntologyPropertyOf(param)
9:              paramMapping ← uriTemp.getMappingOf(param)
10:             paramMapping.property ← op
11:         else
12:             V ← GetValuesOf(param)
13:             json.add(param, V)
14:             P_¬SM.insert(param)
15:         end if
16:     end for
17:     if P_¬SM ≠ ∅ then
18:         SendToOntoGenesis(json)                        ▷ Updates: SM and Ontology
19:         for each param ∈ P_¬SM do
20:             op ← SM.getOntologyPropertyOf(param)
21:             paramMapping ← uriTemp.getMappingOf(param)
22:             paramMapping.property ← op
23:         end for
24:     end if
25: end procedure
```

input the URI requested by the consumer as well as the HTTP method used for the request (GET, POST, DELETE, etc.). Initially, the Semantic Adapter seeks in the Service Description Repository for the IRI Template associated to the URI called by the consumer. If the URI is not described in the description, a new *IRITemplate* is created for such URI (line 2). Afterwards, all URI parameters are extracted (line 3). For each existing parameter, it is verified if there is a mapping in the SM (lines 7-8). If so, the ontology property associated with the syntactic parameter is retrieved and added as the property of the *IRITemplate* mapping (lines 9-10). Otherwise, the parameter and its values are added to JSON structure, in addition to adding the parameter in an auxiliary variable, $P_{\neg SM}$ (lines 12-14).

After processing all the URI parameters, the algorithm verifies if there is any parameter that was not associated with any semantic concept (due to the lack of semantic mappings) and, if so, it sends to the OntoGenesis Engine a JSON containing all these parameters and their values (lines 17-18). When OntoGenesis receives such JSON, follows the entire flow for adding to the ontology the semantic concepts related to the parameters contained in the JSON document. At the end of the process, the updated ontology is returned to the service along with new semantic mappings. Lines 19 to 23 represent the same process of enriching the parameters described previously for lines 8 through 10. In the end, all URL parameters have a property in the service ontology, and they are associated by the `mapping` of Hydra description.

In addition, Algorithm 4 associates the semantic concept to the `returns` field in the Hydra description. It is done based on the entity type of the representation. The type can be found in the `@type` property in the generated JSON-LD for the representation, as shown in Listing 3.

In summary, the Semantic Adapter provides a way to dynamically enrich service description and representations. A non-intrusive implementation of the Semantic Adapter is provided for the JAX-RS specification so as to intercept the requests coming to a data service. This implementation transparently

Algorithm 4 Operation returns enrichment.

```
1: procedure EnrichSupportedOperation(URI, httpMethod, jsonLD)
2:     uriTemp ← GetOrCreateUriTemplate(URI, httpMethod)
3:     operation ← GetOperation(uriTemp, httpMethod)
4:     type ← ExtractTypeFrom(jsonLD)
5:     operation.returns ← type
6: end procedure
```

diverts responses with representations generated by the data service to the OntoGenesis API, and i) replaces them with JSON-LD, and ii) semantically enriches its description. Moreover, the Semantic Adapter provides the data service with an endpoint to share the constructed ontology and its Hydra description in accordance with its URI base. Semantic applications can thereby retrieve the service description and the ontology in order to perform reasoning tasks.

5 EVALUATION

In this section we describe the scenario in which the evaluation was performed, the experimental methodology and the obtained results. The aim of this evaluation is to show the applicability of OntoGenesis and to analyze the compliance and performance measures using real world data. Regarding compliance, we analyze the recall, precision and F-measure obtained by the proposed algorithm, and in the performance measures we observed the average processing time, memory and CPU consumption. In the end, we briefly describe a test performed with two ontology matchers in order to compare them with our approach.

5.1 Scenario

The scenario is based on open government data published by SSP/SP, which discloses police reports in a non-structured format. Two categories of reports were selected: "intentional homicide" and "suspicious death", due to the large attention given to these types of crime/death. The police report provides information on those involved (victims and perpetrators), such as name, document number, birth date, nationality, place of birth, gender, among others. In spite of being a substantial initiative of information transparency to society, SSP/SP does not publish information in a suitable format to be integrated and reused by other data sources or even analyzed by researchers. In order to enrich this information, we have deployed a data service that provides non-semantic data of those who have been involved in any police report filed in the year of 2015.

