Service Selection Based on a Practical Interface Assessment Scheme

Abstract Service-Oriented Computing promotes building applications by consuming reusable services. However, facing the selection of adequate services for a specific application still is a major challenge. Even with a reduced set of candidate services, the effort ~~on~~ assessing candidates could be overwhelming. We have defined an approach to assist developers in the selection of services, which mainly comprises an assessment process for service *Interface Compatibility*. This assessment process is based on a comprehensive structural scheme for services' interfaces matching. The scheme allows developers to gain knowledge on likely services' interactions and their required adaptations to achieve a successful integration. We evaluated the performance of the *Interface Compatibility* analysis with a data-set of 453 services and two different service discovery registries. The experiments shown an improvement of up to 17% in precision and up to 8% in the DCG usefulness metric, with regard to the previous results obtained using only textual (syntactical) queries.

Keywords: Service oriented Computing, Web Services, Interface Compatibility, Web Service Discovery, Web Service Selection

1 Introduction

Service-Oriented Computing (SOC) is a paradigm that promotes the development of rapid, low-cost, interoperable, evolvable, and massively distributed applications through a network of services, which can create dynamic business processes that span organizations and computing platforms (Papazoglou et al., 2008). SOC lets developers dynamically grow application portfolios rapidly by creating compound solutions using organizational software assets – including enterprise information and legacy systems (Rodriguez et al., 2013b) – and combining them with external components residing in remote networks. From a business perspective, a *service-oriented application* implies a business facing solution which consumes services from one or more providers and integrates them into the business process (Sprott and L., 2004). From an architectural perspective, it can be viewed as a component-based application that is created by assembling both *internal* and *external* components, where the latter are statically or dynamically bound to services.

Mostly, the software industry has adopted the SOC paradigm by using Web Service technologies. A Web Service is a program with a well-defined *interface* (contract) and an *id* (URI), which can be located, published and invoked through standard Web protocols (Papazoglou et al., 2008). The Web Service contract (mostly specified in WSDL) exposes public capabilities as operations without any ties to proprietary communication frameworks. The terms "Web Service" and "service" will be used interchangeably in this paper.

However, a broadly use of the SOC paradigm requires efficient approaches to allow service consumption from within applications (McCool, 2005). Currently, developers are required to manually search for candidate services mainly exploring web catalogs usually showing poorly relevant information. This implies a large effort into discovering services and analyzing the suitability of retrieved candidates. Even with a reduced set of services, the required assessment effort could be overwhelming. Exploration on candidate services includes envisioning the required adaptations for a correct integration

and safe consumption of services. Without properly extracting meaningful information, selecting the most suitable candidate service resembles to fortune-telling.

In order to ease the development of service-oriented applications we have defined an approach to assist developers in the *selection* of services. This proposal is based on a recent approach (Flores and Polo, 2012), that was initially developed to select the most suitable off-the-shelf (OTS) software components as a solution for substitutability of component-based systems. Since Web Services involve a special case of software component (Kung-Kiu and Zheng, 2007), we were able to apply adjustments to such component selection to define an approach for service *selection*.

Particularly, this paper is focused in an assessment process for service *Interface Compatibility*, which mainly supports the selection method. This process has been defined to identify different structural aspects concerning the interfaces of candidate services. Through a comprehensive Assessment Scheme, interfaces are evaluated according to requirements of internal components from a service-oriented application. The matchmaking process is characterized through a series of structural compatibility cases. This conveys not only the usual programming standards (e.g., operation names and parameters), but also differentiating strong and potential similarity cases. Our previous work (Flores and Polo, 2012) had a reduced underlying model to cover just few matching cases, what made it unable to outline a likely solution for mismatching cases. Currently, the approach is implemented for the widely adopted Java platform[a]. In this context, Java interfaces are automatically derived from services' WSDL specifications – through the Apache Axis2 framework[b]. This encourages regular Java developers to adopt the SOC paradigm without a deep expertise in service technologies.

In this work, the Assessment Scheme has been divided into two main parts: automatic-strong matchings and semiautomatic-potential matchings. The former involves similarity cases directly recognized from Java interfaces of candidate services. The latter involves cases that could be solved trough a semi-automatic assistance. The whole information package gathered from this process provides an important insight about candidate services and their required adaptations for integration.

In addition, the process to yield such information can be seen as a white-box model that provides an explicit view of intermediate results from carried out evaluations. Conversely, other approaches usually provide just a high-level view of the final results (e.g., a synthesized numeric value), without a clear view of the underlying rationale. This avoids developers to gain knowledge about the reasons from accepting/rejecting candidate services, which eventually may help to improve design skills for architecting a system that might be filled up with third-party Web Services.

To sum up, the main contributions of this paper are:

- A comprehensive Assessment Scheme for service *Interface Compatibility* analysis.

- A semi-automatic assistance to fix service interface incompatibities.

- A white box model to improve design skills for architecting a service-oriented system.

- An early view of candidate services adaptation effort for integration.

In addition, performance of the *Interface Compatibility* analysis has been validated by running different experiment settings with a data-set of 453 services. Two different

---

[a]http://www.java.com/
[b]http://axis.apache.org/axis2/

service discovery registries were populated and queried. First, using only textual (syntactic) information. Then, executing the *Interface Compatibility* analysis with structural queries. The *precision-at-n* in the first positions of the retrieved lists of candidate services has shown an improvement of up to 17%.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 gives details of *Interface Compatibility* analysis, also introducing a case study to illustrate the assessment process. Section 4 describes the experimental evaluation of the *Interface Compatibility* analysis. Conclusions and future work are presented afterwards.

2 Related Work

First, two surveys of service discovery approaches are introduced. Then, current approaches are presented. Finally, a summary of the related work is outlined.

The surveys in (Kokash, 2006, Crasso et al., 2011) provide a comparative analysis of existing approaches to improve Web Services discovery. This is closely related to service selection, since an improved discovery method performs a partial preliminary selection among candidates. Particularly, Information Retrieval (IR) techniques have been used on several approaches as an effort to increase precision of Web Service discovery without involving any additional level of semantic markup. Although such approaches report concrete improvements, they seem to be insufficient for automatic retrieval if they are applied without using any complementary technique. A strategy for a semantic basis consists of formal ontology-based methods, which yet involve a high cost making service designers be alienated from their use in practice (Kokash, 2006). One of the main differences is what such approaches consider/require as service descriptions. Semantic approaches depend on shared ontologies and annotated resources, whereas IR-based ones depend on textual descriptions. Although those service discovery systems strive to solve the same problem, they may be appropriate in a particular environment but not in others (Crasso et al., 2011).

The approach in (Stroulia and Wang, 2005) is focused in the support of programmatic service discovery. The authors have developed a suite of methods to assess the similarity between two WSDL specifications based on the structure of their data types and operations, and the semantics of their natural language descriptions and identifiers. Given only a textual description of the desired service, a semantic IR method can be used to identify and rank the most relevant WSDL specifications. If a (potentially partial) specification of the desired service behavior is also available, this set of likely candidates can be further refined by a semantic structure-matching step. The structural and semantic similarity of the retrieved services are summarized through a similarity score. However, this score is not intended to represent the underlying adaptation effort after selecting certain candidate service. In our approach, the assessment process is mainly concerned on addressing adaptation factors, which fosters safe interaction with other services to deliver truthful business processes.

