SADIO Electronic Journal of Informatics and Operations Research

http://www.dc.uba.ar/sadio/ejs

vol. 11, no. 1, pp. 16-30 (2012)

Practical Assessment Scheme to Service Selection for SOC-based Applications*

Martín Garriga[1,3] Andres Flores[1,3] Alejandra Cechich[1] Alejandro Zunino[2,3]

[1]GIISCo Research Group
Facultad de Informática
Universidad Nacional del Comahue
Neuquén, Argentina
[martin.garriga,andres.flores,acechich]@fai.uncoma.edu.ar
[2]ISISTAN Research Institute
UNICEN
Tandil, Argentina
azunino@isistan.unicen.edu.ar
[3]CONICET (National Scientific and Technical Research Council)
Argentina

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 assessment effort could be overwhelming. On a previous work we have presented a novel approach to assist developers on discovery, selection and integration of services. This paper presents the selection method, which is based on a comprehensive scheme for services' interfaces compatibility. The scheme allows developers to gain knowledge on likely services' interactions and their required adaptations to achieve a positive integration. The scheme is also complemented by a framework based on black-box testing to verify compatibility on the expected behavior of a candidate service. The usefulness of the selection method is highlighted through a series of case studies.

Keywords: Service oriented Computing, Component-based Software Engineering, Web Services

* This work is supported by projects: ANPCyT–PAE-PICT 2007-02312 and UNCo–IEUCSoft (04-E072)

1 Introduction

Service-Oriented Computing (SOC) is a paradigm that promotes the development of distributed applications in heterogeneous environments [Erickson and Siau, 2008]. Service-oriented applications are developed by reusing existing third-party components or services that are invoked through specialized protocols. 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 [Bichler and Lin, 2006]. 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 suitable services to then provide the adequate "glue-code" for assembly into the application under development. This implies a large effort into discovering services, analyzing the suitability of retrieved candidates and identifying the set of adjustments for the final assembly of a selected candidate service [Cavallaro and Di Nitto, 2008].

In order to ease the development of SOC-based applications we have presented in a previous work [Flores et al., 2010] an approach which helps at discovery, selection and integration of services. This proposal is based on two recent approaches, each one focused on different aspects of maintainability. The first approach, called *EasySOC* [Crasso et al., 2010], provides specific semi-automated methods for both discovery and integration of services, for which a comprehensive review of current methods and techniques was previously carried out – as can be seen in [Crasso et al., 2008; Crasso et al., 2010; Mateos et al., 2010]. The second approach, called *TestOOJ* [Flores and Polo, 2010] was initially developed to work with off-the-shelf (OTS) software components as a solution for substitutability of component-based systems. This approach supplies a method for selection of the most appropriate third-party candidate component. Since web services involve a special case of software component [Stuckenholz, 2005; Canfora and Di Penta, 2006; Kung-Kiu and Zheng, 2007], few initial adjustments were required to apply this selection method for SOC-based application development.

The whole approach is fully supported by two semiautomatic tools, named *EasySOCPlugin* and *TestOOJ* respectively, which have been conveniently integrated to validate the ideas proposed in this paper.

Particularly, the selection method provides two main assessment procedures: an Interface Compatibility analysis and a Behavioral Compatibility evaluation. The former is made at a syntactic level to identify different aspects concerning the interface of a candidate service, for which details are provided throughout this paper. The latter is based on a testing framework to complement the previous analysis, by verifying compatibility on the expected behavior of candidate services.

This paper is focused on the Interface Compatibility step, which has been conveniently extended to provide a comprehensive Assessment Scheme to evaluate interfaces from candidate services according to requirements of internal components from a SOC-based application. This scheme allows characterizing the matchmaking process through a series of syntactic compatibility cases conveying not only the usual programming standards (e.g. names on operations and parameters), but also differentiating strong and potential similarity cases. In our previous work, this step of Interface Compatibility analisys had a very reduced underlying model just to cover a few matching cases that were mostly the trivial ones, and making it unable to aid developers to outline a likely solution upon different missmatching cases [Flores et al., 2010].

In this work, the Assessment Scheme has been divided into two main parts: automatic-strong matching and semiautomatic-potential matching, where the former involves similarity cases directly recognized from candidate's interfaces, and the latter involves those cases initially analyzed as a mismatching that could be solved by a decision of a developer based on a semi-automatic assistance. The whole package of information achieved from this process provides developers an important insight on candidate services and the required adaptations for integration.

The second evaluation of the selection method addressing the behavior evaluation of candidate services considers current techniques from [Jaffar-Ur Rehman, 2007; Orso, 2006; Mariani et al., 2007]. This step has been particularly conceptualized based on the *observability* testing metric [Freedman, 1991; Jaffar-Ur Rehman, 2007] that identifies a component operational behavior by analyzing data transformations (input/output). This testing metric helps to understand the functional mapping performed by a component and

therefore its behavior. Hence, a potential compatibility of a candidate service could be exposed – as we analyzed on a previous work [Flores and Polo, 2010] and was also discussed in [Cechich and Piattini, 2007; Alexander and Blackburn, 1999]. In addition, [Canfora and Di Penta, 2006] presents a summary of fair techniques to test SOC-based systems, pointing out their close relation to component-based systems.

The paper is organized as follows. Section 2 presents an overview of the whole process for SOC-based application development. Section 3 gives details of the Assessment Scheme of the Selection Method. Section 4 presents a series of case studies. Section 5 presents related work. Conclusions and future work are presented afterwards.