As external data sources of OntoGenesis, we have used subsets of both DBpedia and Geonames. DBpedia is a huge cross-domain database that offers diverse Linked Data extracted from information boxes in Wikipedia pages. Geonames is a geographical database containing more than 160 million RDF triples available on the Web, and also comprises a vocabulary for representing information on places, such as latitude, longitude, name, etc. A pair of subsets of Geonames was utilized due to their potential for data intersection: Places in Brazil and Country names. In addition, we have selected the 40% most frequent terms of the DBpedia-Person dataset, which contains more than 3.5 million triples representing attributes of a person, such as name, birth date, birth place, among others.

5.2 Methodology

The deployed data service provides OntoGenesis with JSON representations that may contain up to ten attributes of a person. Nine of these attributes have equivalent properties in external sources and only one has no correspondences. Therefore, this one must have a datatype property in the ontology, but with no

equivalent property. It is worth mentioning that all representation attributes provided by the data service were labeled in Portuguese, literally as in the data source, whereas the properties of external sources remained in English.

We analyzed the behavior of our approach setting three different values for the threshold of equivalent properties strength: 0.4, 0.6 and 0.8. For experimental purposes, we have configured the representation size buffer to 1, that is, OntoGenesis will process each request at a time. When it receives a representation from the data service, OntoGenesis extracts all values, checks overlapping of terms with external sources and, finally, updates the data service ontology with new properties and equivalent properties, in accordance with the configured strength threshold.

The experiments were performed using random person data, with 7 rounds of 100 requests sent by the data service to OntoGenesis for each of the thresholds. All experiments were performed on an Intel i7 processor running at 2.5GHz, equipped with 8GiB of memory, using Oracle Java Development Kit (JDK) 8 with Xms=128M and Xmx=1024M, and Apache Jena 3.3.0 for RDF and ontology processing. Source code and instructions for setting up the evaluation as well as the detailed experimental results are available in a public repository[2].

5.3 Compliance Evaluation

Figure 6 shows the average of precision and recall per request, considering the 7 rounds of execution and their confidence interval in the shadowed area surrounding the lines. It also depicts the F-measure, representing the harmonic mean of precision and recall. Figure 6 (a) shows that in the scenario with a strength threshold of 0.4, the recall is kept stable, reaching almost 0.9 after 20 requests, while the precision slightly decreases to 0.3. In the second scenario, with a threshold of 0.6 (Figure 6 (b)), the recall increases before executing 10 requests, while the precision slightly increases, reaching almost 0.4. On the other hand, in the scenario with a threshold of 0.8 (Figure 6 (c)), the precision reaches nearly 0.6, with recall and F-Measure of approximately 0.85 and 0.7, respectively.

In all scenarios, after some time, OntoGenesis was capable of enriching, with equivalent properties, 8 out of 9 properties that could be enriched (approx. 0.89 of recall). In addition, it has also found false positive equivalent properties for the same datatype property. The reason behind this is that there are few different properties sharing the same frequent terms. For instance, `pessoa:nome` (a property of the data service ontology that represents a person's name) has correspondences with `dbpedia:name`, `dbpedia:surname` and `dbpedia:givenName`. Nevertheless, we consider only `dbpedia:name` as the correct match. In the same way, `pessoa:naturalidade` (a property of the data service ontology related to the place of birth) has correspondences with `dbpedia:name` and `geo:name`, since there are several places in Brazil that are named after a person. These situations can be dealt with increasing the strength threshold. The higher the threshold, the higher is the precision. Nonetheless, we must pass a balanced value to maintain a suitable recall score, otherwise, the recall tends to decrease. Both recall and precision are also related to the overlapping of data existing between service representations and external sources. In the first requests in scenario with threshold $\alpha = 0.8$, we observed a slight decrease in the recall, since many property values of service did not match those of DBpedia nor Geonames, which leads to a strength $\leqslant \alpha$. Therefore, in scenarios with higher thresholds, the strength tends to balance as new values are passed to each request.

[2]Repository: https://github.com/brunocnoliveira/ijwis2018-ontogenesis-experiments.

