Behavior Assessment based Selection Method for Service Oriented Applications Integrability[*]

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. [martin.garriga, andres.flores, alejanda.cechich]@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 previous work we have presented an approach to assist developers on the selection of services from a syntactic viewpoint of a matchmaking process for interfaces compatibility. In this paper we extend the approach to assess the behavior of services taking advantage of a black-box testing framework to verify compatibility on the expected execution behavior of a candidate service. This paper analyzes the selection method through a case study, to show its potential on determining the best choice of a service among a set of candidates.

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

1. Introduction

Service-Oriented Computing (SOC) promotes building distributed applications in heterogeneous environments [1]. Service-oriented applications are developed by reusing existing third-party components or services that are invoked through specialized protocols. The SOC paradigm has been widely adopted by using the Web Services technology [2], which leads to a concrete decentralization of business processes and a low investment of new technologies and execution platforms. However, the efficient reuse of existing Web Services is still a major challenge. On one side, searching for candidate services on the Web implies a manual task yet, mainly exploring web catalogs usually showing poorly relevant information. On the other side, the result of a prosperous search requires skillful developers to deduce the most appropriate service to be selected from the set of candidates, for the subsequent integration tasks. Even with a reduced set of services, the required assessment effort could be overwhelming. Not only functional and non-functional properties must be explored on candidates, but also the required adaptations for a correct integration allowing client applications to consume services while enabling loose coupling for maintainability.

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

In order to ease the development of SOC-based applications we presented on previous work [3,4] a proposal for *discovery*, *selection* and *integration* of services, which is based on two recent approaches particularly concerned on development and maintainability. The first approach, called EasySOC [5], provides specific semi-automated methods for both *discovery* and *integration* of services. The second approach [6], was initially developed to work with software components by supplying a method for *selection* of the most appropriate third-party candidate component, as a solution for substitutability of component-based systems. In fact, this paper is focused on the *selection method* to detail the last extensions carried out, which allow to steadily use it in the context of service-oriented applications. Both approaches supply a semi-automatic tool support, which have been conveniently integrated to validate the ideas proposed in our work.

The main aspect of the *selection method* is the use of testing techniques to achieve a reliable level on the required compatibility of candidate services. This is based on the observability testing metric [7] that observes a component operational behavior by analyzing the functional mapping of data transformations (input/output) performed by a component. Therefore, a candidate service is assessed by an execution behavior process which requires a compliance test set to reveal a potential compatibility – as we analyzed on previous work [3,4,6] and was also discussed in [7].

The whole *selection method* comprises two assessment procedures: an *Interface Compatibility* analysis and a *Behavioral Compatibility* evaluation. The former is made at a syntactic level, by means of a comprehensive scheme to evaluate the interface provided by candidate services. The latter is based on a specific *Test Suite* (TS) which has been designed from a particular selection of testing coverage criteria, to achieve a behavior dynamic representation of services, viz. a Behavioral Test Suite.

The paper is organized as follows. Section 2 presents an overview of the Selection Method. Section 3 describes the Behavioral Test Suite, Section 4 focuses in the Interface Compatibility analysis, and Section 5 details the Behavior Compatibility evaluation. Finally, Section 6 presents the Related Work, while Conclusions and future work are presented afterwards.

2. Service Selection Method

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. However, the decision could also involve the acquisition of third-party components, which in turn could be solved with the connection to web services. When many candidates are discovered a developer still needs to deduce the most appropriate candidate service. Fig. 1 depicts our proposal intended to assist developers in the process of *selection* of web services, which is briefly described as follows:

The *selection method* requires the definition of a simple specification (in the form of a required interface I_R) as input for its two main assessment procedures. The Interface Compatibility evaluation is based on a comprehensive Assessment Scheme to recognize strong and potential matchings from a required interface (I_R) and the interface provided by candidate services (I_S). The outcome of this step is an Interface

Matching List where each operation from I_R may have a correspondence with one or more operations from I_S [6].

The Behavioral Compatibility evaluation is intended to analyze the execution of candidate services by means of a Behavioral Test Suite (TS), which is built to represent behavioral aspects from a third-party service. For this evaluation, the Interface Matching List produced in the previous step is processed, and a set of wrappers W (adapters) is generated, where remote invocations to I_S are solved through a proxy (P_S) derived from its WSDL description. Thus, a candidate service is evaluated by executing the TS against each $w \in W$, where at least 70% successful tests must be identified on some wrapper to confirm a behavioral compatibility [6]. Besides, such successful wrapper allows an in-house component to safely call the candidate service once integrated into a client application.