2 A Process for SOC-based Application Development

During development of a service-oriented application, a developer may decide to implement specific parts of a system in the form of in-house components. Additionally, for some of the comprising components the decision could be the acquisition of third-party components, which in turn could be solved with the connection to web services. Figure 1 depicts our proposal intended to assist developers in the process of discovery, selection and integration of web services. Following are briefly described the steps for each of the three main phases of the process.

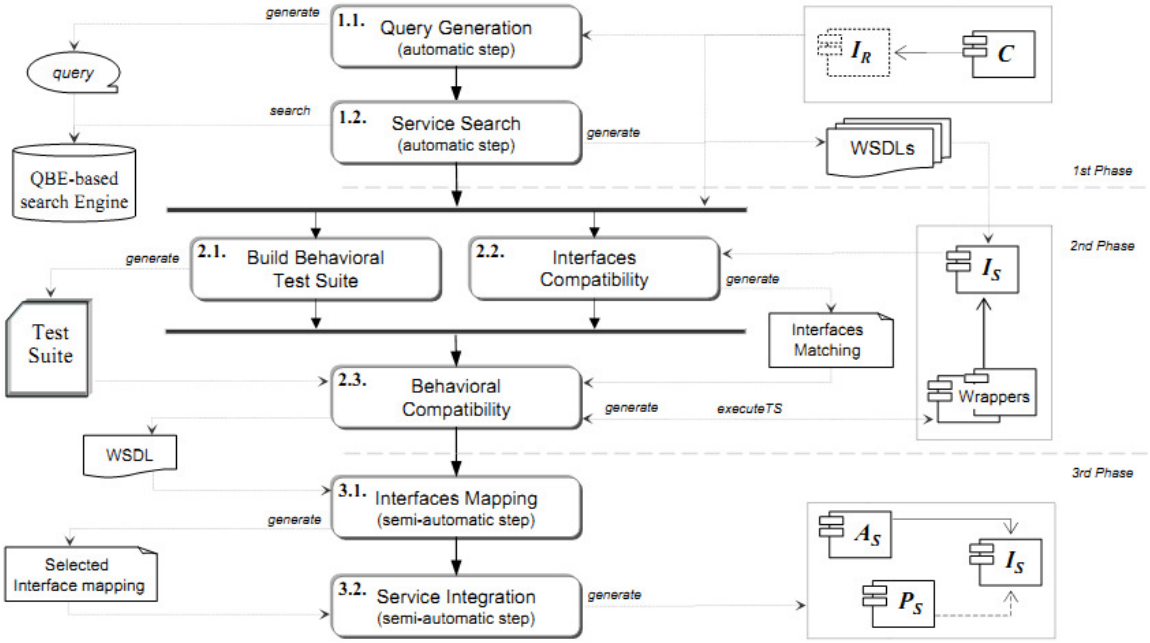


Figure 1. Process for SOC-based Application Development

1st Phase – Service Discovery

Being *C* a client component that requires certain services to be fulfilled. A specification for a required service could be described in the form of an interface I_R and a dependency from *C* to I_R. Additional annotations as JavaDoc comments could be attached to them [Flores et al., 2010].

1.1. Query Generation. Information gathered from *C* and I_R is processed by applying text mining techniques to form an initial *query* comprised of relevant terms. This query can also be properly refined and expanded by exploring other internal components from the client application and analyzing superclasses from the I_R's hierarchy [Crasso et al., 2010].

1.2. Service Search. The final query becomes the input for a search method called WSQBE that uses a Query-by-Example search engine [Crasso et al., 2008]. An initial step deduces the most related category

to the query (or example functionality), to then look for relevant services within its registry. The developer may also set a specific category in order to get a more focused and reduced search. The outcome is a wieldy list of candidate services.

2nd Phase – Service Selection

Although the list of candidate services is not too large, still a decision must be made about the most appropriate service S (with interface I_S) for the consumer's application. This phase is intended to help not only to identify an adequate candidate service, but also to confirm that its behavior match the requirements of the client application. Each service S is evaluated at a time, by previously deriving a Java version of the WSDL description of its interface I_S [Flores et al., 2010].

2.1 Build Behavioral Test Suite. A test suite TS is generated with the purpose to represent behavioral aspects from a third-party service, with required interface I_R. This TS complies with certain criteria that help describing different facets of interactions of component C with the required service (through I_R). Notice that the goal of this TS is not to find faults but to represent behavior [Flores and Polo, 2010].

2.2. Interface Compatibility. Both the required interface I_R and the provided interface I_S are syntactically compared. The evaluation is based on a comprehensive *Assessment Scheme* to recognize either automatic-strong or semiautomatic-potential matchings, from the set of operations of I_R and the operations offered by I_S. The *Assessment Scheme* provides a chance to not discard potential candidate services in which operations do not completely coincide on their names, order of parameters, etc. The outcome of this step is an *Interfaces Matching List* where each operation from I_R may have a correspondence with one or more operations from I_S. Since this step is the main focus of this paper, details are given in Section 3.

