A Software Tool for Selection and Integrability on Service Oriented Applications[*]

Martín Garriga[1,3], Alan De Renzis[1], Andres Flores[1,3],

Alejandra Cechich[1], Alejandro Zunino[2,3]

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

Abstract. Connecting services to rapidly developing service-oriented applications is a challenging issue. Selection of adequate services implies to face an overwhelming assessment effort, even with a reduced set of candidate services. On previous work we have presented an approach for service selection addressing the assessment of WSDL interfaces and the expected execution behavior of candidate services. In this paper we present a plugin for the Eclipse IDE to support the approach and to assist developers' daily tasks on exploring services integrability. Particularly for behavioral compatibility we make use of two testing frameworks: JUnit and MuClipse to achieve a compliance testing strategy.

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

1. Introduction

Service-oriented applications development implies a business facing solution which consumes services from one or more providers and integrates them into the business process [1,2]. From an architectural perspective developing service-oriented applications involve to reuse existing third-party components or services that are invoked through specialized protocols. Particularly, the industry has adopted the Web Services technology [3], which leads to a concrete decentralization of business processes and a low investment on new technologies and execution platforms. However, the efficient reuse of existing Web Services is still a major challenge. After searching for candidate services, a developer still requires high skills 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. Besides, the set of meaningful properties to explore on candidates also involve the required adaptations for a correct integration allowing client applications to safely consume services while enabling loose coupling for maintainability.

In order to ease the development of service-oriented applications we presented on previous work [4,5] a proposal for *service selection*, which is based on a recent approach [6] that was initially developed to work with software components as a solution for substitutability of component-based systems. The approach was properly adjusted and extended to be applied in the context of service-oriented applications.

The *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.

In this paper we present the architecture of a software tool, which supports the development of service-oriented applications. In particular, we have developed the tool support as a plug-in for the Eclipse IDE[†], to agile developers' daily tasks through an environment that provides a way to integrate different tools to improve developers' productivity and code quality. As Eclipse has been adopted as the most popular IDE nowadays, adding new functionality as a plugin reduces the learning curve of developers. Besides, from an organizational software production perspective, software vendors are keen to benefit from the increased productivity and quality that a good IDE promises to deliver [8].

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

2. Service Selection Method

During the 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 candidate services are discovered a developer still needs to deduce the most appropriate candidate. **Fig. 1** depicts the 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 (*step 1.1*) is based on a comprehensive Assessment Scheme to recognize direct (strong) and potential matchings between a required interface (I_R) and the interface provided by a candidate service (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. If some mismatching is detected, a developer may apply a solution through a semi-automatic procedure (*step*

[†] Integrated Development Environment (IDE)

1.2). The same step can also be used to set up a different matching for some operation of I_R, even when an initial matching had been initially identified [4].

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 (adapters) W is generated (*step 2*), where remote invocations to I_S are solved through a proxy (P_S) derived from service's WSDL description. Thus, a candidate service is evaluated by executing the TS against each $w \in W$ (*step 3*), where at least 70% successful tests must be identified on some wrapper to confirm a behavioral compatibility [5]. Besides, such successful wrapper allows an in-house component to safely call the candidate service once integrated into a client application.
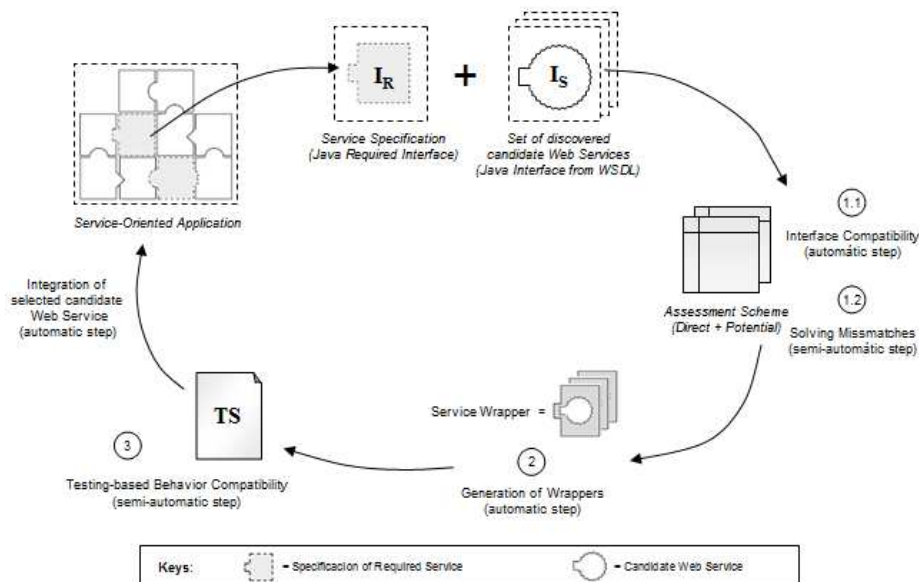

Fig. 1. Service Selection Method

Next section describes the software architecture of the supporting tool. A simple example will be used to illustrate the usefulness of the Selection Method.

