Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 321 (2016) 89–112

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Case-based Reasoning for Web Service Discovery and Selection

Alan De Renzis, Martin Garriga, Andres Flores, Alejandra Cechich[1,2]

GIISCo Research Group
Faculty of Informatics, UNComa University
Nuequen, Argentina

Alejandro Zunino[3]

ISISTAN Research Institute
UNICEN University
Tandil, Buenos Aires, Argentina

Abstract

Web Service discovery and selection deal with the retrieval of the most suitable Web Service, given a required functionality. Addressing an effective solution remains difficult when only functional descriptions of services are available. In this paper, we propose a solution by applying Case-based Reasoning, in which the resemblance between a pair of cases is quantified through a similarity function. We show the feasibility of applying Case-based Reasoning for Web Service discovery and selection, by introducing a novel case representation, learning heuristics and three different similarity functions. We also experimentally validate our proposal with a dataset of 62 real-life Web Services, achieving competitive values in terms of well-known Information Retrieval metrics.

Keywords: Web services, Service Selection, Service Discovery, Case-based Reasoning, Service Oriented Application.

1 Introduction

Service-Oriented Computing (SOC) has seen an ever increasing adoption by providing support for building distributed, inter-organizational applications in heterogeneous environments [14]. Mostly, the software industry has adopted SOC by using Web Service technologies. A Web Service is a program with a well-defined interface that can be located, published, and invoked by using standard Web protocols [5].

[1] This work is supported by projects: PICT 2012-0045 and UNCo–Reuse(04-F001).

[2] Email: {alanderenzis,martin.garriga,andres.flores,alejandra.cechic}@fi.uncoma.edu.ar

[3] Email: alejandro.zunino@isistan.unicen.edu.ar

http://dx.doi.org/10.1016/j.entcs.2016.02.006

However, a broadly use of the SOC paradigm requires efficient approaches to allow service discovery, selection, integration and consumption from within applications [29]. Currently, developers are required to manually search for suitable services to then provide the adequate "glue-code" for their assembly into a service-oriented application [9]. Even with a wieldy candidates list, a skillful developer must determine the most appropriate service for the consumer application. This implies a prohibitive effort into discovering services, analyzing the suitability of retrieved candidates (i.e., service selection) and identifying the set of adjustments for the final integration of a selected candidate service.

In this work we make use of Case-based Reasoning (CBR) [1]– from the Artificial Intelligence (AI) field – to overcome the aforementioned problems in Web Service Discovery and Selection. A Case-based Reasoner solves problems by using or adapting solutions from old recurrent problems [35]. Sometimes called similarity searching systems, the most important characteristic of CBR systems is the effectiveness of the similarity function used to quantify the degree of resemblance between a pair of cases [25].

Our proposal models a Case-based Reasoner for Service Selection, where the main contribution is threefold. We define a case representation capturing information in Web Services functional descriptions (typically WSDL). Moreover, we draw a parallel among the key steps in CBR and the problem of Web Service Discovery and Selection. Finally, we provide three implementations for the similarity function, concerning structural and semantic aspects from functional service descriptions.

The rest of the paper is organized as follows. Section 2 details the service selection process. Section 3 presents the application of CBR in the context of service selection. Section 4 details the alternatives for the similarity function. Section 5 presents the experimental validation of the approach. Section 6 discusses related work. Conclusions and future work are presented afterwards.

2 Service Selection

During development of a Service-oriented Application, some of the comprising software pieces could be fulfilled by the connection to Web Services. In this case, a list of candidate Web Services could be obtained by making use of any service discovery registry. Nevertheless, even with a wieldy candidates' list, a developer must be skillful enough both to determine the most appropriate service and to shape the adaptation artifacts for seamless integration of the selected service. Therefore, a reliable and practical support is required to make those decisions. For this, in previous work [12,16] we defined an approach for service selection.

The service selection method is based in an Interface Compatibility assessment of the candidate Web Services and the (potentially partial) specification of the required functionality – depicted in Figure 1. The procedure matches the required interface I_R and the interface (I_S) provided by a candidate service S (previously discovered). All available information from the two interfaces is gathered to be assessed at semantic and structural levels. The semantic assessment makes use

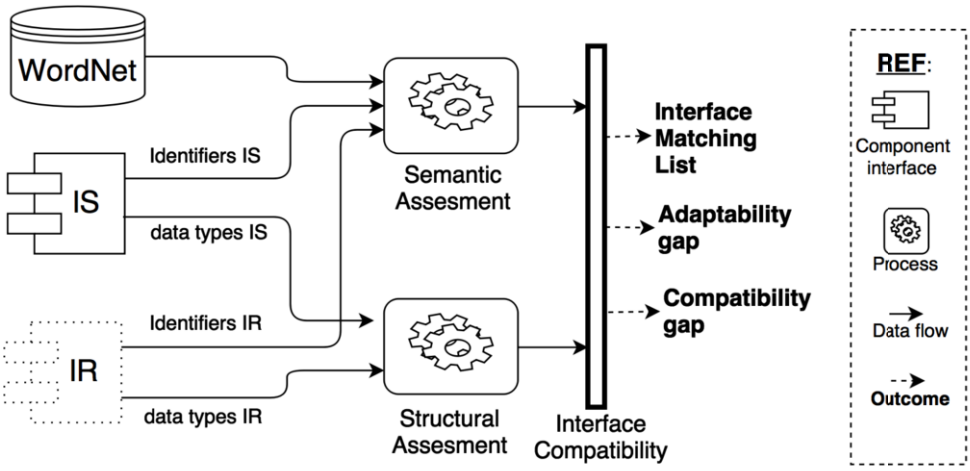


Fig. 1. Interface Compatibility Scheme

of the WordNet lexical database [30] for identifiers evaluation, by means of terms separation, stop words removing, stemming and terms similarity (synonymy and hypo/hyperonymy). The structural evaluation considers data types equivalence and subtyping.

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, two appraisal values are calculated: *compatibility gap* (concerning functional aspects), and *adaptability gap* – which reflects the required effort for integrability of the selected service.

3 Case-based Reasoning for Service Selection

This work extends the Interface Compatibility Scheme by means of a CBR methodology [1]. The main goal is to capture the knowledge obtained from successive service selections as a set of cases in the form of problem-solution pairs. Figure 2 shows the CBR approach (adapted from [1]).

Let be a *knowledge base KB* containing an initial set of cases. Each case consists of a pair (problem, solution): the *problem* is a description of certain functionality, and the associated *solution* is the candidate service that fulfills such functionality. A *new case* C is a problem part (required functionality) that has to be paired with its corresponding solution (candidate service). For this, the first step compares C with all the problem parts in KB, according to a similarity function. The outcome of this step is the most similar case to C (*retrieved case*) – i.e., the pair ($functionality$, $service$) with the most similar functionality w.r.t. C.

Then, the retrieved case is reused to generate the *solved case*, by combining its solution part with the new case C as the problem part. The solved case is then a pair ($required\_functionality$, $service$). At this point, the solved case is returned as the *suggested solution*, which can be revised by expert users. If the suggested solution succeed in the revision, it then becomes a *confirmed solution* (*tested case*).

