Testing-based Behavioral Assessment for Service Selection

Martín Garriga[1,3], **Alan De Renzis**[1], **Andres Flores**[1,3], **Alejandra Cechich**[1], and **Alejandro Zunino**[2,3]

[1] *GIISCo Research Group, Facultad de Informática, Universidad Nacional del Comahue*
[2] *ISISTAN Research Institute, UNICEN*
[3] *CONICET (National Scientific and Technical Research Council.*

Abstract

Building Service-oriented Applications implies the selection of adequate services to fulfill required functionality. Even a reduced set of candidate services involves an overwhelming assessment effort. In a previous work we have presented an approach to assist developers in the selection of Web Services. In this paper we detail its behavioral assessment procedure, which is based on compliance testing. This is done by a specific Behavioral Test Suite for exposing required messages interchange from/to a client application and a Web Service. In addition, helpful information takes shape to build the needed adaptation logic to safely integrate the selected candidate into a Service-oriented Application. A concise case study shows the potential of this approach for both selection and integration of a service among a set of candidates.

Key Words

Service Oriented Applications, Software Testing, Web Services.

1. Introduction

Service-oriented Applications imply a business facing solution that consumes services from one or more providers and integrates them into the business process. Certainly, it is not expected to deliver core enterprise applications purely by assembling services from multiple sources [23]. Thus, a Service-oriented Application can be viewed as a component-based application that is created by assembling two types of components: internal that are locally embedded into the application, and external that are statically or dynamically bound to a service [5]. Although developers of consumer applications do not need to know the underlying model and rules of a third-party service, its proper reuse still implies quite a big effort. On one side, yet searching for candidate services is mainly a manual exploration of Web catalogs usually showing poorly relevant information. On the other side, even a favorable search result requires skillful developers to deduce the most appropriate service to be selected for subsequent integration tasks. The effort on assessing candidate services could be overwhelming. Not only services contracts must be assessed, but also correct adaptations so client applications may safely consume services while enabling loose coupling for maintainability.

To ease the development of Service-oriented Applications we presented in a previous work [12] a proposal to assist developers in the selection of services. This proposal is based on a previous approach in which we addressed a solution for substitutability of component-based systems [10]. The approach comprised a selection method to recognize the most appropriate third-party candidate component, where the conventional compatibility assessment was complemented by using black-box testing criteria to explore components behavior. Since Web Services are considered as software components [16], such selection method was easily adapted to the Service-oriented Applications context. The initial approach was based in the *observability* testing metric [10,15] that observes a component operational behavior by analyzing its functional mapping of data transformations (input/output). In the context of Service-oriented Applications the observability testing metric has been discussed in [25,2]. In addition, a proper service definition entails as a key feature an operational behavior description, besides its interface and identifier [17].

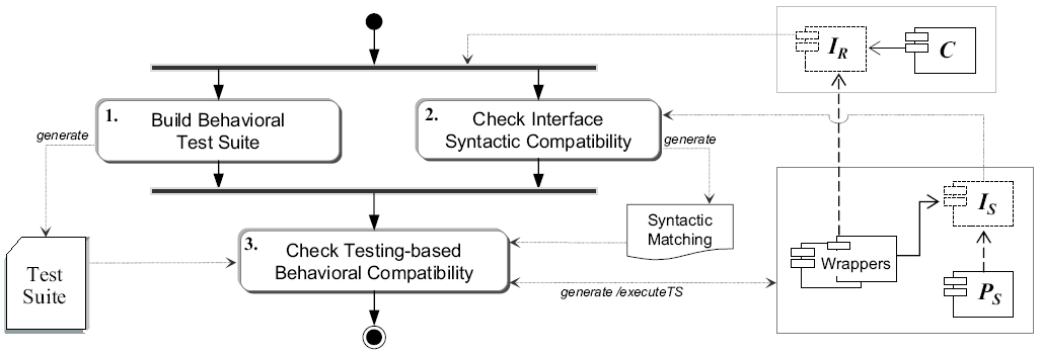


Figure 1. *Testing-based Selection Method*

Although exploring functional mappings could be extensive, attending certain aspects and representative data is more efficient without neglecting effectiveness. The choice on testing criteria has been essential to design a compliance Test Suite (TS) to represent required service behavior, namely *Behavioral TS*. Therefore, a candidate service is assessed by an execution behavior process to reveal its potential compatibility. This selection method thus reaches a reliable level since a service compatibility can be confirmed when the applied testing criteria is adequately satisfied.

In this paper, we detail the testing-based behavioral assessment process, in which helpful information also takes shape to build the needed adaptation logic to safely integrate the selected candidate into a Service-oriented Application. Through a concise case study we illustrate the Selection Method showing its usefulness for both selection and integration of a service among a set of candidates.

