Computing and Informatics, Vol. 36, 2017, 1173–1206, doi: 10.4149/cai_2017_5_1173

ASSESSING WEB SERVICES INTERFACES WITH LIGHTWEIGHT SEMANTIC BASIS

Martin Garriga, Alan De Renzis, Andres Flores

GIISCo Research Group, Facultad de Informática
Universidad Nacional del Comahue, Neuquén, Argentina
&
CONICET (National Scientific and Technical Research Council), Argentina
e-mail: martin.garriga@fi.uncoma.edu.ar

Alejandra Cechich

GIISCo Research Group, Facultad de Informática
Universidad Nacional del Comahue, Neuquén, Argentina

Alejandro Zunino

ISISTAN Research Institute, UNICEN, Tandil, Argentina
&
CONICET (National Scientific and Technical Research Council), Argentina
e-mail: azunino@exa.unicen.edu.ar

Abstract. In the last years, Web Services have become the technological choice to materialize the Service-Oriented Computing paradigm. However, a broad use of Web Services requires efficient approaches to allow service consumption from within applications. Currently, developers are compelled to search for suitable services mainly by manually exploring Web catalogs, which usually show poorly relevant information, than to provide the adequate "glue-code" for their assembly. This implies a large effort into discovering, selecting and adapting services. To overcome these challenges, this paper presents a novel Web Service Selection Method. We have defined an Interface Compatibility procedure to assess structural-semantic aspects from functional specifications – in the form of WSDL documents – of candidate Web Services. Two different semantic basis have been used to define and implement the approach: WordNet, a widely known lexical dictionary of the English language; and DISCO, a database which indexes co-occurrences of terms in very large text collections. We performed a set of experiments to evaluate the approach regarding the underlying semantic basis and against third-party approaches with a data-set of real-life Web Services. Promising results have been obtained in terms of well-known metrics of the Information Retrieval field.

Keywords: Web services, service oriented computing, service discovery, service selection, WordNet, DISCO

1 INTRODUCTION

Service-Oriented Computing (SOC) has seen an ever increasing adoption by providing support for building distributed applications in heterogeneous environments [1], by reusing existing third-party components or services that are invoked through specialized protocols. Nowadays, Web Services are the technological weapon-of-choice to materialize the SOC paradigm. A Web Service is a program with a well-defined interface that can be located, published, and invoked by using standard Web protocols [2].

However, a broad use of the SOC paradigm requires efficient approaches to allow service consumption from within applications [3]. Currently, developers are required to manually search for suitable services to then provide the adequate "glue-code" for assembly into the application under development. Even with a wieldy candidates list, a developer must be skillful enough to determine the most appropriate service for the consumer application. This implies a large effort into discovering services, analyzing the suitability of retrieved candidates (i.e., selecting services) and identifying the set of adjustments for the final assembly of a selected candidate service.

In order to ease the development of Service-Oriented Applications we have presented in previous work [4, 5, 6] an approach for service selection based in an Interface Compatibility procedure. All available information from interfaces of candidate services (previously discovered) is gathered to be assessed at structural and semantic levels. The Interface Compatibility procedure leverages as semantic basis an underlying lightweight ontology – in the form of a concept hierarchy – to compare terms from interfaces. We have originally used the lexical database WordNet [7] as a semantic basis, particularly manipulated through the MIT Java library JWI [8] (Java Wordnet Interface) from which we developed a similarity formula.

The main contributions of this paper are threefold. First, we present three semantic-based alternatives to analyze functional (WSDL) descriptions of Web Services, which use generic, domain-independant concept hierarchies rather than ad-hoc heavyweight domain ontologies: WordNet, accessed through two java libraries that provide different similarity metrics, namely JWI and JWNL (Java WordNet Library); and DISCO (extracting DIStributionally related words using CO-occurrences) [9], a lexical database based on co-occurrencies in a large corpus of text. Second, we implemented these three semantic-based alternatives in the context of the Interface Compatibility assessment procedure, achieving its independence from the underlying semantic basis. Finally, we performed a suite of experiments with a large dataset of real-world Web Services, comparing our approach with well-known service discovery/evaluation algorithms [10, 11]. The Interface Compatibility procedure performed well with regard to categorization and matching algorithms based in heavyweight ontologies, in terms of well-known Information Retrieval (IR) metrics – such as *precision* and *recall.*

The rest of the paper is organized as follows. Related work are presented in Section 2. Section 3 introduces the Interface Compatibility procedure, along with a motivating example. Section 4 presents a brief description of the Structural assessment. Section 5 describes three versions of the Semantic assessment. Section 6 presents a set of experiments to validate our approach. Finally, Section 7 concludes the paper.

2 RELATED WORK

This section presents related work according to the two main topics of this paper. Section 2.1 analyzes approaches in the field of Web Service discovery, selection and adaptation. Section 2.2 focuses on semantic-based similarity measures and their classification.

2.1 Web Service Selection

The surveys in [12, 13] provide a comparative analysis of existing approaches to improve Web Services discovery considering the typical scenario where users perform queries against a service registry. This is closely related to service selection, since an improved discovery method performs a partial, preliminary selection among candidates in a registry: a service has to be discovered before it can be selected. Several approaches used IR techniques in an effort to increase precision of Web Service discovery without involving any additional level of semantic markup. Although such approaches achieve concrete improvements, they seem insufficient for automatic retrieval if applied without any complementary technique [13]. A semantic-based strategy can consist in formal ontology-based methods, which yet involve high costs in contract and query specification, making service designers being alienated from their use in practice [12, 14]. Indeed, one of the main differences is what such approaches consider/require as service descriptions. Semantic approaches depend on shared ontologies and annotated resources [11], whereas IR-based ones depend on (exiguous) textual descriptions. Those service discovery systems may be appropriate in different environments, since they strive to solve the same problem [13].

The work in [15] distinguishes two types of service mismatches: interface-level (operation definition) and protocol-level. The former are closely related to our approach, and are addressed through a matching component implemented on the top of COMA++[1] [16], a matching tool that uniformly supports schemes and ontologies – e.g., XML Schema (XSD) and OWL. Thereby, interface-level mismatches are identified primarily by assessing service's XML schemes.

Similarity is also addressed in [17] to find relevant substitutes for failing Web Services. This approach is parameterized by weighted scores and a set of (lexical and semantic) similarity metrics calculated over service names, operations, input/output messages, parameters, and documentation. Message structures and complex XML schema types are compared through schema matching, representing complex types as labeled directed graphs. However, complex types similarity exploiting a lightweight semantic basis can be performed without dealing with the complexity of an XML schema, as in our approach [4].

The Woogle search engine [18] returns similar Web Services for a given query based on similarity search. Woogle is focused on operation parameters as well as operations and services descriptions, through a clustering algorithm for grouping descriptions in a reduced set of terms. After that, similarity between terms is measured using a classical IR metric such as TF/IDF. The provided solution is limited to evaluating similarity using semantic relations between clustered terms.

The work in [19] presents UDDI Registry By Example (URBE), a Web Service retrieval algorithm for substitution purpose, based on WSDL. A substitute Web Service has to expose an interface functionally equal or richer than the interface of the failing Web Service. The similarity function is calculated upon operation name similarity and parameters data type similarity, and a maximization function obtains the maximum similarity. When available, URBE compares annotations extracted from semantically annotated WSDL (SA-WSDL) descriptions rather than names of operations and parameters. This is combined with the analysis of data types resulting in a hybrid approach. Although having high precision, the weak point of the approach is that providers do not annotate their services often in practice, since it is costly, and also relevant and commonly used ontologies in different domains are not usually available [20, 11, 21, 13].

2.2 Similarity Measures

In general terms, similarity measures quantify how much two entities are alike. In the context of this paper, entities can be concepts expressed in different ontologies as in ontology matching [22], or operations in Web Service specifications. The work in [11] presents a categorization of (WordNet-based) similarity measures. Similarity measures based on *path length* either compute the shortest path between concepts, normalized by the maximum path length of the concepts hierarchy; or find the path length of common ancestors of two concepts. *Relatedness* measures are based on

[1] `http://dbs.uni-leipzig.de/de/Research/coma.html`

relations between concepts. This can be computed by analyzing either path shapes (length and direction changes) between concepts, or the overlapping of concept definitions, or the co-occurrences of certain concepts by a co-occurrence matrix. Finally, similarity measures based on *information content* leverage the specificity of concepts, by analyzing the shared information content between two concepts (i.e., the information content of their common ancestor) and the information that each specific concept adds to such shared information.

Our work can be analyzed according to the aforementioned categorization. Particularly, the WordNet-based Interface Compatibility procedures are implemented using similarity measures based in path length. The considered relationships between terms are synonymy ($path\, lenght = 0$), and a single level of hypo/hyperonymy ($path\ length = 1$). The DISCO-based Interface Compatibility procedure is a straightforward example of a *relatedness* measure, since the construction of a co-occurence matrix is a key step of this procedure. In particular, DISCO computes co-occurrences of terms based in a very large corpus of text extracted from Wikipedia.

With regard to *information content* measures, the work in [23] proposes a Similarity Metric that calculates the *msca* (Most Specific Common Abstraction) between two terms t_1 and t_2, considering the intersection of their features. The metric relies in the meaningful and structured organization of WordNet as the underlying lightweight ontology, where concepts with many hyponyms convey less information than leaf concepts. We are working to adapt such similarity metric as an alternative to implement the Interface Compatibility procedure. However, the execution time of the algorithm that calculates the metric is prohibitive in the context of our work, thus it still needs some adjustments.

3 INTERFACE COMPATIBILITY PROCEDURE

The Interface Compatibility procedure (Figure 1) assesses structural and semantic aspects from the interface (I_S) of a candidate service S against a simple specification of an interface (I_R) from a required functionality [4, 5].

Structural aspects involve data types equivalence (mainly subtyping), while semantic aspects relate to concepts in identifiers. The procedure characterizes different similarity cases through a set of constraints for pairs of operations ($op_R \in I_R, op_S \in I_S$). Constraints involve structural and/or semantic conditions for each element (identifier/data type) of an operation signature: return type, exceptions, operation names and parameters (both type and name).