2.3. Behavior Compatibility. Service S, which has passed the previous step, must be evaluated on its behavior. This implies to execute the TS generated from I_R, against S (through IS). The purpose is to find the true operation correspondences from the *Interfaces Matching List* generated in the previous step, from which a set of wrappers (W) for S (through I_S) is generated. Another goal is to find a wrapper $w \in W$ to be placed between I_R and I_S to allow the client component C to safely call service S. For this, each $w \in W$ is taken at a time as the target class under test by running the TS from I_R. After the whole set W has been tested, the percentage of successful tests should be higher than 70% to have a final conclusive result on compatibility. This also implies that at least one wrapper can be taken as the most suitable to allow the integration of service S to the client component C [Flores and Polo, 2010].

3rd Phase – Service Integration

After a candidate service S has passed the evaluations from the Selection Phase, the most adequate wrapper $w \in W$ can be used to proceed with the integration of service S to the client component C [Flores et al., 2010].

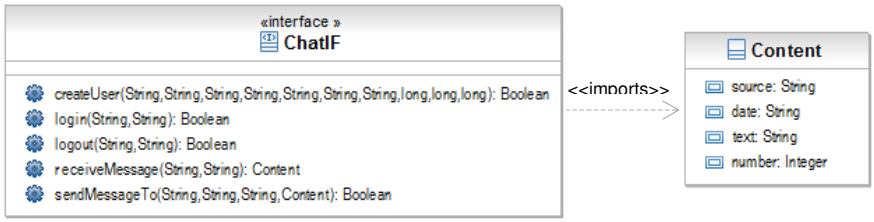
3.1. Mapping of Selected Interface. From the most adequate wrapper $w \in W$ and making use of the *Interfaces Matching List* is generated a specific *Interface Mapping* comprised of concrete correspondences between the required interface I_R and the interface I_S (of the selected web service S). The *Interface Mapping* adopts the form of an XML file.

3.2. Integration of Selected Service. From the *Interface Mapping* defined into the XML file in the previous step, the *Adapter* design pattern is applied to generate an adapter A_S, where each operation from the required interface I_R will invoke a specific operation from the selected interface I_S. In addition, the physical connection to S for allowing invoking operations exhibited in I_S, is managed through the *Dependency Injection* design pattern [Johnson, 2005]. Thus, a proxy for S (P_S) is generated, from where C will end up calling the operations declared in I_S through P_S, which transparently invokes the remote service S. Interestingly, this mechanism is not intrusive, since the code of C remains untouched still on dependency with I_R, from where the adapter A_S and the proxy P_S have been generated.

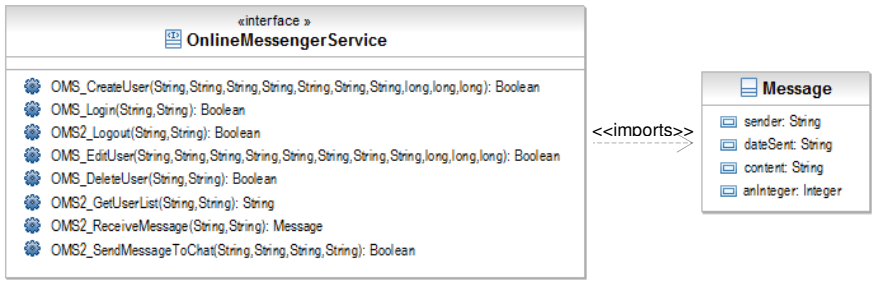
Next sections provide detailed information particularly related to the Interface Compatibility step. A case study will be used to illustrate the usefulness of the Assessment Scheme into the Selection Method.

2.1 Case Study

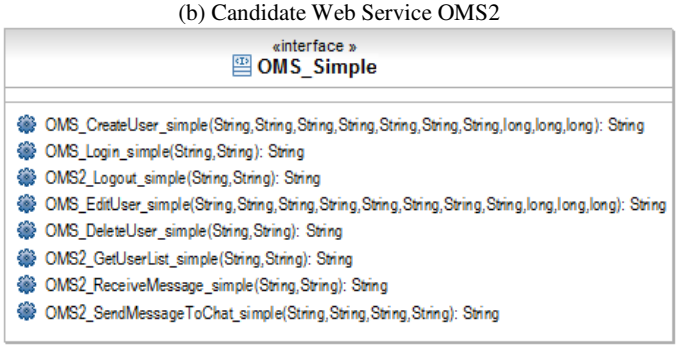
Let us suppose the development of a communication tool for exchanging instant messages with contacts from a user's contact list. We have specified the behavior of the required service in the form of operations defined into a Java interface I_R, named `ChatIF`. Figure 2(a) shows the required interface `ChatIF`, which includes a complex type named `Content`. By running the 1st Phase of the process, a web service called `OMS` (Online Messenger Service) has been discovered at *http://www.nims.nl/*. Particularly we are interested in two of those services: `OMS2`[2] and `OMS2Simple`[3]. The former provides an interface I_{S1} comprised of 38 operations. The most relevant ones can be seen in Figure 2(b), where another complex type named `Message` is used for enclosing the contents to be exchanged. The latter, whose interface I_{S2} is shown in Figure 2(c), uses the `String` type for the operations' return, instead of any other built-in or complex type.



(a) Required Interface



(b) Candidate Web Service OMS2



(c) Candidate Web Service OMS2_Simple

Figure 2. Instant Messenger Application – Chat

[2] *http://www.nims.nl/soap/oms2.wsdl*
[3] *http://www.nims.nl/soap/oms2simple.wsdl*

3 Interface Compatibility Analysis

The Selection Method, corresponding to the 2nd phase of the whole process for SOC-based application development, entails handling certain context information from the application's business domain. Such information is vital to understand the functionality that will be fulfilled by a third-party service. We assume the availability of the documentation artifacts describing the expected software architecture, including the Requirement Specification document (as knowledge source).