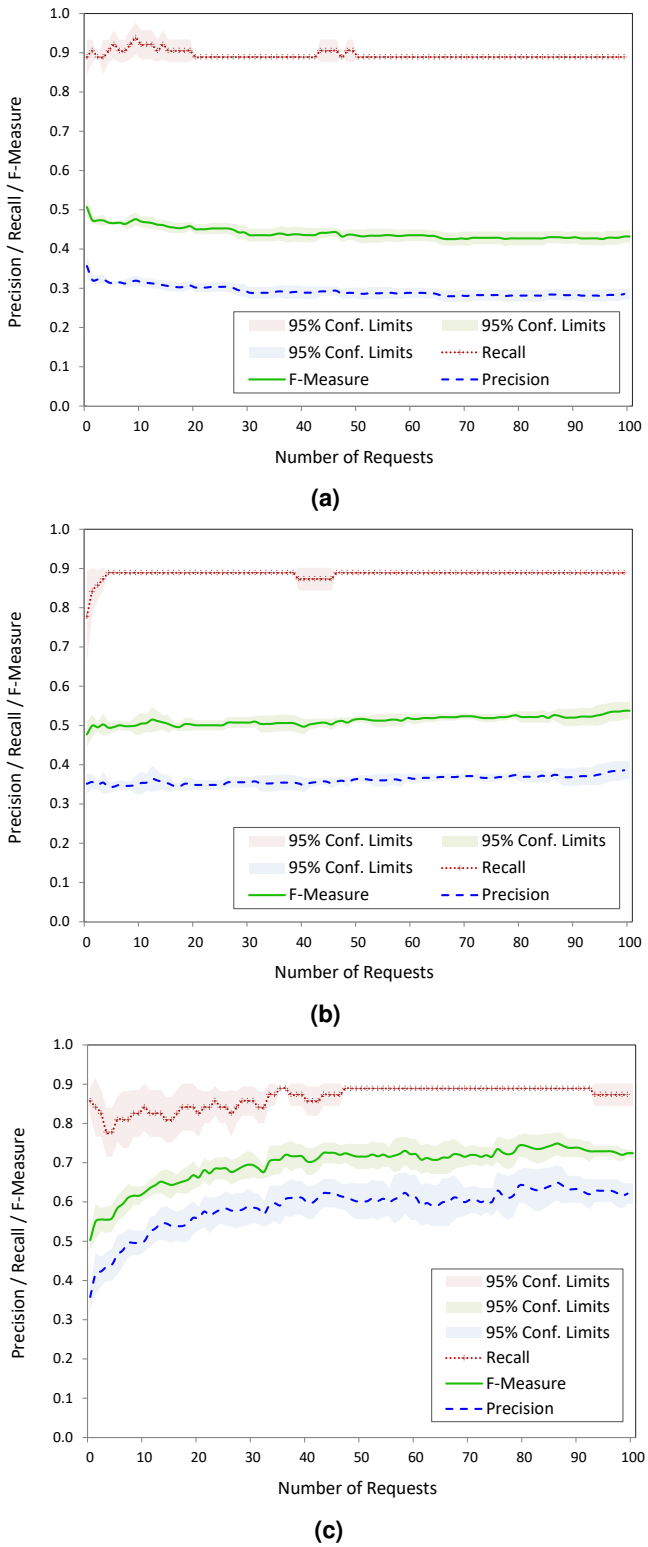

Figure 6. Precision, Recall and F-Measure curves for strength threshold (a) $\alpha = 0.4$; (b) $\alpha = 0.6$; and (c) $\alpha = 0.8$.

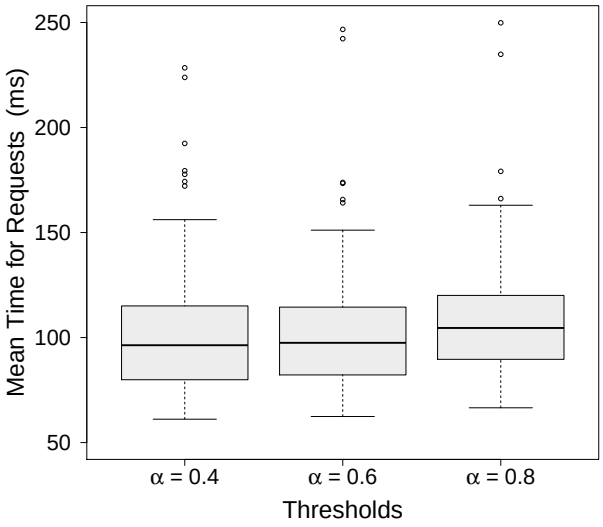

Figure 7. Requests Time Analysis.

5.4 Performance Evaluation

According to Euzenat et al. (2005), the two most commonly used performance measures are the speed (i.e., the processing time) and the memory consumption. We have analyzed both measures in addition to the CPU consumption rate. The performance evaluation reuses the same methodology discussed in subsection 5.2.