The paper is organized as follows. Next is described the Selection Method while an illustrative case study is also introduced. Section 3 explains the steps to build a Behavioral TS. Section 4 briefly describes the Interface Compatibility analysis. Section 5 describes the Behavioral Compatibility evaluation. Section 6 details the case study. Section 7 presents related work. 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, some internal components could be bound to reusable Web Services, requiring a search for candidates.When many services are discovered a developer still needs to determine the most appropriate candidate. Figure 1 depicts our proposal to assist developers in the selection of Web Services, which is briefly described as follows:

The Selection Method needs the definition of a simple specification, in the form of a required interface I_R (linked to an in-house component *C*), as input for its two main assessment procedures. The *Interface Compatibility* analysis is based on a comprehensive Assessment Scheme to recognize strong and potential matchings from I_R and the interface (I_S) provided by a candidate service *S* previously discovered. 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.

The *Behavioral Compatibility* evaluation analyzes the execution of candidate services by means of a *Behavioral TS*, which is built to describe required messages interchange from/to a third-party service *S*. For this evaluation, the *Interface Matching* list produced in the previous step is processed, and a set of wrappers *W* (adapters) is generated. Remote invocations to *S* are

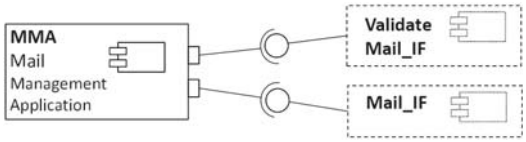
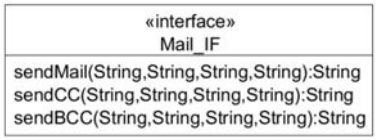


Figure 2. *Structure of Mail Management Application (MMA)*

(a) I_R *for Mail Validation*



(b) I_R *for Mail Sending*

Figure 3. *Required Interfaces for MMA's main features*

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 behavior compatibility. Besides, such successful wrapper allows an in-house component (C) to safely call the candidate service S (trough P_S) once integrated into the client application.

Next sections provide detailed information particularly related to the testing-based activities mentioned above. A case study will be used to illustrate the usefulness of the Selection Method.

2.1. Case Study

The case study has been outlined as a *Mail Management Application* (MMA) being developed under the Java platform. Figure 2 depicts the component structure of the application, with the invoking and coordinating component MMA and the interfaces for its required key features, which will be fulfilled by third-party Web Services. Responsibilities behind these features are: (1) a Mail Validation tool, to validate an email address; (2) a Mail Sending tool, to send emails to one or multiple receivers, in a blind (bcc) or the usual (cc) copy mode –emails must include both their subject and body.

Figure 3 shows a concrete structure for the required interfaces (I_R) of MMA, namely ValidateMail_IF and Mail_IF. Two sets of candidate services (one per required interface) have been built, as presented in Table 1.

To clearly illustrate the Service Selection steps, the case study is initially reduced to the required interface Mail_IF (Figure 3(b)) and one of its candidates: the AtMessaging service (Figure 4). The evaluation of the remaining candidate services for the two required interfaces is presented in Section 6, where the whole case study is developed.

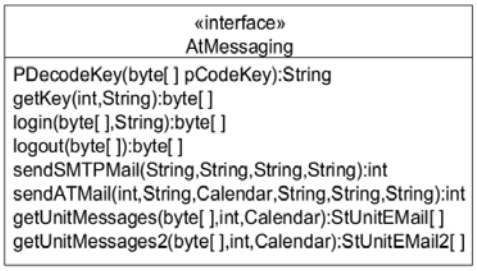


Figure 4. *Candidate Web Service AtMessaging*

Table 1. *Candidate Services for MMA features*

Required Interface	Candidate Service	WSDL Specification URI
Validate Mail IF	EmailVer	ws.cdyne.com/emailverify/emailvernotestemail.asmx?WSDL
	ValidateEmail	api.earnmydegree.com/emailvalidation/validateemailaddress.asmx?WSDL
	Email-Verify	ws.strikeiron.com/EmailVerify5?WSDL
Mail IF	ATMessaging	wd.air-trak.com/atmessagingws/ATMessaging.asmx?WSDL
	Communication Service	local repository
	cemailService	sal006.salnetwork.com:83/lucin/Email/CEmail.xml

3. Behavioral Test Suite

To build a TS as a behavioral representation of services, specific coverage criteria for component testing has been selected to fulfill the *observability* testing metric [10, 15]. A component operational behavior is observed by exploring its functional mapping of data transformations (input/output).