As explained in Section 2, the Selection Method concerns two main evaluations on candidate services, from which a concrete recommendation concerning the most appropriate service is achieved. The final evaluation procedure (*Step 2.3*) takes the set of candidate services to be put under test with the purpose to discover a compatibility with respect to the expected behavior for the client application. Nevertheless, such final evaluation requires a previous assessment at a syntactic level on Interface Compatibility (*Step 2.2*), which may provide useful preliminary information to help developers gain knowledge on several aspects. The outcomes may help elude to early discard a candidate service upon simple mismatches but also preventing from a serious incompatibility. In addition, helpful information about the adaptation effort of a candidate service may take shape for a positive integration into the consumer application.

Particularly, the Interface Compatibility analysis is comprised of a practical Assessmen Scheme that has been conveniently extended to cover a comprehensive range of matching cases, from which a developer may easily understand causes of a certain compatibility result. Besides, the scheme has also been divided into two parts: automatic matching cases and semi-automatic potential matchings. Both parts characterize syntactic similarity cases into four levels of compatibility, to help analyzing operations from the interface I_S (of a candidate service S), with respect to the required interface I_R.

Following is presented the first part of the scheme which recognizes automatic matching cases. Section 3.2 presents the second part of the scheme, intended to be applied for solving mismatching cases.

3.1 Assessment Scheme: Automatic Matchings

The Assessment Scheme is focused on characterizing operations equivalence from a required interface I_R when is compared to an interface I_S (of a candidate service S). Table 1 presents the first part of the Assessment Scheme, which is divided into four levels to describe different syntactic constraints for a pair of corresponding operations. Such syntactic constraints are based on individual conditions for each element comprising the operations' signature of an interface (*return*, *name*, *parameter*, *exception*). Table 2 summarizes the set of operation matching conditions, according to the elements of an operation's signature.

Table 1. Assesment Scheme: Automatic Matchings

Level	Constraints
■ Exact Match (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 (13 cases)	Three or two identical conditions. The remaining might be second conditions: (R2/N2/P2/E2). Exceptional cases: three identical conditions with a remaining third condition (N3/P3/E3). This implies equivalence values between 5 and 6.
	Example: operation `logout` of `ChatIF` has *near-exact_2* match to `OMS2_Logout` of `OMS2` with a substring equivalence for the operation name ("logout"): [R1,N2,P1,E1].
■ Soft Match (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.
■ Near Soft Match (14 cases)	There cannot be two identical conditions, i.e. all conditions can be relaxed simultaneously. This implies equivalence values between 9 and 11.

Those conditions concerning data type equivalence involve 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]. It is expected that types on operations from I_S have at least as much precision as types on I_R. However,

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. A criterion of "no inclusion" has been defined about conditions `R3` and `P4` that are evaluated in this first part of the scheme as incompatibilities (treated as conditions `R0` and `P0` respectively). For example, operation `sendMessageTo` of `ChatIF` could have a correspondence with operation `OMS2_SendMessageToChat` because there is identical return and exceptions with an equivalent operation name (`R1`,`N2`,`E1`). However, in `sendMessageTo` there is a parameter of complex type (`Content`) without a counterpart into the operation `OMS2_SendMessageToChat` – i.e. `P4` that is initially evaluated as `P0`. In Section 3.2 is shown how this incompatibility can be solved.

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, there must be a correspondence for each field of a complex type from an operation $op_R \in I_R$ – though extra fields from interface I_S may be initially left out of any correspondence. For example, the operation `receiveNextMessage` of `ChatIF` has a complex type as a return (`Content`), and operation `OMS_ReceiveMessage` of `OMS2` also has a complex type as a return (`Message`). Both complex types are equivalent because their fields are equivalent one-to-one. Therefore, operation `receiveNextMessage` has equivalence *near_exact_12* with `OMS_ReceiveMessage`, since they coincide on number, type and order for parameters and exceptions (`P1`,`E1`) and there is a substring equivalence for their names (`N2`) – common words "*receive*" and "*message*". Finally, from the previous comments there is an equivalent complex type as a return (`R2`).

Table 2. Syntactic Matching Conditions for Interface Compatibility

Signature Element	Condition	Description
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 number, type and order for parameters into the list
	P2	Equal number and type for parameters into the list
	P3	Equal number and type at least equivalent (including 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 number and type, and also order for exceptions into the list
	E2	Equal number and type for exceptions into the list.
	E3	If non-empty original's exception list, then non-empty candidate's list (no matter the type).

The first part of the Assessment Scheme in Table 1 is finally comprised of 54 cases, from the combination of individual conditions (classified into the four levels of compatibility). In the following section is addressed the possibility to solve certain cases of mismatch by means of a semi-automatic assistance based on the second part of the Assessment Scheme for Interface Compatibility.

3.2 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 and particularly the Requirement Specification document (as source of knowledge). We have identified specific cases in which a concrete compatibility can be set up 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. Table 3 presents the second part of the Assessment Scheme, in which only new cases are described for all but the first level of compatibility (*exact-match*). This time, the lowest individual conditions for return and parameters (`R3`,`P4`) are considered likely possibilities to solve mismatch cases.