The outcome of these evaluations is an Interface Matching list where each operation from I_R may have a correspondence with one or more operations from I_S. In addition, an appraisal value named *adaptability value* is calculated upon the signature constraints – which reflects the integrability factor of the selected service. The impact on the precision achieved during the Interface Compatibility analysis is essential to reduce the subsequent adaptation effort when integrating a candidate service into a Service-oriented Application. This is the main reason for the

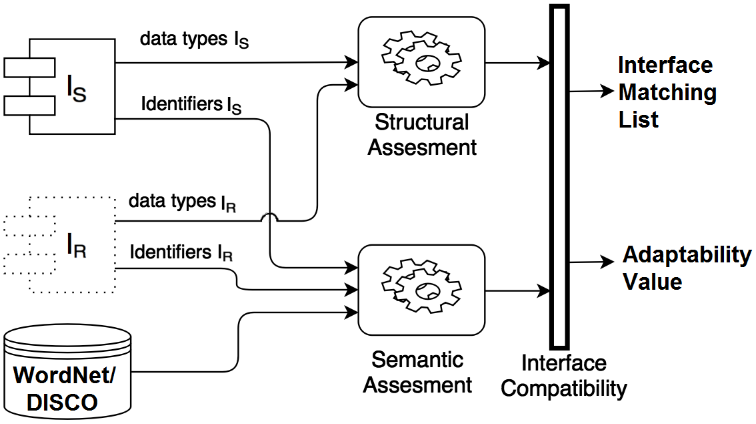


Figure 1. Interface compatibility procedure

definition of the whole procedure, in which different semantic heuristics have been applied, mostly from a practical and domain-independant perspective.

3.1 Motivating Example

Let us consider the Car Rental domain, where the main features required by a hypothetical client application are represented as a required interface I_R, namely `CarRentalBrokerService`. Such interface defines three operations and three complex data types. The operations and data types were not defined from scratch, but adapted (simplified) with illustrative purposes from an existing Web Service[2] from the Car Rental domain.

The required functionality will be fulfilled by engaging a third-party Web Service[3] – whose simplified interface I_S is called `RentaCar`. This interface defines a subset of the original operations from the Web Service comprised of four operations and three complex data types. Figure 2 depicts the structure of interfaces I_R and I_S, which were defined under the Java platform. Thus, the goal of this example is to compare similarity between these two interfaces I_R and I_S and determine if the candidate Web Service `RentaCar` is suitable to implement the required functionality `CarRentalBrokerService`.

4 STRUCTURAL ASSESSMENT

Let be I_R the interface of certain required functionality, and I_S the interface of a candidate Web Service S. For each pair of operations (op_R, op_S), a likely equivalence is

[2] `rpc.test.sunnycars.com/CarRentalAgentService121/CarRentalBrokerService.asmx?WSDL`

[3] `www.icelandcarrental.is/RentacarServices.asmx?WSDL`

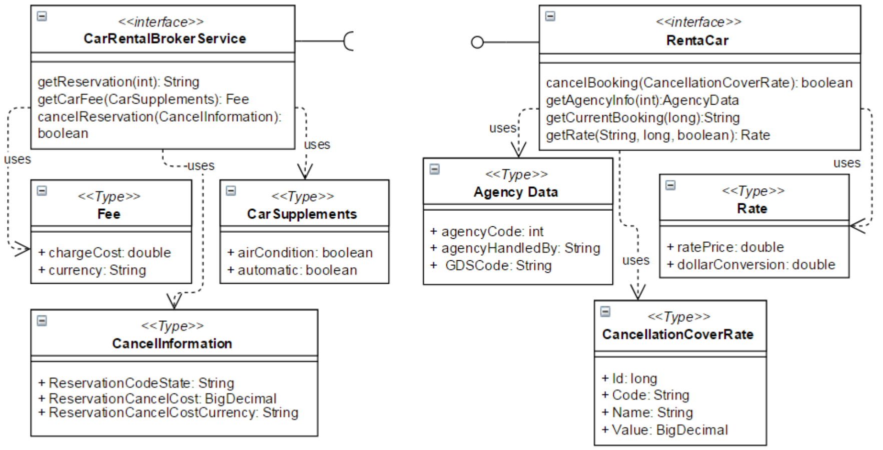

<<interface>>
CarRentalBrokerService
getReservation(int): String
getCarFee(CarSupplements): Fee
cancelReservation(CancelInformation): boolean
uses
uses
uses
<<Type>>
Fee
+ chargeCost: double
+ currency: String
<<Type>>
CarSupplements
+ airCondition: boolean
+ automatic: boolean
<<Type>>
CancelInformation
+ ReservationCodeState: String
+ ReservationCancelCost: BigDecimal
+ ReservationCancelCostCurrency: String
<<interface>>
RentaCar
cancelBooking(CancellationCoverRate): boolean
getAgencyInfo(int):AgencyData
getCurrentBooking(long):String
getRate(String, long, boolean): Rate
uses
uses
uses
<<Type>>
Agency Data
+ agencyCode: int
+ agencyHandledBy: String
+ GDSCode: String
<<Type>>
Rate
+ ratePrice: double
+ dollarConversion: double
<<Type>>
CancellationCoverRate
+ Id: long
+ Code: String
+ Name: String
+ Value: BigDecimal

Figure 2. Required interface and candidate service interface for the Car Rental example

foremost based in structural conditions for some signature elements – namely, return type, parameter types and exceptions. Notice that elements are named according to Java terminology, rather than using WSDL conventions for Web Service interfaces. The reason is that structural assessment is performed upon Java interfaces previously derived from WSDL specifications. The main aspects of structural assessment are presented below. For details of the structural assessment, interested readers could refer to [4].

4.1 Return Type

Conditions for return type equivalence involves the subsumes relationship or subtyping, which implies a direct subtyping in case of built-in types in the Java language [24], as shown in Table 1. It is expected that types on operations from a *required interface* have at least as much precision as types on operations from a *candidate service*. For example, if $op_R \in I_R$ includes an `int` type, a corresponding operation $op_S \in I_S$ should not have a smaller type (among numerical types) such as `short` or `byte`. However, the `String` type is a special case, which is considered as a wildcard type – it is generally used in practice by programmers to allocate different kinds of data [25]. Thus, we consider `String` as a supertype of any other built-in type.

Complex data types imply a special treatment in which the comprised fields must be equivalent one-to-one with fields from a counterpart complex type. This means, each field of a complex type from an operation $op_R \in I_R$ must match a field from the complex type in $op_S \in I_S$ – though extra fields from the latter may be initially left out of any correspondence. This means, if all fields from the required complex type in $op_R \in I_R$ are fulfilled, then extra fields in $op_S \in I_S$ do not necessarily

op_R **type**	op_S **type**
char	string
byte	short, int, long, float, double, string
short	int, long, float, double, string
int	long, float, double, string
long	float, double, string
float	double, string
double	string

Table 1. Subtype equivalence

cause incompatibility. However, the input data to fulfill those extra fields should be adequately defined to allow the proper execution of the service [26, 27].

The return type similarity value is calculated according to the following cases:

- $Ret = 1$: Equal Return Type.
- $Ret = 0.66$: Equivalent Return Type (Subtyping, String or Complex types).
- $Ret = 0.33$: Non-equivalent complex types or precision loss.
- $Ret = 0$: Not compatible.

Example 1. Figure 3 shows the field-to-field equivalence (considering only data types) for two complex types of the Car Rental example, which contains information about booking cancellation rates. The three fields of the `CancelInformation` type have a one-to-one correspondence with three fields of the `CancellationCoverRate`. The dotted arrows indicate a likely correspondence between the `String` types. For this example the return type similarity value is $Ret = 1$. Notice that the extra field in `CancellationCoverRate` does not cause an incompatibility since all the required fields in `CancelInformation` were properly matched.

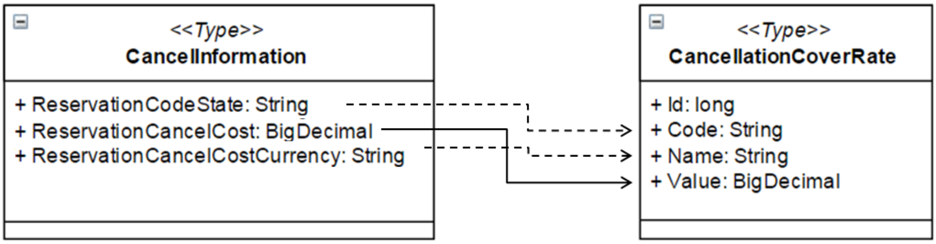


Figure 3. Equivalence of complex data types for the CarRental example

4.2 Exceptions

Any operation op_R may define *default* exceptions – i.e., using the `Exception` type – or ad-hoc exceptions. Likewise, an operation op_S from a candidate service may define a *fault* (the WSDL name for non-standard outputs of operations) as a message including a specific attribute. The exceptions similarity value is calculated according to the following cases:

- $Exc = 1$: op_R and op_S have equal amount, type and order for exceptions.
- $Exc = 0.66$: op_R and op_S have equal amount and type for exceptions into the list.
- $Exc = 0.33$: If op_R defines exceptions, then op_S defines at least one exception.
- $Exc = 0$: Not compatible, since op_R defines exceptions and op_S does not define exceptions.

Interestingly, in the context of Web Services, *faults* definitions have not become a common practice [28, 29]. Even so, exceptions are evaluated in the order that they are specified into an operation signature, being desirable that compatible exceptions appear in the same position of exceptions lists ($Exc = 1$).

Example 2. Let us consider the following operations for obtaining rates for Car Rental, according to different vehicles and conditions:

```
getCarFee(requiredCarSupplements: CarSupplements): Fee
throws unavailableSupplementsException;

getRate(currencyCode: String, vehicleTypeId: long,
AutomaticGearPreference: boolean): Rate
throws vehicleNotFoundException, rateNotFoundException;
```

Being `getCarFee` as $op_R \in I_R$ and `getRate` as $op_S \in I_S$, respectively, the required operation throws an exception that may have two likely exceptions (from different types) in the I_R operation. Then the exceptions similarity value is $Exc = 0.33$.

5 SEMANTIC ASSESSMENT

We developed three alternatives for the semantic similarity basis and criteria: two of them based on WordNet and the third one based on DISCO. WordNet is a widely used lexical database for the English language that is structured as a lexical tree. WordNet groups terms in *synsets* (synonym sets) that represent the same lexical concept. Several relations connect different synsets, such as hyperonymy/hyponymy, holonymy/meronymy and antonymy. We make use of two Java libraries to manipulate the WordNet database: JWI[4] and JWNL[5]. These libraries provide different criteria for analyzing semantic equivalence of terms. They also provide the most complete and easy Java-based access to the WordNet lexical tree, according to a recent comparative work [8]. JWI is more focused on performance, usability and

[4] Java WordNet Interface: `http://projects.csail.mit.edu/jwi/`

[5] Java WordNet Library: `http://sourceforge.net/projects/jwordnet/`

flexibility; whilst JWNL is more focused on offering a wide variety of similarity metrics implementation.