Table 1 presents the mean processing time for 4 variables (semantic mappings (*SM*) and ontology construction; property matching process; JSON-LD generation; and description enrichment), along with their confidence interval for the mean of all observations for each combination of variable and α level. One can notice that there is no significant alteration in execution time between the adopted thresholds. Additionally, the processing time is independent of the number of requests, since previous requests are not reprocessed, but only the strength of equivalent properties is recalculated. Moreover, Figure 7 shows a boxplot for the average time (across the 7 rounds) of each of the 100 requests. Such boxplot represents the 50% quantile around the median, and the 1.5 inter-quantile range threshold for outliers. While the mean observed for the $\alpha = 0.8$ is slightly higher, the three sampling distributions largely overlap, suggesting that, in practice, the effect of the threshold on time is of little importance.

Unlike the compliance measures, performance measures depend on the benchmark processing environment. In view of that, the same environment settings previously described was used. However, in order to analyze the average of memory and CPU consumption we have conducted an additional experiment in which a consumer sends random requests to OntoGenesis for a period of time. We started measuring after

Table 1. Processing times with 95% Confidence Interval.

α^*	*SM* and Ontology Construction	Properties Matching	JSON-LD Generation	Description Enrichment
0.4	19.10 ± 1.17 ms	19.57 ± 2.48 ms	3.57 ± 0.32 ms	1.21 ± 0.027 ms
0.6	17.98 ± 0.97 ms	18.05 ± 1.42 ms	3.39 ± 0.26 ms	1.12 ± 0.024 ms
0.8	19.58 ± 1.08 ms	21.22 ± 1.75 ms	4.23 ± 0.35 ms	1.19 ± 0.032 ms

*α = Strength threshold

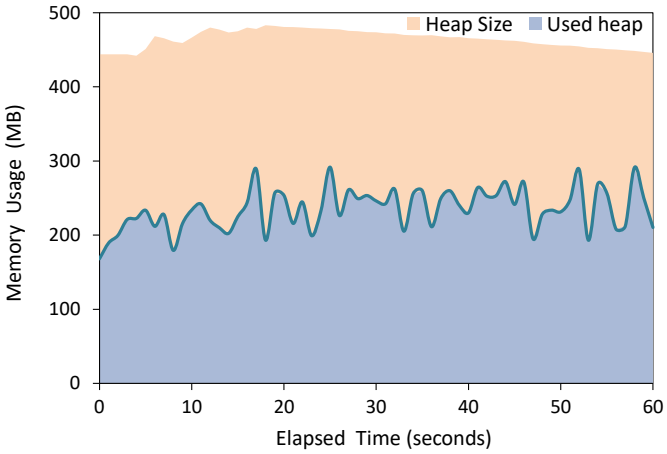

Figure 8. Average Memory Consumption of OntoGenesis.

an initial preheating phase consisting of a portion of requests during 5 seconds, in order to reduce the variance introduced by the Java Just-in-Time compiler. It is worth noting that the resources consumption regarding the OntoGenesis setup (including external data source loading and the automaton construction) is not accounting in this experiment. Therefore, both memory and CPU consumption variables were monitored during 60 seconds after OntoGenesis executes the preheating phase. Such measures were observed in the JVM in which OntoGenesis was executed.

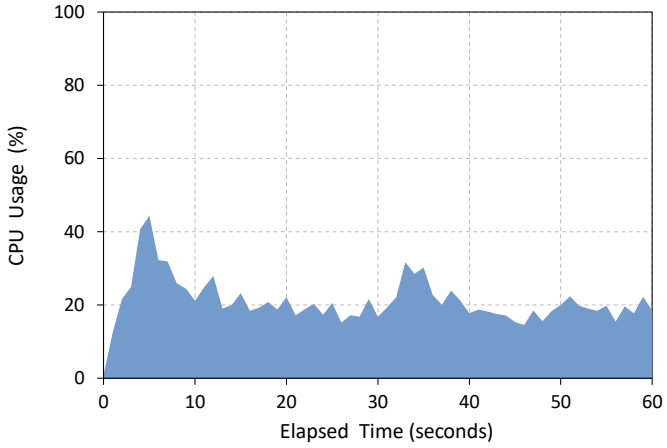

Figure 9. Average CPU Consumption of OntoGenesis.