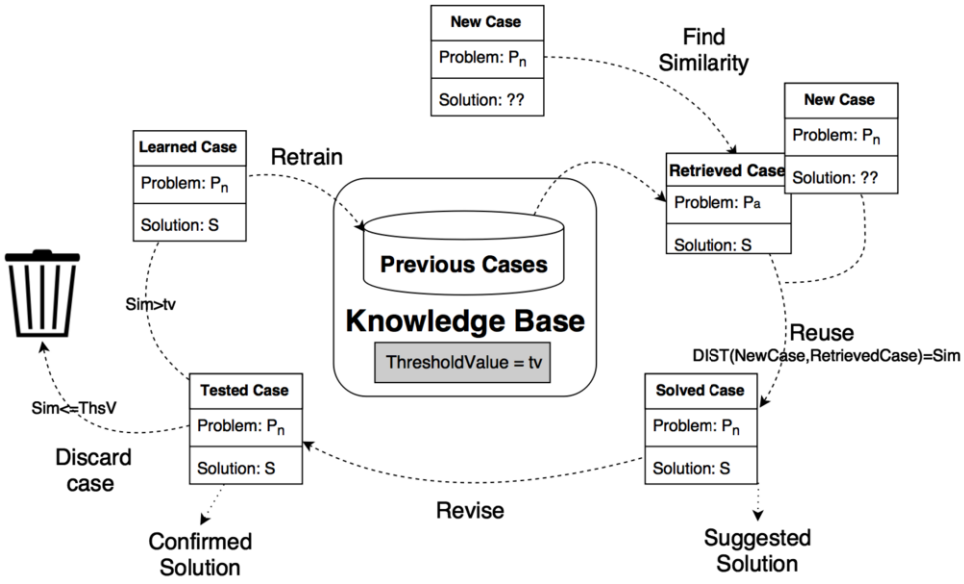


Fig. 2. Case-based Reasoning for Service Selection

If it fails, the case is discarded.

Finally, the last step decides whether or not to include the confirmed solution (tested case) in *KB*. The *learning case* decision can rely upon different criteria. In this approach, we use a threshold value (*th*) over the similarity function: if the similarity function returns a value higher than the threshold, then the case is added to the *KB*.

In the following sections, we describe the application of CBR concepts to Web Service discovery and selection.

3.1 Case Representation

First, it is essential to define an adequate case representation in the context of service selection. We have used an object-oriented (OO) case representation, where the cases are represented as object collections described by a set of attribute-value pairs. Object-oriented representations are appropriate for complex domains where different case structures may occur [7]. Figure 3 shows the OO case representation structure for service selection. As stated earlier, the Case class is divided into two parts – namely Problem and Solution.

The Problem part captures the required functionality to be fulfilled by a candidate service. The required functionality is composed of three simple attributes and a collection of a complex attribute. The simple attributes are *Service name* (`String`), *Category* (`String`), and *Operations number* (a positive integer including zero). The complex attribute is *Operations*, which represents the required operations. Each *Operation* contains two attributes – *Operation name* and *Return type* (Simple or Complex type) – and two collections of complex attributes – *Parameter*s and *Exception*s. *Parameter* contains two attributes: *parameter name* and *parameter type* (Simple or Complex type). *Exceptions* contains a simple attribute: *Type*

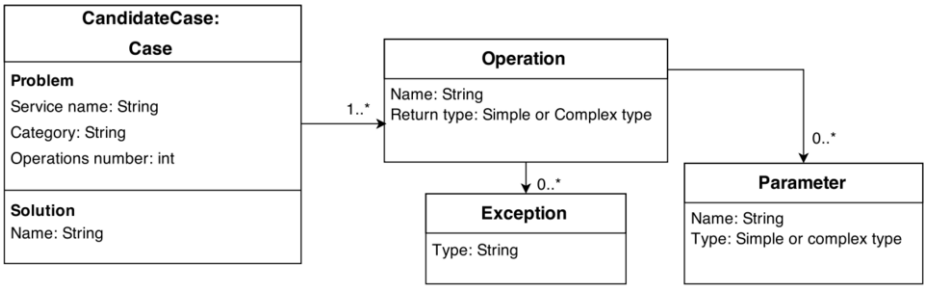


Fig. 3. Object-oriented Case representation

(`String`).

The Solution part is a simple `String` attribute *Candidate Service* that represents the name of the service associated to the problem description.

3.2 Case Retrieval

New cases are given as input to the Case-based Reasoner (Figure 2) in the form of a required functionality – i.e., a problem part (P_{n}) . To find a solution to a *new case*, the first step calculates the similarity function ($DIST$) as a distance between the new case and each case in the knowledge base KB – according to Formula 1, extracted from [25]. For each attribute in the case representation, we have defined specific similarity functions sim and weights w_i (where the sum of all weights w_i is 1) – that are presented in the following sections.

$$DIST(C^N, C^C) = \sum_{1}^{n} (w_i * sim_i(C_i^N, C_i^C)) \tag{1}$$

where C^N is a new case to evaluate and C^C is the candidate case in KB.

Service name and Category

The *Service name* and *category* evaluation consists in comparing the `String` values of these attributes between the new case under analysis and the problem part of each case in the *KB*. Similarity is calculated using an algorithm for Identifiers evaluation which considers semantics of terms in identifiers – discussed in detail in Section 4.2. The given weight for service name and category attributes is low ($w = 0.1$) as they do not directly express functional aspects.

Operations number

The *operations number* evaluation consists in comparing the numerical value for such attribute between the new case under analysis and the problem part of each case in the *KB*. Similarity is calculated according to Formula 2. Candidate services (solutions) in the *KB* with fewer operations than the required functionality (problem) are considered as incompatible, being discarded as potential solutions.

The given weight for this attribute is higher ($w = 0.3$) as it directly expresses a functional aspect (i.e., the number of required operations).

$$Sim(\#op^N, \#op^C) = \begin{cases} \#op^N <= \#op^C & 1 \\ otherwise & 0 \end{cases} \quad (2)$$

where $\#op^N$ and $\#op^C$ are the values of the operation number attribute in the new case and a candidate case of the *KB respectively*

Operations

The *operations* evaluation calculates similarity between this complex attribute in the new case, and the analogous attribute in the problem part of each case in *KB*. Since the main criterion of our service selection approach is functional similarity, this attribute presents the highest weight ($w = 0.6$) and the most complex similarity function in this Case-based Reasoner. Details of the similarity function for the *operations* attribute are presented in Section 4.

3.3 Case Reuse and Revision

We use the first nearest neighbor (1-NN) strategy for case adaptation, which implies that the most similar case is chosen as the best solution [38]. Therefore, the solution part into the new case will be the solution part (S) – i.e., the candidate service – of the most similar *retrieved case* according to Formula 1.

The *solved case* is then returned as the *suggested solution* (Figure 2), and can be revised by expert users. Expert users are people with high knowledge about the domain and the service-oriented application under construction. Thus, an expert user decides if the solution is suitable for the target application or relevant for the underlying domain. Otherwise, the *solved case* is rejected. Relevant cases can then be part of the *KB*'s initial state. Experts feedback is not mandatory but necessary to improve the reasoner performance and to determine the *threshold value* according to the state (i.e., the number of cases) of the *KB* at a given time.