DISCO [9] is a pre-computed database of collocations and distributionally similar words over very large text collections (e.g., Wikipedia). DISCO's Java library[6] allows retrieving the semantic similarity between arbitrary words. For each word, DISCO stores the first and second order vectors of related words using a Lucene[7] index. To analyze the similarity between two words, DISCO retrieves the corresponding word vectors from the index and computes the similarity according to a similarity formula based in co-occurrence functions.

In the following sections we describe the Identifiers Evaluation Algorithm implemented using WordNet and DISCO as semantic basis. First, two identifiers under evaluation are given as input for three different pre-processing steps, namely term separation, stop words removal and stemming [5].

Term Separation. Identifiers are normally restricted to a sequence of one or more letters in ASCII code, numeric characters and underscores ("_") or hyphens ("-"). The algorithm supports the usual programming naming conventions, such as text casing or hyphenation [30]. Additionally, a semantic level is applied to recognize identifiers that do not follow those conventions, recognizing potential terms (uppercase sequences and lowercase sequences) using WordNet. *Term Separation* step is crucial to consider the correct terms as input to the semantic analysis.

Example 3. Let us consider the identifier `GDSCode` from the Car Rental domain. This identifier does not strictly follow the Java Beans notation. An initial analysis identifies an uppercase sequence (`GDSC`), and a lowercase sequence (`ode`). Then, the sequence `C + ode = Code` is also given as input to WordNet. As it is an existing word in the WordNet dictionary, `Code` is considered as a term and `GDS` as an acronym (an abbreviation of Global Distribution System) that is also considered as a term.

Stop Words Removal. Stop words are meaningless words that are filtered out when processing natural language data (text) [31]. By removing symbols and stop words our approach attempts to "clean" service interface descriptions, as a simple but useful technique for filtering non-relevant terms. We defined a stop words list containing articles, pronouns, prepositions, each letter of the alphabet, and words from other stop words lists[8, 9], commonly used in natural language processing tools [10].

Example 4. Let us consider the identifier `AgencyHandledBy` which corresponds to a field in the data type `AgencyData` of the Car Rental example. According to the Term Separation step, the identifier is decomposed in three terms: [`Agency`,

[6] `http://www.linguatools.de/disco/disco-download_en.html`

[7] `https://lucene.apache.org/core/`

[8] `http://www.csee.umbc.edu/courses/691p/code/wf/stop_words.txt`

[9] `http://algs4.cs.princeton.edu/35applications/stopwords.txt`

`Handled, By`]. As `'By'` belongs to the stop words list, it is removed, obtaining: [`Agency, Handled`].

Stemming is the process for reducing inflected or derived words to their stem or root form [32]. As standard stemmers (e.g., Porter's algorithm) usually generate incorrect data (incurring in understemming or overstemming [33]), we decided to leverage the semantic stemming facilities provided by WordNet. For each term, the *Stemming* algorithm verifies if it belongs to the WordNet dictionary. If it does so, their corresponding stems are added to the result list. Otherwise, the original term is added to the result list, considering that it can be an acronym or abbreviation.

These techniques are well-known and widely adopted in the field of IR, particularly in text mining [34]; and also in recent work in natural language processing in the context of Web Services [35], improving the precision of any text/document analysis approach [36, 31]. In the context of this work, the preprocessing techniques are combined with other semantic techniques and the structural assessment of data types. Thus, their final impact is not directly observable in a quantitative validation comprising hundreds of services.

5.1 WordNet-Based Identifiers Evaluation with JWI

After the preprocessing steps described above, the information to calculate the compatibility value of two terms lists must be extracted using the JWI library. This information includes:

- $terms = \#(\{termsOp_R\} \cup \{termsOp_S\})$: total terms between both terms lists (counting only once duplicated terms).
- *exact*: Number of identical terms.
- *syn*: Number of synonyms (words with the same meaning) of the op_R terms belonging to the op_S terms list.
- *hyper*: Number of hyperonyms of the op_R terms belonging to the op_S terms list.
- *hypo*: Number of hyponyms of the op_R terms belonging to the op_S terms list.

To evaluate synonyms, hyperonyms and hyponyms we considered the following aspects:

1. A single level of hypo/hyperonymy; i.e., direct hypo/hyperonymy, since the most distant hypo/hyperonyms have less to do with the meaning of original terms.
2. Total synonymy, where two terms are synonyms if they are interchangeable in the same context without affecting the semantics. In the context of WordNet, this implies that two terms are synonyms if they are grouped in the same *synset*.

Example 5. Let us consider the identifiers `GetReservation` and `GetCurrentBooking` extracted from the Car Rental example. According to the semantic comparison, these identifiers present four distinct terms (`Get`, `Reservation`, `Current`, `Booking`), one identical term `(Get)` and one synonym (`Reservation`, `Booking`).

Using the semantic information extracted from identifiers in the previous step, identifiers compatibility is calculated according to Formula (1), as the weighed average of the compatibility among all terms in the two term lists under analysis. This is calculated according to the notions of semantic relatedness of terms described above (exact or identical terms, synonyms and hypo/hyperonyms). Notice that the formula assigns for hypo/hyperonyms half the weight assigned for exact terms and synonyms, because the former convey a meaning that is similar but not exactly the same, while the latter convey an identical meaning. This notion of weighting the terms according to their relative importance has been widely used in the field of IR [31], and particularly in the context of Web Services similarity [10, 37].

$$identComp = \frac{exact + syn + 0.5 * (hypo + hyper)}{terms - syn}. \tag{1}$$

Example 6. Let us consider the identifiers `GetReservation` and `GetCurrentBooking` extracted from the Car Rental example. By replacing the values in Formula (1), we obtain:

$$identComp = \frac{1 + 1 + 0.5 * (0 + 0)}{4 - 1} = \frac{2}{3} = 0.66.$$

Considering that the maximum value for *identComp* is 1, the obtained value of 0.66 between identifiers `GetReservation` and `GetCurrentBooking` indicates a moderate to strong compatibility.

5.2 WordNet-Based Identifiers Evaluation with JWNL

First, a matrix named Normalized Depth matrix (*ND* matrix) is generated. This matrix contains the normalized depth between each term from both terms lists. Thus, the cell nd_{ij} from the *ND* matrix contains the normalized depth between the i^{th} term of an identifier in op_R and the j^{th} term of an identifier in op_S – according to the WordNet hierarchy. The depth is defined as the shortest path between two terms in the WordNet hierarchy. These values are normalized according to the maximum depth of the WordNet tree D (which currently is 16).

Formally, the normalized depth is calculated according to Formula (2), where $length(t_i, t_j)$ is the shortest path between t_i and t_j in the WordNet hierarchy, and D is the maximum tree depth (16). $NormalizedDepth(t_1, t_2)$ ranges from 0 (when t_i and t_j are completely unrelated) to 1 (when t_i and t_j are identical or synonyms).

$$NormalizedDepth(t_i, t_j) = 1 - \frac{length(t_i, t_j)}{2D}. \tag{2}$$

Example 7. Accounting this notion of length, let us consider the two identifiers `GetReservation` and `GetCurrentBooking` (analyzed in Section 5.1). The *ND* matrix will be a 2×3 matrix containing the length between each pair of terms in the identifiers, as shown in Table 2. Notice that ND(`Reservation`,`Booking`) $= 1$ since these terms are synonyms in the WordNet hierarchy (their path length is zero).

$op_R \downarrow op_S \longrightarrow$	`Get`	`Current`	`Booking`
`Get`	1.00	0.56	0.72
`Reservation`	0.72	0.72	1.00

Table 2. Normalized Depth matrix for a pair of identifiers

After calculating the *ND* matrix, the best term matching (among all possible pair-wise combinations) must be selected – i.e., the combination of terms from both terms lists that maximizes their compatibility. Each possible term matching consists of taking each row of the *ND* matrix (i.e., a term of the first list) and choosing a column (i.e., a term of the second list), without repeating columns (i.e., matching each term only once). Thus, it is ensured that each term from the first list corresponds to one and only one term for the second list. The similarity values are obtained from the corresponding cells of the *ND* matrix.

The total value of each possible term matching is the sum of all pair-wise assignments that compose it (*assignSum*). The matching with the highest value is obtained through the Hungarian method [38], which models the allocation problem as a cost matrix $n \times m$ – in this case n and m are the number of terms of both terms lists.

Finally, the identifiers compatibility value *identComp* is calculated through Formula (3), which is the average of the pair-wise assignments of terms *assignSum*, according to the maximum number of terms in the identifiers under analysis $\max(n, m)$ where n and m are the number of terms in both term lists. The denominator determines the number of simultaneous assignments; i.e., if the maximum number of terms from both lists is 3, the *assignSum* will consist of three pairs of terms and will be averaged accordingly. Formula (3) is the alternative to Formula (1) to calculate identifiers compatibility (*identComp*) when using JWNL instead of JWI.

$$identComp = \frac{assignSum}{\max(n, m)}. \tag{3}$$

Example 8. Considering the *ND* matrix shown in Table 2, the term matching that maximizes the compatibility between the identifiers consists of the following pair-wise assignments:

- [`Get`,`Get`], stored in the cell $nd_{1,1} = 1.00$
- [`Reservation`, `Booking`], stored in the cell $nd_{2,3} = 1.00$

Then, replacing the corresponding values in Formula (3), the compatibility value between identifiers `GetReservation` and `GetCurrentBooking` is calculated as:

$$identComp = \frac{1+1}{max(2,3)} = \frac{2}{3} = 0.66.$$

Considering that the maximum value for *identComp* is 1, the obtained value of 0.66 between identifiers `GetReservation` and `GetCurrentBooking` indicates a moderate to strong compatibility. This ratifies the compatibility value obtained in Section 5.1 when using WordNet through JWI.

5.3 DISCO-Based Identifiers Evaluation

First, a matrix named Co-occurrences matrix (*Co* matrix) is generated. This matrix contains the similarity values between each term from both terms lists. Thus, the cell Co_{ij} from the *Co* matrix contains the similarity value between the i^{th} term of an identifier in op_R and the j^{th} term of an identifier in op_S – according to their co-occurrences in the indexed text collections. The similarity notion of DISCO was introduced in Section 5.