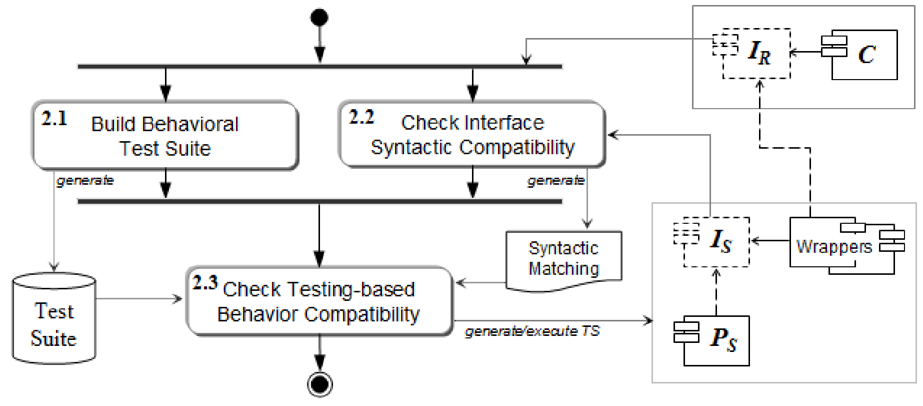

2.1 Build Behavioral Test Suite
2.2 Check Interface Syntactic Compatibility
generate
generate
Syntactic Matching
Test Suite
2.3 Check Testing-based Behavior Compatibility
generate/execute TS
I_R
C
I_S
Wrappers
P_S

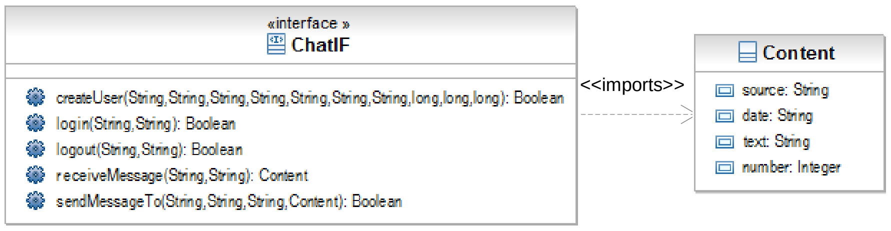
Fig. 1. Service Selection Method

Next sections provide detailed information particularly related to the aforementioned activities. A case study will be used to illustrate the usefulness of the Selection Method.

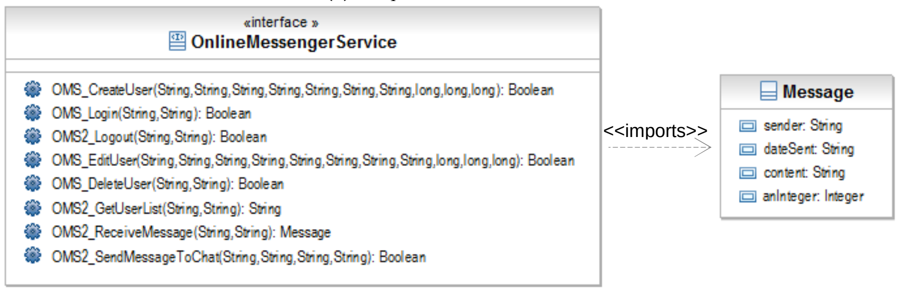
2.1. Case Study

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 that is showed in Fig. 2(a), and includes a complex type structure named Content for exchanging messages. By running the first phase of the process, a set of web services called OMS (Online Messenger Service) has been discovered at *http://www.nims.nl/*. Particularly we are interested in two of those services: OMS2 and OMS2_Simple. The former (*http://www.nims.nl/soap/oms2.wsdl*) provides an interface I_{S1} comprising 38 operations, and the most relevant ones are shown in Fig. 2(b), where another complex type structure named Message is used for enclosing the contents to be exchanged. The latter (*http://www.nims.nl/soap/oms2_simple.wsdl*), whose interface I_{S2}

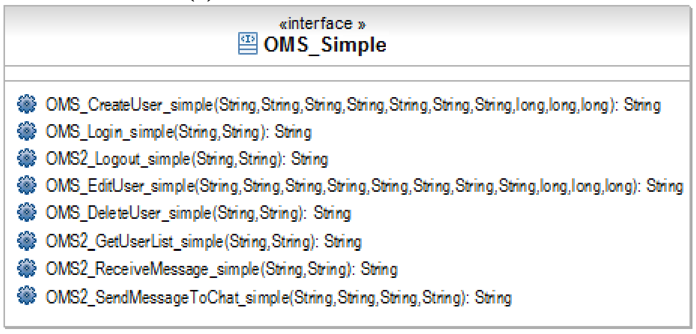
is shown in Fig. 2(c), uses the `String` type for the operations' return, instead of any other type (built-in or complex).


«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
<<imports>>
Content
source: String
date: String
text: String
number: Integer

(a) Required Interface ChatIF


«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
<<imports>>
Message
sender: String
dateSent: String
content: String
anInteger: Integer

(b) Candidate Web Service OMS2


«interface »
OMS_Simple
OMS_CreateUser_simple(String,String,String,String,String,String,String,long,long,long): String
OMS_Login_simple(String,String): String
OMS2_Logout_simple(String,String): String
OMS_EditUser_simple(String,String,String,String,String,String,String,String,long,long,long): String
OMS_DeleteUser_simple(String,String): String
OMS2_GetUserList_simple(String,String): String
OMS2_ReceiveMessage_simple(String,String): String
OMS2_SendMessageToChat_simple(String,String,String,String): String

(c) Candidate Web Service OMS2_Simple

Fig. 2. Instant Messenger Application – Chat

3. Behavioral Test Suite

In order to build a TS as a behavioral representation of services, specific coverage criteria for component testing has been selected. The goal of this TS is to check that a candidate service *S* with interface I_S coincides on behavior with a given specification described by a required interface I_R. Therefore, each test case in TS will consist of a set of calls to I_R's operations, from where the expected results were specified to determine acceptance or refusal when the TS is exercised against *S* (through I_S).