In the context of Service-oriented Applications, functional mappings imply messages interchange from/to a client component *C* and a third-party service *S*. Hence, 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 (on dependence with *C*). Therefore, each test case in TS will consist of a sequence of calls to I_R's operations, from where a set of specified expected results determine acceptance/refusal when the TS is exercised against *S* (through I_S).

To fulfill the *observability* testing metric, the *Behavioral TS* is based on the *all-context-dependence* criterion [15], 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. This means, specific messages interchange from/to a candidate service, where exceptions are denoted as *faults* in terms of WSDL terminology. The criterion requires to traverse each operational sequence at least once.

Operational sequences are represented in our approach with regular expressions, describing the protocol of use for a service interface [19]. The alphabet for regular expressions comprises signatures from service's operations. For interested readers, a complete description of basic coverage notions is given in [9].

By means of the case study presented in Section 2.1, the procedure to build a *Behavioral TS* is explained next.

3.1. Test Suite for Mail Feature

To build a *Behavioral TS* for Mail_IF, a concrete (*shadow*) class implementing this interface must be initially created to describe the required operational behavior. This *shadow* class is called Mail and simply resembles specific behavior in the form of expected results for some representative *test data*, for each operation within the Mail_IF interface. For example, the operation sendMail receives as input four Strings (sender, receiver, subject and body), and returns a String containing a control data (success/error).

The expected behavior is checking that the email is able to be sent to the receiver address by a successful return code. For this case study, the *test data* involve two valid email addresses (authors personal mails) for sender/receiver, and the subject and body is always "*hello*" and "*message*" respectively.

The next step implies to define the protocol of use (in the form of a regular expression). For Mail could be as follows:

Mail sendMail sendCc* sendBcc*

This regular expression is processed to derive sentences (describing operational sequences) according to the number of calling operations, from where a set of *test templates* is generated. In terms of testing, the kleene (∗) operator implies zero/one and at least one more occurrence of the operand. Therefore, the minimum calling operations in this case study would be 4, producing 6 *test templates* with one occurrence of sendMail operation followed by zero/single/double/combined occurrences of operations sendCc and sendBcc. Detailed explanations of this step can be seen in [9].

Next, the *test data* is combined with the 6 *test templates* (operational sequences) to generate the *Behavioral TS* in a specific format, based on the MuJava framework [18]. This combination was based on the *all–combinations* algorithm [13], from where 228 test cases were generated in the form of methods inside a test driver file called MuJavaMail.

```
public String testTS 5 1() {
        Mail obtained= null;
        obtained = new Mail();
        java.lang.String arg1= (java.lang.String) "martin.garriga@fi.uncoma.edu.ar";
        java.lang.String arg2= (java.lang.String) "derenzis.alan@gmail.com";
        java.lang.String arg3= (java.lang.String) "hello";
        java.lang.String arg4= (java.lang.String) "message";
        java.lang.String result0= obtained.sendMail(arg1, arg2, arg3, arg4);
        java.lang.String arg5= (java.lang.String) "andres.flores@fi.uncoma.edu.ar";
        java.lang.String result1= obtained.sendCc(arg1, arg5, arg3, arg4);
        java.lang.String arg6= (java.lang.String) "azunino@isistan.unicen.edu.ar";
        java.lang.String result2= obtained.sendBcc(arg1, arg6, arg3, arg4);
        return " " + result0 + " " + result1 + " " + result2;
    }
```

Figure 5. *MuJava Test Case for Mail*

Test cases return a String value that can be used for a comparative evaluation. Figure 5 shows the test method testTS_5_1, which exercises the following sequence: Mail, sendMail, sendCC, and sendBcc.

4. Interface Compatibility

The *Interface Compatibility* analysis comprises a practical Assessment Scheme that consist of two parts: automatic and semi-automatic matching cases –as shown in Table 2. Both parts characterize structural similarity cases into 4 levels of compatibility to analyze operations from the interface I_S (of a candidate service *S*), regarding the required interface I_R. Each compatibility level encloses a set of equivalence degrees defined as combinations of structural conditions.

Table 3 defines those conditions for operations signatures (return, name, parameter, exception). Types on operations from I_S should have at least as much precision as types on I_R. 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 [21]. Conditions R3 and P4 are the weakest being evaluated as incompatibilities in the automatic part of the scheme (treated as R0 and P0 respectively). The semi-automatic part of the scheme deals with those weakest conditions –as described in Table 2. Details of this step can be seen in [12].