Example 9. Let us consider the pair of identifiers of the previous section – namely `GetReservation` and `GetCurrentBooking`. The *Co* matrix will be a 2×3 matrix containing the co-occurrence values between each pair of terms in the identifiers, as shown in Table 3. Notice that, when using DISCO rather than WordNet, synonyms do not present a co-occurrence value of 1 – as can be seen for the pair (`Reservation`, `Booking`).

$op_R \downarrow op_S \longrightarrow$	`Get`	`Current`	`Booking`
`Get`	1.00	0.006	0.02
`Reservation`	0.01	0.01	0.1

Table 3. Co-occurrences matrix for a pair of identifiers

After calculating the *Co* matrix, the best term matching (among all possible pair-wise combinations) must be selected – i.e., the combination of terms from both terms lists that maximizes their compatibility. Similarly to the JWNL-based implementation, for each possible pair-wise term assignment between both lists, the similarity value is obtained from the corresponding cell of the *Co* matrix. The value of each possible term matching is the sum of all pair-wise assignments that compose it. The matching with the highest value will be the most compatible. Such matching is also obtained through the Hungarian method – introduced in Section 5.2. Finally, the identifiers compatibility value using DISCO is also calculated according to Formula (3).

Example 10. Considering the *Co* matrix shown in Table 3, the term matching that maximizes the compatibility between the identifiers consists of the following pair-wise assignments:

- [`Get`,`Get`], stored in the cell $co_{1,1} = 1.00$
- [`Reservation`, `Booking`], stored in the cell $co_{2,3} = 0.1$

Then, replacing the corresponding values in Formula (3), the compatibility value between identifiers `GetReservation` and `GetCurrentBooking` is calculated as follows:

$$identComp = \frac{1 + 0.1}{max(2,3)} = \frac{1.1}{3} = 0.36.$$

A potential issue that may arise when using DISCO is the generation of apparently inconsistent results, since words with very different meaning will appear closely related. For example, *Casino* and *Vegas* clearly are words with different meaning. However, these words are closely related according to DISCO, since they co-occur in many indexed documents. This is mitigated by the supplementary structural evaluation.

5.4 Parameters Evaluation

The algorithm for Parameters Evaluation consists of calculating three matrices: Type matrix (T), Name matrix (N) and Compatibility matrix (C). For the three matrices, the cell m_{ij} represents the compatibility value between the i^{th} parameter of $op_R \in I_R$ and the j^{th} parameter of $op_S \in I_S$.

The T matrix contains the structural compatibility value among parameter types from op_R and op_S. The notions of structural data type equivalence and subtyping are used to assess parameter types. A cell t_{ij} contains the compatibility value between the i^{th} parameter's type of op_R and the j^{th} parameter's type of op_S, according to Formula (4), where two identical types have a compatibility value of 1; a subtyping relationship (represented as $<:$) of parameters results in a compatibility value of 0.75 and two different parameter types are given a compatibility value of 0.5. The ranges were empirically determined according to previous experience and related work in structural assessment of service interfaces, as detailed in [4].

The last value $t_{ij} = 0.5$ is intended to represent a low structural compatibility, but it does not determine an incompatibility (given by $t_{ij} = 0$) since it has to be weighted with the semantic analysis, as described below.

$$t_{ij} = \begin{cases} 1 & \mathit{Type}(P_i) = \mathit{Type}(P_j), \\ 0.75 & \mathit{Type}(P_i) <: \mathit{Type}(P_j), \\ 0.5 & \text{otherwise.} \end{cases} \tag{4}$$

The N matrix contains the compatibility values between the name of each parameter from op_R and the name of each parameter from op_S. The underlying rationale is similar to the T matrix. The cell n_{ij} contains the compatibility value between the i^{th} parameter's name of op_R and the j^{th} parameter's name of op_S. This value is the result of applying the Identifiers Evaluation Algorithm – thus depends on the chosen implementation for the Semantic Assessment.

Then, the C matrix is generated from the T and N matrices. The goal of the C matrix is to store the compatibility value between all parameter pairs from operations op_R and op_S, considering structural and semantic aspects – collected in the T matrix and the N matrix respectively. Each cell c_{ij} stores the product between corresponding cells t_{ij} and n_{ij}. Thus: $c_{ij} = t_{ij} * n_{ij}$.

After calculating the C matrix, the best parameter matching (among all possible pair-wise combinations) must be selected – i.e., the combination of parameters from op_R and op_S that maximizes their compatibility. This step applies the Hungarian method to calculate the best pair-wise parameters assignments – similarly to the term matching maximization in JWNL-based Identifiers Evaluation (Section 5.2).

5.5 Outcomes

The final outcome of the Interface Compatibility procedure is an *Interface Matching* list, where each correspondence is characterized according to the compatibility values obtained from the structural and semantic analysis of signature elements. For each operation $op_R \in I_R$, a list of compatible operations from I_S is obtained, ordered by their compatibility degree. For example, let I_R be with three operations op_{Ri}, $1 \leq i \leq 3$, and I_S with five operations op_{Sj}, $1 \leq j \leq 5$. After the procedure, the Interface Matching list might result as follows:

$$[(op_{R1}, \{op_{S1}, op_{S5}\}), (op_{R2}, \{op_{S2}, op_{S4}\}), (op_{R3}, \{op_{S3}\})].$$

An additional aspect can be highlighted from the Interface Compatibility procedure. Considering the structural and semantic assessments presented in the previous sections, an appraisal value can reflect the required adaptation effort to integrate a candidate service into a Service-oriented Application under development. For this, we defined the Formula (5) *adaptability value*, which represents the achieved compatibility between assessed interfaces, as the average of the best adaptability values obtained for all required operations. Only the strongest equivalences (higher values) from the Interface Matching list are considered for each operation in I_R. In the formula, N is the interface's size of I_R, and $\max(AdapOp)$ is the best value for $op_{Ri} \in I_R$ from the list of equivalence values $adapOpValue(op_{Ri}, op_{Sj})$ with $op_{Sj} \in I_S$.

$$adapVal(I_R, I_S) = \frac{\sum_{i=1}^{N}(\mathrm{Max}(AdapOp(op_{Ri}, I_S)))}{N}. \qquad (5)$$

The operation adaptability value ($adapOpValue$) between an operation op_R and a potentially compatible operation op_S is calculated according to Formula (6).

The *identComp* value is used in *adapOpValue* to calculate the compatibility between identifiers in service descriptions, according to the alternative semantic basis discussed in previous sections (Formulas (1) and (3)). The *identComp* value is used as-is to calculate operation names compatibility (*nameComp*) and parameters compatibility (*paramComp*), which also considers structural aspects (described in Section 5.4). Meanwhile, *Ret* and *Exc* are the structural equivalence values for return type and exceptions respectively.

$$adapOpValue(op_R, op_S) = Ret + Exc + nameComp + paramComp. \qquad (6)$$

The highest (better) adaptability value is 4, since the highest value for *Ret*, *Exc*, *nameComp*, *paramComp* is 1. The *adaptability value* synthesizes the structural and semantic information gathered from interfaces in a single numeric value. Although all operations in the Interface Matching list presenting an exact or identical equivalence may resemble to a perfect interface match, this initially means that I_R is included into I_S. The size of interface I_S of the candidate service may be larger, including additional operations. The alternatives for semantic assessment are weighted with the structural analysis, which prevents spurious compatibilities between terms in operations by also considering the structure of data types, parameters and return types. Finally, the precision achieved during the Interface Compatibility procedure is essential to reduce the subsequent adaptation effort when integrating a candidate service into a Service-oriented Application.

Example 11. Let us consider the full interfaces of the Car Rental example, presented in Section 3.1. Table 4 shows, for each operation in the required interface `CarRentalBrokerService`, the operation with higher compatibility (according to their *adapOpValue*) in the interface of the candidate `RentaCar`. Calculations were done using the semantic basis accessed through the JWI, JWNL and DISCO, respectively.

Notice that operations `getCarFee` and `getRate` obtained the lower (worse) *adapOpValue*. This indicates a low compatibility between those operations in terms of adaptability, due to the complexity of their signatures (see Figure 2) featuring complex return types, multiple parameters and ad-hoc exceptions, which may hinder their seamless integration.

As the better adaptability value is 4, the obtained *adaptabilityValue* (2.78, 2.69 and 2.48) with the three implementations can be considered as moderate to high – as shown in Table 4. This suggests an easy adaptation of the `RentaCar` service to be integrated into the client application, considering the required interface `CarRentalBrokerService`. However, we should notice that this example was designed only to illustrate our approach and the compatibility was straightforward. Real-world scenarios are usually more complex in terms of number of candidate services and potentially compatible operations, complex data types and parameters. Therefore, experiments with large datasets of real-world Web Services are presented in the following section.

CarRentalBrokerService (I_R)	RentaCar (I_S)			
	Compatible Operation	*AdapOpValue*		
		JWI	JWNL	DISCO
`getReservation`	`getCurrentBooking`	2.97	3.08	2.76
`getCarFee`	`getRate`	2.41	1.83	1.79
`cancelReservation`	`cancelBooking`	2.96	3.17	2.89
	Adaptability Value:	2.78	2.69	2.48

Table 4. Operations matching for the Car Rental example

6 EXPERIMENTAL EVALUATION

In the first experiment (Section 6.1), we executed the Interface Compatibility procedure with the two different underlying concept hierarchies over a data-set of approximately 500 services extracted from the literature. Then we compared the results of the executions against the original discovery results for the same dataset, and a foundational algorithm for structural-semantic Web Services discovery, the Stroulia and Wang algorithm [10]. The goal of the first experiment is to compare the visibility of suitable candidate services when using the Interface Compatibility procedure with the different underlying concept hierarchies, and w.r.t. a well-known algorithm for Web Service discovery and selection. We then performed a second experiment with a similar goal (Section 6.2) but a different experimental configuration and dataset, consisting of approximately 1 000 services crawled from the Mashape.com Web repository.

In the third experiment (Section 6.3), we executed the Interface Compatibility procedure based on WordNet (through JWI) replicating the experimental settings of Bouchiha et al. [11], which tested two algorithms for categorization and matching of Web Services based in domain ontologies and WordNet. The goal of this experiment was to compare the performance of the Interface Compatibility procedure in a third-party designed experiment, w.r.t. ontology-based algorithms for categorization and matching of Web Service specifications.

