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

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

[1] GIISCo Research Group, Facultad de Informática, Universidad Nacional del Comahue, Neuquén, Argentina. [aflores,acechich]@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

1 Introduction

Service-Oriented Computing (SOC) is a paradigm that promotes the development of distributed applications in heterogeneous environments [8]. 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 [2]. However, a broadly use of the SOC paradigm requires efficient approaches to allow service consumption from within applications [18]. 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 [4].

In order to ease the development of SOC-based applications we have presented in a previous work [9] 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* [6], 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 [7, 6, 17]. The second approach, called *TestOOJ* [10] 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 [20, 3, 15], few initial adjustments were required to apply this selection method for SOC-based application development.

In addition, the selection method has been extended to provide a comprehensive scheme for assessing interfaces from candidate services according to requirements of internal components from a SOC-based application. This scheme allows to characterize 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 mainly differentiating strong and potential similarity cases. Thus the scheme is divided into two main sections: automatic-strong matching and semiautomatic-potential matching, where the former involves similarity cases directly recognized from candidate's interfaces, while the later involves those cases initially analyzed as a mismatching that could be solved by a decision of a developer based on a semiautomatic 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 assessment scheme is also complemented by a framework based on black-box testing with the purpose to verify compatibility on the expected behavior of a candidate service. After a review of the literature, this step consider current techniques from [13, 19, 16] and has been particularly conceptualized based on the *observability* testing metric [11, 13] 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 [10] and was also discussed in [1, 5]. In addition, [3] presents a summary of fair techniques to test SOC-based systems, pointing out their close relation to component-based systems.

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

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. Conclusions and future work are presented afterwards.

2 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.
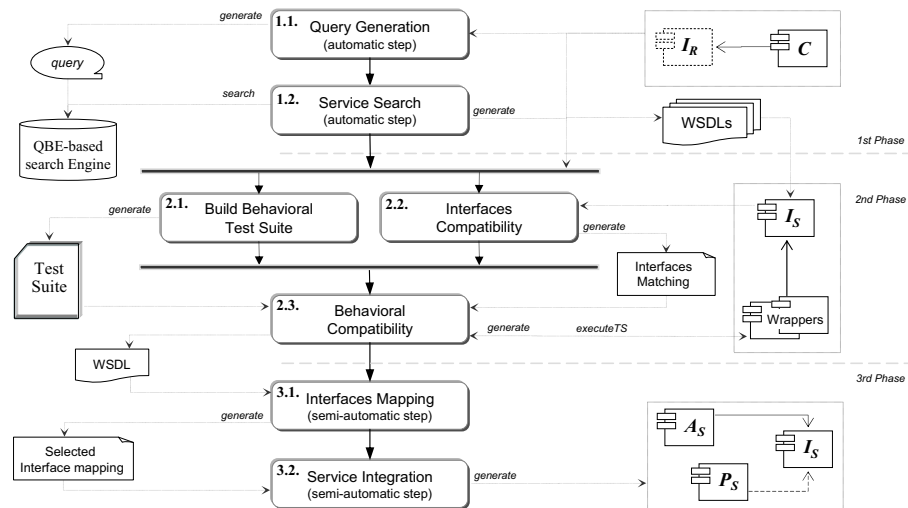

Fig. 1. Process for SOC-based Application Development

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 [9].

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 [6].

1.2. Service Search. The final *query* becomes the input for a search method called WSQBE that uses a Query-by-Example search engine [7]. 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 wieldly 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 certify 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 [9].

2.1. Build Behavioral TS. 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 [10].

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 the 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 I_S). 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 [10].

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 [9].