The Assessment Scheme in Table 2 is able to recognize 108 cases of *Interface Compa-*

Table 2. *Assessment Scheme: Automatic and Semi-Automatic Matching Cases*

Level	Part	Constraints
■ Exact Match	Auto (1 case)	Identical signatures (four identical conditions): [R1,N1,P1,E1] Equivalence value = 4 (by adding the value 1 of each condition).
■ Near Exact Match	Auto (13 cases)	Three or two identical conditions. Remaining might be second conditions: (R2/N2/P2/E2). Exceptional cases: three identical conditions with a remaining third condition (N3/P3/E3). Equivalence values = [5-6].
	Semi-Auto (1 case)	Three identical conditions with return that may have a not equivalent complex type or lost precision: [R3,N1,P1,E1]. Equivalence value = 6.
■ Soft Match	Auto (26 cases)	Similar to previous level, but only two identical conditions. Previous exceptional cases may occur with lower conditions. Equivalence values = [7-8].
	Semi-Auto (13 cases)	Two identical conditions, similar to automatic. Either return or parameter (not both) with a nonequivalent complex type or lost precision (R3/P4). Equivalence values = [7-8].
■ Near Soft Match	Auto (14 cases)	There cannot be two identical conditions, i.e., all conditions can be relaxed simultaneously. Equivalence values = [9-11].
	Semi-Auto (40 cases)	Either two identical conditions with P4 or relaxing all conditions simultaneously (with R3/P4). Equivalence values = [9-13].

Table 3. *Structural 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	N0: Not compatible	N1: Equal operation name
	N2: Equivalent operation name (substring)	N3: Operation name ignored
Parameters	P0: Not Compatible	P1: Equal amount, type and order for parameters
	P2: Equal amount and type for parameters	P3: Equal amount and type at least equivalent (including subtyping, Strings or Complex types) for some parameters into the list
	P4: Nonequivalent complex types or lost precision	
Exceptions	E0: Not compatible	E1: Equal amount, type, and order for exceptions
	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).

tibility (each part comprises 54 cases). When certain mismatch cases are detected for I_R, a developer may outline a likely solution using context information from the application's business domain. We have identified specific cases in which a concrete compatibility can be set up by a semi-automatic mechanism. In addition, for a specific operation $op_R \in I_R$, there could be another correspondence that better fits for the application's context. Then, a developer is enabled to "*prioritize*" such correspondence even when an automatic match was identified. Besides, the success on the precision achieved during this step is essential to reduce computation effort for the subsequent step of *Behavioral Compatibility* evaluation (see Section 5).

The final outcome of this step is an *Interface Matching* list, in which for each operation $op_R \in I_R$, a list of compatible operations from I_S is shaped. For instance, 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 Interface Matching list might result as follows:

{ (opR1, {opS1, opS5}), (opR2, {opS2, opS4}), (opR3, {opS3}) }

Compatibility cases represent specific numeric values on the Assessment Scheme in Table 2 –e.g., the value of *exact* equivalence is 4. Therefore, from the *Interface Matching* list, a totalized equivalence value can be calculated to synthesize the achieved degree of *Interface Compatibility* between I_R and I_S. Only the highest compatibility level for each operation is considered to calculate that value, named *Compatibility Gap*, according to formula (1).

4.1. Interface Compatibility for Mail Feature

Table 4 shows the *Interface Compatibility* analysis for Mail_IF and the AtMessaging service. As can be seen, the same operation sendSMTPMail of the AtMessaging service matches the three operations from Mail_IF: sendMail, sendBcc and sendCc with a *near-exact_12* equivalence. They coincide on parameters and exceptions, with a subtype for return and a substring for operations names –i.e., the conditions [R2,N2,P1,E1] in Table 3.

In fact, the two last correspondences could be quite reasonable considering that after sending the main email copy, additional

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

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

Table 4. *Interface Compatibility between Mail_IF and AtMessaging*

Mail_IF	AtMessaging	Degree	Case	Conditions	Value
sendBcc	sendSMTPMail	n-exact_12	13	R2, N2, P1, E1	6
sendCc	sendSMTPMail	n-exact_12	13	R2, N2, P1, E1	6
sendMail	sendSMTPMail	n-exact_12	13	R2, N2, P1, E1	6
Total best value: 12 (based on Mail_IF size)			*Total Equivalence*		18

copies (Cc/Bcc) could also be iteratively sent with a similar procedure afterwards.

The total compatibility value between Mail_IF and the AtMessaging service is 18. This means, the *compatibility gap* can be calculated as: 18/12−1=0.5 according to formula (1). Although all operations from the required interface Mail_IF found a match, a conclusive decision to accept/discard the AtMessaging service must be made through the subsequent step of *Behavioral Compatibility* evaluation.