The Behavioral TS is based on the *all-context-dependence* criterion [3], in which synchronous events (e.g., invocations to operations) and asynchronous ones (e.g., exceptions) may have sequential dependencies on each other, causing distinct behaviors according to the order in which they (i.e., operations or exceptions) are called. The criterion requires traversing each operational sequence at least once.

Into our approach, *operational sequences* are represented by using regular expressions, where its alphabet is comprised of signatures from services' operations. This helps to describe a general pattern referred to as the "*protocol of use*" for a service interface [13,14].

Following with the case study presented in Section 2.1, to build a Behavioral TS for ChatIF, some steps supported by the TestOOJ tool [15] must be done. Initially a concrete class implementing the ChatIF interface must be created to describe the required behavior in the form of expected results for some representative selected test data. This *shadow* class is called Chat and it simply resembles an expected behavior according to some specific input data (or return a particular output data) for each operation within the ChatIF interface.

For example, the operation receiveNextMessage receives as input two `Strings` (user and password), and returns a `String` containing a message. The expected behavior is checking that the user has been previously created and logged-in, to then return a `String` containing a message. For this case study, the *test data* involve two users with their corresponding passwords, and the message is always "hello".

The next step implies defining the protocol of use (in the form of a regular expression). For the *shadow* class Chat could be as follows:

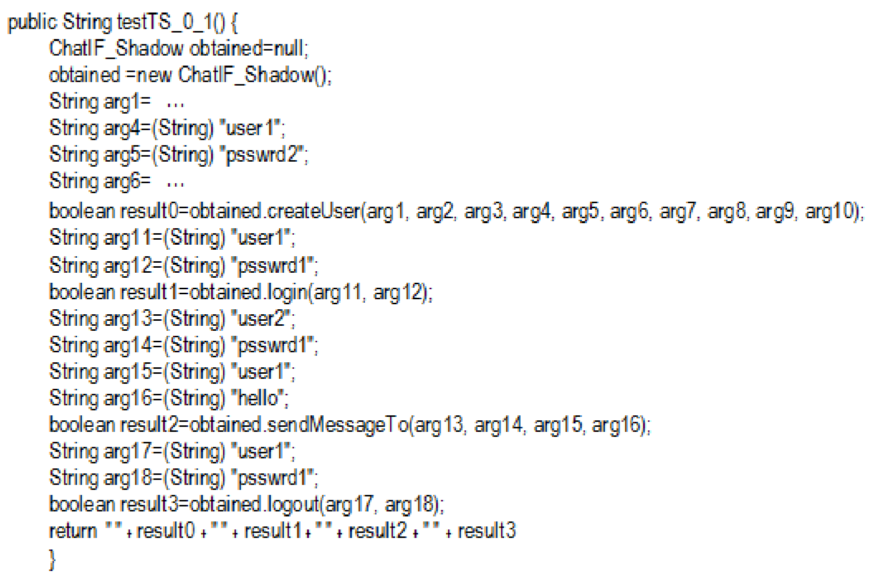
Chat createUser+ login (receiveNextMessage | sendMessageTo)$*$ logout

This regular expression is processed to derive sentences (describing operational sequences) according to a certain number of operations to be invoked, from where a set of test templates is generated. In this case, the minimum number would be 7, which produces 19 *test templates* with one or two occurrences of createUser operation, and single, alternated or combined occurrences of operations to send and receive messages. Detailed explanations of this step can be seen in [8].

After this, the selected *test data* values must be combined with the 19 *test templates* (operational sequences) to generate a TS in a specific format: based on the *MuJava* framework [16]. This combination was based on the *pairwise* algorithm [16], from where 468 test cases were generated in the form of methods inside a test driver file called MujavaChat. Fig. 3 shows the test method testTS_0_1, which exercises the following sequence: createUser, login, sendMessageTo, and logout.