The second part of the Assessment Scheme is comprised of additional 54 cases therefore making the whole scheme able to recognize 108 cases for Interface Compatibility. In addition, this second part not only is intended to assist on solving mismatch cases, but also to allow a developer to "force" 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. Then, the developer is enabled to make such prioritization for a particular matching, which then is considered in first order for the processing on the Selection Method's subsequent step (see Section 2).

Table 3. Assessment Scheme: Solving Mismatches

Level	Constraints
■ Near Exact Match (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 (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 implies equivalence values between 7 and 8.
	Example: operation `sendMessageTo` of `ChatIF` could match operation `OMS2_SendMessageToChat`. However, the first operation includes a parame-ter of complex type (`Content`) without a match into the other operation that has only String parameters (initially evaluated as `P0`). This can be re-evaluated considering that the wildcard type String might contain a chain of all fields from the complex type – i.e. an equivalence *soft_25*: [`R1`,`N2`,`P4`,`E1`].
■ Near Soft Match (40 cases)	Either two identical conditions with the condition P4 or relaxing all conditions simultaneously. This implies equivalence values between 9 and 13.

3.3 Assessment Scheme: Syntactic Distance

The final outcome of the Interface Compatibility step is a matching list characterizing each correspondence according to the four levels of the Assessment Scheme, named *Interface Matching List*. 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}, $1 \leq j \leq 5$. The 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 in Table 2 and 3. 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, which is the result of adding the value 1 of each condition. Therefore, from the Interface Matching List, a totalized equivalence value could be calculated, to synthesize the achieved degree of Interface Compatibility between a required interface I_R and a candidate interface I_S (from a service *S*). Only the higher compatibility level for each operation is considered to calculate that value, named *Syntactic Distance*. The corresponding formula is shown in (1).

$$syntDist(I_R,I_S) = \frac{\sum_{i=1}^{N} Min(op_{Ri}, MapComp(I_R,I_S)) - 1}{N * 4} \quad (1)$$

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

The *Syntactic Distance* concerns a helpful metric for comparison when a set of candidate services is under evaluation, providing for a developer a better understanding of the achieving results. In the case that all operations in the *Interface Matching List* presents an *exact* equivalence, the *Syntactic Distance* between I_R and I_S is zero. Although this might give the feeling of a perfect interface match, this iniatilly only means that I_R is included into I_S, while the interface I_S may be larger on size – i.e., including additional operations.

The success on the precision achieved during the Interface Compatibility step is essential to reduce the computation effort for the subsequent step of behavior evaluation (see Section 2). 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 experience perspective.

4 Case Studies

This section shows in detail the evaluation results for the example presented in Section 2.1. Then two other case studies are briefly described.

4.1 Instant Messenger – Chat

In the automatic matching results for `ChatIF` and service `OMS2`, a mismatch is identified for operation `sendMessageTo` of `ChatIF` for which a semi-automatic solution could be set up by a *soft_25* (`R1,N2,P4,E1`) match to operation `OMS2_SendMessageToChat` of `OMS2`. The rest of the `ChatIF` interface has found a match – as shown in Table 4. For example, operation `createUser` has a *near-exact_2* match to operation `OMS_CreateUser` (due to the substring equivalence). Operations `login` and `logout` obtained similar result by a *near-exact_2* match to alike operations, and four *near-exact_7* matches to other operations. Finally, operation `receiveNextMessage` obtained a *near-exact_12* match to operation `OMS_ReceiveMessage` of `OMS2` service.

As no automatic matching has been found for `ChatIF` and `OMS2Simple`, Table 5 shows the final matching results for `ChatIF` and service `OMS2Simple`, where mismatches have been solved in the semi-automatic step by the notion of the String type as a *wildcard* type (see Section 3.1).

Table 4. Final Interface Compatibility between `ChatIF` and `OMS2`

ChatIF	OMS2		
boolean sendMessageTo (String,String, String,Content)	soft 25, boolean OMS2 SendMessageToChat (String, String, String, String), R1,N2,P4,E1		
boolean createUser (String, String, String, String, String, String, String, long, long, long)	n_exact_2, boolean OMS_CreateUser (String, String, String, String, String, String, String, long, long, long), R1, N2, P1, E1		
Content receiveNextMessage (String, String)	n_exact_12,Message OMS_ReceiveMessage (String, String,), R2, N2, P1, E1		
boolean logout(String, String)	n_exact_2,boolean OMS2_Logout(String,String), R1, N2, P1, E1]	n exact 7, boolean OMS_Login (String, String), R1, N3, P1, E1	n_exact 7, boolean OMS DeleteUser (String, String), R1, N3, P1, E1
boolean login(String, String)	n_exact_2, boolean OMS_Login(String, String), R1, N2, P1, E1	n exact 7, boolean OMS2 Logout (String, String), R1, N3, P1, E1	n exact 7,boolean OMS DeleteUser (String, String), R1, N3, P1, E1

At this point, the *Interface Matching List* for both candidate services is available. Thus, the *syntactic distance* could be used to determine which of them is better to continue with the Behavioral Compatibility (*step 2.3*). Table 6 summarizes the best values found for each candidate service and each operation in `ChatIF`.

The *syntactic distance* between `ChatIF` and `OMS2` is 29/20-1 = 0,45 according to formula (1), and considering `OMS2_Simple` the *syntactic distance* is 40/20-1 = 1. Because the lower value is the better, the suggested candidate service is `OMS2`.