5. Behavioral Compatibility Evaluation

This step helps to differentiate from structurally similar operations, mainly assuring that interface correspondences also match at the behavioral level by providing the required functionality. The purpose is finding operations from a candidate service *S* that expose a similar behavior with respect to those specified in the required interface I_R. In our approach, this implies to exercise the *Behavioral TS* against the candidate service *S*.

The Interface Matching list is used to build a wrappers' set *W* for the candidate service *S*. Wrappers are based on the adapter pattern [11] simply forwarding requests to the candidate service *S* (through I_S). The size of *W* derives from combinations of operations matching. Instead of making a blind combination, a reduced number can be reach from the best correspondences in the *Interface Matching* list.

The wrapping approach makes use of *Interface Mutation* [14,7] by applying mutant operators to change invocations to operations and parameter values. The former is done through the list of matching operations. The latter, by varying arguments with the same type, for a given operation correspondence. If the size of *W* is too high, a developer may decide to either manually set correspondences for building just one wrapper or generate a manageable subset of wrappers.

Since the *Behavioral TS* will be finally executed against a candidate service *S*, then an additional consideration implies the physical connection to *S* to enable invoking operations exhibited in I_S. Thus, a proxy for *S* (P_S) is generated, from where the *Behavioral TS* will end up invoking the operations declared in I_S through P_S, which then invokes the remote service *S*.

After building wrappers, the testing step may proceed by taking each wrapper as the target testing class and executing the *Behavioral TS*. Test cases evaluation is done by means of the returned String value – e.g., the test case in Figure 5– which thus gives a binary result: success/failure. The percentage of successful tests for each wrapper determines its acceptance/refusal – i.e., either killing the wrapper (as a mutation case) or allowing it to survive. The greater the killed wrappers the better, because it makes easy to conclude (in)compatibility for a candidate service.

5.1. Running TS of Mail on AtMessaging

To initiate the *Behavioral Compatibility* between Mail_IF and AtMessaging, the *Interface Matching* list from the previous step must be processed to build the set of wrappers (*W*). A *Wrapper Generation Tree* is created, where each level of the tree adds the set of correspondences for a different operation of Mail_IF, as shown in Figure 6. Each path from the root to a leaf node represents a different wrapper to be generated. Notice that for all matchings between Mail_IF and AtMessaging the P1

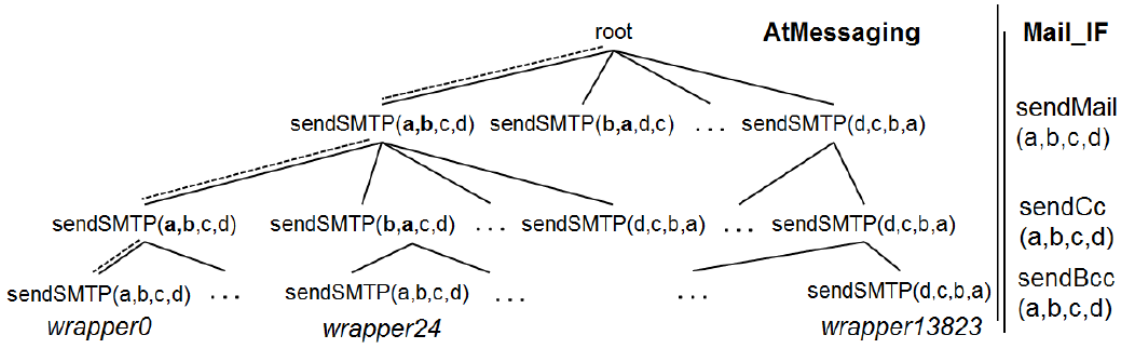


Figure 6. *Wrapper Generation Tree for Mail_IF and AtMessaging*

condition was found, implying a coincidence on parameters' type and order –as defined in Table 3.

This means, initially just one wrapper could be generated, corresponding to the first path (to the left) from the tree in Figure 6 –i.e., wrapper0. However, all operations of Mail_IF includes 4 String parameters, which may imply permutations when the parameters order is not considered. This could be done to find an adequate arrangement of parameters that might result in a successful wrapper, in case wrapper0 is killed (as a mutation case). Permutations rises to 24 for each level, making the whole number of wrappers to be 24∗24∗24=13824. Although this wrappers' set becomes unwieldy to be tested, a partial generation can be performed. Therefore, initially a subset (*W1*) of 24 wrappers was generated as a result of this step –from wrapper0 to wrapper23.

To run the Behavioral TS (MujavaMail) against the wrappers' set a specific structure (with a proxy for remote calls to the AtMessaging service) must be created, as shown in Figure 7. After this, the MujavaMail test file can be run against the subset (*W1*) of 24 wrappers. In this case, only wrapper0 passed successfully the tests, which confirms the expected behavior specified for the required interface Mail_IF.