Figures 8 and 9 show the average memory consumption and the mean CPU consumption, respectively, considering 7 rounds of execution. As can be seen, the average of used heap size reaches up to 300MB and there are few variations over time. This experiment considers synchronous requests of a single client, which avoid full occupancy of the CPU by the engine process, as a large portion of time dedicated to the synchronous HTTP communication with the client occurs while the engine process is deemed not runnable by the operating system scheduler. Therefore, the average CPU consumption reaches up to 40% and remains below this value during execution.

5.5 Other Ontology Matchers

Extensional-based techniques can be adapted for service enrichment (Salvadori et al., 2017) and, similarly to OntoGenesis, exploit instance data to find ontology alignments. They rely on individuals obtained from

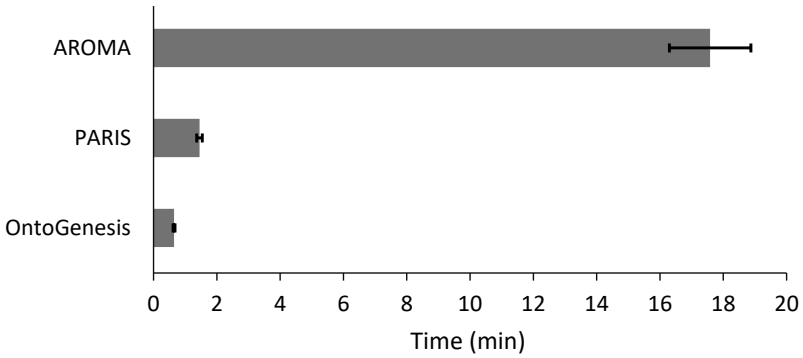

Figure 10. Processing Time Comparison.

data services and from external sources to check if they have some intersecting properties, i.e., properties with matching identifiers or with values in the same domain. This correference or value sharing among properties is used to infer property and class equivalences among different ontologies.

Firstly, we converted our scenario from service semantic enrichment to ontology alignment. The goal is to align the ontology constructed by OntoGenesis (without alignments) against DBpedia ontology, FOAF and Geonames. To this end, two files are generated. The first contains the constructed ontology along with 1021 person instances described in this ontology. The second file contains external ontologies and all instances from the external datasets. We evaluated two extensional matchers, PARIS (Suchanek et al., 2011) and AROMA (David et al., 2007), executing 7 rounds in each test. Nevertheless, none were able to produce property alignments. The results were also not favorable performance-wise: on average, AROMA took almost 18 minutes, PARIS took 1 minute 45 seconds and OntoGenesis took only 39 seconds. Figure 10 shows the processing time for both extensional matchers in comparison with OntoGenesis.

The challenge to these otherwise successful ontology matchers is in the absence of correferent and data sharing between the service data and the external data. For instance, we observed that the only complete literal shared between the two service data and external data is "Preta"[3]. Extending the comparison to Levenshtein with threshold 1, there are 2,324 (0.20%) subjects in the external data that share a single literal value with the service data. However, no such subject shares the value of more than one property with a counterpart in the service dataset, which becomes a challenge for extensional matchers that are based on the similarity of individuals.

6 RELATED WORK

The topic of semantic enrichment of Web services is related to some similar problems, such as ontology construction, ontology matching, automatic service representation and description generation, among others. A variety of research works have been proposed to address these problems. In view of that, the related work is divided into two major categories. The first category discusses research on data-based ontology construction and ontology matching approaches. The second category concerns proposals related to the semantic enrichment of services.

As discussed in section 2, Euzenat and Shvaiko (2007) identify 4 groups of ontology matching approaches, each with specific challenges in the scenario of semantic enrichment of services. Name-based techniques require ontology elements to be labeled in the same language (which was not the case for the

[3]"Preta" means black skin color in Portuguese and is also a place in the GeoNames dataset (`http://www.geonames.org/3391216`).

scenario in section 5) and to have elements with similar names. Semantic and structure-based techniques analyze ontology features, e.g., the class hierarchy and relations between classes. Thus, OntoGenesis constructs ontologies with a flat class hierarchy (without subclasses) and the only relations between classes occur through domains and ranges of properties. Although extensional-based techniques can be adapted for service enrichment (Salvadori et al., 2017), such techniques present poor performance (as shown in section 5.5) when data obtained through the service interface is unrelated to data from the external sources and when individuals are not the same. In light of such evidence, we argue that extensional ontology matching algorithms present distinct characteristics from the problem we tackle, specifically property matches.