2.1. Software Tool Architecture

In order to provide support for the Selection Method we have developed a software tool into the Java language, which has adopted the form of a plug-in for the Eclipse IDE. In this way, developers are provided with an augmented environment, in which building service-oriented applications is now assisted by an automated and guided process easing evaluation and selection of Web services.

Fig. 2 depicts the plug-in's software architecture, in which a central component is the *Testing Meta-Model*: a Java representation of the OMG UML Testing Profile [9] that is used to build the *Behavioral TS*. In addition, both checker components:

Interface Compatibility and *Behavior Compatibility* make use of the meta-model as a way to manage, store and assess the structure of candidate services' specifications (static and dynamic aspects).
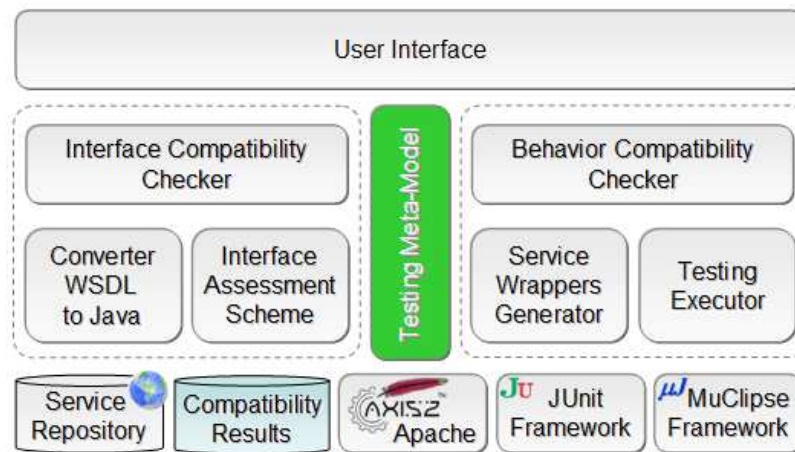
Fig. 2. Software Architecture of Plug-in for Eclipse IDE

The *Interface Compatibility Checker* receives as input a Java required interface (I_R) and a set of WSDL files of candidate services (S) from the (remote) *ServiceRepository*. From those WSDL files a set of Java interfaces (I_S) are derived by the *Converter* component, which is based on the *Apache Axis2* framework. After applying the *Interface Assessment Scheme* their *Results* are properly stored and shown in a *User Interface*'s view. The *Behavior Compatibility Checker* makes use of those *Results* as input for the *Service Wrappers Generator*, where the *Axis2* framework is used to build a service proxy (P_S) to allow safe execution of an evaluated candidate (S). Then the *Testing Executor* component evaluates such candidate by exercising the *Behavioral TS* that could be formatted under the *JUnit* [10] framework or *MuClipse* [11] – an Eclipse plug-in version of the MuJava framework [12] to address Mutational Testing. Final stored *Results* from both checker components are shown into the corresponding *User Interface*'s view to present a selected candidate Web service.

2.2. Example

Let us suppose the development of a Mail Management Application (MMA) being developed under the Java platform. **Fig. 3** depicts the invoking and coordinating component MMA and the interfaces for its required key features: 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. To clearly illustrate the use of the plug-in, the example is reduced to the second required interface (I_R), named Mail_IF, shown in **Fig. 4(a)**, and one candidate Web service: the AtMessaging service, whose interface (I_S) is shown in **Fig. 4 (b)**.

This example is used in the following sections to explain the basis of the two main evaluations: Interface Compatibility and Behavior Compatibility, which also corresponds to the *checker* components in the plug-in software architecture (**Fig. 2**).
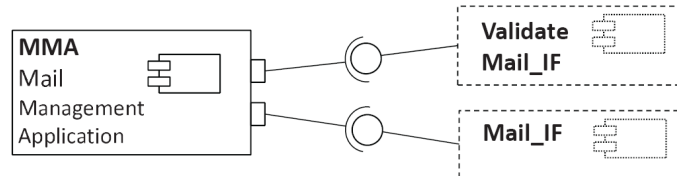

Fig. 3. Structure of Mail Management Application – MMA

«interface» Mail_IF
sendMail(String,String,String,String):String
sendCC(String,String,String,String):String
sendBCC(String,String,String,String):String