Details of the whole case study, involving the remaining candidate services, are given in the following section.

6. Details of the Case Study

This section details the evaluation procedure for the whole case study about the Mail Management Application (MMA) presented in Section 2.1. The evaluation procedure for the Mail functionality is described in Section 6.1. Then a synthesis from evaluating the Mail Validation functionality is presented in Section 6.2.

6.1. MMA - Mail Sending Feature

Since the AtMessaging candidate service was taken for an initial example in the previous sections, only the remaining candidates are analyzed below.

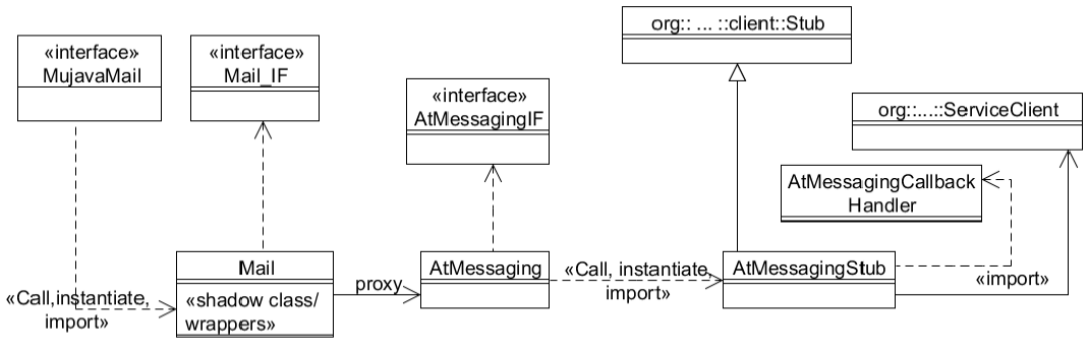


Figure 7. *Structure of Service Wrappers to test a candidate service S (AtMessaging)*

Candidate CommunicationService:
When analyzing on *Interface Compatibility*, no matching was found for operations of the required interface Mail_IF and the CommunicationService candidate. Although this candidate provides one operation to send emails, it makes use of a complex type (namely Mail), which does not coincide with the email definition of Mail_IF. Hence, this candidate is considered as incompatible regarding Mail_IF.

Candidate cemailService:
The *Interface Compatibility* analysis of the cemailService candidate involves a similar result to the AtMessaging service. Only one operation (sendAnonymousEmail) matches the three operations from Mail_IF with a *near-exact_12* equivalence. However, the sendAnonymousEmail operation presents a Boolean return type that matches the String *wildcard* type of Mail_IF operations.

For the *Behavioral Compatibility* evaluation, the number of wrappers could be overwhelming, similarly to the AtMessaging service. Again, the decision is to generate only a subset (*W2*) of the whole wrappers' set –from wrapper0 to wrapper23. After running the MujavaMail TS against *W2*, all wrappers failed the tests. The reason is that the Boolean return value (true/false) is casted to String ("true"/"false") as defined in Mail_IF. However, the expected return values (for the *shadow* class) are numeric codes ("1":success/"-1":error). Therefore, executing the TS against any wrappers' subset (*Wi*) of cemailService will always fail due to the return type. Hence, the cemailService candidate is not considered as an eligible service.

Web Service Selection for Mail_IF:
From the previous results the only candidate that successfully passed all evaluation procedures is the AtMessaging service. Hence, the successful wrapper0 of this service (identified in Section 5.1) could be used to allow the MMA application to safely call the selected AtMessaging service.

6.2. MMA - Mail Validation Feature

For each candidate to fulfill the required interface ValidateMail_IF, meaningful details of acceptance/rejection are explained below.

Candidate EmailVer:
When analyzing on *Interface Compatibility*, no matching was found for operations of the required interface ValidateMail_IF and the EmailVer service. Particularly, the candidate provides an operation to validate mail addresses including an additional parameter (licenseKey) that results in a mismatch by the P0 condition (in Table 3). Hence, the EmailVer service can be discarded as a candidate.

Candidate Service: EmailVerification
Similarly to the previous candidate, nomatching was found for the Email-Verification service due to a P0 condition. Hence, this candidate is considered as incompatible regarding ValidateMail_IF.

Candidate ValidateEmail:
When running the *Interface Compatibility* analysis for the ValidateEmail service, two matchings were found for operation isValidMail to operations isValidEmail and isValidEmailAddress, both with a *near-exact_12* equivalence –as shown in Table 5. Therefore, the total compatibility value is 6, from where the *compatibility gap* value can be calculated as: 6/4−1=0.5 according to formula (1).