3.4 Case Retraining

At this point, the Case-based Reasoner has compared the new case against the problem part of all the cases in the *KB* and (expectedly) has found a solution in terms of the attribute similarity presented in the previous sections. Also, the solution had been revised and acknowledged as valid by expert users – i.e., it is a confirmed solution (*tested case*). The next step consists in deciding if the *tested case* will be added to the *KB*.

On the one hand, too many retrained cases can generate noise in the evaluation, decreasing the performance of the reasoner in the long term. On the other hand, if no new cases are added, no learning occurs, so the reasoner will not be capable to deal with new cases. In order to prevent these problems, we have defined a threshold value (th) over the *similarity function* ($DIST$). The threshold value determines

whether or not a new case is retrained (*learned case*) in the *KB*. If the *similarity function* returns a value higher than the threshold, then the case is added to the *KB* as a new meaningful case, otherwise it is discarded.

The goal is to prevent the uncontrollable growth of the *KB* while improving the performance of the reasoner. The threshold value is a configurable constant that depends on the reasoner implementation and the initial number of cases. If the initial number of available cases is low (e.g., with regard to the total number of services in a given domain), the threshold value will also be settled low, allowing the reasoner to add new cases and to enrich the *KB*. If the number of available cases grows in a certain moment, the threshold value can be increased to add only new cases with a significant similarity.

4 Operations Similarity

As we stated earlier, operation similarity is the key attribute for assessing new cases against the potential candidate services, included as cases in the *KB*. The operations similarity evaluation accounts semantic and structural aspects extracted from operation definitions. Structural aspects involve data types equivalence (subtyping), while semantic aspects involve concepts from terms and identifiers. We have defined three implementations for operations similarity that mainly differ in their semantic basis. This resulted in three different *similarity functions* for operations evaluation. In the following sections we describe the similarity evaluation for each element in operations: identifiers, operation name (evaluated as a special case of identifiers), parameters, return type, and exceptions.

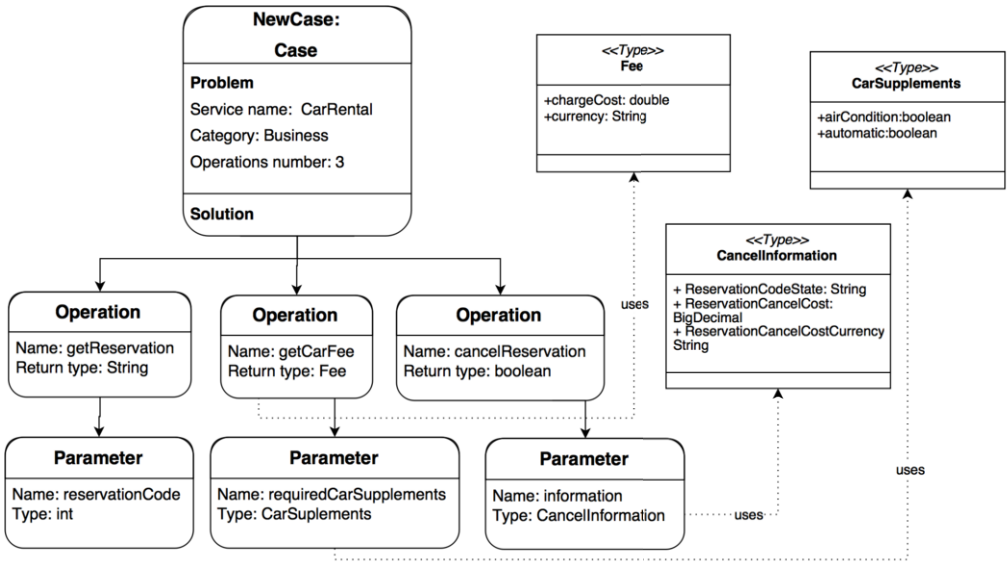
4.1 Case Study

A simple case study has been outlined to illustrate key steps of our proposal. We considered the Car Rental domain, where the required features are portrayed according to the OO case representation (Figure 3). Thus Figure 4a shows a new case (*newCase*), which contains the description of a proposed service named `RentaCar`. Such interface defines three operations and three complex data types. Note that the solution part of the *newCase* is not instantiated since the case has not been evaluated.

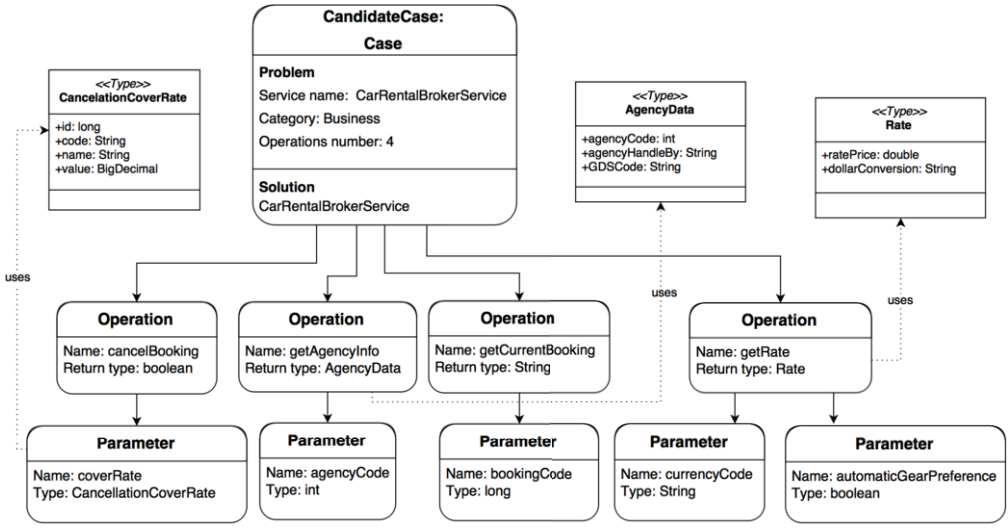
The functionality of the required interface will be fulfilled by engaging a third-party Web Service. The case in the *KB candidateCase* contains the representation of the service `CarRentalBrokerService`. The *candidateCase* defines four operations and three complex data types – as shown in Figure 4b.

Both cases were built by adapting real Web Services [4, 5] from the Car Rental domain to illustrate our proposal.

[4] http://goo.gl/MC7uXh
[5] http://goo.gl/LL0k0w



(a) New case instantiation



(b) Candidate case instantiation

Fig. 4. Object cases representation for the Car Rental example

4.2 Identifiers Evaluation

To evaluate semantic aspects the similarity functions compare terms and identifiers from operations. We implemented three alternatives for these functions. The first two make use of WordNet [30]. WordNet is a domain-independant lexical database of the english language that is structured as a lexical tree. WordNet groups terms in *synsets* (synonym sets) that represent the same lexical concept. Several relationships connect different synsets, such as hypo/hyperonyms, holonyms/meronyms and antonyms. All hierarchies ultimately go up the root node {*entity*}. The WordNet

Table 1
Rules for Decomposing Identifiers

Notation	Rule	Source	Result
Java Beans	Splits when changing text case	getZipCode	get Zip Code
Special symbols	Splits when either "_" or "-" occurs	get_Quote	get Quote

structure can be accessed through different Java libraries, each one implementing different metrics and features [15]. Particularly, in this work we used JWI [6] (in the first similarity function) and JWNL [7] (in the second similarity function). These libraries are among the most complete and easy to use for WordNet lexical tree manipulation [15].