4. Interface Compatibility

Particularly, the Interface Compatibility analysis is comprised of a practical scheme to analyze operations from the interface I_S (of a candidate service S), with respect to the required interface I_R. The outcome of this step may avoid early discarding 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.



```
public String testTS_0_1() {
      ChatIF_Shadow obtained=null;
      obtained =new ChatIF_Shadow();
      String arg1=  ...
      String arg4=(String) "user1";
      String arg5=(String) "psswrd2";
      String arg6=  ...
      boolean result0=obtained.createUser(arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8, arg9, arg10);
      String arg11=(String) "user1";
      String arg12=(String) "psswrd1";
      boolean result1=obtained.login(arg11, arg12);
      String arg13=(String) "user2";
      String arg14=(String) "psswrd1";
      String arg15=(String) "user1";
      String arg16=(String) "hello";
      boolean result2=obtained.sendMessageTo(arg13, arg14, arg15, arg16);
      String arg17=(String) "user1";
      String arg18=(String) "psswrd1";
      boolean result3=obtained.logout(arg17, arg18);
      return "" + result0 + "" + result1 + "" + result2 + "" + result3
      }
```

Fig. 3. MuJava Test Case for shadow class of ChatIF

Table 1 presents the Assessment Scheme that is comprised of four compatibility levels to define different syntactic constraints for a pair of corresponding operations. Constraints are based on individual conditions, summarized in Table 2, according to the elements of an operation's signature (return, name, parameter, exception). Types on operations from I_S should have at least as much precision as types on I_R. However, the `String` type is a special case, being considered as a *wildcard* type since it is generally used in practice to allocate different kinds of data. *Parameters* (`P`) and *return type* (`R`) are the most significant signature elements of the scheme.

Table 1. Assessment Scheme: Automatic Match and Semi-Automatic Mismatch Solving

Level	Part	Constraints
🟩 Exact Match	Auto (1 case)	Two operations must have identical signatures. (four identical conditions): [`R1,N1,P1,E1`]
🟨 Near Exact Match	Auto (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`)
	Semi-Auto (1 case)	Three identical conditions with the return that may have a nonequivalent complex type or lost precision: [`R3,N1,P1,E1`]
🟧 Soft Match	Auto (26 cases)	Similar to the previous level, but only two identical conditions. Previous exceptional cases may occur with lower equivalence conditions.
	Semi-Auto (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`).
🟦 Near Soft Match	Auto (14 cases)	There cannot be two identical conditions, i.e. all conditions can be relaxed simultaneously.
	Semi-Auto (40 cases)	Either two identical conditions with the condition `P4` or relaxing all conditions simultaneously.

The Assessment Scheme in Table 1 is able to recognize 108 cases for Interface Compatibility (where each part is comprised of 54 cases), from the combination of individual conditions (classified into the four levels of compatibility). For complex data types their comprising fields must be equivalent one-to-one with fields from a complex type counterpart.

Table 2. Syntactic Operation Matching Conditions for Interface Compatibility

Return	R0: Not compatible	R1: Equal return type
	R2: Equivalent return type (subtyping, Strings or Complex types)	R3: Non equivalent complex types or lost precision
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
	P2: Equal number and type for parameters	P3: Equal number and type at least equivalent (including subtyping, Strings or Complex types) for some parameters into the list
	P4: Nonequivalent complex types or lost precision	
Excep-tions	E0: Not compatible	E1: Equal number, type, and order for exceptions
	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 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 $op_S \in I_S$ is shaped. For example, let be I_R with three operations and I_S with five operations. The matching list might result as follows:

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

Each compatibility case represents a specific numeric value in the Assessment Scheme. For example, the value of *exact* equivalence is 4. Therefore, a totalized value could be determined to synthetize the *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 formula can be seen in [4].

If all operations in the *Interface Matching List* presents an *exact* equivalence, the *Syntactic Distance* between I_R and I_S is zero. This initially means that I_R is included into I_S, though I_S may have 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.

Following the case study, in Table 3 the matching result for ChatIF and service OMS2 is shown. No automatic matching has been found for ChatIF and OMS2Simple, and the mismatches have been solved in the semi-automatic step, by the notion of the `String` type as a *wildcard* type. At this point, the *Interface Matching List* for both candidates is available. Thus, the syntactic distance could be used to determine which of them is better to continue with the step of Behavior Compatibility. Based on the summary shown in Table 4, the syntactic distance between ChatIF and OMS2 is `29/20−1=0.45`, while the syntactic distance for OMS2_Simple is `38/20−1=0.9`. Because the lower value is better, the suggested candidate service would be OMS2. However, a conclusive decision to either accept or reject a candidate service *S* must be made through the step of Behavior Compatibility. The following section gives details of the step in which a required service's functionality is represented as a particular Test Suite.

Table 3. Interface Compatibility between ChatIF and OMS2

ChatIF	createUser	login	logout	receiveNextMessage	sendMessageTo
OMS2	OMS_CreateUser [R1, N2, P1, E1]	OMS_Login [R1, N2, P1, E1]	OMS2_Logout [R1,N2,P1,E1]	OMS_ReceiveMessage [R2, N2, P1, E1]	OMS2_SendMessage ToChat [R1,N2,P4,E1]
		OMS2_Logout [R1, N3, P1, E1]	OMS_Login [R1, N3, P1, E1]		
		OMS_DeleteUser [R1, N3, P1, E1]	OMS_DeleteUser [R1, N3, P1, E1]		

Table 4. Interface Compatibility Summary for ChatIF and services OMS2, OMS2_Simple

ChatIF	createUser	login	logout	receiveNextMessage	sendMessageTo	***Total Compatibility***
OMS2	5	5	5	6	8	29
OMS2_Simple	7	7	7	7	10	38

Total Best Compatibility = 20 (based on ChatIF size)

5. Behavior Compatibility

To carry out the Behavior Compatibility evaluation for a candidate service *S*, a wrappers set *W* needs to be built. Those wrappers will be necessary to execute the Behavioral TS (designed for the required interface I_R) against each $w \in W$. Initially, only the higher compatibility level of the *Interface Matching List* is considered.