«interface» AtMessaging
PDecodeKey(byte[ ] pCodeKey):String
getKey(int,String):byte[ ]
login(byte[ ],String):byte[ ]
logout(byte[ ]):byte[ ]
sendSMTPMail(String,String,String,String):int
sendATMail(int,String,Calendar,String,String,String):int
getUnitMessages(byte[ ],int,Calendar):StUnitEMail[ ]
getUnitMessages2(byte[ ],int,Calendar):StUnitEMail2[ ]

(a) Required Interface (I_R) – Mail_IF   (b) Candidate Web Service (I_S) – AtMessaging
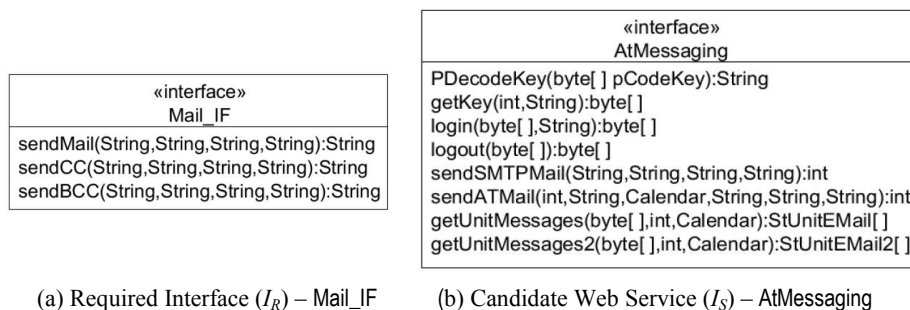
Fig. 4. Mail Management Application – I_R and I_S for Mail sending feature

3.   Interface Compatibility

The Interface Compatibility analysis – presented in a previous work [4] – comprises a practical Assessment Scheme to analyze operations from the interface I_S (of a candidate service S), with respect to the required interface I_R. This step may avoid 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.

The Assessment Scheme is divided in two parts: direct (strong) and potential (weaker) matching cases, which are automatically identified. Weaker matching cases can also be used to solve incompatibilities in a semi-automatic manner. Both parts consists of four compatibility levels (*exact*, *near-exact*, *soft*, *near-soft*) to classify matching cases, defined as syntactic constraints, applied on a pair of corresponding operations. Constraints are based on conditions for elements of an operation's signature (return, name, parameter, exception).

The outcome of this step is an *Interface Matching* list that characterizes each correspondence according to the four levels of the Assessment Scheme. 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:

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

The *Interface Matching* list is also used to calculate a structural dissimilarity value, named *Compatibility Gap*, based on specific equivalence values assigned to different syntactic constraints of the Assessment Scheme – e.g., the value of *exact* equivalence is 4. The *Compatibility Gap* between I_R and I_S can be calculated by taking the highest compatibility level for each operation $op_R \in I_R$ – the formula can be seen in [4]. This value also gives evidence of the expected adaptation effort for the candidate service integrability.
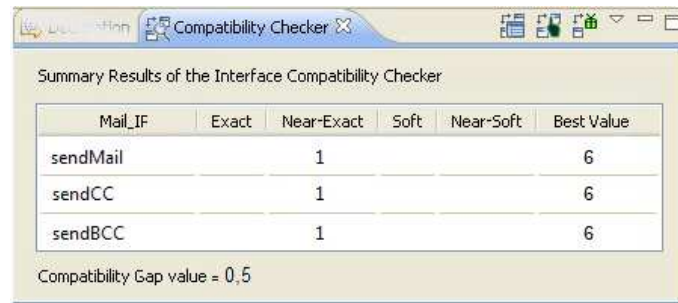
3.1. Interface Compatibility for the Mail feature

