Adaptability-based Service Behavioral Assessment

Diego Anabalón[1,3], Martin 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, Neuquen, Argentina.
[diego.anabalon, martin.garriga, andres.flores,
alejandra.cechich]@fi.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. 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 testing and adaptation. This is done by using black-box testing criteria to explore services behavior. 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 candidate Web Service.

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 [13]. Although developers do not need to know the underlying model and rules of a third-party service, its proper reuse still implies quite a big effort. Yet searching for candidate services is mainly a manual exploration of Web catalogs usually showing poorly relevant information [12]. 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 interfaces must be assessed, but also their operational behavior as key feature of a service contract. Besides, correct adaptations must be identified so client applications may safely consume services while enabling loose coupling for maintainability.

To ease the development of Service-oriented Applications we presented in previous work [3,6] a proposal to assist developers on service selection by means of testing and adaptation. This approach complements the conventional compatibility assessment by using black-box testing criteria to explore services behavior. The aim is to fulfill the *observability* testing metric [8,1] that observes a service operational behavior by analyzing its functional mapping of data transformations (input/output). In addition, a helpful information takes shape concerning the adaptation logic to integrate a service

into a client application. Hence, a wrapping algorithm was defined based on *mutation testing* [4,9], to identify the right adapter configuration. However, mutation testing carries a high effort (cost) both on generation and execution.

In this work, we improve the wrapping algorithm based on a set of adaptability factors recently defined [3]. In this way, we were able both to be more accurate on setting the best adapter and to highly reduce the involved costs on mutation testing. A concise case study shows the potential of improvements implemented into our approach.

The rest of the paper is organized as follows. Section 1 presents an overview of the *Selection Method*. Section 3 explains the steps to build a *Behavioral TS*. Section 4 briefly describes the *Interface Compatibility* procedure. Section 5 describes the *Behavioral Compatibility* procedure. Section 6 presents related work. Conclusions and future work are presented afterwards.

2 Service Selection Method

During development of Service-Oriented Applications, specific parts of the system may be implemented in the form of in-house components. Besides, some of the comprising software pieces could be fulfilled by the connection to Web Services. A set of candidate services could be obtained by making use of any service discovery registry. Even with a wieldy candidates' list, a developer must be skillful enough to determine the most appropriate service for the consumer application. Figure 1 shows our proposal to assist developers in the selection of Web Services, which is briefly described as follows:

As an initial step, a simple specification is needed, in the form of a required interface I_R, as input for the three comprising procedures.

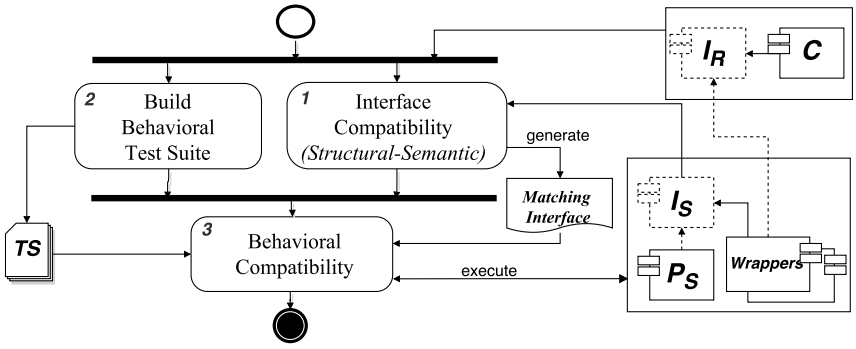


Fig. 1: Service Selection Method

The *Interface Compatibility* procedure (step 1) matches the required interface (I_R) and the interface (I_S) provided by a candidate service S. A structural-semantic analysis is performed to characterize operation signatures (return, name, parameters, exceptions) at four compatibility levels: *exact*, *near-exact*, *soft*, *near-soft*. This analysis also considers adaptability factors to reduce the integration effort. The outcome of this step is an *Interface Matching* list where each operation from I_R may have a matching with one or

more operations from I_S [3]. Particularly, operations from I_R with multiple matchings are considered as "*conflictive operations*" in this approach – i.e., they must be disambiguated yet.