This process is based on the *Interface Mutation* technique [18, 19], and it applies the mutation operator to change invocations to operations and another operator to change arguments for parameters. Then a Wrapper Generation Tree is created, where in each level of the tree is added the set of correspondences ($op_S \in I_S$) for a different operation $op_R \in I_R$.

When a list contains various parameters of the same or equivalent type, a combination of arguments is needed. Each combination arising from different parameters ordering should be added into the Wrapper Generation Tree, in the form of a new branch. For example, considering the case study, the operation sendMessageTo implies a likely case in which its complex parameter Content could match any of the `String` arguments from operation OMS2_SendMessageToChat, due to the `P4` condition. Therefore, in order to find the right match, there should be a swap into the parameter list, to successfully identify the behavior compatibility for those operations. Then, since the parameters list has a size of 4, the number of combinations rises to 16, as shown in Fig. 4, where each path from the root to a leaf node represents a different wrapper to be generated.

The combination process for parameters considers the syntactic equivalence conditions from Table 2, i.e., `[P1,P2,P3,P4]`, which impact in the tree in the following way:

- `P1`: No arguments' combination is needed.
- `P2`: Parameters of the same type are grouped and permutations are applied into each group. Then the whole solution is generated combining the permutations.
- `P3`: This case is similar to `P2`, but considering subtypes and the `String` type as a *wildcard*. This implies the following cases:
  - When the amount of numeric type parameters is equal between the evaluated operations, if there are `String` parameters in both operations, they have to be combined among each other. We assume as a good programming practice that if in the signature of an operation there are both numeric parameters and `String` parameters, the latter should not allocate numeric values.
  - When the amount of numeric type parameters is not equal between the operations, at most one of the `String` parameters is being used as a *wildcard*.
  - We assume that one complex counterpart exists for each complex type. Their comprising fields must be of an equivalent type and they have to be defined in the same order.
- `P4`: This case is similar to previous but it considers lost precision. Once again, there are two possibilities:
  - When the amount of numeric type parameters is equal, String types are paired between themselves.
  - When the amount of numeric type parameters is not equal, all parameters are used to generate combinations (except for complex types).
  - Complex parameters are treated as described earlier, but without restricting the order inside the structure.

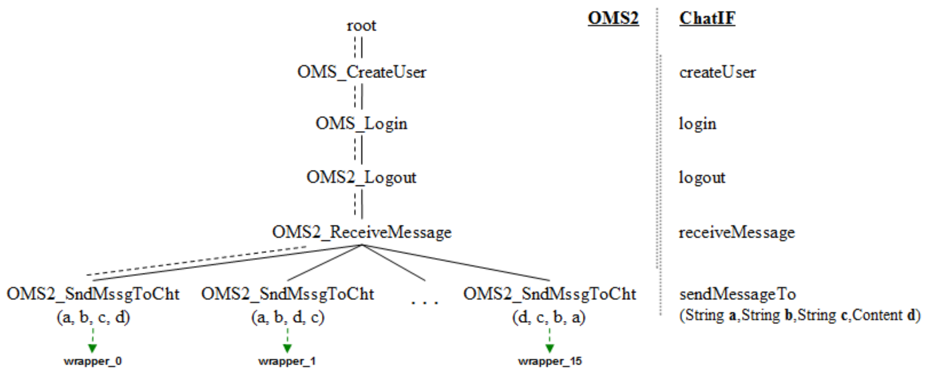
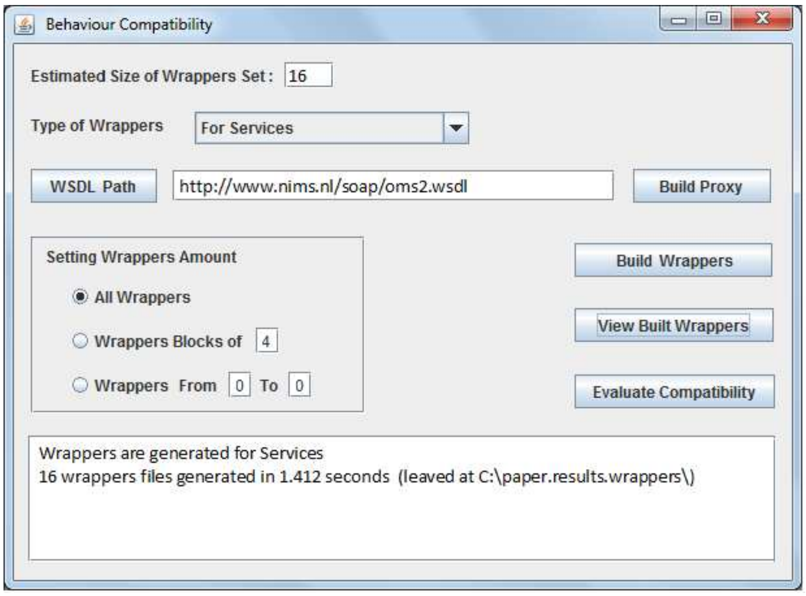