On the other hand, there are few efforts focusing on matching properties in ontologies. Tran et al. (2011) introduce a cluster-based similarity aggregation methodology for ontology matching, which relies on some different similarity measures to align object properties based on their domains and ranges. The authors, however, mention that the evaluation results are not strong enough to distinguish matching and non-matching property names. Zapilko and Mathiak (2014) identify the exact relationship between two objects in large scale linked data, using governmental data. Their goal is to align instances of ontologies separately and to compute the overlap between them in order to improve the matching of object properties. Complementarily, the overlap scores use some variations of the Jaccard coefficient. A drawback in the aforementioned approaches is that they are limited to object properties. On the other hand, Pereira Nunes et al. (2013) use genetic algorithms to match complex Datatype properties. However, their approach also differs from ours in the sense that their goal is the matching of one to many complex relationships in different ontologies. In other words, the authors are interested in mapping properties that are composed of other properties (e.g., mapping “first name” and “last name” to “full name”).

Many tools and approaches have been designed aiming to support users in the ontology construction process (Cimiano and Völker, 2005; Salem and AbdelRahman, 2010; Nguyen and Lu, 2016). Nonetheless, they all suffer from some shortcomings. First, most of them depend either on very specific or proprietary ontology models, which hinders their wide applicability. Second, such approaches are not fully automatic, since they focus on assisting expert domain users. Finally, traditional ontology construction methods commonly require as input a huge set of unstructured text or Web page data (Nguyen and Lu, 2016), differently from our approach, in which data is provided on demand by service interfaces.

An interesting approach is proposed by Yao et al. (2014). The authors introduce a framework for transforming semi-structured data – specifically a set of JSON documents provided by Web services – into a unified ontology. Firstly, the framework parses the JSON and yields RDF triple sets. Afterwards, multiple independent ontologies are created based on the triple sets and, meanwhile, an ontology merging process is performed to achieve one unified ontology model. The resulting ontology must be validated by domain expert users, who can validate and edit the final ontology. Although it demonstrates to be a relevant study related to ontology construction based on JSON representations, the authors do not tackle the automatic semantic enrichment of Web services, neither consider the generation of a semantic description. Furthermore, the framework only considers JSON as input and outputs a single ontology, while OntoGenesis can support different representation formats and derive one domain ontology for each data service. Moreover, we propose a property matching technique in order to optimize the reuse of concepts defined in external resources, whilst they do not use any external source to enhance the constructed ontologies. The dataset used in their experiments is not provided in the paper, but based on the presented results, we infer that their approach, in the best case, is able to process 2,038 triples/sec. In contrast, OntoGenesis processed approximately 96,300 triples/sec in the scenario described in section 5.

Several authors have been directing efforts towards automatic/semi-automatic semantic enrichment of Web services (Tosi and Morasca, 2015). These works can be divided into two subgroups: service description and service representation enrichment. The former concentrates on adding semantic information to describe service interfaces, while the second aims to enrich service representations with semantic features. Little attention is given to the latter group. Most of the proposals found in the literature focus on enhancing service descriptions with semantics, such as SDWS (Bravo et al., 2014) and ASSARS (Luo et al., 2016). Saquicela et al. (2011) propose an approach for generating semantic annotation of RESTful services using external resources, such as DBpedia ontology and a synonymous database. Nevertheless, semantic data is added only to the service description.

A composition method that exploits the potential data intersection observed in data-based microservice descriptions is proposed by Salvadori et al. (2017). The proposed method focus on creating links between semantic resources, so that representations provided by microservices are enriched with `owl:sameAs` and `rdfs:seeAlso` links. Additionally, the authors propose a framework called Alignator, which employs ontology matching techniques to identify equivalences between different ontologies that describe different microservices. However, this approach only considers services that i) already adopts a previously defined domain ontology and ii) already provide Linked Data representations and semantic descriptions. Therefore, such framework does not support services that do not provide any description handle syntactic data.

7 CONCLUSIONS AND FUTURE WORK

This paper presented an architecture to automatically enhance data services with semantic features. The architecture, called OntoGenesis, aims to construct domain ontologies for describing data provided by data services. Additionally, it improves the constructed ontology through a property matching mechanism, which reuses well-known concepts used by external data sources. Finally, an adapter plugged into the service transparently produces a semantic description and enriches the output representations with semantic concepts defined in the domain ontology. Similar approaches found in the literature focus on generating semantic descriptions in a semi-automatic way and do not cover semantic enhancement of data provided by a service. Furthermore, solutions for ontology construction and alignment, in general, are not suitable for scenarios in which data instances are dinamically obtained through a service interface.