The approach in (Wu and Wu, 2005) presents a service discovery process in which a Web Service Conceptual Model must be built-up and registered into a UDDI-based registry. The conceptual model consists of four categories: common properties, special properties, interface and quality of service (QoS). Common properties include service name, key, description, owner and URI. Special properties include domain knowledge. Interfaces consists of sets of operations, described by operation names and input/output

parameters. QoS refers to the measurement of Web Service's usability, reliability and fidelity. These categories are built by parsing services' WSDL documents when stored into the registry. Necessarily, this approach relies on service providers that develop Web Services in a *contract-first* manner, a method that encourages designers to first derive a service's WSDL document to then supply an implementation, achieving better and highly discoverable services. However, the most used approach to build Web Services by the industry is *code-first* (Rodriguez et al., 2013b). This means that a developer first implements a service and then generates the corresponding WSDL document by using language-specific tools that automatically derive the interface from the service code. Then, WSDL documents are not directly created by humans but are instead automatically derived via language-dependent tools. Consequently, the resulting WSDL documents may manifest *anti-patterns*, a set of indicators of poor quality service interfaces (O. Coscia et al., 2013). Our approach, instead, relies only in service contract information that can be gathered from a WSDL specification generated either in a *code-first* or in a *contract-first* manner.

In (M. Nezhad et al., 2010) authors distinguish between two types of service mismatches: interface-level and protocol-level. Mismatches at the interface-level characterize heterogeneities related to operation definition in WSDL interfaces. This level includes message signature (different name and/or data types), message split/merge and missing/extra messages (M. Nezhad et al., 2007). Interface mismatches are addressed through a matching component implemented on top of the COMA++ tool[c] (Aumueller et al., 2005). COMA++ is a matching tool that uniformly supports schemes and ontologies – e.g., XML Schema and OWL. Thereby, interface-level mismatches are identified primarily by assessing service's XML schemes. As a complement, contextual information of message types (input/output) is gathered from the WSDL specifications. The message-level matching implies a high overhead by raising the number of required matchings up to the Cartesian product of number of messages in the two analyzed interfaces. Considering multiple candidate services (e.g., a window of 10 candidates as the experiment in Section 4), this strategy may not be applicable in practice. In our work, essential matching information is gathered from WSDL documents, since they usually declare most of the service's functionality-wise (M. Nezhad et al., 2007).

Likewise, Web Service similarity is addressed in (Tibermacine et al., 2013), as a key solution for service interoperability, mainly to find relevant substitutes for failing Web Services. This approach is parameterized (customized) by weighted scores and a set of similarity metrics, which are measured by analyzing WSDL specifications. The measurement process encompasses calculating (at the same time) 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 through a similarity flooding algorithm, representing complex types as labeled oriented graphs. As we stated earlier, an initial comparison of complex type similarity can be performed without dealing with the complexity of a XML schema. In our approach, the analysis of WSDL documents allows addressing complex data types while aiming a lightweight proposal.

The work in (Ouederni, 2011) is based on formal comparisons through a generic flooding-based technique for measuring the compatibility degree of service protocols. A generic framework is proposed, where the interfaces compatibility degree can be automatically measured according to different compatibility notions. A formal model has

_____

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

been defined for describing service interfaces with interaction protocols. A global and unique compatibility degree is calculated from the detailed measures to help in ranking and selecting some services from many possible candidates. This proposal seems to be more appropriate for critical systems, such as the automotive or aerospace industries where a formal basis may become crucial. Our approach can be considered as a programmatic-oriented model, more suitable for software developers engaged with business-centered requirements without a high-risk factor.

The work in (Ait-Bachir, 2008) presents a similarity measure between behavioral interfaces of Web Services by simulation. The authors differentiate among services' structural and behavioral aspects. The former implies provided operations and messages' schema expressed by WSDL specifications. The latter is defined by the control flow and interdependencies of the involved operations. In conversational services, such behavioral interfaces can be described using BPEL, for instance. Nevertheless, Finite State Machines is the formal model adapted in this work to describe behavioral interfaces. In particular, such structural aspect is related to our goals. Though, the authors express that their work does not include neither detecting nor fixing (structural) complex incompatibilities between compared services' interfaces.

In (De Antonellis and Melchiori, 2003) a comparison of services' structure based on a semantic markup is presented. This comparison is performed through a tool named ARTEMIS, which calculates a set of similarity coefficients and clusters the services to evaluate their level of compatibility. In this work, the assessment is accomplished between an abstract service and a concrete service instance from a certain category. Instead, we compare a required interface against the interface of a candidate service. Our work is structurally oriented, including an aspect which is usually neglected, related to failed function executions represented by exceptions, as mentioned in (Crasso et al., 2010). This aspect is even important on service protocols affecting expected execution sequences.

Finally, there is a competing architectural style for building Web Services besides the WS-* standards (SOAP): *RESTful* services. REST is an alternative approach to SOC which uses basic HTTP methods (PUT, POST, GET, and DELETE) applying their intended semantics to access a resource (Fielding, 2000). A resource is any information that could be referenced by an URI such as a document, an image, a tweet or a weather forecast[d]. In (Richardson and Ruby, 2008) authors identify four system properties of RESTful services: uniform interface, addressability, statelessness, and connectedness, that are embodied in resources. On the contrary, WS-* services exhibit all but the first property. A uniform interface shared by all services plus non-standardized documentation make the automatic discovery of RESTful services very hard in public repositories (Adamczyk et al., 2011). Besides, RESTful Web Services currently do not use a standardized format for representing resources – e.g., XML messages. Given their lack of formally described interfaces, they are also cumbersome to compose (Pautasso et al., 2008). In terms of architectural principles, conceptual decisions and technological decisions, the authors in (Pautasso et al., 2008) recommend using REST for ad-hoc integration and using WS-* for enterprise-level application integration where transactions, reliability, and message-level security are critical. Recent efforts in large-scale legacy system migration to SOA have demonstrated the suitability of WS-* technologies and standards (Rodriguez et al., 2013b).

---

[d]e.g., http://www.weather.gov/forecasts/xml/rest.php

Summary of related work.

First, some proposals rely on semantic descriptions of services (Di Martino, 2009) that generally are not available, since publishers must put extra effort into describing services by means of semantic meta-data (Crasso et al., 2011). As stated in (Brogi, 2011), a true spread of semantic services will only start when the derived advantages become of clear interest for the market. Second, there is an evident lack of meaningful information in WSDL documents due to the proliferation of code-first Web Services (Mateos et al., 2011) that decreases precision of service discovery registries. Finally, services are usually assessed to be integrated in business processes, interacting with other services and components. This creates the need of service *adaptation*, which is addressed mostly at protocol level (M. Nezhad et al., 2007). However, it is also necessary to support interface level adaptation as a non-trivial activity.