root
OMS2
ChatIF
OMS_CreateUser
createUser
OMS_Login
login
OMS2_Logout
logout
OMS2_ReceiveMessage
receiveMessage
OMS2_SndMssgToCht (a, b, c, d)
OMS2_SndMssgToCht (a, b, d, c)
. . .
OMS2_SndMssgToCht (d, c, b, a)
sendMessageTo (String a,String b,String c,Content d)
wrapper_0
wrapper_1
wrapper_15

Fig. 4. Wrapper Generation Tree for ChatIF and OMS2

These conditions may be simplified, especially for `P3` and `P4`, by establishing a blind combination among parameters. However, by assuming those conditions, the number of combinations (and the generated wrappers) substantially decreases. Since scalability is a key factor when generating wrappers, we introduced a partial wrappers' set generation procedure. This avoids reaching the physical limit imposed by the file system. A developer may generate the whole set, separated blocks or a subset of specific wrappers. For this reason, wrappers are numerated in sequence. In Fig. 5 is showed how to interact with the process of wrappers generation.



Estimated Size of Wrappers Set : 16

Type of Wrappers: For Services

WSDL Path: http://www.nims.nl/soap/oms2.wsdl

Setting Wrappers Amount: [✓ All Wrappers] Wrappers Blocks of 4; Wrappers From 0 To 0

```
Wrappers are generated for Services
16 wrappers files generated in 1.412 seconds  (leaved at C:\paper.results.wrappers\)
```

Fig. 5. Wrappers generation for Behavior Compatibility

Service Wrappers evaluation.

At this point, the Behavioral Assessment activity requires executing the Behavioral TS (built through the required interface I_R) against candidate services through the generated wrappers.

In this process, the wrappers are generated with an additional responsibility of *auto-configuration*, by instantiating the corresponding subclass for I_S (of a service S). In addition, the subclass implementing the interface I_S, which links wrappers to the proxy P_S, is also auto-configurable by instantiating classes comprising the generated proxy. Fig. 6 depicts the class structure for the ChatIF case study. The TS MujavaChat instantiates and invokes the Chat class, which represents not only the shadow class for the required interface ChatIF, but also represents the wrappers. This is done to avoid name modifications into the TS (designed for the *shadow* class).

Thus, if a wrapper successfully passes at least 70% of the Behavioral TS, it will be correctly describing the required behavior defined by the *shadow* class. Finally, this wrapper may be used instead of the *shadow* class allowing a safe integration of a candidate service.

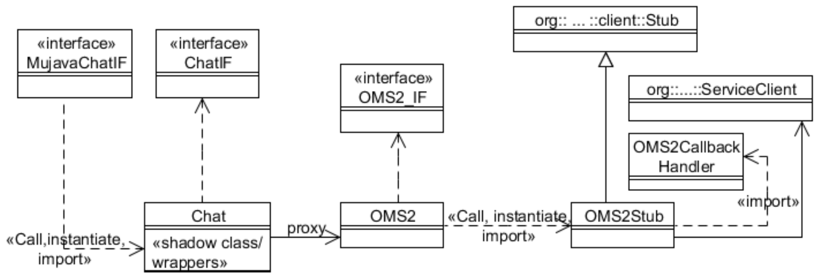


Fig. 6. TS for ChatIF to evaluate Wrappers through the Proxy.

Table 5 shows a summary of results from running the MujavaChat TS against the 16 wrappers for OMS2 service, where only one wrapper passed successfully the tests. For OMS2_Simple, were also generated 16 wrappers by the same situation with the parameter list in the operation OMS2_SendMessageToChat. Only one successful wrapper had been identified. Although, for both services was possible to find a compatibility on execution behavior. Again, the *syntactic distance* is the key factor to determine the definite service selection. In this case, the lower and best syntactic distance value corresponds to the OMS2 service.

Table 5. Compatibility Summary for ChatIF and services OMS2, OMS2_Simple

Services	Compatibility Value	Syntactic Distance	Amount of Wrappers	Number of Successful Wrappers
OMS2	29	0.45	16	(1)wrapper0
OMS2_Simple	38	0.9	16	(1)wrapper0

Since the selection method has been defined from a testing based assessment model, intermediate processes were defined not only to perform an evaluation of candidate services, but also to provide an early solution through the testing activity. The process offers a pragmatic guide to analyze any *off-the-shelf* component, including web services as a particular form of software component [10].