6.1 Experiment 1 – Interface Compatibility and the Stroulia Algorithm

We used a data-set that comprised 479 services extracted from the literature [39, 40, 4], from which we randomly selected 65 services as *target* services, and the rest as *noise* services. The only condition for the target services is that their Java interface could be generated automatically using the library WSDL2Java[10]. The remaining services were used as 'noise' to increase the size of the experiment in terms of the amount of WSDL specifications in the registry.

A set of 531 queries was automatically generated from the operations of the 65 target services of the data-set. Notice that noise services could also include these operations (or very similar ones) as well. Then we applied interface mutation techniques [41] (Section 6.1.1) to the operations in the queries, to generate different structural information by combining signature elements (return types, parameters). Mutation operators were applied in a random and probabilistic way.

Methodologically, the execution of the first experiment consisted in the following steps. First, the data-set of 479 WSDL specifications was used to populate the EasySOC service discovery registry [37]. EasySOC provides a simple support for service discovery based in document matching techniques, leveraging only syntactic aspects of queries and service specifications. Thus, their results provide a baseline to compare other comprehensive service selection approaches. Thereby, we queried

[10] `http://axis.apache.org/axis2/java/core/tools/CodegenToolReference.html`

that registry with the 531 queries. Then, the registry returned the 10 most relevant services for each query (potentially containing the target service for the query), according to a simple syntactical evaluation.

Then, upon discovery results, the Interface Compatibility procedure presented in Section 3 was executed using the three different lightweight semantic basis to compare the Java interfaces of services retrieved by the discovery registry against the mutated queries as required interfaces. We also implemented the WordNet-based Stroulia algorithm [10] that, given a query, identifies and ranks the most relevant WSDL specifications for that query, by recursively analyzing data types, messages and operations through a structural IR-based method. This set of candidates can be further refined by a semantic matching step based on a Vector Space Model (VSM) [42] and WordNet to expand queries, obtaining the overall matching score between a WSDL service specification and a query as a weighted mean from both steps (interested readers could refer to [10]).

Finally, we compared the results in terms of visibility of target services (Section 6.1.3) according to a suite of metrics from the IR field – *Precision-at-n*, *Recall*, *F-Measure* and *NDCG* (Normalized Discounted Cumulative Gain). Details of the experimental procedure are given in the following sections.

6.1.1 Query Set Generation

The original set of queries consisted in the operations extracted from target services. Then, we implemented an interface mutator[11] to generate different structural information according to four different mutation operators. Interface Mutation is a technique to evaluate how well the interactions between various units have been tested [43]. When applying Interface Mutation, unlike traditional mutation, the changes are made only at the interface related points or connections between units. In this experiment, such points are the parameters and the return types from the queries. Four different mutation operators were probabilistically applied to process the queries prior to the execution of the service selection algorithms, generating different data types and parameters for each query.

Encapsulation. A random number of parameters are encapsulated as fields of a new complex data type. The name of the complex data type is the concatenation of the name of the operation and the word "Request". The name of the complex parameter is a concatenation of the encapsulated parameters.

Example 12. Let us consider the operation `AddUser(String userName, String password, int sessionId)`, the first two parameters can be encapsulated as `AddUser(AddUserRequest userName_password, int sessionId)`, where the complex type `AddUserRequest` is composed of two `String` fields – namely `userName` and `password`.

[11] `http://code.google.com/p/querymutator/`

Flatten. A random number of complex type parameters are flattened, generating as many parameters as fields in the complex type. The resulting parameters can be primitive or complex, according to the types of the fields in the original data type.

Example 13. Let us consider the operation `AddUser(UserData userData, int sessionId)` with the complex type `UserData` containing two `String` fields (`user` and `password`) the mutated interface after applying the flatten operator could be: `AddUser (String user, String password, int sessionId)`.

Upcasting. The return type and/or a random number of parameters of numeric types are upcast to a direct numeric supertype in the Java language, according to Table 5.

Type	Direct Supertype
byte	short
short	int
int	long
long	float
float	double

Table 5. Built-in direct supertyping for Java

Wildcard Supertype. The return type and/or a random number of parameters are cast to the wildcard supertype `String`, including `void` return types. As stated earlier, the `String` type is generally used as a wildcard in practice to allocate different kinds of data.

Example 14. Let us consider the operation `void AddUser(String userName, String password, int sessionId)`, the Wildcard Supertype operator could generate `String Add User (String userName, String password, String sessionId)`, where both the return type and the parameter `sessionId` have been cast to `String`.

In a previous work [4], we manually expanded the syntactic queries to include structural information. For the present experiment, the interface mutator enables an automatized and probabilistic application of the mutation operators, generating queries with unique structural characteristics. Also, mutated queries are similar in a greater or lesser extent to operations defined by the services in the dataset – since they are based on original operations of such services. This emulates the (potentially partial) specification of the desired functionality that developers may define when looking for suitable services. After applying the mutation operators, each mutated query can be encapsulated as a Java (required) interface with only one operation, which acts as the required interface (I_R).

6.1.2 Experiment Execution

We queried the EasySOC registry with the syntactic queries. Then, obtained results were post-processed by executing the WordNet and DISCO-based Interface Compatibility procedures, and the Stroulia-based algorithm. We considered an initial ranked list of the first 10 services retrieved by the EasySOC discovery registry per query. When the target service is not retrieved by the discovery registry, it is given in the 11th position as input for the other algorithms. Thus, the goal of this experiment is to analyze how the Interface Compatibility procedure could improve the visibility of a suitable candidate service in a result list, to be selected for its integration into a Service-oriented application.

Example 15. Let us consider the query `getUser` extracted from service `AccountingService`. Thus, `AccountingService` is the target service for that query, and is expected to be retrieved in the topmost positions. Then, the results list (ordered by position) retrieved by the EasySOC service discovery registry could be:

{`(1, VomsAdminService), (2, VomsTrustedAdminService), (3, Service6.Accounts), (4, Service7.Accounts), (5,` *AccountingService*`),...`}.

As can be seen, the target service was retrieved in the fifth position. The first four services also provide an operation named `getUser`, which satisfies the syntactic query. Through the different service selection methods, this list is rearranged considering structural and semantic information both from the mutated query and the services in the list. Thus, the target service is ranked in a better position of the list – e.g., among the first three positions. In this case, the re-ordered list for a certain service selection method could allow the target service to be retrieved in the second position, as follows:

{`(1, VomsAdminService), (2,` *AccountingService*`), (3, VomsTrustedAdminService), (4, Service6.Accounts), (5, Service7.Accounts), ...`}.

6.1.3 Results

We compared results according to four well-known IR metrics: *Precision-at-n*, *Recall*, *F-Measure* and *NDCG* (Normalized Discounted Cumulative Gain). These metrics have been broadly used in the context of Web Service discovery, selection and semantic matching [44, 11, 4, 10]. Considering the results for the 531 queries, an average of each metric was generated. Figure 4 depicts the cumulative average precision-at-n values corresponding to the original discovery registry EasySOC, the Interface Compatibility procedures and the Stroulia-based algorithm. Results show that:

- The WordNet-based (JWI) Interface Compatibility procedure increases the precision-at-n value between 19 % and 34 % for the first three positions of the list (with $n \in [1, 3]$) w.r.t. original results from the discovery registry.

- The WordNet-based (JWNL) Interface Compatibility procedure increases the precision-at-n value between 24 % and 38 % for the first three positions of the list (with $n \in [1,3]$) w.r.t. original results.

- The DISCO-based Interface Compatibility procedure increases the precision-at-n value between 25 % and 40 % for the first three positions of the list (with $n \in [1,3]$) w.r.t. original results.

- The Stroulia-based algorithm increases the precision-at-n value between 2 % and 4 % for the first three positions of the list (with $n \in [1,3]$) w.r.t. original results.

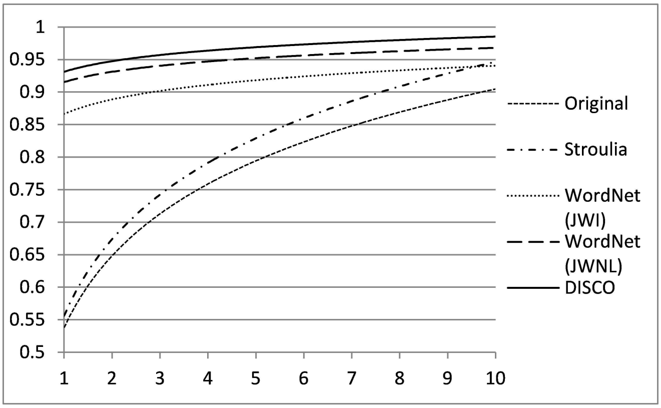
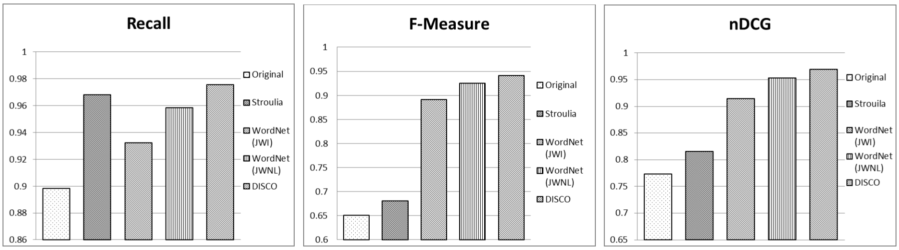


Figure 4. Precision-at-n for original discovery registry, Interface Compatibility, and the Stroulia algorithm

Figure 5 a) shows the Recall result, where the Interface Compatibility procedures outperformed original results by 3 % to 8 %, and the Stroulia-based algorithm outperformed original results by 7 %. All algorithms presented high recall, over 90 %.

Figure 5 b) shows the F-Measure results, where the Interface Compatibility procedures outperformed original results by 23 % to 29 %, and the Stroulia-based algorithm also outperformed original results by 3 %, being the only selection algorithm under 85 % for F-Measure.