The third alternative for the similarity function is based upon DISCO [22], a pre-computed database of collocations and distributionally similar words. DISCO's Java library [8] allows to retrieving the semantic similarity between arbitrary words. The similarities are based on the statistical analysis of very large text collections (e.g., Wikipedia), through co-occurrence functions. For each word, DISCO stores the first and second order vectors of related words using a Lucene index [19]. To determine the similarity between two words, DISCO retrieves the corresponding word vectors from the index and computes the similarity based in co-occurrences.

Following we describe the similarity functions implemented using JWI, JWNL and DISCO. To determine the similarity between two identifiers, these implementations share two preliminary common steps. Thus, two identifiers are initially pre-processed through term separation and stop words removal [12]:

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 rules in Table 1 – i.e., the usual programming naming conventions – plus a semantic level to consider identifiers that do not follow those conventions. The Term Separation step analyzes the identifiers, recognizing potential terms (uppercase sequences and lowercase sequences). Then, WordNet is used to analyze all the potential terms and determines the most adequate term separation. The term separation step is crucial to consider the correct terms as input to the semantic analysis.

Example. Let be 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 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

[6] http://projects.csail.mit.edu/jwi/
[7] http://web.stanford.edu/class/cs276a/projects/docs/jwnl/overview
[8] http://www.linguatools.de/disco/disco-api-1.4/

(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 prior to, or after, processing natural language data (text) [3]. We defined a stop words list containing articles, pronouns, prepositions, words from other stop words lists and each letter of the alphabet. The terms lists obtained from the previous step are analyzed to remove any occurrence of a word belonging to the stop words list.

Example. Let consider the identifier `AgencyHandledBy` which corresponds to a field in the Data Type `AgencyData` of the Car Rental example. According to the Java Beans notation, the identifier is decomposed in three terms: [`Agency, Handled, By`]. As `'By'` belongs to the stop words list, it is removed from the terms list.

4.2.1 JWI-based Identifiers Evaluation

The JWI implementation comprises three main additional steps: stemming, terms lists semantic comparison and identifiers compatibility calculation.

Stemming is the process for reducing words to their stem, base or root form. Due to common problems of standard syntactical stemmers [39], we adapted the semantic stemmer provided by WordNet. The Stemming step receives as input a terms list. For each term in the list is verified that it belongs to the WordNet dictionary. If it does so, the corresponding stems are added to the result list. Otherwise, the original term is added to the result list, considered as an abbreviation or acronym.

After generating both lists of stems, their compatibility is calculated considering semantic information. This information is expressed as a vector of integers $v = \{t, e, s, h_1, h_2\}$ including: the total terms between both lists (t), the identical (exact) terms (e), synonyms (s), hyperonyms (h_1) and hyponyms (h_2). For example, let be the identifiers `GetReservation` and `GetCurrentBooking` extracted from the cases of the Car Rental example in Section 4.1. According to the term lists semantic comparison, these identifiers present:

- Four distinct terms: (`Get`, `Reservation`, `Current`, `Booking`)
- One exact (identical) term: `Get`
- One synonym: (`Reservation`, `Booking`)
- No hypo/hyperonyms

Using these values in the vector v as input, the *Identifiers Compatibility Value* is calculated according to Formula 3.

$$ICValue = \frac{e + s + 0.5 * (h_1 + h_2)}{t - s} \tag{3}$$

Example. Let be the identifiers `GetReservation` and `GetCurrentBooking` extracted from the interfaces of the Car Rental example, by replacing the values in Formula 3 we obtain:

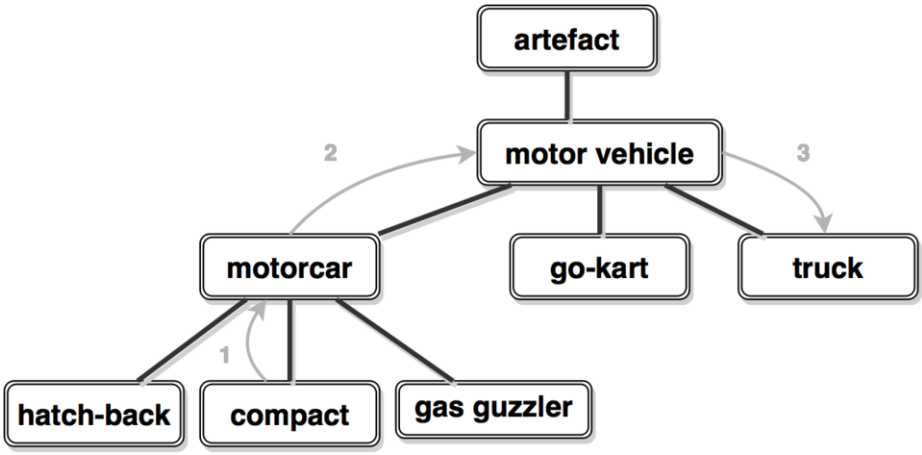


Fig. 5. Length between "compact" and "truck" in the WordNet hierarchy

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

Which indicates a compatibility value of 0.66 between the identifiers `GetReservation` and `GetCurrentBooking` – considering that the maximum value for $ICValue$ is 1, the obtained value indicates a moderate to strong compatibility between the identifiers.

4.2.2 JWNL-based Identifiers Evaluation

The JWNL implementation calculates the compatibility value according to two main additional steps: generation of the normalized depth matrix and term matching maximization.

First, the *Normalized Depth* matrix (ND) is generated. The depth is defined as the shortest path between two terms in the WordNet hierarchy. These values are normalized by the maximum depth of the WordNet hierarchy (16). Formally, the normalized depth is calculated according to Formula 4.

$$NormalizedDepth(t_i, t_j) = \frac{2D - length(t_i, t_j)}{2D} \tag{4}$$

where $length(t_i, t_j)$ = shortest path between t_i, t_j in the WordNet hierarchy, D is the maximum tree depth (16)

Example. Figure 5 shows an excerpt of the WordNet hierarchy, showing different types of vehicles. It shows that the *length* between the concepts `compact` and `truck` is 3, and the length between `compact` and `motor vehicle` is 2. These values indicate that `compact` and `motor vehicle` are more similar than `compact` and `truck` – according to JWNL.

Accounting this notion of length, lets consider the two identifiers `GetReservation` and `GetCurrentBooking` (analyzed in Section 4.2.1). The ND matrix will be a 2x3 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).

Table 2
Normalized Depth matrix for the identifiers `GetReservation` and `GetCurrentBooking`

	`Get`	`Current`	`Booking`
`Get`	1.00	0.56	0.72
`Reservation`	0.72	0.72	1.00

Higher is better. Maximum and minimum values are 1 and 0 respectively.

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. For each possible pair-wise term assignment (t_i, t_j) between both lists, the similarity value is obtained from the corresponding matrix cell ND_{ij}. The 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 [23], as an instance of the allocation problem.