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.
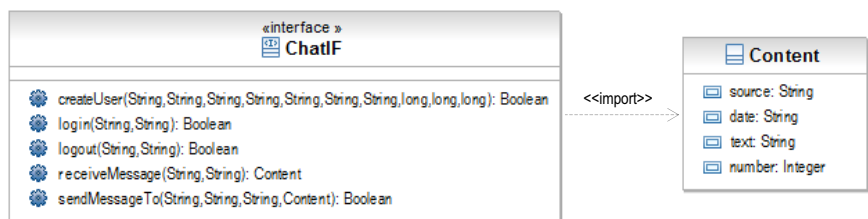
3.2. Integration of Selected Service. From the *Interface Mapping* defined into the XML file in the previous step, is applied the *Adapter* design pattern to generate an adapter A_S, where each operation from the required interface I_R will invoke a specific operation from the selected interface I_S. In addition, the physical connection to S for allowing invoking operations exhibited in I_S, is managed through the *Dependency Injection* design pattern [14]. 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.
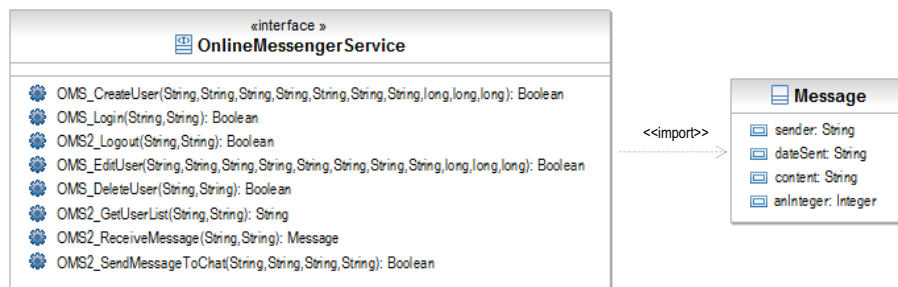
Next sections provide detailed information particularly related to the Interface Compatibility step. A case study will be used to illustrate the usefulness of the Assessment Scheme into the Selection Method.

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 on service OMS2 (*http://www.nims.nl/ soap/oms2.wsdl*) whose interface I_S is comprised of 38 operations, and 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.

«interface »
ChatIF

createUser(String,String,String,String,String,String,String,long,long,long): Boolean
login(String,String): Boolean
logout(String,String): Boolean
receiveMessage(String,String): Content
sendMessageTo(String,String,String,Content): Boolean

<<import>>

Content

source: String
date: String
text: String
number: Integer

(a) Required Interface

«interface »
OnlineMessengerService

OMS_CreateUser(String,String,String,String,String,String,String,long,long,long): Boolean
OMS_Login(String,String): Boolean
OMS2_Logout(String,String): Boolean
OMS_EditUser(String,String,String,String,String,String,String,String,long,long,long): Boolean
OMS_DeleteUser(String,String): Boolean
OMS2_GetUserList(String,String): String
OMS2_ReceiveMessage(String,String): Message
OMS2_SendMessageToChat(String,String,String,String): Boolean

<<import>>

Message

sender: String
dateSent: String
content: String
anInteger: Integer

(b) Candidate Web Service
Fig. 2. Instant Messenger Application – Chat

3   Interface Compatibility Analysis

The Selection Method, corresponding to the 2nd phase of the whole process for SOC-based application development, entails handling a 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 source of knowledge).

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 a useful preliminary information to help developers gain knowledge on several aspects. The outcomes may help eluding to early discard a candidate service upon simple mismatches but also preventing from a serious incompatibility. In addition, a 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 scheme that is divided into two parts: automatic matching cases and semi-automatic potential matchings. Both parts characterize syntactic similarity cases into 4 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.

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 [12]. 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 criteria 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_SendMessage-ToChat – i.e. P4 that is initially evaluated as P0. In Section 3.2 is shown how this incompatibility can be solved.

Level	Constraints
🟩 Exact Match (1 case)	Two operations must have identical signatures. (four identical conditions): [R1,N1,P1,E1]
🟨 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). Example: operation `logout` of `ChatIF` has equivalence *near-exact_2* with `OMS2_Logout` of `OMS2` by three identical conditions and 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.
🟦 Near-Soft Match (14 cases)	There cannot be two identical conditions, i.e. all conditions can be relaxed simultaneously.