Fig. 5 shows the initial user interface of the plug-in, a view of the Interface Compatibility checker, to analyze the required interface Mail_IF and the candidate AtMessaging service (through its WSDL document). A summary result is shown in **Fig. 6**, where all operations from Mail_IF – i.e., sendMail, sendBcc and sendCc – obtained one *near-exact* match with an *equivalence value* of *6*. From the total equivalence value (18) and the best possible value (12) the *compatibility gap* between Mail_IF and the AtMessaging service can be calculated as: *18/12−1 = 0.5*.
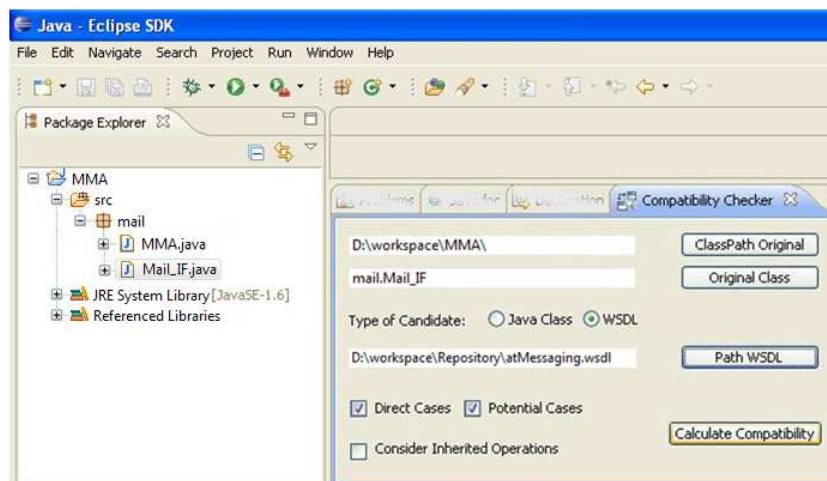
Fig. 5. Interface Compatibility Checker – Mail_IF and AtMessaging service

When asking for detailed results, **Fig. 7** shows that all operations from Mail_IF match the same operation sendSMTPMail of the candidate AtMessaging service, with a *near-exact_12* equivalence. Operations coincide on the parameters list (P1) and neither of them have exceptions (E1). For the return type, the String type is considered as a *wildcard* type allowing equivalence or subtyping (R2) with the int type. They also have substring equivalence on operations names (N2) – terms 'send' and 'mail'. In fact, the last two correspondences could be quite reasonable considering that after sending the main email copy, additional copies (Cc/Bcc) could also be iteratively sent with a similar procedure afterwards.
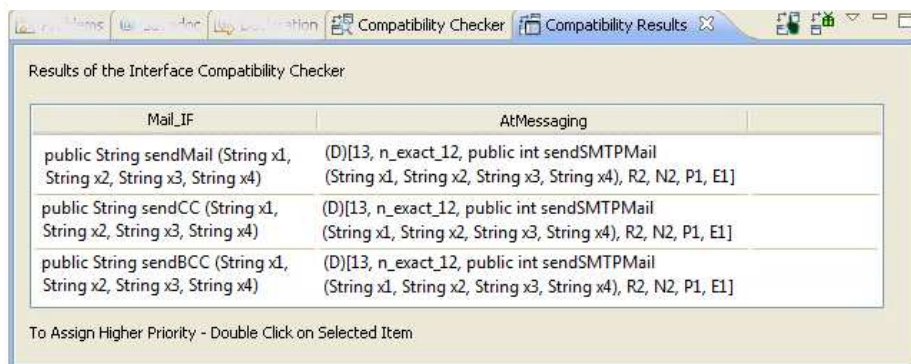
Fig. 6. Interface Compatibility Checker – Result summary for Mail_IF and AtMessaging service
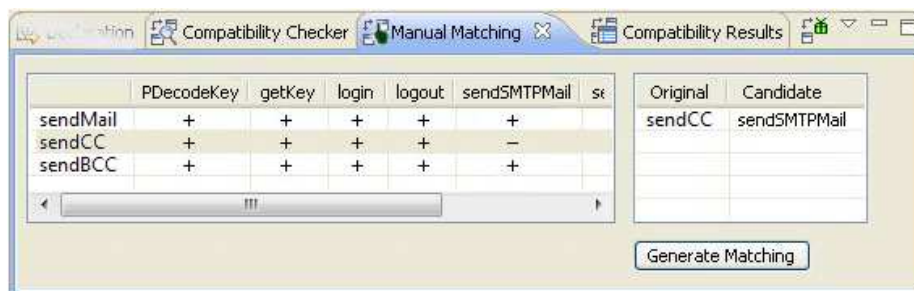
Fig. 7. Interface Compatibility Checker – Detailed Results for Mail_IF and AtMessaging service

If a mismatch is found for some operation, a developer can solve it by the semi-automatic facility. **Fig. 8** shows the view of manual matching with a hypothetical case where a developer is setting up a specific operation-pair correspondence.

Fig. 8. Interface Compatibility Checker – Manual Matching example
for Mail_IF and AtMessaging service

After that, the *Interface Matching* list for Mail_IF and the candidate AtMessaging service is available – i.e., the table in **Fig. 7**. 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 these step in which a required service's functionality is represented as a particular Test Suite.

5. Behavior Compatibility

To carry out the Behavior Compatibility evaluation – presented in [5] – 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 [13,14], 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 the set of correspondences ($op_S \in I_S$) is added for a different operation $op_R \in I_R$. When a operations' pair contains various parameters of the same or equivalent type, also combination of arguments is needed. Each combination arising from different parameters matching should be added into the Wrapper Generation Tree, in the form of a new branch.