For these reasons, we propose a practical service selection method, which assesses services mainly through their WSDL specifications that are usually available. Through gathering available structural information from code-first (or contract-first) Web Service contracts, our approach has shown an improvement in precision w.r.t. initial results from service discovery registries. The assessment process is mainly concerned on comprehensively addressing interface adaptation, achieving early and meaningful amending information. Thereby, this can be seen as an adaptability-oriented approach.

3 Interface Compatibility Analysis

During development of a service-oriented application, specific parts of a system may be implemented in the form of in-house components. Besides, 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 to determine the most appropriate service for the consumer's application. Therefore, a reliable and practical support is required to make those decisions. In this approach, the *Interface Compatibility* analysis is the main assessment process to select the most appropriate candidate service. We assume for this analysis the availability of the documentation artifacts describing the expected software architecture (as knowledge sources). From these documents, useful information is outlined to avoid early discarding a candidate service upon simple mismatches, also preventing from a unmanageable incompatibility. In addition, the adaptation effort of a candidate may take shape for a successful integration into the consumer application.

This section provide details of the *Interface Compatibility* analysis. First, a case study is presented to illustrate the comprising steps of such analysis.

3.1 Proof-of-Concept

A simple case study has been outlined as a *Personal Communication Application* (PCA) being developed under the Java platform. The main required key feature is a *Chat tool*, which allows creating a user and a chat session (in a synchronous way) with any instant messaging client. This feature will be fulfilled by a third-party Web Service. Figure 1 shows a concrete structure for the PCA's required interface (I_R), namely ChatIF. By searching on web-cataloges, a list of candidate services has been built-up, as presented

in Table 1. Notice that although the corresponding *URI* for each candidate's WSDL specification is shown, due to a common volatility factor of prototype Web Services such *URI*s may change or even become unavailable after some time.
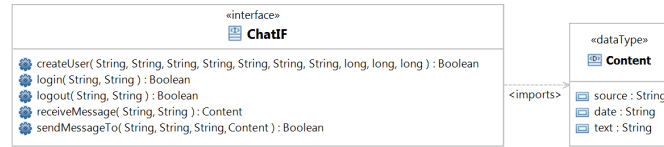

Figure 1: Required Interface for `PCA`'s main feature

Table 1 Candidate Services for `Chat` feature of `PCA`

Required Interface	Candidate Service	URI	Candidate Service	URI
ChatIF	OMS	www.nims.nl/soap/oms.wsdl	OMS2_simple	www.nims.nl/soap/oms2_simple.wsdl
	OMS2	www.nims.nl/soap/oms2.wsdl	OnlineMessenger[†]	127.0.0.1:8080/TestWebServices/ services/OnlineMessenger?wsdl

[†]deployed locally

In order to clearly illustrate the *Interface Compatibility* analysis, the candidate service `OMS2` is taken from Table 1, whose interface comprises a set of 38 operations using some complex types such as `Message`. The evaluation of the remaining candidate services is presented in Section 3.5.

3.2   Assessment Scheme

The main asset of the *Interface Compatibility* analysis is a practical Assessment Scheme that covers a comprehensive range of matching cases, from which a developer may easily understand causes of compatibility results. The scheme has been divided into two parts: automatic matching cases and semi-automatic potential matchings. Both parts characterize structural similarity cases into four compatibility levels (named: *exact, near-exact, soft, near-soft*). This allows comprehensively describing similarity cases representing structural constraints for pairs of operations ($op_R \in I_R, op_S \in I_S$), where I_R is a required interface and I_S is the interface of a candidate service S. Particularly, those constraints are based on individual conditions for each element comprising operations signature: the *Return type* (R), the *operation Name* (N), the *Parameters list* (P), and the *Exceptions list* (E). Table 2 summarizes the set of operation matching conditions.

Table 2 Structural Operation Matching Conditions for *Interface Compatibility*

Return Type	R0: Not compatible	R1: Equal return type
	R2: Equivalent return type (subtyping, Strings or Complex types)	R3: Not equivalent complex types or lost precision
Operation Name		N1: Equal operation name
	N2: Equivalent operation name (substring)	N3: Operation name ignored
Parameters	P0: Not compatible	P1: Equal amount, type and order for parameters
	P2: Equal amount and type for parameters	P3: Equal amount and type at least equivalent (subtyping, Strings or Complex types) for some parameters into the list
	P4: Not equivalent complex types or lost precision	
Exceptions	E0: Not compatible	E1: Equal amount and type, and also order for exceptions
	E2: Equal amount and type for exceptions	E3: If non-empty original's exception list, then non-empty candidate's list (no matter the type)

Table 3 presents the Assessment Scheme that is able to recognize 108 cases for *Interface Compatibility*, divided in two parts of 54 cases, from the combination of individual conditions. Following is presented the first part of the scheme which can automatically recognize direct matching cases. Then, the second part of the scheme applied for solving mismatching cases is detailed.

Table 3 Assessment Scheme: Automatic and Semi-Automatic Matching

Level	Part	Constraints
■ Exact Match	Automatic (1 case)	Two operations must have identical signatures (four identical conditions): [R1,N1,P1,E1]. This implies an equivalence value of 4 (by adding the value 1 of each condition)
■ Near-Exact Match	Automatic (13 cases)	Three or two identical conditions. The remaining might be conditions (R2/N2/P2/E2). Exceptional cases: three identical conditions with a remaining third condition (N3/P3/E3), implying equivalence values between 5 and 6.
	Example:	operation logout of ChatIF has a *near-exact* equivalence with OMS2_Logout of OMS2 by three identical conditions and a substring equivalence for the operation name: [R1,N2,P1,E1]
	Semi-Automatic (1 case)	Three identical conditions with the return that may have a no equivalent complex type or lost precision: [R3,N1,P1,E1]. This implies an equivalence value of 6.
■ Soft Match	Automatic (26 cases)	Similar to the previous level, but only two identical conditions. Previous exceptional cases may occur with lower equivalence conditions. This implies equivalence values between 7 and 8.
	Semi-Automatic (13 cases)	Two identical conditions, similar to automatic scheme. Either return or parameter (not both) with a nonequivalent complex type or lost precision (R3/P4). This also implies equivalence values between 7 and 8.
	Example:	operation sendMessageTo of ChatIF could match operation OMS2_PostMessage. However, the first operation includes a complex parameter (Content) without an automatic match. This can be re-evaluated considering that the wildcard type String might contain a concatenation of fields from the complex type -- i.e. a *soft* equivalence [R1,N2,P4,E1]
■ Near-Soft Match	Automatic (14 cases)	There cannot be two identical conditions, i.e. all conditions can be relaxed simultaneously. This implies equivalence values between 9 and 11.
	Semi-Automatic (40 cases)	Either two identical conditions with the condition P4 or relaxing all conditions simultaneously. This implies equivalence values between 9 and 13.

Assesment Scheme: Automatic Matching

This first part of the Assessment Scheme has been defined with a set of specific constraints describing highly meaningful matching cases between operations from I_R and I_S. The four compatibility levels are outlined from such constraints, by different combinations of structural conditions for an operation signature – according to Table 2.