Table 5. Final Interface Compatibility between `ChatIF` and `OMS2Simple`

ChatIF	OMS2
boolean createUser(String, String,String,String,String, String, String,long,long,long)	[soft_16, String OMS_CreateUser_simple (String, String,String,String,String,String,String,long,long,long), R3, N2, P1, E1]
boolean sendMessageTo (String,String, String,Content)	[n_soft_39, String OMS2_SendMessageToChat_simple (String, String, String, String), R3, N2, P4, E1]
Content receiveNextMessage (String, String)	[soft_16, String OMS_ReceiveMessage_simple(String, String), R3, N2, P1, E1]
boolean logout(String, String)	[soft_36, String OMS2_Logout_simple (String, String), R3, N3, P1, E1]
boolean login(String, String)	[soft_36, String OMS_Login_simple(String, String), R3, N3, P1, E1]

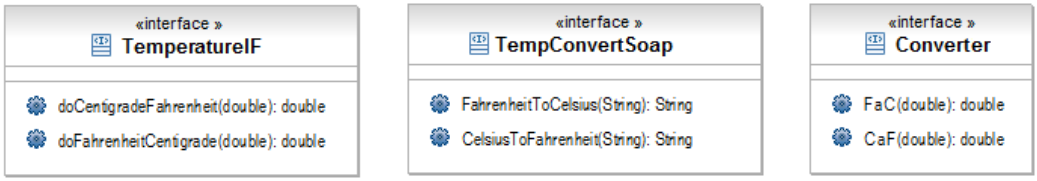
Table 6. Interface Compatibility summary for `ChatIF`, `OMS2`, and `OMS2Simple`

ChatIF Operations	OMS2 Best Value*	OMS2_Simple Best Value*
createUser	5	6
sendMessageTo	8	11
receiveNextMessage	6	7
Logout	5	8
Login	5	8
Total	29	40
Syntactic Distance	0,45	1

* Total Best Value 20 (based on `ChatIF` size)

4.2 Weather System

This case study is a system in which it is required to provide temperature information on both Celsius and Fahrenheit scales. A required interface I_R has been defined in the Java format, named `TemperatureIF`, which is shown in Figure 4(a). Candidate web services are named `TempConvert`[4] and `Converter`[5], whose interfaces I_{S1} and I_{S2} are shown in Figure 4(b) and 4(c) respectively.



(a) Required Interface (b) Candidate Service (c) Candidate Service

Figure 4. Weather System

[4] *http://www.w3schools.com/webservices/tempconvert.asmx?WSDL*
[5] *http://www.elguille.info/Net/WebServices/CelsiusFahrenheit.asmx?WSDL*

When running the automatic Interface Matching between `TemperatureIF` and service `TempConvert`, the results reveal that all operations from `TemperatureIF` have found a match – as shown in Table 7. In this case, both operations from `TemperatureIF` obtained similar result by two matches to both operations of `TempConvert` service. The `String` type which is recognized as a *wildcard* type allows to have an equivalence on types for return and parameters (`R2`,`P3`).

Table 7. Interface Compatibility for `TemperatureIF–TempConvert`

TemperatureIF	TempConvertSoap	
double doFarenheitCentigrado (double)	[n_soft_12, String fahrenheitToCelsius (String), R2, N3, P3, E1]	[n_soft_12, String celsiusToFahrenheit (String), R2, N3, P3, E1]
double dofahrenheitCentigrado (float)	[n_soft_12, String fahrenheitToCelsius (String), R2, N3, P3, E1]	[n_soft_12, String celsiusToFahrenheit (String), R2, N3, P3, E1]

Table 8 shows the results of automatic Interface Matching for `TemperatureIF` and `Converter`. Again, the results are not conclusive because two matches have been found for each operation. Table 9 shows the *syntactic distance* calculated between `TemperatureIF` and both candidate services, where is 1 for `TempConvert` and 0,5 for `Converter`. Thus, the suggested candidate for the next step of Behavioral Compatibility is the `Converter` service.

Table 8. Interface Compatibility for `TemperatureIF–Converter`

TemperatureIF	Converter	
double doFarenheitCentigrado (double)	[n_exact_7, double faC (double), R1, N3, P1, E1]	[n_exact_7, double caF (double), R1, N3, P1, E1]
double doCentigradoFarenheit (double)	[n_exact_7, double faC (double), R1, N3, P1, E1]	[n_exact_7, double caF (double), R1, N3, P1, E1]

Table 9. Interface Compatibility summary for `TemperatureIF` and the candidates

Operations of TemperatureIF	TempConvert Best Value*	Converter Best Value*
doFarenheitCentigrado	8	6
doCentigradoFarenheit	8	6
Total	16	12
Syntactic Distance	1	0,5

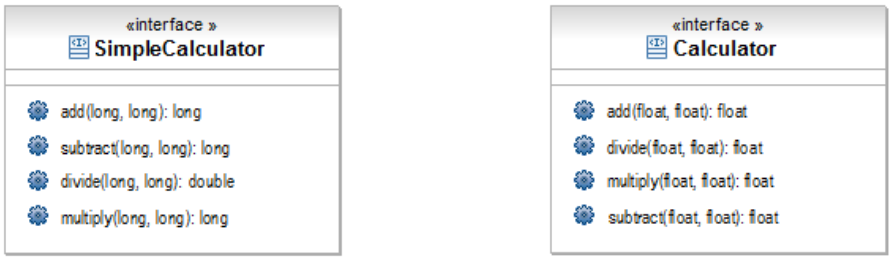
*Total Best Value 8 (based on `TempConvert` size)

4.3 Math Application

This case study is a Math Application where a calculator component is needed, for which a Java required interface I_R, named `SimpleCalculator`, has been defined, which is shown in Figure 5(a). The candidate web service is named `Calculator`[6] and its interface I_S is shown in Figure 5(b).