Experiments were performed to analyze both compliance and performance measures. The former indicates the equivalence strength between properties of the constructed ontology and the external data sources according to defined thresholds. The precision is tightly related to the strength threshold, so better F-Measure scores are achieved when the threshold is increased. In some cases, however, we observed that similar values may also lead to wrong matches. For instance, a person's surname could match with a place name. The performance measures show that, in virtue of the proposed automata-based indexing mechanism, OntoGenesis achieved faster property matching in comparison with extensional ontology matchers. Moreover, in comparison with an ontology construction approach (Yao et al., 2014), OntoGenesis was 47 times faster in the number of triples processed per second. Overall, the evaluation results show that OntoGenesis is a promising approach that can boost the provisioning of semantically enhanced data by service providers, reducing the efforts involved in the process of developing semantic data services.

As future work, we intend to propose a mechanism to identify class equivalence, in addition to the equivalent properties. We also aim to extend the strength equation so as to consider the frequency of each term in the data service and in the external data sources so as to reduce false positives. Another promising

future development concerns a mechanism for filtering specific external data sources in accordance with the domain of a data service, with the aim of achieving better performance in the execution of the property matching algorithms.

REFERENCES

Alfaries, A. (2010). *Ontology Learning for Semantic Web Services*. PhD thesis, Brunel University, UK.

Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The Semantic Web. *Scientific American*, 284(5):34–43.

Bianchini, D., De Antonellis, V., and Melchiori, M. (2015). Developers' networks contribution to web application design. In *Proceedings of the 17th International Conference on Information Integration and Web-based Applications & Services*, iiWAS '15, pages 55:1–55:10, New York, NY, USA. ACM.

Bravo, M., Rodríguez, J., and Pascual, J. (2014). SDWS: Semantic description of web services. *International Journal of Web Services Research*, 11(2):1–23.

Carey, M. J., Onose, N., and Petropoulos, M. (2012). Data Services. *Communications of the ACM*, 55(6):86.

Cimiano, P. and Völker, J. (2005). Text2onto: A framework for ontology learning and data-driven change discovery. In *Proc. of the 10th Internat. Conf. on Natural Language Processing and Information Systems*, NLDB'05, pages 227–238, Berlin, Heidelberg. Springer-Verlag.

David, J., Guillet, F., and Briand, H. (2007). Association Rule Ontology Matching Approach. *International Journal on Semantic Web and Information Systems*, 3(2):27–49.

Euzenat, J., Ehrig, M., and Castro, R. G. (2005). Towards a methodology for evaluating alignment and matching algorithms. Technical report, Ontology Alignment Evaluation Initiative (OAEI).

Euzenat, J. and Shvaiko, P. (2007). *Ontology Matching*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Fellah, A., Malki, M., and Elçi, A. (2016). Web services matchmaking based on a partial ontology alignment. *International Journal of Information Technology and Computer Science (IJITCS)*, 8(6):9–20.

Gregorio, J., Fielding, R., Hadley, M., Nottingham, M., and Orchard, D. (2012). URI Template. RFC 6570.

Guarino, N. (1997). Semantic matching: Formal ontological distinctions for information organization, extraction, and integration. In *Information Extraction A Multidisciplinary Approach to an Emerging Information Technology: International Summer School, SCIE-97 Frascati, Italy, July*, pages 139–170. Springer Berlin Heidelberg.

Guizzardi, G. (2007). Summary for Policymakers. In Intergovernmental Panel on Climate Change, editor, *Climate Change 2013 - The Physical Science Basis*, volume 155, pages 1–30. Cambridge University Press.

Hopcroft, J. E. and Ullman, J. D. (1990). *Introduction To Automata Theory, Languages, And Computation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition.

Lanthaler, M. and Guetl, C. (2013). Hydra: A vocabulary for hypermedia-driven web apis. In Bizer, C., Heath, T., Berners-Lee, T., Hausenblas, M., and Auer, S., editors, *LDOW*, volume 996 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Lanthaler, M., Sporny, M., and Kellogg, G. (2014). JSON-ld 1.0. W3C recommendation, W3C. http://www.w3.org/TR/2014/REC-json-ld-20140116/.