Finally, the identifiers compatibility value ($ICValue$) using JWNL is calculated according to Formula 5, which weights the sum of the pair-wise assignments of terms according to the maximum number of terms in the identifiers under analysis.

$$ICValue = \frac{assignSum}{max(n, m)} \tag{5}$$

where n and m are the number of terms in both terms lists.

Example. 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 5, the compatibility value between identifiers `GetReservation` and `GetCurrentBooking` is calculated as follows:

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

4.2.3 DISCO-based Identifiers Evaluation

The DISCO-based implementation calculates the compatibility value according to two main steps: generation of the co-occurrences matrix and term matching maximization.

First, the Co-occurrences matrix (Co) is generated. This matrix contains the similarity values between each term from both terms lists. These values are the result of applying the co-occurrences similarity notion of DISCO, explained earlier. After calculating the Co matrix, the best term matching (among all possible pair-wise combinations) must be selected. Similarly to the JWNL-based implementation, this step uses the Co matrix as input for the Hungarian algorithm. The matching with the highest value will be the most compatible. Such matching is also obtained through the Hungarian method – introduced in Section 4.2.2. Finally, the identifiers compatibility value using DISCO is calculated according to Formula 5.

Example. Lets consider the pair of identifiers of the previous section – namely `GetReservation` and `GetCurrentBooking`. The Co matrix will be a $2x3$ 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`).

Table 3
Co-occurrences matrix for the identifiers `GetReservation` and `GetCurrentBooking`

	`Get`	`Current`	`Booking`
`Get`	1.00	0.006	0.02
`Reservation`	0.01	0.01	0.1

Higher is better. Maximum and minimum values are 1 and 0 respectively.

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 5, the compatibility value between identifiers `GetReservation` and `GetCurrentBooking` is calculated as follows:

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

4.3 Return type

Data Type Equivalence

Conditions for data type equivalence involves the subsumes relationship or subtyping, which implies a direct subtyping in case of built-in types in the Java language [18], as shown in Table 4. It is expected that types on operations from a

new case have at least as much precision as types on operations from a candidate service (case in the *KB*). For example, if $op^N \in newCase$ includes an `int` type, a corresponding operation $op^C \in candidateService$ 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 since it is generally used in practice by programmers to allocate different kinds of data [31]. Thus, we consider `String` as a supertype of any other built-in type.

Table 4
Subtype Equivalence

op^N **type**	op^C **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
double	string

Complex Data Types

Complex data types imply a special treatment in which the comprising 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 $op^N \in newCase$ must match a field from the complex type in $op^C \in candidateService$ – though extra fields from $newCase$ may be initially left out of any correspondence.

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

- $Ret = 3$: Equal Return Type.
- $Ret = 2$: Equivalent Return Type (Subtyping, String or Complex types).
- $Ret = 1$: Non-equivalent complex types or precision loss.
- $Ret = 0$: Not compatible .

Example. Figure 6 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 $Ret = 2$.

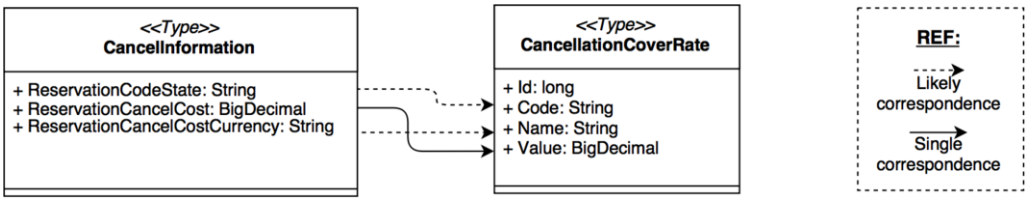


Fig. 6. Equivalence of Complex Data types for the CarRental example

4.4 Parameters evaluation

The algorithm for Parameters Evaluation consists of calculating three matrices: Type (T), Name (N) and Compatibility ($Comp$). For the three matrices, the cell M_{ij} represents the compatibility value between the *i-th* parameter of op^N and the *j-th* parameter of op^C – where op^N is an operation of the required functionality (*new case*) and op^C is an operation of a candidate service (case in the KB).

In the T matrix, the notions of structural data type equivalence and subtyping are used to assess parameter types. The goal of the T matrix is to store the relationship between all pairs of parameter types from both operations. A cell T_{ij} contains the compatibility value between the *i-th* parameter's type of op^N and the *j-th* parameter's type of op^C, according to Formula 6.

$$\begin{cases} Type(P_i) = Type(P_j) & T_{ij} = 2 \\ Type(P_i) <: Type(P_j) & T_{ij} = 1.5 \\ otherwise & T_{ij} = 1 \end{cases} \tag{6}$$

where $<:$ represents the sutyping relationship

The N matrix contains the compatibility values between the name of each parameter from op^N and the name of each parameter from op^C. 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^N and the *j-th* parameter's name of op^C. This value is the result of applying the Identifiers Evaluation Algorithm presented in Section 4.2. Therefore, these values depend on the chosen similarity function implementation – from the three alternatives.

Then, the $Comp$ matrix is generated from the T and N matrices. The goal of the $Comp$ matrix is to store the compatibility value between all parameter pairs from operations op^N and op^C, considering structural and semantic aspects – collected in the T matrix and the N matrix respectively. Each cell $Comp_{ij}$ stores the product between T_{ij} and N_{ij}. Thus: $Comp_{ij} = T_{ij} * N_{ij}$.

After calculating the $Comp$ matrix, the best parameter matching (among all possible pair-wise combinations) must be selected – i.e., the combination of parameters from op^N and op^C that maximizes their compatibility. This step applies the Hungarian algorithm to calculate the best pair-wise parameters assignments – similarly to the term matching maximization in JWNL-based Identifiers Evaluation (Section 4.2).

4.5 Exceptions

Structural conditions for exceptions are evaluated as follows. First, any operation op^N may define *default* exceptions – i.e., using the `Exception` type – or ad-hoc exceptions. Likewise, an operation op^C from a candidate case may define a *fault* (the WSDL name for non-standard outputs of operations) as a message including an specific attribute. The exceptions similarity value is calculated according to the following cases:

- $Exc = 1$: op^N and op^C have equal amount, type and order for exception.
- $Exc = 2$: op^N and op^C have equal amount and type for exception into the list.
- $Exc = 3$: if nonempty op^N exception list then nonempty op^C exceptions list.
- $Exc = 0$: op^N and op^C exceptions are not compatible.

In fact, in the context of Web Services, faults definitions have not become a common practice [8]. However, the Case-based Reasoner considers this simple schema to analyze exceptions.

Example. Lets consider the following operations for obtaining rates for Car Rental, according to different vehicles and conditions (from the cases presented in Section 4.1):

- `getCarFee(requiredCarSupplements: CarSupplements): Fee throws unavailableSupplementsException;`
- `getRate(currencyCode: String, vehicleTypeId: long, AutomaticGearPreference: boolean): Rate throws vehicleNotFoundException, rateNotFoundException;`

If we consider `getCarFee` as $op^N \in newCase$ and `getRate` as $op^C \in candidateCase$ respectively, the exceptions analysis shows that the required operation throws an exception that may have two likely exceptions (form different types) in the candidate service's operation – as defined in case (2).