When a functional requirement (I_R) from an application can be fulfilled by a potential candidate Web Service, a *Behavioral Test Suite* (TS) is built (step 2) [6]. This TS describes the required messages interchange from/to a third-party service, upon a selected testing coverage criteria [8,1], to fulfill the *observability* testing metric.

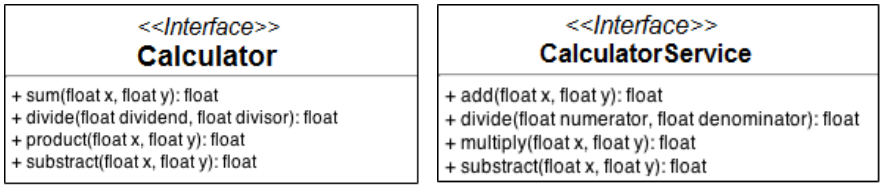
The *Behavioral Compatibility* procedure (step 3) evaluates the required behavior of candidate Web Services by executing the *Behavioral TS*. For this the *Interface Matching* list is processed to generate a set of wrappers W (adapters) – based on the identified *conflictive operations* – allowing to run the TS against the candidate service S.

After exercising the TS against each wrapper $w \in W$, at least one wrapper must successfully pass most of the tests to confirm both the proper matching of *conflictive operations* and the behavioral compatibility of the candidate service S. Besides, such successful wrapper allows an in-house component to safely call service S once integrated into the client application.

Next sections provide detailed information particularly related to the aforementioned procedures. A simple example will be used to illustrate the usefulness of the Selection Method.

2.1 Proof-of-Concept

To illustrate our proposal, we assume a simple example of a calculator application, with the four basic arithmetic operations. Figure 2a shows the required interface (I_R) called Calculator and Figure 2b shows the interface (I_S) of a candidate Web Service named CalculatorService.



(a) Required Interface Calculator (b) Service Interface CalculatorService

Fig. 2: Case Study of Calculator service

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 [6]. 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 [8,1], where synchronous events (e.g., invocations to operations) and asynchronous (e.g., exceptions) may have sequential dependencies on each other, causing distinct behaviors according to the order in which operations and exceptions are called. The criterion requires traversing each operational sequence at least once. In our approach, this is called "interaction protocol" [2], formalized by using *regular expressions*, which allows to automatize test case generation. The alphabet for regular expressions comprise the signature of service operations.

In addition, an imperative specification must be built to describe the expected behavior of the interface I_R, with a set of representative test data. This is called *shadow class* and takes the same name as I_R. Hence, each test case uses these test data as input for parameters on each call to operations of the I_R's interface. This means a black box relationship or input/output functional mapping.

3.1 TS for Calculator

For the interface (I_R) Calculator, a shadow class was defined using the values 0 and 1 as *test data* to the four arithmetic operations. Then, the *interaction protocol* (in the form of a regular expression) is defined as follows:

Calculator (sum | subtract | product | divide)

This regular expression implies operational sequences limited to an only operation to be invoked, since Calculator is a *stateless* service without dependencies between operations. A set of *test templates* is generated from the regular expression, representing each operational sequence. In this case, 4 test templates are derived, each one composed of the constructor operation and one arithmetic operation.

Then, the selected test data is combined with the 4 test templates to generate a TS in a specific format: based on the MuJava framework [10]. From this combination, 8 test cases were generated in the form of methods into a test file called MujavaCalculator. Code Listing 1.1 shows the test case testTS_3_1, which invokes the sum operation.