Lira, H. A., Dantas, J. R. V., de Azevedo Muniz, B., Chaves, L. M., and Farias, P. P. M. (2014). Semantic

Data Services: An approach to access and manipulate Linked Data. In *XL Latin American Computing Conference (CLEI)*, pages 1–12. IEEE.

Luo, C. C., Zheng, Z. c., Wu, X. X., Yang, F. F., and Zhao, Y. Y. (2016). Automated structural semantic annotation for RESTful services. *International Journal of Web and Grid Services*, 12(1):26–41.

Maedche, A. and Staab, S. (2001). Ontology learning for the semantic web. *IEEE Intelligent Systems*, 16(2).

McIlraith, S., Son, T., and Zeng, H. Z. H. (2001). Semantic Web services. *IEEE Intelligent Systems*, 16(2):46–53.

Nguyen, T. T. S. and Lu, H. (2016). Domain ontology construction using web usage data. In *Advances in Artificial Intelligence*, pages 338–344. Springer.

Oliveira, B. C. N., Huf, A., Salvadori, I., and Siqueira, F. (2017). Automatica semantic enrichment of data services. In *Proc. of Int. Conf. on Information Integration and Web-based Applications & Services*. ACM.

Pavel, S. and Euzenat, J. (2013). Ontology matching: State of the art and future challenges. *IEEE Transactions on Knowledge and Data Engineering*, 25(1):158–176.

Pereira Nunes, B., Mera, A., Casanova, M. A., Fetahu, B., P. Paes Leme, L. A., and Dietze, S. (2013). Complex matching of rdf datatype properties. In Decker, H., Lhotská, L., Link, S., Basl, J., and Tjoa, A. M., editors, *Database and Expert Systems Applications: 24th International Conference, DEXA 2013, Prague, Czech Republic, August 26-29, 2013. Proceedings, Part I*, pages 195–208, Berlin, Heidelberg. Springer Berlin Heidelberg.

Salem, S. and AbdelRahman, S. (2010). A multiple-domain ontology builder. In *Proc. of the 23rd Internat. Conf. on Computational Linguistics*, pages 967–975, Stroudsburg, USA. Association for Computational Linguistics.

Salvadori, I., Huf, A., Oliveira, B. C. N., Mello, R., and Siqueira, F. (2017). Improving Entity Linking with Ontology Alignment for Semantic Microservices Composition. *International Journal of Web Information Systems*, 13(3):302–323.

Saquicela, V., Vilches-Blazquez, L. M., and Corcho, O. (2011). Lightweight Semantic Annotation of Geospatial RESTful Services. In *Proceedings of the 8th Extended Semantic Web Conference on The Semanic Web: Research and Applications - Volume Part II*, ESWC'11, pages 330–344, Berlin, Heidelberg. Springer-Verlag.

Schulz, K. U. and Mihov, S. (2002). Fast string correction with Levenshtein automata. *Int. Journal on Document Analysis and Recognition*, 5(1):67–85.

Suchanek, F. M., Abiteboul, S., and Senellart, P. (2011). Paris: Probabilistic alignment of relations, instances, and schema. *Proceedings of the VLDB Endowment*, 5(3):157–168.

Sycara, K., Paolucci, M., Ankolekar, A., and Srinivasan, N. (2003). Automated discovery, interaction and composition of semantic web services. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(1):27–46.

Tosi, D. and Morasca, S. (2015). Supporting the semi-automatic semantic annotation of web services: A systematic literature review. *Information and Software Technology*, 61:16–32.

Tran, Q.-V., Ichise, R., and Ho, B.-Q. (2011). Cluster-based similarity aggregation for ontology matching. In *Proceedings of the 6th International Conference on Ontology Matching - Volume 814*, OM'11, pages 142–147, Aachen, Germany. CEUR-WS.org.

Yao, Y., Liu, H., Yi, J., Chen, H., Zhao, X., and Ma, X. (2014). An automatic semantic extraction method for web data interchange. *2014 6th International Conference on Computer Science and Information*

Technology (CSIT), pages 148–152.

Zapilko, B. and Mathiak, B. (2014). Object property matching utilizing the overlap between imported ontologies. In Presutti, V., d'Amato, C., Gandon, F., d'Aquin, M., Staab, S., and Tordai, A., editors, *The Semantic Web: Trends and Challenges: 11th International Conference, ESWC 2014, Anissaras, Crete, Greece, May 25-29, 2014. Proceedings*, pages 737–751. Springer International Publishing.