The automatic part of the Assessment Scheme comprises the strongest constraints, to clearly identify direct matching cases. A criterion of "no-inclusion" was defined involving two low compatibility conditions from Table 2: R3 and P4. These conditions are evaluated in the first part of the scheme as incompatibilities – R0 and P0 respectively. For R3, either the return type of operation op_R is a complex type without equivalence to the return type of operation op_S, or the return type of op_S implies losing precision w.r.t. the return type of op_R – e.g., the candidate has an int type but a double type is required. For P4, either some parameter of op_R is defined through a complex type without equivalence to any parameter into op_S, or a pair of corresponding parameters implies losing precision. This occurs between operations sendMessageTo of ChatIf and Oms2_PostMessage, as shown in the example of Table 3.

Assessment Scheme: Solving Mismatches

In general, when certain mismatch cases are detected for the interface I_R, a developer may outline a likely solution with the support of context information from the application's business domain. We have identified specific cases in which a concrete compatibility can be set up by providing a semi-automatic mechanism to ease this procedure. Thus, a given operation $op_R \in I_R$ can be linked to a specific operation $op_S \in I_S$ (of a candidate Web Service *S*), with which initially there was no correspondence through the automatic interface assessment.

The second part of the Assessment Scheme shown in Table 3 has been defined upon the 4 compatibility levels as well. The *no-inclusion criterion* applied to identify incompatibilities during the automatic assessment process becomes now the base to recognize potential cases to save a candidate service *S* from being early discarded. Therefore, conditions R3 (return) and P4 (parameters) from Table 2, are considered for combinations with other conditions for signature elements. This part of the Assessment Scheme is comprised of 54 additional cases making the whole scheme able to recognize 108 cases for *Interface Compatibility*.

In addition, the goal of this second part is not only to assist on solving mismatch cases, but also to "prioritize" certain correspondences even when an automatic match has been previously identified. In this case, a developer may consider that for a specific operation $op_R \in I_R$, there is another correspondence that better fits for the application's context. Such correspondence may be considered first for adapting candidate service operations for integration in a service-oriented application. For example, operation `SendMessage` from `ChatIF` could match operation `OMS2_SendMessageToChat`. However, a developer may outline a likely matching with operation `OMS2_PostMessage` that is prioritized for adapting candidate service operations.

The set of structural conditions summarized in Table 2 are detailed in the following section to identify equivalence on operations' signatures.

3.3 Assessment Scheme: Structural Equivalence Conditions

Let I_R be an interface of a required functionality for a client application and I_S the interface of a candidate Web Service *S*. For every pair of operations (op_R,op_S), where $op_R \in I_R$ and $op_S \in I_S$, a likely equivalence between those operations is based on the structural conditions for each element of an operation's signature presented in Table 2.

Notice that signature elements are named according to the Java terminology, rather than using the WSDL convention for Web Service contracts. The reason is that assessments are performed upon Java interfaces, previously derived from WSDL specifications. Thus, in case of *response* and *fault* from a *message* definition, terms *return* and *exception* are used instead respectively.

Main aspects concerning the set of structural conditions are detailed below.

Data Type Equivalence – Subtyping

Conditions R2 and P3 concern data type equivalence according to Table 2. This involves the *subsumes* relationship or subtyping (written <:), which implies a *direct* subtyping (written $<_1$) in case of built-in types in the Java language (Gosling et al., 2005), as shown in Table 4. Thus, subtyping relations are considered for operations' signatures as shown in Table 5, where it is expected that types on operations from I_S have at

least as much precision as types on I_R. For example, if operation op_R includes an `int` type, then a corresponding operation op_S cannot have a lower precision like `short` or `byte` (among numerical types). There is a special case with the `String` type, which is considered as a "*wildcard*" type since it is generally used in practice to allocate different kinds of data. This practice is defined in (Pasley, 2006) as a risk factor in the attempt to make Web Services' interfaces extensible. As a negative side effect, this increases complexity and results in ambiguous interface definitions. Since this practice is still commonly used, in this approach the `String` type is considered as a supertype for all numerical types.

Table 4 Built-in Direct Subtyping

1st Type		2nd Type
byte	$<_1$	short
short	$<_1$	int
int	$<_1$	long
long	$<_1$	float
float	$<_1$	double

Table 5 Subtyping Equivalence for Operations

Type on op_R	Type on op_S
char	String
byte	short, int, long, float, or double, or String
short	int, long, float, or double, or String
int	long, float, or double, or String
long	float, or double, or String
float	double, or String
double	String

Complex Data Types

Complex data types require a special treatment in which there must be a correspondence for each field of a complex type from an operation op_R to a counterpart complex type of an operation op_S. For conditions (`R2`, `P3`), if operation op_R includes a complex type in return or parameters, there must be an equivalent counterpart complex type into operation op_S. This means, they must coincide on number and order of fields (one-to-one) inside the corresponding complex data types. For conditions (`R3`, `P4`), if operation op_R includes a complex type in return or parameters, there must be an equivalent counterpart complex type into operation op_S, where the order of fields' types is relaxed.
Example. Operation `receiveMessage` of `ChatIF` returns a complex type (`Content`), and operation `OMS_ReceiveMessage` of the `OMS2` service also returns a complex type (`Message`). Both complex types are equivalent because their fields have equivalent types (`R2`). Therefore, operation `receiveMessage` has a *near-exact* equivalence with `OMS_ReceiveMessage`, since they coincide on parameters and exceptions with an equivalent operation name – i.e., [R2,N2,P1,E1].

Operation Name Equivalence

Condition `N2` implies that an operation op_S includes an equivalent operation name with regard to operation op_R. Name equivalence concerns identifying a substring similarity, by considering the current naming conventions, as presented in Table 6. In general, developers combine a verb and a noun for denoting an operation name, such as `getQuote` or `get_quote`, from where the name can be decomposed into words `get` and `quote`, and a likely string coincidence could be found.

Table 6 Rules for decomposing an Operation Name

Notation	Rule	Source	Result
Java Beans	Split when changing text case	getZipCode	get Zip Code
Hungarian	Split when changing text case	ulAccountNum	ul Account Num
Special Symbols	Split when either "_" or "-" occurs	get_Quote	get Quote

Analyzing Exceptions

In general, a functionality enclosed into an operation requires input and output data represented by the operation's parameters and return respectively. However, a complete and more adequate design of any operation's functionality should consider exceptions (i.e., *faults* in WSDL terminology). In fact, in the context of Web Services, faults definitions have not become a common practice. This phenomenon is defined in (Crasso et al., 2010) as an anti-pattern for well-defined WSDL interfaces, named "*Undercover fault information within standard messages*", where output messages are used to notice about a service's operation error. Therefore, developers should be prevented from handling such hidden faults in an incorrect manner. Hence, by assuming software development best practices, exceptions or faults must be analyzed accordingly.