Figure 5 c) shows the NDCG result, where the Interface Compatibility procedures outperformed original results by 14 % to 19 %, and the Stroulia-based algorithm outperformed original results by 4 %. All algorithms presented competitive values for DCG, over 80 %. The discussion of these results is presented after the next experiment.



a) Recall b) F-Measure c) NDCG

Figure 5. Results for original discovery registry, Interface Compatibility, and the Stroulia algorithm

6.2 Experiment 2 – Interface Compatibility with the Mashape Dataset

In the second experiment, we used another data-set comprised of 1 146 services crawled from the Mashape[12] service repository. Unlike the previous experimental scenario, queries were not automatically generated by mutation, but rather were constructed by expert software engineers. The WSDL specifications were given to two groups of engineers. Each group randomly selected 15 services as target services to generate the queries. Then, these experts exchanged their services and completely rewrote the signature of the operations in the selected services. They were asked to preserve the intended semantics of the operations while altering the identifiers (i.e., names from operations, parameters and fields of complex types, among others) and data types (in parameters, return and fields). Then, they splitted these operations in groups of one, two or three operations, to generate the queries. Thus, each query consisted of up to three related operations, in contrast to the single operation queries in the previous experiment. For example, if a target service consisted of 7 operations, the experts might have generated three queries with 3, 3 and 1 operations; or 3, 2 and 2 operations. From this process of manual query generation we obtained a total of 42 fully described, multi-operation queries. Each query was associated with the original service from which it was generated as its target service (thus having a gold-standard for evaluation).

We used two versions of the EasySOC service discovery registry as baseline: the VSM-based original version [45] and an enhanced version featuring query expansion [46, 37]. Both registries were populated with the 1 146 services in the dataset. From the 42 multi-operation queries from experts, a corresponding set of syntactic queries was generated as input to the service discovery registries, by concatenating operation names. Finally, the Interface Compatibility procedure used for this experiment was the WordNet (JWI) based implementation.

Figure 6 depicts the cumulative average precision-at-n values corresponding to both versions of EasySOC, and the Interface Compatibility procedure. The

[12] `http://www.mashape.com`

WordNet-based (JWI) Interface Compatibility procedure increases the precision-at-n value between 19 % and 34 % for the first three positions of the list (with $n \in [1, 3]$) w.r.t. original results from the discovery registries. With regard to recall, Interface Compatibility obtained a 93 % while original and enhanced EasySOC versions presented 82 % and 84 %, respectively.

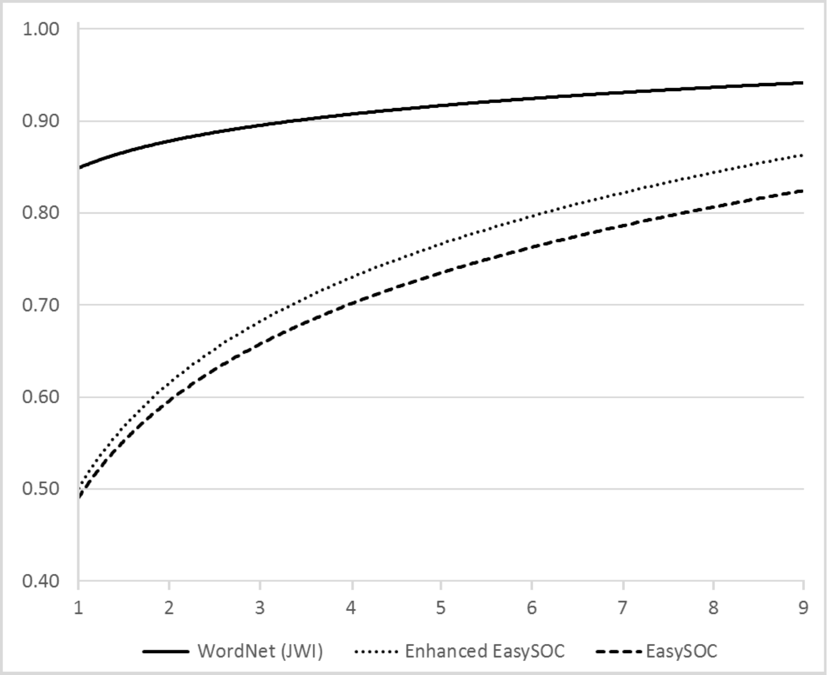


Figure 6. Precision-at-n for the discovery registry (original and enhanced) and Interface Compatibility

Discussion. The results show that the proposed Interface Compatibility procedures increase the visibility of the most suitable candidate services, independently of the adopted underlying semantic basis. The higher values for precision, recall, F-Measure and NDCG support such improvement. The comparison with the classic Stroulia-based algorithm shows encouraging results as well. The Stroulia-based algorithm did not present significant improvements over original results. The reason for this is twofold. First, EasySOC already uses some concepts of the vector space model to represent Web Service descriptions and queries, making the EasySOC registry more efficient with respect to traditional discovery methods. Thus, the advantages of using VSM in the Stroulia-based algorithm are overlapped in the service discovery step of the experiment. Second, as stated by the original authors of the Stroulia algorithm, the similarity assessment methods developed are neither precise nor robust enough to discover the desired service without a developer's intervention [10].

For the first experiment, the DISCO-based Interface Compatibility procedure outperformed other approaches. In fact, the three implementations of Interface Compatibility performed competitively. Particularly, precision-at-n greatly improves for the most relevant results – i.e., 1^{st} to 3^{rd} position of results lists. Although precision tends to converge when n approaches to 10, the first positions are the most significant since users tend to select higher ranked search results, regardless of their actual relevance [47].

For the second experiment, we used a different dataset consisting of more than 1 000 services crawled from the Mashape repository. Also, the query generation approach introduced two groups of software engineers to manually generate multi-operation queries. Results with this configuration confirmed previous insights, showing an improvement both in precision and recall in the first positions of results lists.

It is important to notice that the experiments are empirical, thus the obtained results can be specific for the data-set and queries used, and cannot be merely generalized "as-is" to other experimental configurations. However, the empirical validation is a common practice for the knowledge area of Web Service discovery and selection. Similar experimental configurations and datasets were used not only in foundational papers [10, 39, 45] but also in recent papers in the field [4, 48, 46, 17].

6.3 Experiment 3 – Interface Compatibility and Ontology-Based Categorization and Matching

In the third experiment, we compared the WordNet-based (JWI) Interface Compatibility procedure against the experimental results obtained by Bouchiha et al. in [11] for categorization and matching of Web Services.

Categorization and matching algorithms are based in domain ontologies. Ontologies represent the semantic resources with which WSDL descriptions can be annotated. The annotation process relates WSDL descriptions with concepts in ontologies. Annotation poses several problems, e.g., finding the relevant ontology or ontologies, and matching large Web Service descriptions to the ontology vocabulary. Because of these factors, it is necessary to have a semi-automatic and scalable way for recognizing the functional category (domain) of a given Web Service description and then annotating such description with real world ontologies – namely, categorization and matching algorithms.

Categorization algorithms aim to classify WSDL service descriptions according to their corresponding domain – represented by an ontology. To do this, the fundamental WSDL elements are identified (XSD data types, interface, operations and messages) and then compared with concepts in different domain ontologies. The result of the categorization algorithm is a likely ontology that is used as input for the following matching algorithm. Matching algorithms aim to map WSDL elements to concepts in a pre-selected domain ontology. Categorization and Matching algorithms not only rely on ontology matching techniques but also use WordNet-based similarity measures [49]. For further details interested readers could refer to [11].

Although the goal of the Interface Compatibility procedure differs from the categorization and matching algorithms, we find that they are comparable in the context of this experiment. Particularly, the notions of semantic and structural assessment described along this work have been used in previous work for classification and categorization. On the one hand, to categorize software components according to a given taxonomy [50]; on the other hand, to classify cases in a Case-based Reasoning approach for service selection [51].

For this experiment, we used the data-set of 424 Web Services from [39]. Bouchiha et al. evaluated categorization and matching algorithms using only Web Services of the *business* domain from such data-set, due to the lack of relevant and descriptive ontologies for other domains. Hence, 13 services from the *business* domain are considered as the target services, and the remaining 411 services are noise for this experiment. Using the experimental steps described in Section 6.1.1 we extracted operations' signatures from the 13 *business* (target) services, which contained 71 operations. Then the mutation operators were randomly applied to these operation signatures, generating 71 mutated queries with different structural information. We defined two different experimental scenarios for the dataset and queries, whose details and results are described below.

6.3.1 Scenario 1 – Interface Compatibility vs. Categorization Algorithm

For each query, we associated the 13 services from the business domain as *target* services. Then, we analyzed the first 10 results retrieved by the interface compatibility procedure. If at least 5 out of the first 10 retrieved services are *target* services for a query, then such query is categorized under the *business* category; i.e., it is a *hit* (according to categorization notions). Otherwise, the query is considered as a *miss*.

Table 6 summarizes the precision, recall and f-measure values for the categorization algorithm and Interface Compatibility procedure applied with categorization purposes. Results show that the Interface Compatibility procedure outperformed the categorization algorithm in precision (5 %) but underperformed in recall (24 %) and F-measure (8 %). This means that the overall accuracy (considering both precision and recall) of Interface Compatibility used for categorization is slightly worse than the ontology-based categorization algorithm. Nevertheless, the former is widely applicable, while the latter is restricted to domains with descriptive ontologies available.

	Categorization Algorithm	Interface Compatibility
Precision	0.61	0.66
Recall	0.85	0.61
F-Measure	0.71	0.63

Table 6. Interface Compatibility vs. Categorization Algorithm experimental results

6.3.2 Scenario 2 – Interface Compatibility vs. Categorization and Matching

We used the aforementioned dataset of 424 services, with 71 queries extracted and mutated from the 13 *business* services. For each query, a group of experts selected *one* service from the business domain as *target* service. Then, we analyzed the first 10 results retrieved by the interface compatibility procedure according to the IR metrics precision, recall and F-Measure. Finally, we compared the results with the ontology-based categorization and matching algorithms.