After this, the ValidateEmail service is suitable to continue with the *Behavioral*

Table 5. *Interface Compatibility between ValidateMail_IF and ValidateEmail*

ValidateMail_IF	ValidateEmail	Degree	Case	Conditions	Value
isValidMail	isValidEmail	n-exact_12	13	R2, N2, P1, E1	6
	isValidEmailAddress	n-exact_12	13	R2, N2, P1, E1	6
Total best value: 4 (based on ValidateMail_IF size)			*Total Equivalence*		6

Compatibility step. According to the procedure in Section 3 a *Behavioral TS* was built, with the authors personal mails as test data, from where 4 test cases were created enclosed in a test driver file named MujavaValidate. Next, the wrappers' set was generated (as explained in Section 5) containing two wrappers, namely wrapper0 and wrapper1.

After running the MujavaValidate TS, the 2 wrappers successfully passed the tests. Therefore, either one of both wrappers could be used as the artifact to integrate the ValidateEmail service into the MMA application. A detailed analysis from the WSDL description of such candidate shows that both operations provide the same behavior, but implemented by different end points (URIs). Hence, this increases the availability of this service, allowing calls to any end point through the corresponding wrapper.

Interestingly, changes between both wrappers into the *Mail Management Application* could be transparently done without affecting its client coordinating component (MMA).

A final remark about performance of the Selection Method. This case study was done on a regular Pentium 1.83GHz 2GB RAM, where the TS was generated in about 1.5 minutes, and the step of Interface Compatibility is done in about 1 minute. The generation of both subsets of 24 wrappers required 2.5 minutes and the execution of the TS against both wrappers' subsets was done in about 25.5 minutes.

7. Related Work

Due to lack of space this section briefly presents related work without a detailed comparison with our approach.

The work in [8] 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/output data registered after testing individual operations for each candidate service. A different TS is built for each service to be evaluated, which is based on a selected input data (either randomly or manually).

Another work based on testing individual or atomic services' operations is presented in [24], which implies a specification-based strategy by means of OWL-S.

The work in [4] is concerned with substitutions of inoperable services with compatible ones. Automatically finding 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.

The work in [25] 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.

Another work [27] 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. The proposed framework verifies a proper service execution through strategies to find faults, and also using a semantic Web Service approach.

For further references other important related work about testing service-oriented systems is summarized in [3,20,1].

8. Conclusions and Future Work

In this paper we have presented an approach to assist developers in the selection of services, when developing a Service-oriented Application. Particularly, our approach addresses two main aspects: confirming the suitability of a candidate service by a dynamic behavioral evaluation (execution behavior), and attending a pragmatic issue related to the required

testing task that inevitably follows any integration process.

Our current work implies test selection for the Behavioral TS, for which we are applying minimization strategies to structure a manageable set of test cases. A straightforward solution is designing a reduced TS based on the result of the Interface Compatibility step. Test cases could be generated only for operations without a single structural matching. This avoids executing the whole TS against the wrappers' set built in the Behavioral Compatibility step.

Another concern implies the scalability upon generation and management of wrappers (as mutation cases). To deal with this, only those wrappers (or even a unique wrapper), with a major probability of success can be generated to substantially reduce the computation effort. In addition, we are exploring Information Retrieval techniques to get more precision in analyzing concepts from interfaces –e.g., semantic equivalence in parameters and operations names.

Finally, we will address the composition of candidate services, which is particularly useful when an atomic candidate service cannot fulfill the whole required functionality.

We will extend the current procedures and models mainly based on business process descriptions and service orchestration [22, 26,6].