Table 1. Assessment Scheme: Automatic Matchings

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 amount, type and order for parameters into the list
	P2	equal amount and type for parameters into the list
	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 into the list
	E2	equal amount 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)

Table 2. Syntactic Operation Matching Conditions for Interface Compatibility

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 amount, 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).

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.

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]
■ Soft Match	Two identical conditions, similar to automatic scheme. Either return or parameter (not both) with a non equivalent complex type or lost precision (R3/P4).
(13 cases)	Example: operation `sendMessageTo` of `ChatIF` could match operation `OMS2_SendMessageToChat` of `OMS2`. However, there is a parameter of complex type (`Content`) on operation `sendMessageTo` without a match into the operation `OMS2_SendMessageToChat` in which all parameters are String types (initially evaluated as P0). Now they can be re-evaluated considering that the wildcard type String might contain a chain of all fields from the complex type (`Content`) – 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.

Table 3. Assessment Scheme: Solving Mismatches

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 fit 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).

The final outcome of the Interface Compatibility step is a matching list characterizing each correspondence according to the four levels of the Assessment Scheme. 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 matching list might result as follows:

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

The success on the precision achieved during the Interface Compatibility step is very important 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

In this section is shown in detail the evaluation's results for the example presented in Section 2.1. Then another case study is briefly described.

4.1   Instant Messenger – Chat

Figure 3 shows the automatic matching results for `ChatIF` and service `OMS2`, where a mismatch is identified for operation `sendMessageTo` of `ChatIF` (depicted with a red cell) for which a semi-automatic solution has been set up by a *soft_25* 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. Although a specific correspondence was identified for each operation $op_r \in I_R$, a conclusive decision to accept or reject the candidate service S must be made through the subsequent step of behavior compatibility. This step is omitted for brevity reasons, though a similar procedure can be seen in [9], where the expected compatibility had been found.

4.2   Weather System

This case study is a system in which it is required to provide temperature information on both celsius and fahrenheiht scales. A required interface I_R has been defined in the Java format, named `TemperatureIF`, which is shown in Figure 4(a). The candidate web service is named `TempConvert`[4] and its interface I_S is shown in Figure 4(b). 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 5. In this case, both operations from `TemperatureIF` obtained similar result by two *soft_32* 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
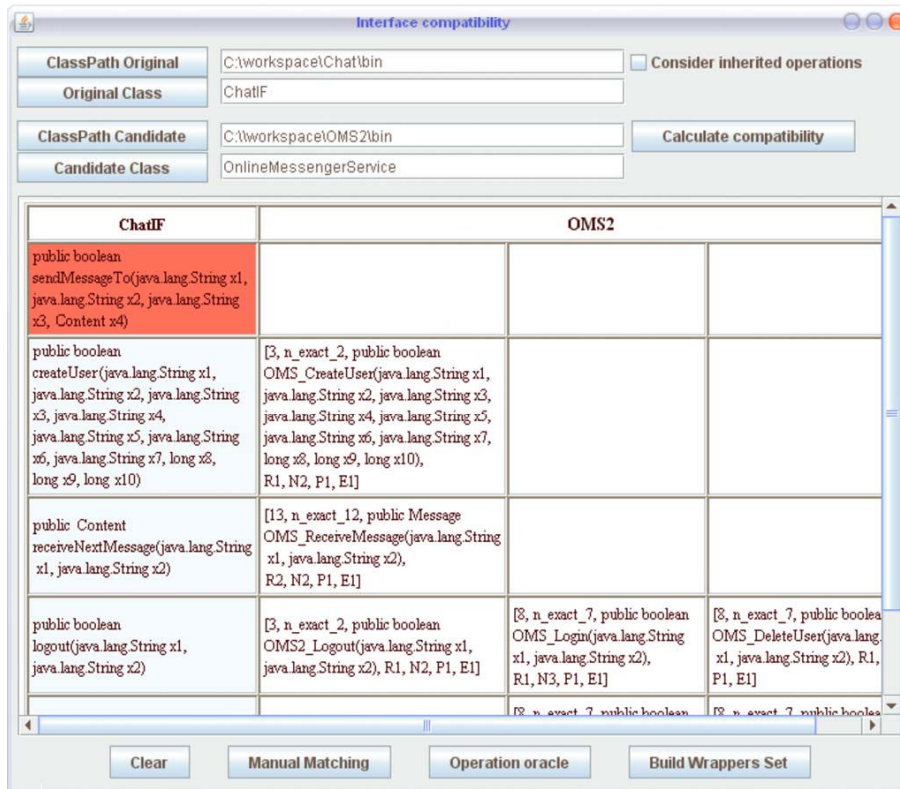
---