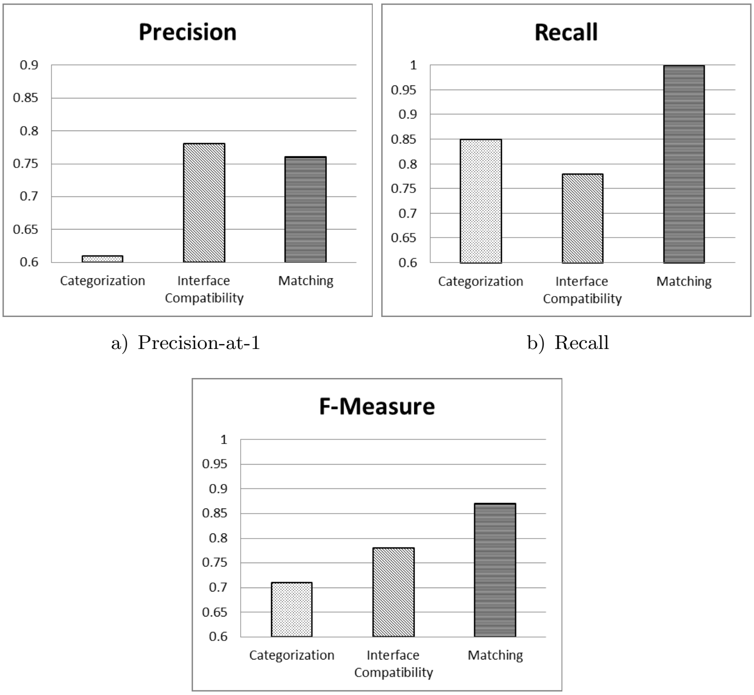
Figure 7 a) depicts the Precision-at-1 values, where the Interface Compatibility procedure outperformed categorization and matching algorithms by 17 % and 2 %, respectively. This means, Interface Compatibility was more precise for this dataset. Figure 7 b) depicts the Recall values, where the Interface Compatibility procedure underperformed categorization and matching by 7 % and 22 %, respectively. Figure 7 c) depicts the F-Measure values, where the Interface Compatibility procedure performed in between of categorization and matching, with a difference of 7 % and 9 %, respectively. This means that the overall accuracy (considering both precision and recall) of Interface Compatibility is better than the categorization algorithm and worse than the matching algorithm in average.

Discussion. From this experiment we can highlight that Interface Compatibility performs as well as ontology-based algorithms. Furthermore, the former is widely applicable, as the latter are restricted to domains with descriptive ontologies available. When such domain ontologies are available, the matching algorithm performs better than Interface Compatibility, in terms of precision, recall and F-Measure. But, according to different works in the field [20, 11, 21, 13], most domains lack relevant specific ontologies, whose availability is mandatory for the matching algorithm. Even more, the ontologies (when available) sometimes are not comprehensive enough to express all the relevant concepts in a domain [11, 52, 53].

The goal of matching algorithms is different from the Interface Compatibility procedures. Nevertheless, the notions of the Interface Compatibility procedures can be used with categorization purposes [51, 50], performing as well as ontology-based categorization algorithms while being widely applicable. However, the analysis in the Interface Compatibility procedure is restricted to concepts in operations, as the procedure ignores potentially meaningful concepts in comments, documentation and other parts of service descriptions. As we stated earlier, it is important to notice that the experiments are empirical, thus the obtained results confirm our hypothesis but cannot be merely generalized "as-is" to other experimental configurations, although similar practices are widely used in the field [4, 48, 46].

7 CONCLUSIONS AND FUTURE WORK

In this paper we presented details of the Interface Compatibility procedure, which allows evaluating a candidate Web Service for its likely integration into a client



a) Precision-at-1

b) Recall

c) F-Measure

Figure 7. Precision, Recall and F-Measure for Interface Compatibility, Categorization and Matching algorithms

application. This is performed through gathering and assessing structural and semantic information of service interfaces. The former is done through structural analysis of data types in operation parameters and return. The latter is done by leveraging an underlying lightweight ontology, namely WordNet or DISCO. Several experiments evaluated the performance of the three implementations between each other and against third-party algorithms. Results have been measured in terms of IR metrics: precision-at-n, recall, NDCG and F-measure.

According to the experiments, the Interface Compatibility procedure significantly improved ranking results obtained from the EasySOC service discovery registry, independently of the underlying lightweight ontology. This is initially demonstrated by the precision-at-n increase, ranging between 19 % and 42 % for the first positions of the results lists, w.r.t. the service discovery registry results. Although the DISCO-based implementation performed slightly better for the conducted experiments, the different implementations may be suitable in different contexts: WordNet is a generic dictionary of the English language, and could be suitable for a general-purpose application. In turn, DISCO is based on occurrences of words in large corpus of text; thus, in a corpus with highly-specific terms (e.g., a database of medical bibliography), the DISCO-based approach could outperform others. Furthermore, DISCO and WordNet could be used in conjunction to perform a two-step semantic assessment of terms.

Also, the Interface Compatibility performs as well as ontology-based categorization algorithms and can be used for similar purposes [50, 51]. Furthermore, the former is widely applicable, as the latter are restricted to domains with descriptive ontologies available, or may force developers to construct their own ad-hoc ontologies [54, 20] – which is costly in terms of time and effort. As future work in this direction, we are planning to semi-automatically annotate more services according to well-defined domain ontologies[13]. This would allow validation with a larger and more representative set of semantically annotated services.

Currently, we are working on extending the semantic evaluation considering other semantic relationships (e.g., siblings, or second-level hypo/hyperonymy) and extending the structural-semantic Interface Compatibility procedure to Exceptions, Return and complex types. Also, we are planning to use Named Entity Recognition (NER) techniques [55]. By doing this, concept definitions could be closer to a certain application domain (improving semantic precision) [8].

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments to improve the quality of this paper. We also thank Dr. Cristian Mateos for his valuable collaboration. This work is supported by projects: ANPCyT PICT 2012-0045 and UNCo-SPU Reuse (04-F001).

REFERENCES

[1] Erickson, J.—Siau, K.: Web Service, Service-Oriented Computing, and Service-Oriented Architecture: Separating Hype from Reality. Journal of Database Management, Vol. 19, 2008, No. 3, pp. 42–54, doi: 10.4018/jdm.2008070103.

[2] Bichler, M.—Lin, K.: Service-Oriented Computing. Computer, Vol. 39, 2006, No. 3, pp. 99–101, doi: 10.1109/MC.2006.102.

[3] McCool, R.: Rethinking the Semantic Web. IEEE Internet Computing, Vol. 9, 2005, No. 6, pp. 86–87.

[4] Garriga, M.—Flores, A.—Mateos, C.—Zunino, A.—Cechich, A.: Service Selection Based on a Practical Interface Assessment Scheme. International Journal of Web and Grid Services, Vol. 9, 2013, pp. 369–393, doi: 10.1504/IJWGS.2013.057469.

[5] De Renzis, A.—Garriga, M.—Flores, A.—Zunino, A.—Cechich, A.: Semantic-Structural Assessment Scheme for Integrability in Service-Oriented Applications. Latin-American Symposium of Enterprise Computing, Montevideo,

[13] https://www.w3.org/wiki/Good_Ontologies

Uruguay, Latin-American Conference on Informatics (CLEI), September 2014, doi: 10.1109/CLEI.2014.6965175.

[6] Garriga, M.—De Renzis, A.—Lizarralde, I.—Flores, A.—Mateos, C.—Cechich, A.—Zunino, A.: A Structural-Semantic Web Service Selection Approach to Improve Retrievability of Web Services. Information Systems Frontiers, 2016, pp. 1–26, doi: 10.1007/s10796-016-9731-1.

[7] Miller, G.—Beckwith, R.—Fellbaum, C.—Gross, D.—Miller, K.: Introduction to WordNet: An On-line Lexical Database. International Journal of Lexicography, Vol. 3, 1990, No. 4, pp. 235–244, doi: 10.1093/ijl/3.4.235.

[8] Finlayson, M.: Java Libraries for Accessing the Princeton WordNet: Comparison and Evaluation. Proceedings of the 7th Global Wordnet Conference, Tartu, Estonia, January 2014, pp. 78–85.

[9] Kolb, P.: Experiments on the Difference Between Semantic Similarity and Relatedness. Proceedings of the 17th Nordic Conference on Computational Linguistics (NODALIDA '09), Odense, Denmark, May 2009.

[10] Stroulia, E.—Wang, Y.: Structural and Semantic Matching for Assessing Web Services Similarity. International Journal of Cooperative Information Systems, Vol. 14, 2005, pp. 407–437, doi: 10.1142/S0218843005001213.

[11] Bouchiha, D.—Malki, M.—Alghamdi, A.—Alnafjan, K.: Semantic Web Service Engineering: Annotation Based Approach. Computing and Informatics, Vol. 31, 2012, No. 6, pp. 1575–1595.

[12] Kokash, N.: A Comparison of Web Service Interface Similarity Measures. Starting AI Researchers Symposium, Amsterdam, Netherlands, IOS Press, 2006.

[13] Crasso, M.—Zunino, A.—Campo, M.: A Survey of Approaches to Web Service Discovery in Service-Oriented Architectures. Journal of Database Management, Vol. 22, 2011, No. 1, pp. 102–132, doi: 10.4018/jdm.2011010105.

[14] Bartalos, P.—Bielikova, M.: Automatic Dynamic Web Service Composition: A Survey and Problem Formalization. Computing and Informatics, Vol. 30, 2012, No. 4, pp. 793–827.

[15] Motahari Nezhad, H. R.—Benatallah, B.—Xuyuan, G.: Protocol-Aware Matching of Web Service Interfaces for Adapter Development. International Conference on World Wide Web, Raleigh, North Carolina, USA, 2010, doi: 10.1145/1772690.1772765.

[16] Aumueller, D.—Do, H.—Massmann, S.—Rahm, E.: Schema and Ontology Matching with COMA++. Proceedings of the ACM SIGMOD International Conference on Management of Data, ACM Press, 2005, pp. 906–908, doi: 10.1145/1066157.1066283.

[17] Tibermacine, O.—Tibermacine, C.—Cherif, F.: WSSim: A Tool for the Measurement of Web Service Interface Similarity. Proceedings of the French-Speaking Conference on Software Architectures, Toulouse, France, May 2013.

[18] Dong, X.—Halevy, A.—Madhavan, J.—Nemes, E.—Zhang, J.: Similarity Search for Web Services. Proceedings of the International Conference on Very Large Data Bases (VLDB), 2004, pp. 372–383, doi: 10.1016/B978-012088469-8.50035-8.

[19] Plebani, P.—Pernici, B.: URBE: Web Service Retrieval Based on Similarity Evaluation. IEEE Transactions on Knowledge and Data Engineering, Vol. 21, 2009, No. 11, pp. 1629–1642, doi: 10.1109/TKDE.2009.35.

[20] Garriga, M.—Flores, A.—Mateos, C.—Cechich, A.—Zunino, A.: RESTful Service Composition at a Glance: A Survey. Journal of Network and Computer Applications, Vol. 60, 2016, pp. 32–53, doi: 10.1016/j.jnca.2015.11.020.