In the Assessment Scheme, certain aspects are considered when analyzing exceptions. First, any operation $op_R \in I_R$ may define a default type exception – i.e., named "Exception" – or ad-hoc exceptions. Also, an operation op_S (of a candidate service S), may define a *fault* as a message that includes a specific attribute, which acts as the exception name. In the Assessment Scheme, for conditions `E1` and `E2` each exception type declared into operation op_R must have an identical corresponding exception type into op_S. In addition, the size of both exception's lists must coincide. For condition `E3`, if there is at least one exception type declared in op_R there must be at least one exception type (not necessarily alike) in op_S.

3.4 Assessment Scheme: Compatibility Gap Value

The final outcome of the *Interface Compatibility* analysis is a matching list, where each correspondence is characterized according to the four levels of the Assessment Scheme, named *Interface Matching* list. This outcome makes the whole procedure a white-box model providing an explicit view of carried out evaluations, to understand reasons for accepting/rejecting candidate services. For each operation $op_R \in I_R$, a list of compatible operations from I_S is shaped. For example, let be I_R 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 Assessment Scheme. Each of the four levels of compatibility aggregates different equivalence cases, which also allows generating additional information concerning a specific numeric equivalence value for those cases. For example, the value of *exact* equivalence [R1,N1,P1,E1] is 4 as a result from adding the value 1 of each condition. Therefore, from the *Interface Matching* list, a totalized equivalence value could be calculated, named *Compatibility Gap* (formula 1), which allows to synthesize the achieved degree of *Interface Compatibility* for a candidate interface I_S (from a service S) w.r.t. a required interface I_R. Only the strongest equivalence degree (lower value) from the *Interface Matching* list is considered for each operation in I_R.

$$compGap(I_R, I_S) = \frac{\sum_{i=1}^{N} Min(op_{Ri}, CompMap(I_R, I_S))}{N * 4} - 1 \tag{1}$$

where N is the size of interface I_R, and *CompMap* are the values for the compatibility cases found for operation op_{Ri}.

In case that all operations in the *Interface Matching* list present an *exact* equivalence, the *Compatibility Gap* between I_R and I_S is zero. Although this might seem a perfect interface match, this initially only means that I_R is included into I_S, while the size of interface I_S may be larger, including additional operations. The success on the precision achieved during the *Interface Compatibility* analysis is essential to reduce the subsequent adaptation effort when integrating a candidate service into a SOC-based application. This is the main reason for the definition of the whole Assessment Scheme, in which different design and programming heuristics have been applied, mostly from a practical perspective.

3.5 Results of the Case Study

This section presents the results of the *Interface Compatibility* analysis for the case study about the *Personal Communication Application* (PCA) presented in Section 3.1.

The required feature for PCA was described as a simple *Chat Tool*, from where the required interface ChatIF was built-up. A set of candidate services was also presented in Section 3.1.

After assessing the candidate services, the best (lower) matching value per operation is available. Thus, the *compatibility gap* can be calculated, as shown in Table 7 – except for the OMS service marked as *not-applicable* (N/A), since its interface does not contain an operation to match the logout operation of ChatIF. In the case of ChatIF and the OMS2 service the *compatibility gap* can be calculated as $29/20 - 1 = 0.45$ according to formula (1). Because the lower value is the better, the suggested candidate to fulfill the required *Chat* functionality would be the OMS2 service. The complete set of correspondences between ChatIF and the OMS2 service is shown in Table 8, with the best (lower) associated equivalence degrees, conditions and values.

Table 7 Summary of Interface Compatibility for ChatIF and candidate services

Candidate Service	Total Equivalence[‡]	Compatibility Gap
OMS	N/A	N/A
OMS2	29	0.45
OMS2_Simple	36	0.8
OnlineMessenger	30	0.5

[‡] *Total best value*: 20 (based on ChatIF size)

Table 8 Final Interface Compatibility between ChatIF and the OMS2 service

ChatIF	OMS2	Degree	Conditions	Value
createUser	OMS_CreateUser	near-exact	R1, N2, P1, E1	5
sendMessage	OMS_PostMessage	soft	R1, N2, P4, E1	8
receiveMessage	OMS_ReceiveMessage	near-exact	R2, N2, P1, E1	6
logout	OMS2_Logout	near-exact	R1, N2, P1, E1	5
login	OMS_Login	near-exact	R1, N2, P1, E1	5
ChatIF size: 5			*Total Equivalence*	29

At this point, a selected candidate service is available for the *Chat* feature of the (PCA) application. Candidate services were assessed without any knowledge from their underlying model and logic rules, but only analyzing their provided interfaces (*service contract*). After carrying out this simple case study, it is glimpsed how a developer may gain specific and valuable knowledge about an application's context by the support of the Assessment Scheme upon the *Interface Compatibility* analysis. Moreover, an experimental evaluation of the process is presented in the following section.

4 Experimental Evaluation

To evaluate the *Interface Compatibility* analysis, we used a data-set to populate two service registries with both relevant and irrelevant services. Relevant services were taken from the data-set in (Mateos et al., 2011). Such services came from real-life projects implementing either Web Services or servified EJBs (Enterprise Java Beans). The projects were gathered from Google Code, Exemplar (Grechanik et al., 2010) and Merobase[f].

A set of syntactic queries was automatically generated from the data-set and used to query the service discovery registries. The results for certain subsets of queries were post-processed converting the retrieved WSDL documents to Java interfaces through the Apache Axis2 WSDL2Java[g] tool. Then, the queries were expanded to add structural information by different combinations of operation's signature elements (return types, parameters). This allows representing each structural query as a fully-described single-operation required interface (I_R). Finally, the *Interface Compatibility* analysis was performed with the Java interfaces (I_S) of the candidate services and the expanded structural queries as required interfaces (I_R).

Indeed, most Web Service discovery registries consist on a catalog-style browsing based on keywords matching. Hence, a structural-based service assessment process can be used to identify and to rank the most relevant WSDL specifications (Stroulia and Wang, 2005). Thereby, the goal of this experiment is to analyze whether performing the *Interface Compatibility* analysis after querying a service discovery registry, could increase the probability of selecting a more suitable candidate service.

4.1 Experiment configuration

Data-set Preprocessing.

The considered data-set consisted in 60 relevant services mentioned above, plus 393 noisy WSDL specifications extracted from the data-set in (Heß et al., 2004). The resulting data-set of 453 services was used to populate two service discovery registries:

- *EasySOC* (Rodriguez et al., 2013a) service registry, which maps queries and services onto vectors in the Vector Space Model and uses the Query-by-example search engine (Crasso et al., 2008).

- Apache *Lucene*[h] registry, a well-known cross-platform text search engine based on indexation and Information Retrieval techniques (Hatcher et al., 2004). As Lucene has been designed for indexing any kind of textual documents, reserved words of WSDL may introduce noise to the search engine. For this reason, we used a WSDL-aware Lucene version (Rodriguez et al., 2010).

These service discovery registries implement distinct discovery mechanisms with different performance, which mainly depends on the used queries and data-set. Using two service discovery registries allows us to carry out an unbiased experiment.

Query Expansion.