When running the Interface Matching between the I_R interface and the candidate service's interface I_S, the results reveal that all operations from `SimpleCalculator` have found a match, with an exception on operation `divide` – as can be seen in Figure 6. In this case the incompatibility factor implies the return type in which there is a loss of precision – from `double` to `float` (`R3`), with respect to the homologue operation of `Calculator` service. However, this case can be easily solved with the support of the semi-automatic procedure, where a *soft_39* match was finally set up.

[6] *http://soatest.parasoft.com/axis/calculator.wsdl*



(a) Required Interface (b) Candidate Service

Figure 5. Math Application

These case studies show how a developer may gain specific and valuable knowledge about an application's context by the support of the Assessment Scheme. For each likely equivalence case automatically identified, there is a clear rationale that is also reinforced by the characterization within the four levels of compatibility. In addition, different scenarios of compatibility upon low levels may be analyzed by setting up other correspondences with the semi-automatic assistance based on the second part of the scheme. In this way, a certain web service may be saved from being early discarded as a potential candidate, but also a concrete validation is given for any change on correspondences, which become very helpful for a developer to understand the required adaptation effort to achieve the service integration.

SimpleCalculator1	katze.axis.services.calculator.Calculator1		
public long add(long x1, long x2)	[29, soft_14, public float add(float x1, float x2), R2, N1, P3, E1]	[66, n_soft_12, public float subtract(float x1, float x2), R2, N3, P3, E1]	[66, n_soft_12, public float multiply(float x1, float x2), R2, N3, P3, E1]
public double divide(long x1, long x2)			
public long subtract(long x1, long x2)	[29, soft_14, public float subtract(float x1, float x2), R2, N1, P3, E1]	[66, n_soft_12, public float add(float x1, float x2), R2, N3, P3, E1]	[66, n_soft_12, public float multiply(float x1, float x2), R2, N3, P3, E1]
public long multiply(long x1, long x2)	[29, soft_14, public float multiply(float x1, float x2), R2, N1, P3, E1]	[66, n_soft_12, public float add(float x1, float x2), R2, N3, P3, E1]	[66, n_soft_12, public float subtract(float x1, float x2), R2, N3, P3, E1]

Figure 6. Automatic Interface Compatibility for `SimpleCalculator–Calculator`

5 Related Work

The work in [Kokash, 2006] provides a comparative analysis of existing approaches to improve Web Services discovery. This work is closely related to Service Selection, since an improved discovery method performs a partial preliminary selection among the candidates. In particular *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. Hence, the work in [Kokash, 2006] provides an approach that combines different techniques of lexical and structural information matching. Another strategy more close to a semantic basis implies the use of formal ontology-based methods, which yet involve a high cost making service designers be alienated from their use in practice [Kokash, 2006]. This tradeoff must be thoughtfully considered.

In particular, one work analyzed in [Kokash, 2006] is strongly related to our approach. To support programmatic service discovery, in [Stroulia and Wang, 2005] 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 order the most relevant WSDL specifications based on the similarity of the comprising element descriptions. If a (potentially partial) specification of the desired service behavior is also available (as in our case), this set of likely candidates can be further refined by a semantic structure-matching step, assessing the structural similarity of the desired and retrieved services and the semantic similarity of their identifiers.

Techniques presented on the aforementioned work not only can be applied to improve the service discovery step, but also could be adapted to the service selection process. There is a common rationale between the structural assessment mechanisms proposed by [Stroulia and Wang, 2005] and our practical assessment scheme, since both of them address the comparison of data types, operations and identifiers.

Another work defines the notion of compatibility degree based on formal comparisons [Ouederni, 2011]. In this proposal, a generic flooding-based technique is applied for measuring the compatibility degree of service protocols. The work is focused on the interaction protocol level of service interfaces, for which it is proposed a generic framework where the compatibility degree of service interfaces can be automatically measured according to different compatibility notions. They consider a formal model for describing service interfaces with interaction protocols. Also, the computation of a global and unique compatibility degree from the detailed measures helps in ranking and selecting some services from many possible candidates. As a difference, our approach is based on a more practical perspective attending the industry requirements in which formalisms are not widely accepted since involves higher effort and computational costs.

The work in [Ait-Bachir, 2008] presents a similarity measure between behavioral interfaces of Web Services. The behavioral aspect refers to the control flow between the operations and establishes their interdependencies. In conversational services, such behavioral interfaces can be described using *Business Process Executing Language* (BPEL), for instance. Nevertheless, *Finite State Machines* (FSM) is the formal model adapted in this work to describe behavioral interfaces. In contrast, our approach makes use of compliance assessment by means of behavioral execution of services. This is achieved by applying a testing framework.

The idea of an adapter to address a mismatch between two services is also discussed in [Benatallah and Motahari Nezhad, 2008]. Authors of this work distinguish between two types of mismatches: interface-level and protocol-level. Mismatches at the interface-level characterize heterogeneities related to operation definition in WSDL interfaces. Examples at this level include operation signature mismatch and parameter constraint mismatch. When certain interface mismatch appears, an adapter is settled to solve the incompatibility. In our approach, the assessment scheme for interface compatibility is the basis to generate wrappers or adapters. Particularly, the second part of the scheme (see Section 3.2) which is intended to solve interface mismatches could also be used to automatically provide potential matching cases. This allows developers to realize about non-initially considered compatibilities.