[21] Garriga, M.—Flores, A.—Cechich, A.—Zunino, A.: Web Services Composition Mechanisms: A Review. IETE Technical Review, Vol. 32, 2015, No. 5, pp. 376–383, doi: 10.1080/02564602.2015.1019942.

[22] Shvaiko, P.—Euzenat, J.: Ontology Matching: State of the Art and Future Challenges. IEEE Transactions on Knowledge and Data Engineering, Vol. 25, 2013, No. 1, pp. 158–176, doi: 10.1109/TKDE.2011.253.

[23] Pirró, G.: A Semantic Similarity Metric Combining Features and Intrinsic Information Content. Data and Knowledge Engineering, Vol. 68, 2009, No. 11, pp. 1289–1308.

[24] Gosling, J.—Joy, B.—Steele, G.—Bracha, G.: Java Language Specification. 3rd Ed. Addison-Wesley, Sun Microsystems, Inc., 2005.

[25] Pasley, J.: Avoid XML Schema Wildcards for Web Service Interfaces. IEEE Internet Computing, Vol. 10, 2006, No. 3, pp. 72–79, doi: 10.1109/MIC.2006.45.

[26] Anabalon, D.—Garriga, M.—Flores, A.—Cechich, A.—Zunino, A.: Adaptability-Based Service Behavioral Assessment. Journal of Computer Science and Technology, Vol. 15, 2015, pp. 75–80.

[27] Anabalon, D.—Garriga, M.—Flores, A.—Cechich, A.—Zunino, A.: Test Reduction for Web Service Integration. Proceedings of the Argentinean Symposium of Software Engineering (ASSE 2015), September 2015, pp. 115–129.

[28] Crasso, M.—Rodriguez, J. M.—Zunino, A.—Campo, M.: Revising WSDL Documents: Why and How. IEEE Internet Computing, Vol. 14, 2010, No. 5, pp. 48–56, doi: 10.1109/MIC.2010.81.

[29] Mateos, C.—Crasso, M.—Zunino, A.—Coscia, J. M.: Revising WSDL Documents: Why and How, Part 2. IEEE Internet Computing, Vol. 17, 2013, pp. 46–53, doi: 10.1109/MIC.2013.4.

[30] Elish, M. O.—Offutt, J.: The Adherence of Open Source Java Programmers to Standard Coding Practices. International Conference on Software Engineering and Applications (IASTED), 2002, pp. 374.200–374.207.

[31] Casamayor, A.—Godoy, D.—Campo, M.: Mining Architectural Responsibilities and Components from Textual Specifications Written in Natural Language. SADIO Electronic Journal of Informatics and Operations Research, Vol. 10, 2011, No. 1, pp. 4–19.

[32] Willett, P.: The Porter Stemming Algorithm: Then and Now. Program: Electronic Library and Information Systems, Vol. 40, 2006, No. 3, pp. 219–223.

[33] Paice, C.: An Evaluation Method for Stemming Algorithms. Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval, Springer-Verlag New York, Inc., 1994, pp. 42–50, doi: 10.1007/978-1-4471-2099-5_5.

[34] Meyer, D.—Hornik, K.—Feinerer, I.: Text Mining Infrastructure in R. Journal of Statistical Software, Vol. 25, 2008, No. 5, pp. 1–54.

[35] Bano, M.—Ferrari, A.—Zowghi, D.—Gervasi, V.—Gnesi, S.: Automated Service Selection Using Natural Language Processing. Requirements Engineering in the Big Data Era, Springer, 2015, pp. 3–17, doi: 10.1007/978-3-662-48634-4_1.

[36] Godoy, D.—Schiaffino, S.—Amandi, A.: Interface Agents Personalizing Web-based Tasks. Cognitive Systems Research, Vol. 5, 2004, No. 3, pp. 207–222.

[37] Crasso, M.—Zunino, A.—Campo, M.: Combining Query-by-Example and Query Expansion for Simplifying Web Service Discovery. Information Systems Frontiers, Vol. 13, 2011, No. 3, pp. 407–428, doi: 10.1007/s10796-009-9221-9.

[38] Kuhn, H.: The Hungarian Method for the Assignment Problem. Naval Research Logistic Quarterly, Vol. 2, 1955, pp. 83–97, doi: 10.1002/nav.3800020109.

[39] Hess, A.—Johnston, E.—Kushmerick, N.: ASSAM: A Tool for Semi-Automatically Annotating Semantic Web Services. Proceedings of the International Semantic Web Conference (ISWC), Springer, 2004, pp. 320–334.

[40] Mateos, C.—Crasso, M.—Zunino, A.— Ordiales Coscia, J. M.: Detecting WSDL Bad Practices in Code-First Web Services. International Journal of Web and Grid Services, Vol. 7, 2011, No. 4, pp. 357–387, doi: 10.1504/IJWGS.2011.044710.

[41] Gosh, S.—Mathur, A. P.: Interface Mutation. Software Testing, Verification and Reliability, Vol. 11, 2001, pp. 227–247, doi: 10.1002/stvr.239.

[42] Salton, G.—Wong, A.—Yang, C. S.: A Vector Space Model for Automatic Indexing. Communications of the ACM, Vol. 18, 1975, No. 11, pp. 613–620, doi: 10.1145/361219.361220.

[43] Delamaro, M.—Maldonado, J.—Mathur, A.: Interface Mutation: An Approach for Integration Testing. IEEE Transactions on Software Engineering, Vol. 27, 2001, No. 3, pp. 228–247, doi: 10.1109/32.910859.

[44] Rodriguez, J. M.—Crasso, M.—Zunino, A.—Campo, M.: Improving Web Service Descriptions for Effective Service Discovery. Science of Computer Programming, Vol. 75, 2010, No. 11, pp. 1001–1021, doi: 10.1016/j.scico.2010.01.002.

[45] Crasso, M.—Zunino, A.—Campo, M.: Easy Web Service Discovery: A Query-by-Example Approach. Science of Computer Programming, Vol. 71, pp. 144–164, 2008, doi: 10.1016/j.scico.2008.02.002.

[46] Crasso, M.—Mateos, C.—Zunino, A.—Campo, M.: EasySOC: Making Web Service Outsourcing Easier. Information Sciences, Vol. 259, 2014, pp. 452–473, doi: 10.1016/j.ins.2010.01.013.

[47] Agichtein, E.—Brill, E.—Dumais, S.—Ragno, R.: Learning User Interaction Models for Predicting Web Search Result Preferences. 29th Annual ACM SIGIR International Conference on Research and Development in Information Retrieval, ACM Press, 2006, pp. 3–10, doi: 10.1145/1148170.1148175.

[48] Mateos, C.—Rodriguez, J. M.—Zunino, A.: A Tool to Improve Code-First Web Services Discoverability Through Text Mining Techniques. Software: Practice and Experience, Vol. 45, 2015, No. 7, pp. 925–948.

[49] Pedersen, T.—Patwardhan, S.—Michelizzi, J.: WordNet::Similarity – Measuring the Relatedness of Concepts. Demonstration Papers at HLT-

NAACL 2004, Association for Computational Linguistics, 2004, pp. 38–41, doi: 10.3115/1614025.1614037.

[50] Arias, M.—De Renzis, A.—Buccella, A.—Flores, A.—Cechich, A.: Classification-Based Mining of Reusable Components on Software Product Lines. IEEE Latin America Transactions, Vol. 14, 2016, No. 2, pp. 870–876.

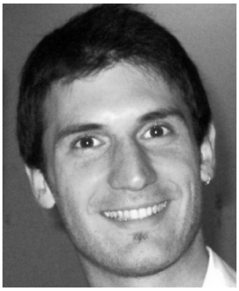
[51] De Renzis, A.—Garriga, M.—Flores, A.—Cechich, A.—Zunino, A.: Case-Based Reasoning for Web Service Discovery and Selection. Electronic Notes in Theoretical Computer Science, Vol. 321, 2016, pp. 89–112, doi: 10.1016/j.entcs.2016.02.006.

[52] Guizzardi, G.: Ontological Foundations for Structural Conceptual Models. 1st Ed. Universal Press, The Netherlands, 2005. ISBN 90-75176-81-3.

[53] Lanthaler, M.—Gutl, C.: A Semantic Description Language for RESTful Data Services to Combat Semaphobia. Proceedings of the 5th International Conference on Digital Ecosystems and Technologies Conference (DEST), IEEE, 2011, pp. 47–53, doi: 10.1109/DEST.2011.5936597.

[54] Funika, W.—Godowski, P.—Pęgiel, P.—Król, D.: Semantic-Oriented Performance Monitoring of Distributed Applications. Computing and Informatics, Vol. 31, 2012, No. 2, pp. 427–446.

[55] Nadeau, D.—Sekine, S.: A Survey of Named Entity Recognition and Classification. Lingvisticae Investigationes, Vol. 30, 2007, No. 1, pp. 3–26.



Martin Garriga received his Ph.D. degree in computer sciences in 2016 at Faculty of Exact Sciences, UNICEN. He is a posdoctoral fellow at Politecnico de Milano (Italy) since 2016, and Lecturer Assisstant at Informatics Faculty, UNComa since 2011. His research interests are service-oriented architectures, web service selection and composition, RESTful Services and microservices architectures.

Alan De Renzis received his B.Sc. degree in computer sciences in 2013 at the Faculty of Informatics, UNComa (Neuquen, Argentina). Since then he is a Ph.D. candidate at Faculty of Exact Sciences, UNICEN (Tandil, Argentina). His research interests are service-oriented architectures, web service discovery and selection and service metamodels.

Andres Flores received his Ph.D. degree in informatics from University of Castilla-La Mancha, Spain in 2009. He is Adjunct Professsor at Informatics Faculty, UNComa since 2010, and Researcher at the Argentinean National Scientific and Technical Research Council (CONICET) since 2012. His research interests are software engineering, service-oriented computing, component-based systems, software testing.

Alejandra Cechich received her Ph.D. degree in informatics from University of Castilla-La Mancha, Spain (2005). She is Fulltime Professor at Informatics Faculty, UNComa since 1996. Her research interests are software engineering, software reuse, software quality and architectures.

Alejandro Zunino received his Ph.D. degree in computer sciences at the Faculty of Exact Sciences, UNICEN university, in 2003. He is Fulltime Professor at Faculty of Exact Sciences, UNICEN since 2006, and Researcher at CONICET since 2005. His research interests are software engineering, mobile computing, service-oriented architectures, and grid and cloud computing.