[4] *http://www.w3schools.com/webservices/tempconvert.asmx?WSDL*

Fig. 3. Automatic Interface Compatibility for ChatIF–OMS2

parameters (`R2,P3`). These results do not give an specific correspondence for each operation $op_r \in I_R$, so this matching list has been evaluated under the subsequent step of behavior compatibility for which the corresponding operations where identified – this step is omitted for brevity reasons.

The case studies above 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, a developer may analyze different scenarios of compatibility when a low level had been identified, 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.

5   Conclusions and Future Work

In this paper we have presented details of a Selection Method which allows to evaluate a candidate web service for its likely integration into a SOC-based application under de-

ChatIF	OMS2		
public boolean sendMessage-To (String x1, String x2, String x3, Content x4)	[40, soft_25, public boolean OMS2_SendMessageToChat (String x1, String x2, String x3, String x4), R1,N2,P4,E1]		
public boolean createUser (String x1, String x2, String x3, String x4, String x5, String x6, String x7, long x8, long x9, long x10)	[3, n_exact_2, public boolean OMS_CreateUser (String x1, String x2, String x3, String x4, String x5, String x6, String x7, long x8, long x9, long x10), R1, N2, P1, E1]		
public Content receiveNext-Message (String x1, String x2)	[13, n_exact_12, public Message OMS_ReceiveMessage (String x1, String x2), R2, N2, P1, E1]		
public boolean logout (String x1, String x2)	[3, n_exact_2, public boolean OMS2_Logout (String x1, String x2), R1, N2, P1, E1]	[8, n_exact_7, public boolean OMS_Login (String x1, String x2), R1, N3, P1, E1]	[8, n_exact_7, public boolean OMS_DeleteUser (String x1, String x2), R1, N3, P1, E1]
public boolean login (String x1, String x2)	[3, n_exact_2, public boolean OMS_Login (String x1, String x2), R1, N2, P1, E1]	[8, n_exact_7, public boolean OMS2_Logout (String x1, String x2), R1, N3, P1, E1]	[8, n_exact_7, public boolean OMS_DeleteUser (String x1, String x2), R1, N3, P1, E1]

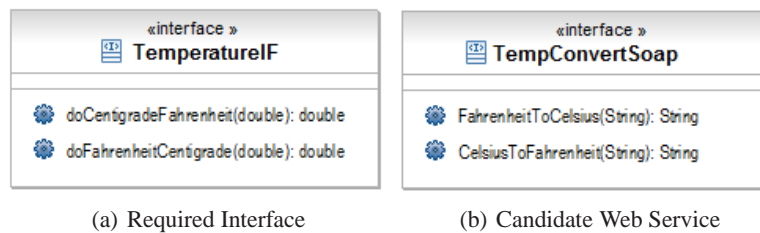**Table 4.** Final Interface Compatibility for ChatIF–OMS2

(a) Required Interface   (b) Candidate Web Service

Fig. 4. Weather System

TemperatureIF	TempConvertSoap	
public double doCentigradeFahren-heit (double x1)	[47, soft_32, public String fahrenheitToCelsius (String x1), R2, N2, P3, E1]	[47, soft_32, public String celsiusToFahrenheit (String x1), R2, N2, P3, E1]
public double doFahrenheitCenti-grade (double x1)	[47, soft_32, public String fahrenheitToCelsius (String x1), R2, N2, P3, E1]	[47, soft_32, public String celsiusToFahrenheit (String x1), R2, N2, P3, E1]

Table 5. Interface Compatibility for TemperatureIF–TempConvert

velopment. 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 where the candidate service will be inserted. 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 tech-

niques to better analyzing concepts from interfaces, which has been initially applied on the EasySOC approach. Another work is related to the behavior compatibility step, for which we are developing strategies for test minimization. The goal is to structure a manageable set of test cases mainly to improve efficiency but also to better understand services' behavior and the rationale behind achieved levels of compatibility.