Considering the case study, the operation sendMail implies a likely case in which its String parameters exactly match (P1) with parameters from operation sendSMTPMail, supposing that parameters lists are defined in the same order.

If it is not the case, 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. Considering the parameters list with 4 String parameters, the number of permutations rises to 24 for each level of the tree, making the whole number of wrappers to be 24*24*24 = 13824. Although this wrappers' set becomes unwieldy to be tested, the partial generation option of the tool (shown in **Fig. 9**) can be used in order to build a manageable wrappers' set.

Fig. 9. Wrappers set generation options

Therefore, initially a sub-set (W_1) of 24 wrappers was generated as a result of this step – from wrapper0 to wrapper23, corresponding to the left branch in the wrapper generation tree shown in **Fig. 10**.
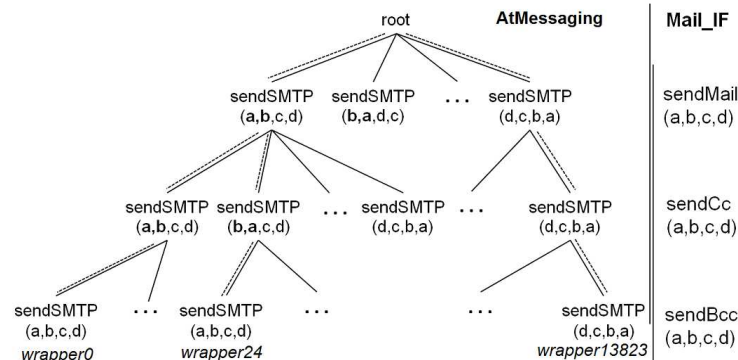
Fig. 10. Wrapper Generation Tree for Mail_IF and AtMessaging
(considering parameter combinations)

Test Suite Generation

To build a Behavioral TS for Mail_IF, a concrete class implementing this interface must be initially created to describe the required behavior in the form of expected results for some representative test data. This shadow class is called Mail and simply resembles an expected behavior according to certain input/output data for each operation within the Mail_IF interface. For example, the operation sendMail receives as input four String parameters (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.

Relevant sequences of operations' invocation are described as *test templates*, which are combined with the test data to generate the Behavioral Test Suite (TS). The TS has been generated inside a test driver file (TestMail) in the specific JUnit format [10]. **Fig. 11** shows the test method testTS_5_1, which exercises the following sequence: sendMail, sendCC, and sendBcc.

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.

```
public void testTS 5 1() {
    Mail obtained=null;
    obtained = new Mail();
    java.lang.String arg1=(java.lang.String)
    "martin.garriga@fai.uncoma.edu.ar";
    java.lang.String arg2=
    (java.lang.String)"andres.flores@fai.uncoma.edu.ar";
    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)"alejandra.cechich@fai.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);
    assertTrue(result0 == result1 == result2 == "success"
}
```

**Fig. 11.** JUnit Test Case for Mail.

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. 12** depicts the class structure for the Mail_IF feature. Also, **Fig. 13** shows the source code of one generated wrapper (wrapper0). This wrapper complies with the structure depicted in **Fig. 12** in which a proxy (P_S) has been generated for invoking operations of service AtMessaging.

The TS TestMail instantiates and invokes the Mail class, which represents not only the shadow class for the required interface Mail_IF, 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.

After setting the specific class structure, the TestMail test file can be run against the generated subset (W_I) of wrappers. The tool support makes use of the *MuClipse* framework [11] to execute the Test, as shown in **Fig. 14**.