6. Related Work

The work in [21] is very close to our goals. The approach intends to evaluate compatibility for services with two purposes: substitutability and composability. The evaluation is based on input and output data registered after testing individual operations for each candidate service. To do this, a different TS is built for each service to be evaluated, which is based on a selected input data (either randomly or manually). The main intent of our approach is fulfilling a required functionality through a selected candidate service. For this, the expected behavior is described in form of a specific and unique TS, which is then exercised against services under evaluation. The main aspect of our TS relies on describing a complex behavior exhibited by operational sequences (instead of testing individual operations), which is more likely on stateful web services [1,11] – i.e., those with a modal characteristic

[22]. Additionally, the behavioral evaluation is only done after passing the syntactic Interface Compatibility analysis, which reduces computation for the testing phase. Thus, only those candidates with a high detected chance of compatibility will be put under test for Behavior Compatibility evaluation, making the whole process more efficient without losing effectiveness.

The work in [23] is also concerned with substitutions of inoperable services with compatible ones. Automatic finding for optimal solutions implies the challenging issue of how to discern the behavior of services. The approach attempts to discover and comprehend services' behavior and classify them into clusters by means of compliance testing. Behavior tables are created to elicit services' behavior by an iterative process that starts with random testing values to achieve the services clustering. As recognized by the authors, the whole process of eliciting service behavior tables implies a costly effort, where performance improvements are an edge of their further work. Similar to our proposal, this approach does not assume the existence of ontologies or any sort of semantic tagging. However, the approach has a very low confidence on any service description, also ignoring WSDL specifications. On the contrary, into our proposal the comparison of WSDL descriptions plays an important role with a high influence on performance. Also, several Information Retrieval-based approaches have shown their effectiveness on facilitating service discovery and selection while working upon WSDL descriptions [20].

The work in [24] is concerned with the improvement of test efficiency during service selection and composition, focusing in dependability and trustworthiness issues. A framework is proposed to support group testing, applied over a set of atomic services that could be potential parts of a service composition. For each service specification, there could be many functional candidates. The group testing mechanism broadcasts the test cases to all atomic candidate services. The oracle for each test case is generated by a voting service based on the majority principle. The same service collects the outputs and then dynamically evaluates the number of disagreements into each service profile. Then a rank is built based on the service's reliability and the test cases' effectiveness, identifying and eliminating test cases with overlapping coverage.

Our work is based on a full coverage TS, particularly applying the all-context-dependence criterion over operational sequences. As we mentioned earlier, we applied minimization strategies to address the unwieldy amount of test cases. Nevertheless, another simple solution to cut down computation could be directly designing a reduced TS based on the result of the Interface Compatibility step. Test cases could be generated only for those operations without a single syntactic correspondence. This avoids executing the whole TS against the wrapper set that is built in the Behavioral Compatibility step.

Another work [25] is intended to cope with Web service testing. A collaborative testing framework has been proposed, where testing tasks are performed through the collaboration of various test services (T-services) that are registered, discovered and invoked at runtime using an ontology of software testing called STOWS. Each functional service should be accompanied with a special T-service to avoid disturbing its normal operation, though managing the T-services' set introduces an inconvenient overhead. For this reason test brokers were introduced to deal with their composition and coordination. However, test brokers must perform a centralized control function, which may derive into a bottleneck. The proposed framework is particularly intended to verify a proper service execution through strategies to find faults, and also using a semantic Web service approach. Instead, our proposal is oriented to compliance testing, since the TS is used to assess candidate services on their expected behavior. As semantic information of web services such as ontologies is rarely available, our TS is built from syntax definitions of Web Services in WSDL language.

Other important related work about testing SOC-based systems is summarized in [26-28], which includes SOAP testing (to check publication and discoverability, among others), model-based SOA testing (using UML, Petri Nets, FSM, BPEL, etc., to describe complex behavior of compositions), agent-based or monitoring approaches (involving performance and reliability issues), and fault-based testing (such as XML perturbation, WSDL mutation, fault injection, etc.). Strategies for test data generation includes specification-based approaches (particularly WSDL-based), model-based approaches (similar to above), domain-slicing and partition-category (using XML Schemas and OWL-S). Some approaches for model-based SOA testing apply symbolic execution (based on extensions of FSMs) or model checking (by deriving OWL-S or BPEL specifications and making use of SPIN, NuSMV, or Blast tools). Although the main goal of those approaches implies to check for correctness of atomic services or compositions, some of the applied strategies are carefully considered to make improvements into our compliance testing oriented approach.

7. 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 Interface Compatibility analysis and a Behavioral Compatibility evaluation. Additionally, such selection might consider other aspects like *Quality of Service* parameters – e.g., performance, security, and so on.

Particularly, the Behavioral Compatibility activity was improved in this work. The *wsdl* specifications were added as a valid input to the process, automatically generating the logic for remote connection (proxy and *stub*). All versions of *wsdl* language are supported now. The wrappers' generation step also gained both flexibility and expressiveness, supporting all the subtypes definition introduced in the Interface Compatibility activity. Finally, when a developer identifies a subset of wrappers with a major probability of success, he or she is allowed to generate only those wrappers (or even a unique wrapper), substantially reducing the computation effort.