References

1. Alexander, R., Blackburn, M.: Component Assessment Using Specification-Based Analysis and Testing. Tech. Rep. SPC-98095-CMC, Software Productivity Consortium, Herndon, Virginia, USA (May 1999)
2. Bichler, M., Lin, K.: Service-oriented computing. Computer 39(3), 99–101 (2006)
3. Canfora, G., Di Penta, M.: Testing Services and Service-Centric Systems: Challenges and Opportunities. IT Professional 8(2), 10–17 (Mar/Apr 2006)
4. Cavallaro, L., Di Nitto, E.: An approach to adapt service requests to actual service interfaces. In: ACM International Workshop SEAMS'08. Leipzig, Germany (2008)
5. Cechich, A., Piattini, M.: Early detection of COTS component functional suitability. Information and Software Technology 49(2), 108–121 (2007)
6. Crasso, M., Mateos, C., Zunino, A., Campo, M.: EasySOC: Making Web Service Outsourcing Easier. Information Sciences (2010)
7. Crasso, M., Zunino, A., Campo, M.: Easy web service discovery: A query-by-example approach. Science of Computer Programming 71(2), 144–164 (April 2008)
8. Erickson, J., Siau, K.: Web service, service-oriented computing, and service-oriented architecture: Separating hype from reality. Journal of BD Management 19(3), 42–54 (2008)
9. Flores, A., Cechich, A., Zunino, A., Polo, M.: Testing-Based Selection Method for Integrability on Service-Oriented Applications. In: 5th IEEE ICSEA'10. pp. 373–379. Nice, France (August 2010)
10. Flores, A., Polo, M.: Testing-based Process for Component Substitutability. Software Testing, Verification and Reliability p. 33 (2010), [early view press]
11. Freedman, R.S.: Testability of Software Components. IEEE Transactions on Software Engineering 17(6), 553–564 (June 1991)
12. Gosling, J., Joy, B., Steele, G., Bracha, G.: Java^{TM} Language Specification. Sun Microsystems, Inc, Addison-Wesley, US, 3rd. edn. (2005), http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html
13. Jaffar-Ur Rehman, M. et.al.: Testing Software Components for Integration: a Survey of Issues and Techniques. Software Testing, Verification and Reliability 17(2), 95–133 (June 2007)
14. Johnson, R.: J2EE Development Frameworks. Computer 38(1), 107–110 (2005)
15. Kung-Kiu, L., Zheng, W.: Software Component Models. IEEE Transactions on Software Engineering 33(10), 709–724 (October 2007)
16. Mariani, L., Papagiannakis, S., Pezzè: Compatibility and Regression Testing of COTS-component-based software. In: IEEE ICSE. pp. 85–95. Minneapolis, USA (May 2007)
17. Mateos, C., Crasso, M., Zunino, A., Campo, M.: Separation of Concerns in Service-Oriented Applications Based on Pervasive Design Patterns. In: 25th ACM SAC'10 (2010)
18. McCool, R.: Rethinking the Semantic Web. IEEE Internet Computing 9(6), 86–87 (2005)
19. Orso, A. et.al.: Using Component Metadata to Regression Test Component-based Software. Software Testing, Verification and Reliability 17, 61–94 (May 2006)
20. Stuckenholz, A.: Component Evolution and Versioning State of the Art. ACM SIGSOFT Software Engineering Notes 30(1), 7–20 (January 2005)
21. Wang, H., Huang, J., Qu, Y., Xie, J.: Web services: problems and future directions. Journal of Web Semantics 1(3), 309–320 (2004)