Finally, in [De Antonellis and Melchiori, 2003] a comparison of services' structure which is 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, in our work the interface compatibility includes different elements (return, name, parameters and exceptions) and it compares a required interface against the interface of a candidate service. Although our work is syntactically oriented, it includes a structural aspect which is usually neglected, related to failed function executions represented by exceptions, as mentioned in [Rodriguez et al., 2010]. This aspect is even important on service protocols affecting expected execution sequences. Then, it is also considered during the behavior assessment into our approach.

6 Conclusions and Future Work

In this paper we have presented details of a Selection Method which allows evaluating a candidate web service for its likely integration into a SOC-based application under development. This method is part of a larger process for discovery and integration of services, and provides a practical Assessment Scheme for

Interface Compatibility where a synthesis of design and programming heuristics have been added, both to improve possibilities to identify potential matchings, but also to help developers to gain knowledge on the application's context for a candidate service. The syntactic distance metric provides a measurable value to mathematically support the candidate selection. Additionally, such selection might consider other aspects like *Quality of Service* parameters – e.g., performance, security, and so on.

The whole process of discovery, selection and integration has a fully support to achieve efficiency and reliability. Our current work is focused on exploring Information Retrieval techniques to better analyzing concepts from interfaces, which has been initially applied on the EasySOC approach. Another concern implies 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, 2005].

References

Ait-Bachir, A. (2008). Measuring Similarity of service interfaces. In *ICSOC PhD Symposium 2008* Australia.

Alexander, R. & Blackburn, M. (1999). *Component Assessment Using Specification-Based Analysis and Testing* (Rep. No. SPC-98095-CMC). Herndon, Virginia, USA.

Benatallah, B. & Motahari Nezhad, H. R. (2008). Developing Adapters for Web Services Integration. In *WWW 2010* Raleigh, North Carolina, USA.

Bichler, M. & Lin, K. (2006). Service-oriented computing. *Computer, 39 (3),* 99-101.

Canfora, G. & Di Penta, M. (2006). Testing Services and Service-Centric Systems: Challenges and Oppotunities. *IT Professional, 8(2),* 10-17.

Cavallaro, L. & Di Nitto, E. (2008). An Approach to Adapt Service Requests to Actual Service Interfaces. In *ACM International Workshop SEAMS'08*.

Cechich, A. & Piattini, M. (2007). Early Detection of COTS Component Functional Suitability. *Information and Software Technology, 49(2),* 108-121.

Crasso, M., Mateos, C., Zunino, A., & Campo, M. (2010). EasySOC: Making Web Service Outsourcing Easier. *Information Sciences*.

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

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

Erickson, J. & Siau, K. (2008). Web service, service-oriented computing, and service-oriented architecture: Separating hype from reality. *Journal of BD Management, 19(3),* 42-54.

Flores, A., Cechich, A., Zunino, A., & Polo, M. (2010). Testing-Based Selection Method for Integrability on Service-Oriented Applications. In *5th IEEE ICSEA'10* (pp. 373-379).

Flores, A. & Polo, M. (2010). Testing-based Process for Component Substitutability. *Software Testing, Verification and Reliability, [early view press],* 33.

Freedman, R. S. (1991). Testability of Software Components. *IEEE Transactions on Software Engineering, 17(6),* 553-564.

Gosling, J., Joy, B., Steele, G., & Bracha, G. (2005). *Java(TM) Language Specification.* (3rd ed.) US: Addison-Wesley.

Jaffar-Ur Rehman, M. (2007). Testing Software Components for Integration: a Survey of Issues and Techniques. *Software Testing, Verification and Reliability, 17(2),* 95-133.

Johnson, R. (2005). J2EE Development Frameworks. *Computer Sciences, 38(1),* 107-110.

Kokash, N. (2006). A Comparison of Web Service Intervace Similarity Measures. In *Starting AI Researchers' Symposium STAIRS 2006* Amsterdam, Netherlands: IOS Press.

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

Mariani, L., Papagiannakis, S., & Pezzé (2007). Compatibility and Regression Testing of COTS component-based software. In *IEEE ICSE* (pp. 85-95). Minneapolis, USA.

Mateos, C., Crasso, M., Zunino, A., & Campo, M. (2010). Separation of Concerns in Service-Oriented Applications Based on Pervasive Design Patterns. In *25th ACM SAC'10*.

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

Orso, A. (2006). Using Component Metadata to Regression Test Component-based Software. *Software Testing, Verification and Reliability, 17,* 61-94.

Ouederni, M. (2011). Measuring the compatibility of service interaction protocols. In *ACM Symposium on Applied Computing, SAC 2011* (pp. 1560-1567).

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

Rodriguez, J. M., Crasso, M., Zunino, A., & Campo, M. (2010). An analysis of frequent ways of making undiscoverable Web Service descriptions. *EJS, 9(1),* 5-23.

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

Stuckenholz, A. (2005). Component Evolution and Versioning State of the Art. *ACM SIGSOFT Software Engineering Notes, 30(1),* 7-20.

Weerawarana, S. (2005). *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More.* (1 ed.) Prentice Hall.