Listing 1.1: MuJava test case for Calculator

```
public String testTS_3_1() {
        calc.calculator obtained=null;
        obtained = new calc.calculator();
        float arg1 = (float) 0;
        float arg2 = (float) 1;
        float result0 = obtained.sum(arg1,arg2);
        return result0.toString();
}
```

4 Interface Compatibility

In the *Interface Compatibility* procedure is determined the level of compatibility between the operations of the interface I_R and the operations of the interface I_S of a

candidate service S [3]. A structural-semantic analysis is performed to operation signatures. Structural aspects consider signatures and data types, while semantic aspects consider identifiers and terms in the names of operations and parameters. Information Retrieval (IR) techniques and the WordNet[4] dictionary are used for semantic aspects. A scheme of constraints allows to characterize pairs of operations ($op_R \in I_R$, $op_S \in I_S$) in four compatibility levels: *exact*, *near-exact*, *soft* and *near-soft*. Such constraints describe similarity cases based on adaptability (structural and/or semantic) conditions for each element of an operation signature (return, name, parameters, exceptions). As a result an *Interface Matching* list is generated, where each operation $op_R \in I_R$ may have a match to one or more operations $op_S \in I_S$, with likely one o more matchings in the parameters list.

In some cases, certain required operations ($op_R \in I_R$) could obtain multiple matchings (with the same compatibility) – at level of operations and/or parameters – to the candidate service interface (I_S). At *operation level*: an op_R has matching to several op_S. At *parameters level*: an op_R has several matchings in the parameters list – i.e., a set of all possible permutations of arguments. These operations need a disambiguation and they are called *"conflicting operations"* in this approach.

For non-conflictive operations it is possible to assume a high reliability in the operation matching – i.e., they may confirm their compatibility through the *Behavioral Compatibility* procedure.

4.1 Calculator-CalculatorService Interface Matching

Table 1 shows the interface matching result for Calculator and CalculatorService. Operations sum and product of Calculator are identified as *conflictive operations* at operation level. They obtained three matchings with operations add, subtract and multiply of CalculatorService, with the same level of compatibility *near-soft* (*n_soft_55*). Operations subtract and divide of Calculator are non-conflictive operations. They obtained a unique correspondence of higher compatibility level to their homonyms from CalculatorService – i.e., *exact* match for subtract operation and *near-exact* (*n_exact_3*) match for divide operation.

Moreover, all operations obtained a unique matching at parameters level. Parameters $(float\,x, float\,y)$ of operations sum, subtract and product of Calculator are identical (in name and type) to their counterparts of CalculatorService. For divide operation of Calculator, its parameters have identical types and equivalent (synonyms) names – *dividend* with *numerator* and *divisor* with *denominator* – with the operation of CalculatorService.

5 Behavior Compatibility

To carry out the *Behavior Compatibility* evaluation for a candidate service S, a set of *wrappers* (adapters) W needs to be built to allow executing the *Behavioral TS* and compare their results with those specified in the interface I_R. The wrappers set is generated

[4] https://wordnet.princeton.edu/

Table 1: Interface Compatibility for Calculator-CalculatorService

Calculator	**CalculatorService**		
float subtract (float x, float y)	[1, exact, float subtract (float x, float y)] {(x:float-x:float),(y:float-y:float)}	[109, n_soft_55, float add (float x, float y)]	[109, n_soft_55, float multiply (float x, float y)]
float sum (float x, float y)	[109, n_soft_55, float add (float x, float y)] {(x:float-x:float),(y:float-y:float)}	[109, n_soft_55, float subtract (float x, float y)] {(x:float-x:float), (y:float-y:float)}	[109, n_soft_55, float multiply (float x, float y)] {(x:float-x:float), (y:float-y:float)}
float divide (float dividend, float divisor)	[4, n_exact_3, float divide (float numerator, float denominator)] {(dividend:float-numerator:float), (divisor:float-denominator:float)}	[116, n_soft_62, float add (float x, float y)]	[116, n_soft_62, float subtract (float x, float y)]
float product (float x, float y)	[109, n_soft_55, float add (float x, float y)] {(x:float-x:float),(y:float-y:float)}	[109, n_soft_55, float subtract (float x, float y)] {(x:float-x:float), (y:float-y:float)}	[109, n_soft_55, float multiply (float x, float y)] {(x:float-x:float), (y:float-y:float)}

by processing the *Interface Matching* list, according to the multiple correspondences from the *conflictive operations* identified – both at operation and parameters levels. Hence, those multiples correspondences could be disambiguated so to identify proper univocal correspondences.