Acknowledgments

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

References

1. Bozkurt, M., Harman, M., Hassoun, Y. *Testing and Verification in Service-Oriented Architecture: A Survey*. Software Testing, Verification Reliability, 23(4): 261-313 (2013)
2. Brenner, D., Paech, B., Merdes, M., Malaka, R. *Web Technologies: Concepts, Methodologies, Tools, and Applications, chap. Enhancing the Testability of Web Services*. IGI Global (2010)
3. Canfora, G., Di Penta, M. *Service Oriented Architectures Testing: A Survey*. ISSSE 2006-2008, LNCS(5413): 78–105 (2009), Springer.
4. Church, J., Motro, A. *Learning Service Behavior with Progressive Testing*. IEEE SOCA'11. Irvine, USA (December 2011)
5. Crasso, M., Mateos, C., Zunino, A., Campo, M. *EasySOC: Making Web Service Outsourcing Easier*. Information Sciences, Special Issue: Applications of Computational Intelligence & Machine Learning to Soft. Eng. (2010), in press
6. Daniel, F., Pernici, B. *Insights into Web Service Orchestration and Choreography*. International Journal of E-Business Research, 2(1): 58–77 (2006)
7. Delamaro, M., Maldonado, J., Mathur, A. *Interface Mutation: An Approach for Integration Testing*. IEEE Transactions on Software Engineering, 27(3): 228–247 (March 2001)
8. Ernst, M., Lencevicius, R., Perkins, J. *Detection of Web Service Substitutability and Composability*. International Workshop on Web Services - Modeling and Testing. pp. 123–135. (June 2006)
9. Flores, A., Polo, M. *Testing-based Process for Component Substitutability*. Software Testing, Verification and Reliability, Wiley Online Library, 22(8): 529–561. (December 2012).
10. Freedman, R.S. *Testability of Software Components*. IEEE Transactions on Software Engineering, 17(6): 553–564 (June 1991)
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley (1995)
12. Garriga, M., Flores, A., Cechich, A., Zunino, A. *Practical Assessment Scheme to Service Selection for SOC-based Applications*. SADIO Electronic Journal (EJS) – Special Issue dedicated to ASSE 2011, 11(1): 16–30 (September 2012)
13. Grindal, M. et al. *Combination Testing Strategies: a survey*. Software Testing, Verification and Reliability, 15(3): 167–199 (2005)
14. Gosh, S., Mathur, A.P. *Interface Mutation*. Software Testing, Verification and Reliability, 11(4): 227–247 (2001)
15. 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)
16. Kung-Kiu, L., Zheng, W. *Software Component Models*. IEEE Transactions on Software Engineering, 33(10): 709–724 (October 2007)
17. Massuthe, P., Reisig, W., Schmidt, K. *An Operating Guideline Approach to the SOA*. Mathematics, Computing & Teleinformatics (2005)
18. μJava Home Page: *Mutation system for Java programs* (2008), http://www.cs.gmu.edu/ offutt/mujava/

19. OMG: *Unified Modeling Language: Superstructure version 2.0*. Tech. Rep., Object Management Group, Inc. (2005), http://www.omg.org
20. Palacios, M., Garcia-Fanjul, J., Tuya, J. *Testing in Service Oriented Architectures with Dynamic Binding: A Mapping Study*. Information and Software Technology, Elsevier, 53(3): 171–189 (March 2011)
21. Pasley, J. *Avoid XML Schema wildcards for Web Service Interfaces*. IEEE Internet Computing, 10(3): 72–79 (2006)
22. Peltz, C. *Web Services Orchestration and Choreography*. IEEE Computer, 36(10): 46–52 (2003)
23. Sprott, D., L., W. *Understanding Service-Oriented Architecture*. The Architecture Journal. MSDN Library. Microsoft Corporation 1, 13 (January 2004), http://msdn.microsoft.com/enus/library/aa480021.aspx
24. Tsai,W., Chen, Y., Paul, R. *Specification-based Verification and Validation of Web Services and Service-oriented Operating Systems*. IEEE International Workshop on Objectoriented Real-time Dependable Systems. pp. 139–147. (February 2005)
25. Tsai,W., Zhou, X., Chen, Y., Bai, X. *On Testing and Evaluating Service-Oriented Software*. IEEE Computer, 41(8): 40–46 (2008)
26. Weerawarana, S.; et al. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL,WS-Reliable Messaging, and More*. Prentice Hall PTR (2005)
27. Zhu, H., Yufeng, Z. *Collaborative Testing of Web Services*. IEEE Transactions on Services Computing, 5(1): 116–130 (2010)

Contact Information

Martín Garriga:
GIISCo Research Group, Facultad de Informática, Universidad Nacional del Comahue, Neuquén, Argentina.
CONICET (National Scientific and Technical Research Council), Argentina.
Email: martin.garriga@fi.uncoma.edu.ar

Alan De Renzis:
GIISCo Research Group, Facultad de Informática, Universidad Nacional del Comahue, Neuquén, Argentina.
Email: derenzis.alan@gmail.com

Andres Flores:
GIISCo Research Group, Facultad de Informática, Universidad Nacional del Comahue, Neuquén, Argentina.
CONICET (National Scientific and Technical Research Council), Argentina.
Email: andres.flores@fi.uncoma.edu.ar

Alejandra Cechich:
GIISCo Research Group, Facultad de Informática, Universidad Nacional del Comahue, Neuquén, Argentina.
Email: alejandra.cechich@fi.uncoma.edu.ar

Alejandro Zunino:
ISISTAN Research Institute, UNICEN, Tandil, Argentina.
CONICET (National Scientific and Technical Research Council), Argentina
Email: azunino@isistan.unicen.edu.ar