An original set of 430 syntactic queries consisting only in operation names was automatically generated from available service operations in the data-set. Then, some query

---

[f]http://merobase.com
[g]http://axis.apache.org/axis2/java/core/
[h]http://lucene.apache.org/core/

expansions were defined to include structural information about the *return type* and *parameters*. After that, each expanded query can be encapsulated as a Java (required) interface with only one operation.

Initially, three different expansions were defined to perform service *Interface Compatibility* analysis, as follows:

- Query Expansion 0 (QE0). A default query expansion, trying to capture the semantics of the original syntactic query which comprised only the operation name. Queries are expanded with a `void` return type, no parameters and no exceptions – e.g. if the syntactic query was `getUser`, applying QE0 results in: `void getUser()`. This expansion is always included, as a representation of the original queries.

- Query Expansion 1 (QE1). A *wildcard* query expansion, including a `String` parameter and return type, and no exceptions. As pointed out in Section 3.3, the `String` type is generally used in practice to allocate different kinds of data – e.g., for the syntactic query `getUser`, applying QE1 results in: `String getUser(String x1)`.

- Query Expansion 2 (QE2). An ad-hoc query expansion. When analyzing the WSDL documents of the data-set, we noticed that many of them defined complex types to encapsulate either return data or parameters. Regardless of their structure (fields), those complex types are in general named as the operation, adding the suffix "Response" for return types. Thereby, QE2 captures this fact adding an ad-hoc complex type for the parameters and another ad-hoc complex type for the return – e.g., the `getUser` query is expanded as: `GetUserResponse getUser(GetUser x1)`.

Although each query expansion was instantiated in a certain way for this experiment, the provided guidelines could lead to the definition of different instantiations. When assessing another set of structural features in services' specifications, the query expansions could be parametrized in a distinct way.

Query Replacement.

To decide which of the 430 syntactic queries will be replaced (with structural queries), we analyzed the results of textually querying both service discovery registries – *EasySOC* and *Lucene* – from a developer's perspective. In this context, finding a needed Web Service in a registry involves two steps. First, ranking WSDL specifications according to a given syntactic query, automatically performed through the discovery step. Then, inspecting the top *n* results, which is usually done by developers manually. When relevant WSDL specifications are not retrieved at the topmost positions – e.g., among the first three or four positions – it would be worthy to count with a mechanism that automatically rearranges (improving) service discovery results. To do this, we have defined two different *cutoff-point* criteria for query replacement:

- *Cutoff-at-4*. The syntactic queries where the relevant service appeared after the 4th position in the result list are replaced by expanded queries.

- *Cutoff-at-5*. The syntactic queries where the relevant service appeared after the 5th position in the result list are replaced by expanded queries.

Interface Compatibility Execution.

To execute the *Interface Compatibility* analysis in the context of this experiment, we defined two query expansion schemes, by combining the query expansions (QE0, QE1, QE2). On one side, QE0 + QE1, including the default query expansion and the *wildcard* one, aiming generality. On the other side, QE0 + QE2, including the default query expansion and the *ad-hoc* one, aiming specialization.

Then, two experimental *Scenarios* were defined bringing together the service registries, query expansions schemes and query replacement cutoff points. In *Scenario 1* services were published in the *EasySOC* registry. Considering the criteria defined above, the experiment configuration implies the execution of *Interface Compatibility* using QE0+QE1 and QE0+QE2 expansion schemes with *cutoff-at-4* and *cutoff-at-5*.

For *Scenario 2* services were indexed in the *Lucene* registry and the experiment configuration implies the execution of *Interface Compatibility* using QE0+QE1 and QE0+QE2 expansion schemes with a *cutoff-at-5* criterion. For this scenario, we have decided to only apply the *cutoff-at-5* criterion. This is based on preliminary experiments, the Scenario 1 results, and the Lucene results using the original syntactic queries. *Cutoff-at-4* would not bring substantial improvements in Scenario 2 – i.e., rearranging relevant services to better positions in candidate lists.

In both scenarios, an initial ranked list of the first 10 retrieved candidate services per syntactic query were considered. When the relevant service does not appear in the first 10 results, the window is extended until that service is found, so its Java interface can be generated. Then, the *Interface Compatibility* analysis is performed when the relevant service is not retrieved at the list's topmost positions.

Example. Let be the syntactic query `getUser` with relevant service `Accounting Service`. The retrieved candidate list (ordered by position) before executing *Interface Compatibility* could be:

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

As can be seen, the relevant service was retrieved in the fifth position. This is because the first four services also provide an operation named `getUser`, from a syntactical standpoint. Services in position 1 and 2 are from the data-set of real-life services and the following two are noisy services.

Through the *Interface Compatibility* analysis this list is rearranged considering structural information both from the expanded query and the services in the list. Thus, the relevant service is arranged in the first positions of the list – e.g., among the first three or four positions. In this case, the rearranged list could be:

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

Thus, the goal of this experiment is to show how the *Interface Compatibility* analysis could increase the visibility of a more suitable candidate service to be selected.

4.2 Results

For the two experimental scenarios, we queried the corresponding registry with the syntactic queries. Then, obtained results were post-processed by executing the *Interface Compatibility* analysis (with different query expansion schemes and cutoff-points). Finally, we compared both results according to three well-known IR metrics:

16

- **Precision-at-n**. Indicates in which position are retrieved the relevant services, at different cut-off points. For example, if the top 5 documents are all relevant to a query and the next 5 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]. This window size was decided considering the balance between number of candidates, relevant services per syntactic query and manageability of results list.

- **Recall**. Formally, Recall is defined as:

$$Recall = \frac{Relevant}{Retrieved} \tag{2}$$

In particular, for this experiment the numerator of the Recall formula could be 0 or 1 – i.e., when the relevant service is included within the results – and the denominator (Retrieved) is always 10.

- **DCG**. The DCG is a measure for ranking quality and measures the usefulness (gain) of an item based on its relevance and position in the provided list. The higher the DCG, the better ranked list. Formally, DCG is defined as:

$$DCG = \sum_{i=2}^{p} \frac{rel_i}{log_2 i} \tag{3}$$

where p is the size of the candidate list, and rel_i indicates if the candidate retrieved in the ith position of the list was relevant.

The DCG values for all queries can be averaged to obtain a measure of the average performance of a ranking algorithm, named Normalized DCG (NDCG).

Those metrics have been broadly used in the context of Web Service discovery and selection (F. Diaz et al., 2006, Rodriguez et al., 2010). Services containing the needed operation (one per query) were arbitrarily selected and associated to the query as the relevant one in order to evaluate the results. Finally, an average of each metric was generated over the total number of queries.

Scenario 1: EasySOC service registry.