Wrappers generation can be seen as applying the Interface Mutation technique [4,9], by using a mutation operator to change invocations to operations and to change arguments in the parameters list. Thus, each wrapper is considered a faulty version (or mutant) regarding the wrapper that contains the proper matchings of operations and parameters.

Previously [6], our approach was only based on structural aspects (signatures and data types) to generate wrappers, producing a larger set of wrappers W. This is because usually a larger number of *conflictive operations* were identified – both at operation and parameters levels.

A major improvement in this work involves to consider the semantic aspects provided by the *Interface Matching* list, in which a less number of *conflictive operations* is identified, effectively reducing the W set.

5.1 Wrappers Generation

A tree structure is built to generate wrappers, where each path from the root to a leaf node represents a specific matching between operations of I_R and I_S (i.e., a wrapper to be generated). Thus, the number of leaf nodes determines the size of the wrappers set W. Each *conflictive operation* produces several branches on the tree. On the contrary, a non-conflictive operation (implying a univocal match) does not involve additional branches in the tree.

In the case of a *conflictive operation* at operation level, a new branch is added for each matching to a service operation. At parameters level, a new branch is added for each arguments matching from the set of permutations – even though there could be a univocal operation matching.

Particularly, in this work was updated the algorithm that implements the mutation operator to change arguments into the wrappers generation. Thereby, the new algorithm to treat parameters matchings considers the following cases:

1. *Without any matching*: if any matching was identified at all (structural and/or semantic), parameters will be permuted between each other, producing branches for each arguments combination.

2. *Only structural matching*: If a semantic matching was not identified, parameters are related only through the structural information (data types). If multiple matchings were identified, for each of them a branch is produced. For the remaining parameters the case 1 is applied.

3. *Structural-semantic matching*: Parameters are related through the structural and semantic information. If multiple matchings were identified, for each of them a branch is produced. For the remaining parameters the case 2 is applied.

4. *Service's extra parameters*: If a service operation contains more parameters than the required operation, then some parameters are left outside of the matchings. For them, a test value is required when invoking the service operation. Hence, in this approach a *default* value is assigned according to each parameter data type – "" (quotes) for strings, ' ' (space character) for characters, *true* value for booleans, and 0 (zero) for numerical types.

5.2 Wrappers for Calculator-CalculatorService

Figure 3 shows the wrapper generation tree for Calculator and CalculatorService. Branches were only produced at operation level according to the *conflictive operations* identified: sum y product of Calculator with respect to add, subtract and multiply of CalculatorService. Regarding to parameters matching, the case 3 was applied since a structural-semantic matching was identified for all parameters.

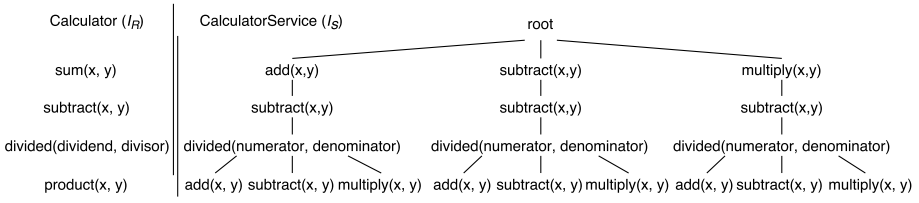


Fig. 3: Wrapper generation tree to Calculator Service

The total number of wrappers (size of W) to be generated is 9, which is the number of leaves on the tree. Notice that without considering semantic aspects, particularly for parameters, a major number of permutations there had been generated. Since all parameters are of the same type, multiple structural matchings there had been identified, making the size of W scaling to 144 wrappers.