4.6 Similarity value

The similarity value between two cases (C^N, C^C) is calculated according to Formula 7.

$$sim(C^N, C^C) = \frac{\sum_{i=1}^{N}(Max(simOp(op_i^N, C^C))}{N} \tag{7}$$

where N is the number of operations in C^N, and $simOp$ is the best equivalence value $simOpValue(op_i^N, op_j^C)$ for all op_j^C in C^C

The value for operation similarity ($simOpValue$) between an operation op^N and a potentially compatible operation op^C is calculated according to Formula 8.

$$simOpValue(op_i^N, op_j^C) = Ret + Exc + Name + Par \tag{8}$$

Table 5
Operations matching for the Car Rental cases

`RentaCar` (*newCase*)	`CarRentalBrokerService` (*candidateCase*)	*simOpValue**
`getReservation`	`getCurrentBooking`	7.3
`getCarFee`	`getRate`	2.4
`cancelReservation`	`cancelBooking`	6.4
	$sim(newCase, candidateCase)$	5.4

* Higher is better

Table 6
Summary of the required calculations

Attribute	**Weight***	**Evaluation**	**Result****
service name (Formula 3): s_N	0.1	$ICValue(CarRental, CarRentalBrokerService)$	0.5
category (Formula 3): *cat*	0.1	$ICValue(Business, Business)$	1
operations number (Formula 2):$\#op$	0.3	$sim(3, 4)$	1
operations (Formula 7): *ops*	0.6	$sim(CarRental, CarRentalBrokerService)$	5.4

* The sum of all weights is 1. The higher the weight the more important the attribute.

** Higher is better.

Example.

Lets consider the full cases of the Car Rental case study, presented in Section 4.1. Table 5 shows, for each required operation $op^N \in newCase$, the operation $op^C \in candidateCase$ with higher compatibility (according to their $adapOpValue$) in the interface of the candidate Web Service. Calculations were done using the WordNet semantic basis accessed through the JWI library.

As the higher (better) *sim* value is 8, the obtained *sim* value (5.4) can be considered as moderate to high.

After obtaining the similarity value for operations, we can calculate the distance between the cases presented in 4 according to the Formula 1. Table 6 shows a summary of the required calculations to obtain the distance value.

Let be new case $C^N = CarRental$ and candidate case $C^C = CarRentalBrokerService$ the distance between the cases is:

$$DIST(C^N, C^C) = 0,1 * s_N + 0,1 * cat + 0,3 * \#op + 0,6 * ops$$
$$DIST(C^N, C^C) = 0,1 * 0,5 + 0,1 * 1 + 0,3 * 1 + 0,6 * 5,4$$

$$DIST(C^N, C^C) = 3,69$$

5 Experimental evaluation

This section describes the experimental evaluation of the CBR for service selection presented in the previous sections. The goal of the experiments is to measure the overall performance of the three alternative implementations of the CBR for service selection in comparative terms. We adopted an empirical, automatized and widely used methodology [6, 16, 28, 36]. Our hypothesis is that CBR for service selection could increase visibility of the most relevant services for certain required functionality.

5.1 Experiment configuration

The considered data-set consisted in 62 services extracted from the data-set of [20]. We have generated (through a tool developed in our group) one case for each service to settle the initial *KB*, according to the Object-oriented case representation presented in Section 3.1.

First, we extracted operation signatures from the 62 relevant services. Each new case consisted of three operations representing required functionality. Then, we applied interface mutation techniques [17] to generate 506 new cases (only problem part). We applied three mutation operators [9] to each operation signature:

- *Encapsulation* – where a random number of parameters are encapsulated as fields of a new complex data type.
- *Flatten* – where a random number of complex parameters are flattened generating as many parameters as fields in the complex type.
- *Upcasting* – where the return type and/or a random number of parameters are upcasted either to a direct supertype or to the wildcard `String` type.

5.2 Case-based Reasoning Execution

To execute the CBR for service selection, we have defined one scenario considering three implementations according to the similarity functions presented in Section 4, and the 506 new cases generated by mutating operation signatures. Considering traditional techniques of service retrieval and selection, we also populated the EasySOC service registry [10] with the relevant services, and then queried such registry with the operation signatures. EasySOC leverages Vector Space Model (VSM) and Web Service query-by-example (WSQBE) to represent Web Service descriptions and queries.

The three versions of the CBR for service selection were executed to rank the retrieved cases. Finally, the results of each new case are measured in terms of two

[9] https://code.google.com/p/querymutator/

well-known Information Retrieval metrics: *recall* and *precision-at-n*.

5.3 Results

Considering the results list as the first 10 retrieved cases for each query, we compared the results according to *precision-at-n* and *recall*.

Precision-at-n

Indicates in which position are retrieved the relevant services, at different cut-off points. For example, if the top five documents are all relevant to a query and the next five are all non-relevant, precision-at-5 is 100%, while precision-at-10 is 50%. In this case, precision-at-n has been calculated for each query with n in [1–10].

Recall

Formally, Recall is defined as:

$$Recall = \frac{Relevant}{R}$$

Where Relevant is the number of relevant services includes in the results list and R is the number of relevant services for a given query.

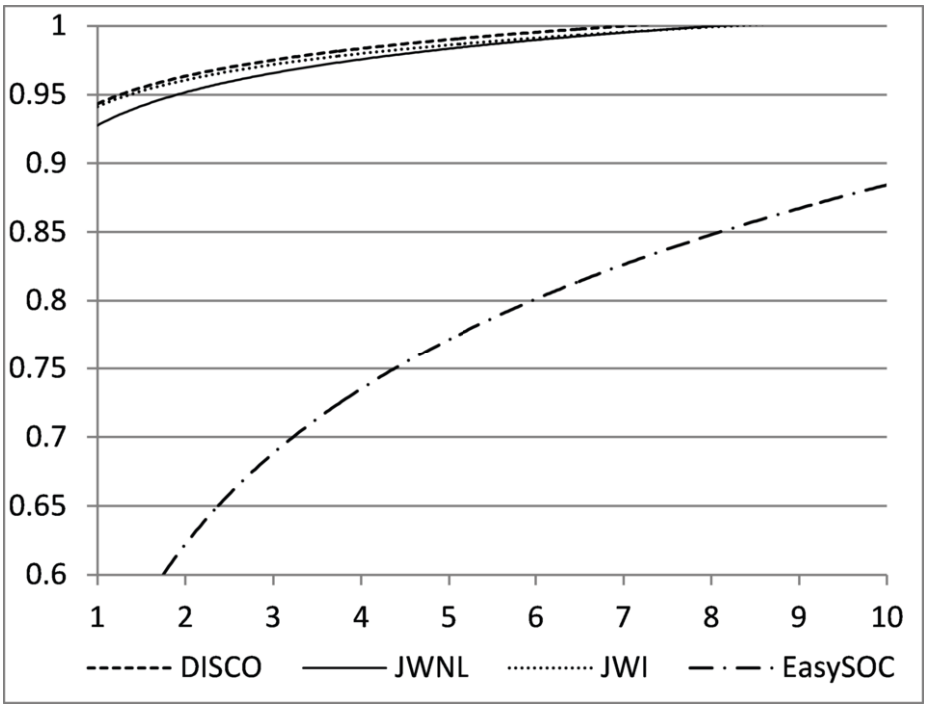
In particular, for this experiment the numerator of the Recall formula could be 0 or 1 – when the relevant service is/is not included within the results respectively – and the denominator (Retrieved) is always 10.