In this case, only wrapper0 successfully passed the tests, which confirms the expected behavior specified for the required interface Mail_IF. A detailed description of the service wrappers evaluation procedure can be found in [5].
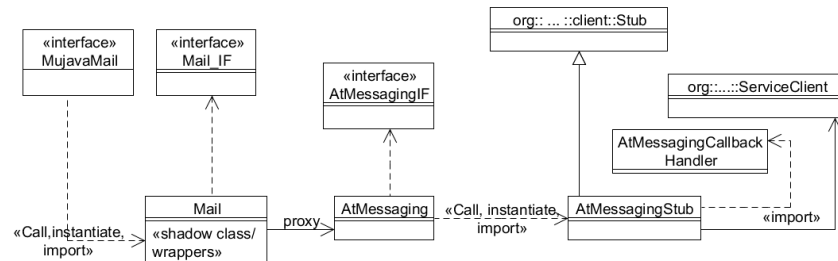
Fig. 12. TestSuite for Mail_IF to evaluate Wrappers through the Proxy

```
public class Mail implements Mail_IF {
private AtMessaging proxy = null;
public Mail() { proxy = new AtMessaging();
} // ---
public java.lang.String sendMail(String x0, String x1,
   String x2, String x3){
      java.lang.String result0 = "";
      try {   result0 = proxy.sendSMTPMail(x0, x1, x2, x3);
      } catch (Exception e) {
      e.printStackTrace();
      throw new RuntimeException(e); }
      return result0;
} // ---
public java.lang.String sendCC(String x0, String x1,
 String x2, String x3){
      java.lang.String result0 = "";
      try {   result0 = proxy.sendSMTPMail(x0, x1, x2, x3);
      } catch (Exception e) {
      e.printStackTrace();
      throw new RuntimeException(e); }
      return result0;
} // ---
public java.lang.String sendBCC(String x0, String x1,
  String x2, String x3){
      java.lang.String result0 = "";
      try { result0 = proxy.sendSMTPMail(x0, x1, x2, x3);
      } catch (Exception e) {
      e.printStackTrace();
      throw new RuntimeException(e); }
      return result0;} // ---
}
```

Fig. 13. Mail wrapper for the AtMessaging service.

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 [15].

Results of the underlying behavior compatibility evaluation against a case study can be seen in [16]. Also, an initial insight into the procedure performance is provided, although it has been improved through fine tuning the test suite and wrappers set generation.
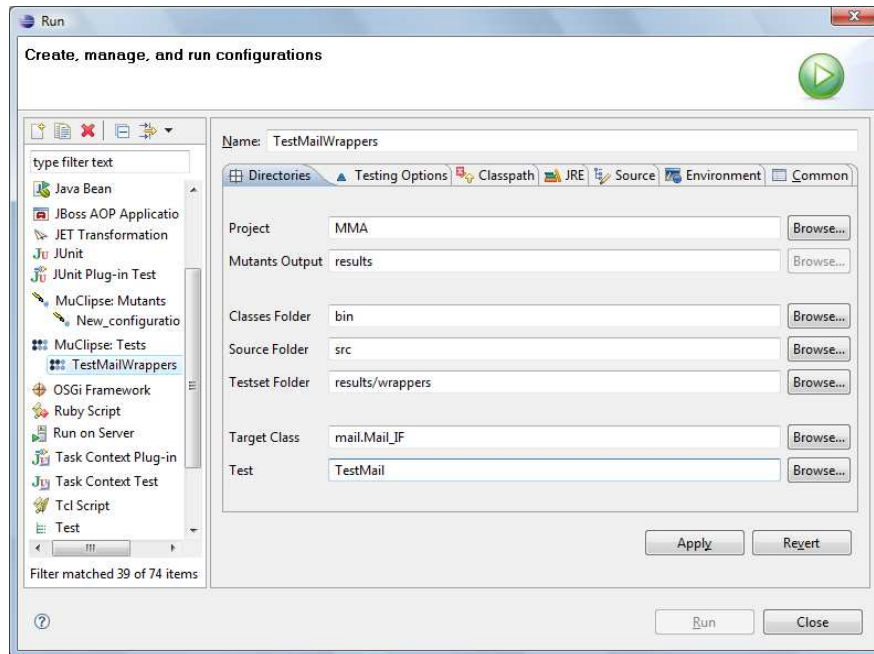
Fig. 14. *MuClipse* test configuration for running the TS

6. Related Work

Several approaches have been proposed towards IDE support for the development of service-oriented applications.

From the Web service composition point of view, in [17] the development of an end-to-end service composition IDE is presented. The tooling support mainly encompasses requirement analysis and design of an IDE, mechanisms for leveraging and integrating existing tools and technologies into the IDE and the development of the IDE as a set of Eclipse plugins. As service selection is a key part of the service composition problem, our plugin can be seen as one of the technologies which could be integrated into such an IDE.

Another Eclipse plug-in for formal verification of Web service composition is presented in [18]. Such plug-in was originally a part of the LTSA tool suite which provides model-checking tools. It was conveniently extended to support the development of (composite) service-oriented applications, featuring UML Message Sequence Charts for scenario modeling, which are then compiled in Finite State Process algebra to formalize behavior. The plugin is architected following the model-view-controller pattern. In contrast, our plugin is intended as a ready-to-use tool to support developers' daily tasks of service selection, without introducing extra complexity.