In Scenario 1, the EasySOC registry was populated with the data-set and then queried with the 430 original syntactic queries. Figure 2 depicts the cumulative average Precision-at-*n* values corresponding to the original results, and the *Interface Compatibility* post-processed results. Figure 2a represents precision values where syntactic queries are replaced with the *cutoff-at-4* criterion, expanding 26% of the original queries. Figure 2b represents precision values where syntactic queries are replaced with the *cutoff-at-5* criterion, expanding 17% of the original queries. Results show that, in general, applying the *Interface Compatibility* analysis improves the Precision-at-*n* between 7% and 14% for the first positions (when n=[1-4]). Although precision tends to converge when *n* approaches to 10, the improvements in the first positions are significant since users tend to select higher ranked search results, regardless of their actual relevance (Agichtein et al., 2006). Moreover, with *cutoff-at-5* – i.e., replacing and expanding 17% of the syntactic queries – results are as good as with *cutoff-at-4*. This implies that executing *Interface Compatibility* analysis over less query results still brings improvements in terms of precision.
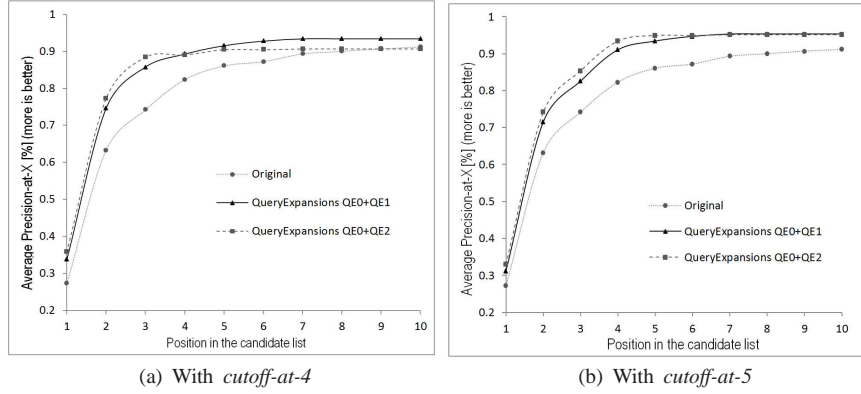
(a) With *cutoff-at-4* (b) With *cutoff-at-5*

Figure 2: Averaged Precision-at-*n* results for Scenario 1 (EasySOC)

Table 9 summarizes the results for Precision-at-*n* (with n=[1-4]), recall and normalized DCG in the first scenario, with either *cutoff-at-4* or *cutoff-at-5* for replacing and expanding queries. Table 9 suggests that performing the *Interface Compatibility* analysis only for queries where relevant services were beyond position 5 in the results list slightly improves both NDCG (between 6% and 8%) and recall (about 4%), besides the above discussed enhancements in precision results.

Table 9 Precision, Recall and NDCG in Scenario 1 – EasySOC

|	Original Results	*Cutoff-at-4* (26% of original queries expanded)		*Cutoff-at-5* (17% of original queries expanded)	
	Syntactic	QE0+QE1	QE0+QE2	QE0+QE1	QE0+QE2
Precision-at-1	0.27	0.34	0.36	0.31	0.33
Precision-at-2	0.63	0.75	0.77	0.72	0.74
Precision-at-3	0.74	0.85	0.88	0.83	0.85
Precision-at-4	0.82	0.89	0.89	0.91	0.93
Recall	0.91	0.91	0.91	0.95	0.95
NDCG	0.78	0.84	0.85	0.84	0.86

Scenario 2: Lucene service registry.

In Scenario 2, the Lucene registry was populated with the data-set and then queried with the 430 original queries. Then, the post-processing steps defined in Section 4.1 were performed settling the corresponding cutoff point and query replacement schemes for the *Interface Compatibility* execution. In this case, *cutoff-at-5* implies replacing 12% of the original queries. The chart in Table 10 depicts the cumulative average Precision-at-*n* values for the original and *Interface Compatibility* post-processed results.

In addition, ~~Additionally,~~ Table 10 summarizes the Precision-at-*n* (with n=[1-4]), recall and normalized DCG values in Scenario 2. As in the previous scenario, the *Interface Compatibility* analysis increased Precision, ranging between 5% and 8% when n=[1-4]. It is worth noticing that the cumulative precision-at-3 when applying *Interface Compatibility* exceeded 90% – i.e., for 9 out of 10 queries, the relevant service was among the three first results. Also, recall and NDCG slightly improved, about 4% and 6% respectively.

These results show that *Interface Compatibility* analysis outperformed original values for precision, recall and NDCG independently of the underlying service discovery registry that constructed the candidate services lists.

Table 10 Precision, Recall and NDCG in Scenario 2 – Lucene

	Original Results	Cutoff-at-5 (12% of original queries expanded)	
	Syntactic	QE0+QE1	QE0+QE2
Precision-at-1	0.39	0.44	0.46
Precision-at-2	0.75	0.82	0.82
Precision-at-3	0.83	0.91	0.91
Precision-at-4	0.88	0.96	0.96
Recall	0.92	0.96	0.96
NDCG	0.84	0.9	0.9
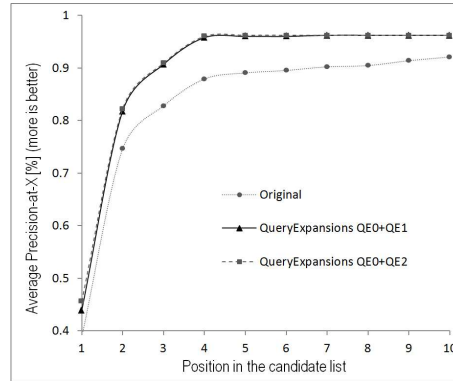
Concluding Remarks.

A final note about execution time of the experiment. Costs of the *Interface Compatibility* in terms of computation time were measured and averaged through several runs. Query expansion step took an average of 125 milliseconds, performing two expansions (QE0 plus QE1 or QE2) per time. The *Interface Compatibility* analysis (average) execution time was 6.65 minutes with a standard deviation of 1.6 minutes. This values depended on the number of replaced and expanded queries. For instance, one of the more intensive experiments was Scenario 1 with *cutoff-at-5* and query expansions QE0 + QE2. Total queries raised to 484 after replacing, expanding and eliminating duplicated queries, and the execution time was about 9.3 minutes. In this case, such time represents an average of 1.15 seconds per query.

Conclusively, the experiments show that executing the *Interface Compatibility* analysis over previously discovered services (published either in EasySOC or Lucene registries) improves the visibility of relevant services. Such an improvement is expressed in terms of slight gains in Precision, Recall and NDCG. It is important to notice that results can be specifics for the data-set and queries used, and cannot be merely generalized to other experimental configurations. However, considering that the selection of candidate services is performed after any discovery process, increasing the visibility of most suitable candidates in a list of previously discovered services may ease the development of service-oriented applications.

5 Conclusions and Future Work

In this paper we have presented details of service *Interface Compatibility* analysis, which allows evaluating a candidate Web Service for its likely integration into a service-oriented application under development. For this, a practical Assessment Scheme is provided where a synthesis of design and programming heuristics have been applied, both to improve possibilities to identify potential matchings, but also to help developers to gain knowledge on an application's context for a candidate service. Additionally, the *compatibility gap* provides an observable value as a meaningful support for candidates selection.