Figure 7a depicts the cumulative average precision-at-n (with n=[1,10]) for the three implementations of the CBR approach (JWI-, JWNL- and DISCO-based) and the EasySOC registry. The CBR for service selection obtained precision values over 90% for the first position of the results (with n = 1) with any implementation. Also, the difference among the precision-at-n of the three CBR implementations was not significant. The CBR for service selection outperformed EasySOC registry between 20% and 40% for the first positions of the results lists (with n=[1,4]).

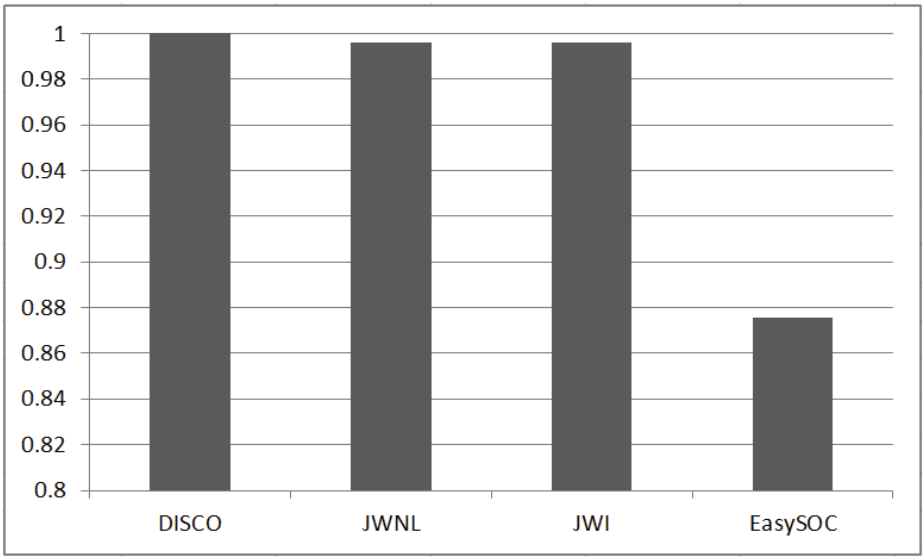
Figure 7b shows Recall results for the three implementations of the CBR-based service selection and the EasySOC registry. The results show that the CBR for service selection presents high recall values – over 98% – independently of the underlying implementation. This means that the relevant case for the given problem is almost always retrieved. The CBR for service selection outperformed recall results for EasySOC by about a 10%, although the EasySOC presented highly competitive values (over 85%) for recall as well.

5.4 Discussion

The results of the experiments have shown that our CBR for service selection approach achieves a high precision and recall with the three alternative implementations of the similarity function. Comparison with a service discovery registry (EasySOC) presented encouraging results as well. This confirms our hypothesis, as



(a) Precision-at-n



(b) Recall

Fig. 7. Results for the CBR implementations and EasySOC

CBR increased visibility of relevant services during service selection. This is significant since users tend to select higher ranked search results, regardless to their actual relevance [2]. The overall performance of the approach with multi-operation queries suggest the suitability for matching complex required functionality with many candidate services. In this direction, the reasoner could be extended to the

Web Service Composition (WSC) problem, by means of case adaptation using the K-nearest neighbors (K-NN) strategy [38]. Finally, the threshold value over the similarity function can be used to fine-tune the reasoner, according to the size of the initial *KB* and the domain. In this experiment, an average of 57% of the solved cases was added to the *KB* as new cases – i.e., 288 of the 506 queries.

As limitations, we can mention that the results can be specific for this experiment, and cannot be merely generalized to other experimental configurations. The dataset was relatively small (62 services), and the threshold values were fine-tunned by trial and error in the experimental scenario. In real scenarios, it would be wise to account the expert feedback from the Case Revision step to adjust the threshold value. Finally, the initial set of cases and solutions in a real scenario has to be manually built, which can be time consuming and also need expert feedback.

6 Related Work

6.1 Case-based Reasoning for Web Services

AI has contributed significantly to the Web Services field, either in the form of planning [4, 32, 34], abstraction and refinement techniques [21], or case based reasoning [24, 26].

The work in [24] presents an approach for WSC using CBR. This approach combines CBR with semantic specifications of services in the OWL-S language [27] to firstly reduce the search space of Web Services (i.e., improve service discovery), and then build an abstract composite process. Authors assume that Web Service providers are in charge of semantically annotating functional service descriptions according to the OWL-S ontology. However, this hardly occurs in practice, and most domains currently lack a descriptive ontology [6]. Our work exploits the most possible information in the (always available) service functional descriptions, to build the case representation.

The work in [26] also applies CBR for WSC. Similarly to our work, CBR is applied for service discovery, as a crucial step in the composition process. This approach requires knowing a priori a set of relationships between the services that compose the KB – e.g., dependence, substitutability and independency. However, this approach is strongly dependant of the Universal Description, Discovery and Integraton (UDDI) registry, that lacks a broad adoption in the industry [11]. Our approach is not tied to any particular discovery registry.

6.2 Structural-semantic service selection

Web Service similarity is addressed in [37] as a key solution to find relevant substitutes for failing Web Services. The approach calculates lexical and semantic similarity between identifiers comprising service names, operations, input/output messages, parameters, and documentation. To compare message structures and complex XML schema types, authors make use of schema matching. However, a straightforward comparison of complex types can be performed without dealing with

the complexity of an XML schema [16].

The Woogle search engine for Web Services is presented in [13]. Based on similarity search, Woogle returns similar Web Services for a given query based on operation parameters as well as operations and services descriptions. Authors introduced 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 [33] extends UDDI with UDDI Registry By Example (URBE), a Web Service retrieval algorithm for substitution purpose. The approach considers the relationships between the main elements composing a service specification (portType, operation, message, and part) and, if available, semantic annotations. The weak point of the approach is, as we stated earlier, that providers do not annotate their services often in practice, even when introducing annotations provides a more accurate description of the service.

7 Conclusions and Future Work

This paper presented the application of CBR to the problem of service discovery and selection. This approach leverages the semantic and structural information gathered from always-available functional descriptions of services. Also, the approach combines notions of CBR with the use of WordNet and DISCO as lightweight semantic basis. This results in a Case-based Reasoner capable of increasing the visibility of relevant services to fulfill certain required functionality – the relevant service was returned as suggested solution in about a 90% of the cases.

The proposed scheme was tested for three different similarity functions, which shown similar performance by considering the whole semantic and structural information available from services. However, it is mandatory to define and fine-tune adequately the threshold values to circunscribe the growing of the knowledge base. This can be done at runtime, accounting feedback from domain-experts and service-experts. As future work, we plan to extend our reasoner to the WSC field, by combining different cases as a solution for complex required functionality [26].

References

[1] Agnar Aamodt and Enric Plaza. Case-based reasoning; foundational issues, methodological variations, and system approaches. *AI COMMUNICATIONS*, 7(1):39–59, 1994.