From the service discovery point of view, the EasySOC project [19] presents a catalog of guidelines to build service-oriented applications and services. This catalog synthesizes best SOC development practices. The corresponding EasySOC plug-in for the Eclipse IDE has been implemented to simplify the utilization of the guidelines for service publication, discovery and consumption. Plug-in provision encourages readily adoption of proposed ideas in the software industry. Recent works have demonstrated the suitability of integrating EasySOC with our proposal featuring a comprehensive process for building Service-oriented Applications [4].

Another approach with a similar scope to ours is presented in [20]. Studying the similarity between (WSDL) Web service descriptions appears as a key solution for service interoperability. The similarity measurement process encompasses calculating similarity between service names, operations, input/output messages, parameters, and documentation, mainly gathered from XML Schema Documents (XSD) associated with services. Those calculations are supported through a stand-alone Java-based tool called WSSim which parses WSDL documents, calculates similarities and returns a final score – equivalent to our Compatibility Gap value. However, an initial comparison of complex type similarity can be performed without dealing with the complexity of a XML schema. In our approach, the analysis of WSDL documents allows addressing complex data types while aiming a lightweight proposal, and reducing the learning curve and the adoption effort of the tool.

With regard to the behavioral and syntactic assessment of services, the following works are also related to our goals.

The approach in [21] evaluates 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. For this, a different TS is built for each operation in each service under evaluation, which is based on selected input data (either randomly or manually). 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. The behavioral evaluation is only done after passing the syntactic Interface Compatibility analysis, which reduces computation for the testing phase.

The work in [22] is also concerned with substitutions of inoperable services with compatible ones. Automatically finding optimal solutions implies the challenging issue of discerning the behavior of services. The approach attempts to discover and comprehend services' behavior and classify them into clusters by means of compliance testing. However, the approach has a very low confidence on any service description, also ignoring WSDL specifications or the derived Java interfaces.

The work in [23] addresses 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. Some of the ideas proposed in this paper are being implemented to prune the wrappers generation tree and minimizing the TS.

Another work [24] 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, though managing the T-services' set introduces an inconvenient overhead. The proposed framework is particularly intended to verify a proper service execution through strategies to find faults, using a semantic Web Service approach. As semantic information of Web Services such as ontologies is rarely available [25], our plugin relies on syntactic and structural definitions of Web Services available in WSDL specifications.

To support programmatic service discovery, in [26] the authors have developed a suite of methods to assess the similarity between two WSDL specifications based on the structure of their data types and operations, and the semantics of their natural language descriptions and identifiers, given only a (potentially partial) description of the desired service.

7. Conclusions and Future Work

In this paper we have presented an Eclipse IDE plug-in as a support of a Selection Method to assess a candidate Web service for its likely integration into a SOC-based application under development. This method provides a practical Interface Compatibility analysis and a Behavioral Compatibility evaluation.

The availability of IDE-integrated tools that aids development processes will help software practitioners to rapidly adopt best practices, particularly for building real service-oriented applications.

The architecture of our plugin is based on integration with well-known frameworks such as Apache Axis2 for service handling, and *JUnit* and *MuClipse* for behavioral evaluation. Besides, a Testing Meta-Model crosses over the entire architecture as a way to manage, store and assess the structure of candidate services' specifications (static and dynamic aspects).

Our current work is focused on integrating in the plug-in some Information Retrieval techniques to better analyzing concepts from interfaces. This will be complemented with a semantic-basis – particularly, through the WordNet lexical dictionary [27] and its Java API JWI‡.

Besides, performance evaluation of the plugin is mandatory. We are carrying out experiments using different data-sets previously used by numerous authors to validate service-oriented proposals [28,29], and including real-word Web Services. Initial results show an improvement of the service selection procedure in terms of classic IR metrics such as *recall* and *precision* independently from the underlying service discovery registry.

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, models and tools mainly focusing service orchestration [30,31,32].