Several experiments have been performed to evaluate the performance of the *Interface Compatibility* analysis. Results were measured in terms of well-known Informa-

tion Retrieval metrics – i.e., Precision-at-n, Recall and Normalized Discount Cumulative Gain (NDCG). We believe that our assessment process will significantly improve ranking results obtained from many service discovery registries.

Our current work is focused on exploring Information Retrieval techniques to better analyzing concepts from interfaces. In particular, considering terms collected from parameters names, in which different heuristics could be applied under the support of some semantic basis – e.g., taxonomies, ontologies or dictionaries. These improvements are oriented to fine-tune the Assessment Scheme to gain accuracy for service selection, while increasing adaptability and integrability of SOC-based applications. Gained accuracy should be measurable in terms of precision, recall, NDCG and other IR metrics.

In addition, we are currently working in a *Behavioral Compatibility* step (Garriga et al., 2012). This analysis is based on assessing the operational behavior execution of services. This is achieved by applying a testing framework, in which a compliance test suite (TS) is generated, based on the required functionality (Flores and Polo, 2012). Then, the TS is executed against candidate services to confirm the outcome from the *Interface Compatibility* analysis. Besides, a wrapper (adapter) could be built to allow the client component to safely call the selected service. Thus, the *Behavioral Compatibility* step complements the *Interface Compatibility* analysis achieving a protocol-level assessment. This makes our proposal a comprehensive selection method.

Moreover, with the increasing number of Web Services with similar or identical functionality, the nonfunctional properties of a Web Service are becoming more and more important (Blanco et al., 2012). Thereby, Quality of Service will be considered for service selection.

Another concern is the composition of candidate services to fulfill functionality, which is particularly useful when a single candidate service cannot provide the whole required functionality. We will expand the current procedures and models mainly based on business process descriptions and service orchestration (Peltz, 2003, Weerawarana et al., 2005).

References

P. Adamczyk, P. H Smith, R. Johnson, and M. Hafiz. REST and Web services: In Theory and in Practice. In Erik Wilde and Cesare Pautasso, editors, *REST: from Research to Practice*, chapter 2, pages 35–57. Springer, 2011.

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.

A. Ait-Bachir. Measuring Similarity of Service Interfaces. In *ICSOC PhD Symposium*, Australia, 2008.

D. Aumueller, H. Do, S. Massmann, and E. Rahm. Schema and Ontology Matching with COMA++. In *ACM SIGMOD International Conference on Management of Data*, pages 906–908. ACM Press, 2005.

E. Blanco, Y. Cardinale, and M. Vidal. Experiences of sampling-based approach for estimating qos parameters in the web service composition problem. *International Journal of Web and Grid Services*, 8(1): 1–30, 2012.

A. Brogi. On the potential advantages of exploiting behavioural information for contract-based service discovery and composition. *The Journal of Logic and Algebraic Programming*, 80(1):3 – 12, 2011.

M. Crasso, A. Zunino, and M. Campo. Easy web service discovery: A query-by-example approach. *Science of Computer Programming*, 71(2):144–164, April 2008.

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

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.

V. De Antonellis and M. Melchiori. An Approach to Web Service Compatibility in Cooperative Processes. In *Applications and the Internet Workshop Symposium*, 2003.

20

B. Di Martino. Semantic web services discovery based on structural ontology matching. *International Journal of Web and Grid Services*, 5(1):46–65, 2009.

O. F. Diaz, R. Santaolaya, and I. Solis. Using case-based reasoning for improving precision and recall in web services selection. *International Journal of Web and Grid Services*, 2(3):306–330, 2006.

R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, CA, USA, 2000.

A. Flores and M. Polo. Testing-based Process for Component Substitutability. *Software Testing, Verification and Reliability*, 22(8):529–561, 2012.

M. Garriga, A. Flores, A. Cechich, and A Zunino. Behavior assessment based selection method for service oriented applications integrability. In *Proceedings of the 41st Argentine Symposium on Software Engineering*, ASSE '12, pages 339–353, La Plata, BA, Argentina, 2012. SADIO. URL http://www.41jaiio.org.ar/sites/default/files/374_ASSE_2012.pdf.

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.

M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby. A Search Engine for Finding Highly Relevant Applications. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 475–484. IEEE Computer Society Press, 2010.

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

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.

N. Kokash. A Comparison of Web Service Interface Similarity Measures. In *Starting AI Researchers Symposium, STAIRS*, Amsterdam, Netherlands, 2006. I O S Press.

L. Kung-Kiu and W. Zheng. Software Component Models. *IEEE Transactions on Software Engineering*, 33(10):709–724, October 2007.

H. R. M. Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-automated Adaptation of Service Interactions. In *16th International Conference on World Wide Web*, pages 993–1002. ACM Press, 2007.

H. R. M. Nezhad, B. Benatallah, and Xuyuan G. Protocol-Aware Matching of Web Service Interfaces for Adapter Development. In *WWW 2010*, Raleigh, North Carolina, USA, 2010.

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

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

J. O. Coscia, C. Mateos, M. Crasso, and A. Zunino. Anti–pattern free code–first web services for state–of–the–art java wsdl generation tools. *International Journal of Web and Grid Services*, 9(2):107–126, 2013. ISSN 1741-1106. In Press.

M. Ouederni. Measuring the Compatibility of Service Interaction Protocols. In ACM, editor, *ACM Symposium on Applied Computing, SAC*, pages 1560–1567, 2011.

M. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: A research roadmap. *International Journal of Cooperative Information Systems*, 17(02):223–255, 2008.

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

C. Pautasso, O. Zimmermann, and F. Leymann. Restful Web services vs. "Big" Web services: Making the Right Architectural Decision. In *17th International Conference on World Wide Web*, pages 805–814. ACM Press, 2008.

C. Peltz. Web Services Orchestration and Choreography. *IEEE Computer*, 36(10):46–52, 2003.

L. Richardson and S. Ruby. *RESTful Web services*. O'Reilly Media, 2008.

J. M. Rodriguez, M. Crasso, A. Zunino, and M. Campo. Improving web service descriptions for effective service discovery. *Science of Computer Programming*, 75(11):1001–1021, 2010.

J. M. Rodriguez, M. Crasso, C. Mateos, and A. Zunino. Best practices for describing, consuming, and discovering web services: a comprehensive toolset. *Software: Practice and Experience*, 43(6):613–639, 2013a.

J. M. Rodriguez, M. Crasso, C. Mateos, A. Zunino, and M. Campo. Bottom-up and top-down cobol system migration to web services: An experience report. *IEEE Internet Computing*, 17(2):44–51, 2013b.

D. Sprott and Wilkes. L. Understanding Service-Oriented Architecture. *The Architecture Journal. MSDN Library. Microsoft Corporation*, 1:13, January 2004. http://msdn.microsoft.com/en-us/library/aa480021.aspx.

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

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.

S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, 2005.

J. Wu and Z. Wu. Similarity-based Web service Matchmaking. In *IEEE International Conference on Services Computing, 2005*, volume 1, pages 287–294. IEEE Computer Society Press, 2005.