[2] E. Agichtein, E. Brill, S. Dumais, and R. Ragno. Learning User Interaction Models for Predicting Web Search Result Preferences. In *29th Annual ACM SIGIR International Conference on Research and Development in Information Retrieval*, pages 3–10. ACM Press, 2006.

[3] Marcelo Armentano, Daniela Godoy, Marcelo Campo, and Analía Amandi. Nlp-based faceted search: Experience in the development of a science and technology search engine. *Expert Syst. Appl.*, 41(6):2886–2896, 2014.

[4] Piergiorgio Bertoli, Marco Pistore, and Paolo Traverso. Automated composition of web services via planning in asynchronous domains. *Artificial Intelligence*, 174(3):316–361, 2010.

[5] M. Bichler and K.J. Lin. Service-Oriented Computing. *Computer*, 39(3):99–101, 2006.

[6] Djelloul Bouchiha, Mimoun Malki, Abdullah Alghamdi, and Khalid Alnafjan. Semantic web service engineering: Annotation based approach. *Computing and Informatics*, 31(6):1575–1595, 2012.

[7] Frans Coenen. Data mining: past, present and future. *The Knowledge Engineering Review*, 26(01):25–29, 2011.

[8] M. Crasso, J. M. Rodriguez, A. Zunino, and M. Campo. Revising WSDL Documents: Why and How. *IEEE Internet Computing*, 14(5):48–56, 2010.

[9] M. Crasso, A. Zunino, and M. Campo. A survey of approaches to web service discovery in service-oriented architectures. *Journal of Database Management (JDM)*, 22(1):102–132, 2011.

[10] Marco Crasso, Cristian Mateos, Alejandro Zunino, and Marcelo Campo. Easysoc: Making web service outsourcing easier. *Information Sciences*, 259:452 – 473, 2014.

[11] Yue Dai, Yongxin Feng, Yuntao Zhao, and Yingchun Huang. A method of uddi service subscription implementation. In *Software Engineering and Service Science (ICSESS), 2014 5th IEEE International Conference on*, pages 661–666. IEEE, 2014.

[12] A. De Renzis, M. Garriga, A. Flores, A. Cechich, and A. Zunino. Semantic-structural assessment scheme for integrability in service-oriented applications. In *Computing Conference (CLEI), 2014 XL Latin American*, pages 1–11, Sept 2014.

[13] Xin Dong, Alon Halevy, Jayant Madhavan, Ema Nemes, and Jun Zhang. Similarity search for web services. In *Proceedings of the International Conference on Very Large Data Bases VLDB*, pages 372–383. VLDB Endowment, 2004.

[14] J. Erickson and K. Siau. Web Service, Service-Oriented Computing, and Service-Oriented Architecture: Separating hype from reality. *Journal of BD Management*, 19(3):42–54, 2008.

[15] Mark Alan Finlayson. Java libraries for accessing the princeton wordnet: Comparison and evaluation. In *Proceedings of the 7th Global Wordnet Conference, Tartu, Estonia*, 2014.

[16] M. Garriga, A. Flores, C. Mateos, A. Zunino, and A. Cechich. Service selection based on a practical interface assessment scheme. *International Journal of Web and Grid Services*, 9(4):369–393, October 2013.

[17] S. Gosh and A. P. Mathur. Interface Mutation. *Software Testing, Verification and Reliability*, 11:227–247, 2001. http://www.interscience.wiley.com.

[18] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java^{TM} Language Specification*. Sun Microsystems, Inc. Addison-Wesley, US, 3rd. edition, 2005. http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html.

[19] E. Hatcher, O. Gospodnetic, and M. McCandless. *Lucene in Action*. Manning Publications Greenwich, CT, 2004.

[20] A. Heß, E. Johnston, and N. Kushmerick. Assam: A tool for semi-automatically annotating semantic web services. In *The Semantic Web–ISWC 2004*, pages 320–334. Springer, 2004.

[21] Hyunyoung Kil, Wonhong Nam, and Dongwon Lee. Efficient abstraction and refinement for behavioral description based web service composition. In *IJCAI*, pages 1740–1745, 2009.

[22] Peter Kolb. Experiments on the difference between semantic similarity and relatedness. *Proceedings of the 17th Nordic Conference on Computational Linguistics - NODALIDA'09*, May 2009.

[23] Harold W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistic Quarterly*, 2:83–97, 1955.

[24] Soufiene Lajmi, Chirine Ghedira, and Khaled Ghedira. Cbr method for web service composition. In *Advanced Internet Based Systems and Applications*, pages 314–326. Springer, 2009.

[25] T. Warren Liao, Zhiming Zhang, and Claude R. Mount. Similarity measures for retrieval in case-based reasoning systems. *Applied Artificial Intelligence*, 12(4):267–288, 1998.

[26] Benchaphon Limthanmaphon and Yanchun Zhang. Web service composition with case-based reasoning. In *Proceedings of the 14th Australasian database conference-Volume 17*, pages 201–208. Australian Computer Society, Inc., 2003.

[27] D. Martin, M. Burstein, D. McDermott, S. McIlraith, M. Paolucci, K. Sycara, D. McGuiness, E. Sirin, and N. Srinivasan. Bringing semantics to web services with owl-s. *World Wide Web*, 10:243–277, 2007.

[28] C. Mateos, M. Crasso, A. Zunino, and J. L. Ordiales. Detecting WSDL bad practices in code–first Web Services. *International Journal of Web and Grid Services*, 7(4):357–387, 2011.

[29] R. McCool. Rethinking the Semantic Web. *IEEE Internet Computing*, 9(6):86–87, 2005.

[30] George Miller, Richard Beckwith, Christiane Fellbaum, Derek Gross, and Katherine Miller. Introduction to Wordnet: An On-line Lexical Database. *International Journal of Lexicography*, 3(4):235–244, 1990.

[31] J. Pasley. Avoid XML Schema Wildcards For Web Service Interfaces. *IEEE Internet Computing*, 10(3):72–79, 2006.

[32] Marco Pistore, Annapaola Marconi, Piergiorgio Bertoli, and Paolo Traverso. Automated composition of web services by planning at the knowledge level. In *IJCAI*, pages 1252–1259, 2005.

[33] Pierluigi Plebani and Barbara Pernici. Urbe: Web service retrieval based on similarity evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 21(11):1629–1642, 2009.

[34] J. Rao and X. Su. A survey of automated web service composition methods. In *International Workshop on Semantic Web Services and Web Process Composition (SWSWPC)*, pages 43–54, 2004.

[35] Christopher K Riesbeck and Roger C Schank. *Inside case-based reasoning*. Psychology Press, 2013.

[36] E. Stroulia and Y. Wang. Structural and Semantic Matching for Assessing Web-Services Similarity. *International Journal of Cooperative Information Systems*, 14:407–437, 2005.

[37] O. Tibermacine, C. Tibermacine, and F. Cherif. Wssim: a tool for the measurement of web service interface similarity. In *Proceedings of the french-speaking Conference on Software Architectures (CAL'13)*, Toulouse, France, May 2013.

[38] Ian Watson. Case-based reasoning is a methodology not a technology. *Knowledge-based systems*, 12(5):303–308, 1999.

[39] Peter Willett. The porter stemming algorithm: then and now. *Program: electronic library and information systems*, 40(3):219–223, 2006.