---

‡ The MIT Java Wordnet Interface – http://projects.csail.mit.edu/jwi/

Acknowledgments

References

1. Sprott D, L W. Understanding Service-Oriented Architecture. The Architecture Journal. MSDN Library. Microsoft Corporation; 1(13). January. http://msdn.microsoft.com/en-us/library/aa480021.aspx. (2004)
2. 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).
3. Bichler, M., Lin, K.: Service-oriented computing. Computer, 39(3), 99-101 (2006)
4. Garriga, M., Flores, A., Cechich, A., Zunino, A.: Practical Assessment Scheme to Service Selection for SOC-based Applications. SADIO Electronic Journal (EJS), vol. 11, no. 1. Special Issue dedicated to ASSE'11. pp. 16-30 (2012)
5. Garriga, M., Flores, A., Cechich, A., Zunino, A.: Behavior Assessment based Selection Method for Service Oriented Applications Integrability. 13th ASSE'12, 41 JAIIO, SADIO. pp. 339-353. Córdoba, Argentina. 27-31 August. (2012)
6. Flores, A., Polo, M.: Testing-based Process for Component Substitutability. Journal STVR, vol. 22. no. 8. pp. 529–561, Wiley. DOI: http://dx.doi.org/10.1002/stvr.438 (2012)
7. The Eclipse Foundation. Eclipse.org Home Page. http://www.eclipse.org/ (2013)
8. Yap, N.; Chiong, H.C.; Grundy, John; Berrigan, R.: Supporting dynamic software tool integration via Web service-based components. *Proceedings of the Australian Software Engineering Conference.* pp.160,169, 29 March- 1 April. (2005).
9. OMG. UML Testing Profile version 1.0. Technical Report formal/05-07-07, OMG, July. (2005).
10. JUnit Home Page. JUnit.org Resources for Test Driven Development. http://www.junit.org/home (2013)
11. MuClipse Home Page. Eclipse Plug-in for the MuJava mutation engine. http://muclipse.sourceforge.net (2013)
12. µJava Home Page: Mutation system for Java programs. http://cs.gmu.edu/~offutt/mujava/ (2011)
13. Gosh, S., Mathur, A. P.: Interface Mutation. Software Testing, Verification and Reliability, 11:227–247. (2001)
14. Delamaro, M, Maldonado, J., Mathur, A.: Interface Mutation: An Approach for Integration Testing. IEEE Transactions on Software Engineering, 27(3):228–247. (2001)
15. Kung-Kiu, L., Zheng, W.: Software Component Models. IEEE Transactions on Software Engineering, 33(10), 709-724 (2007).
16. Garriga, M.; Flores, A.; Cechich, A.; Zunino, A., Testing-Based Process for Service-Oriented Applications, IEEE 30th International Conference of the Chilean Computer Science Society (SCCC), pp.64,73, 9-11 Nov. (2011)
17. Chafle, G.; Das, G.; Dasgupta, K.; Kumar, A.; Mittal, S.; Mukherjea, S.; Srivastava, B.: An Integrated Development Environment for Web Service Composition. IEEE International Conference on Web Services ICWS, pp.839-847, 9-13 July (2007)
18. Foster, H., Uchitel, S., Magee, J., & Kramer, J. LTSA-WS: a tool for model-based verification of web service compositions and choreography. Proceedings of the 28th international conference on Software engineering (pp. 771-774). ACM. (2006)

19. Rodriguez, J. M., Crasso, M., Mateos, C., Zunino, A., & Campo, M. The EasySOC project: a rich catalog of best practices for developing web service applications. CLEI Electronic Journal, 14(3). (2011)
20. Tibermacine, O., Tibermacine, C., & Cherif, F. WSSim: a Tool for the Measurement of Web Service Interface Similarity. To appear in Proceedings of the 25th International Conference on Software Engineering and Knowledge Engineering. (2013)
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. pp. 123–135. Palermo, Italy. (2006)
22. Church, J., Motro, A.: Learning Service Behavior with Progressive Testing. In: IEEE SOCA'11. Irvine, USA. (2011)
23. Tsai, W., Zhou, X., Chen, Y., Bai, X.: On Testing and Evaluating Service-Oriented Software. IEEE Computer 41(8), 40–46 (2008)
24. Zhu, H., Yufeng, Z.: Collaborative Testing of Web Services. IEEE Transactions on Services Computing 5(1), 116–130 (2010)
25. Brogi, A. On the potential advantages of exploiting behavioral information for contract-based service discovery and composition. The Journal of Logic and Algebraic Programming, 80(1):3–12, (2011)
26. Stroulia, E. & Wang, Y. Structural and Semantic Matching for Assessing Web-Services Similarity. *International Journal of Cooperative Information Systems, 14,* 407-437. (2005)
27. WordNet: A lexical database for English. Princeton University, NJ, USA. http://wordnet.princeton.edu/ (2012)
28. Heß, A., Johnston, E., & Kushmerick, N., Assam: A tool for semi-automatically annotating semantic Web Services. The Semantic Web Conference ISWC, pp. 320-334. Springer Berlin Heidelberg. (2004)
29. Mateos, C., Crasso, M., Zunino, A. & Ordiales, J. L., Detecting WSDL bad practices in code–first Web Services. International Journal of Web and Grid Services, 7(4):357–387, (2011)
30. 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).
31. Daniel, F., Pernici, B.: Insights into Web Service Orchestration and Choreography. International Journal of E-Business Research 2(1), 58–77 (2006)
32. Peltz, C. Web Services Orchestration and Choreography. *IEEE Computer, 36(10),* 46-52. (2003)