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 have 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 (BPEL) and service orchestration [10, 11].

References

1. 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).
2. Bichler, M., Lin, K.: Service-oriented computing. Computer, 39(3), 99-101 (2006)
3. Flores, A., Cechich, A., Zunino, A., Polo, M.: Testing-Based Selection Method for Integrability on Service-Oriented Applications. In: 5th IEEE ICSEA'10, 373-379 (2010)
4. Garriga, M., Flores, A., Cechich, A., Zunino, A.: Testing-based Process for Service-oriented Applications. 30th IEEE SCCC'11 (2011) [post-proceedings in press]
5. Crasso, M., Mateos, C., Zunino, A., Campo, M.: EasySOC: Making Web Service Outsourcing Easier. Information Sciences, Elsevier (2010)
6. Flores, A., Polo, M.: Testing-based Process for Component Substitutability. Journal STVR, Wiley, p. 33 (2010) [early view press]
7. Jaffar-Ur Rehman, M., et.al.: Testing Software Components for Integration: a Survey of Issues and Techniques. Journal STVR. Wiley, 17(2), 95-133 (2007)
8. Garriga, M., Flores, A., Cechich, A., Zunino, A.: Practical Assessment Scheme to Service Selection for SOC-based Applications. In: 12th ASSE'11, part of 40 JAIIO, 204-215. (2011)
9. Canfora, G., Di Penta, M.: Testing Services and Service-Centric Systems: Challenges and Opportunities. IT Professional, 8(2), 10-17 (2006).
10. Kung-Kiu, L., Zheng, W.: Software Component Models. IEEE Transactions on Software Engineering, 33(10), 709-724 (2007).
11. Weerawarana, S.; et al., Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WSReliable Messaging, and More. Prentice Hall PTR (2005).
12. Ye, W., Dai, P., Mei-Hwa, C.: Techniques for Testing Component-based Software. In: 7th IEEE ICECCS. Skovde, Sweden, pp. 222–232. (2001).
13. Kirani, S. H., Tsai, W. T.: Method Sequence Specification and Verification of Classes. Journal of Object-Oriented Programming, vol. 7, no. 6, pp. 28–38. (1994)
14. Object Management Group, Inc: Unified Modeling Language: Superstructure version 2.0. OMG Tech. Rep. http://www.omg.org (2005)
15. Polo, M., Tendero, S., Piattini, M.: Integrating Techniques and Tools for Testing Automation. Software Testing, Verification and Reliability. vol. 16, no. 1, pp. 1–37. (2006)
16. µJava Home Page: Mutation system for Java programs. http://www.cs.gmu.edu/offutt/mujava/. (2008)
17. Czerwonka, J.: Pairwise Testing in Real World. In: 24th PNSQC. Portland, OR, US, pp. 419–430. (2006)
18. Gosh, S., Mathur, A. P.: Interface Mutation. Software Testing, Verification and Reliability, 11:227–247. (2001)
19. Delamaro, M, Maldonado, J., Mathur, A.: Interface Mutation: An Approach for Integration Testing. IEEE Transactions on Software Engineering, 27(3):228–247. (2001)
20. Rodriguez, J.M., Crasso, M., Zunino, A., Campo, M.: Discoverability anti-patterns: frequent ways of making undiscoverable Web Service descriptions. In: Proceedings of the 10th ASSE. During 38th JAIIO, ISSN 1850-2792, pp. 1-15. Mar del Plata, Argentina. (2009)
21. Ernst, M., Lencevicius, R., Perkins, J.: Detection of Web Service Substitutability and Composability. In: WS-MaTe 2006: International Workshop on Web Services — Modeling and Testing, Palermo, Italy, pp. 123–135. (2006)

22. Binder, R.: Testing Object Oriented Systems - Models, Patterns and Tools. Addison-Wesley. (2000)
23. Church, J., Motro, A.: Learning Service Behavior with Progressive Testing. In: IEEE SOCA'11, Irvine, USA. (2011)
24. Tsai, W., Zhou, X., Chen, Y., Bai, X.: On Testing and Evaluating Service-Oriented Software. Computer, vol. 41, no. 8, pp. 40–46. (2008)
25. Zhu, H. Yufeng, Z.: Collaborative Testing of Web Services. IEEE Transactions on Services Computing, vol. 5, no. 1, pp. 116–130. (2010)
26. Canfora, G., Di Penta, M.: Service Oriented Architectures Testing: A Survey. ISSSE 2006-2008, vol. LNCS, no. 5413, pp. 78–105, springer. (2009)
27. Palacios, M. Garcia-Fanjul, J., Tuya, J.: Testing in Service Oriented Architectures with Dynamic Binding: A Mapping Study. Information and Software Technology. Elsevier, vol. 53, no. 3, pp. 171–189. (2011)
28. Bozkurt, M., Harman, M. Hassoun, Y.: Testing Web Services: A Survey. Centre for Research on Evolution, Search & Testing, King's College, London, Tech. Rep. TR-10-01, (2010)