Listing Code 1.2 and 1.3 show a fragment of the code from *wrapper2* and *wrapper3* respectively. Where *wrapper2* represents both the tree path down-to the third leaf node and the most appropriate matchings. Likewise, *wrapper3* represents the path down-to the fourth leaf node – being a faulty (mutant) version.

Listing 1.2: Wrapper2 for Calculate-CalculateService

```
public class Calculator{
        protected katze. ... .CalculatorService proxy = null;
        public Calculator(){
                this.proxy = new katze. .. .CalculatorService();
        }
        public float sum (float arg1, float arg2){
                float ret0;
                try{ ret0= candidate.add(arg1,arg2);
                }catch (exception ex){
                        ex.printStackTrace();
                        throw new RuntimeException(ex);
                }
                return ret0;
        }
//...
        public float product (float arg1, float arg2) {
                float ret0;
                try{ret0 = candidate.multiply(arg1, arg2);
                }catch (exception ex){
                        ex.printStackTrace();
                        throw new RuntimeException(ex);
                }
        }
                return ret0;
}
```

5.3 Wrappers Evaluation

Once generated the set of wrappers W, the *Behavior TS* is executed against each wrapper $w \in W$ to assess the behavior of the candidate service S. Using our tool based on the MuJava framework, the TS is exercised against the I_R and iterating over the list of wrappers. After that, results are compared to determine for each wrapper the number of test cases that failed – which produced a result different from the one expected. A wrapper may survive (as mutation case) when most of the test cases are successful. A successful wrapper allows to disambiguate the *conflictive operations,* confirming the right matchings both at operation and parameters levels. In addition, this wrapper may be used as integration artifact allowing a safe communication to the candidate service S.

5.4 Behavioral Evaluation for Calculator-CalculatorService

The TS called *MujavaCalculator* was executed against Calculator (I_R) and the 9 wrappers generated for CalculatorService. Table 2 shows the execution results, where *wrapper2* passed successfully 100% allowing to confirm the behavioral compatibility of CalculatorService. In addition, this wrapper contains the right matchings of operations (sum-add, subtract-subtract, divide-divide, product-multiply). Finally, *wrapper2* can be used as an adapter for the safe integration of CalculatorService in the client application.

6 Related work

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

In [7] we survey current approaches on selection, testing and adaptation of services with focus on composition. Service selection approaches are closely related to discovery, in which IR techniques and/or a semantic basis (e.g., ontologies) are generally used.

Listing 1.3: Wrapper3 for Calculate-CalculateService

```
public class Calculator{
//...
        public float sum (float arg1, float arg2){
                float ret0;
                try{ ret0= candidate.substract(arg1,arg2);
                }catch (exception ex){
                        ex.printStackTrace();
                        throw new RuntimeException(ex);
                }
                return ret0;
        }
//...
        public float product (float arg1, float arg2) {
                float ret0;
                try{ret0 = candidate.add(arg1, arg2);
                }catch (exception ex){
                        ex.printStackTrace();
                        throw new RuntimeException(ex);
                }
                return ret0;
        }
}
```

Table 2: Execution results of TS for Calculator-CalculatorService

Wrappers	**Test Cases**		
	successful	*failed*	*success rate*
wrapper3, wrapper4, wrapper6, wrapper7	0	4	0
wrapper0, wrapper1, wrapper5, wrapper8	2	2	50
wrapper2	4	0	100

Service evaluation mainly use WSDL documents and/or XML schemes of data types, or even WSDL-based ad-hoc enriched specifications. Service implementation may also affect its evaluation: *contract-first* services are designed prior to code, improving their WSDL descriptions; *code-first* services use automatic tools to derive WSDL documents from source code, reducing their description quality.

Regarding service testing, the work in [2] presents a survey of approaches that use strategies of verification and software testing. Some of them evaluate individual operations of atomic services, others also use a semantic basis such as OWL-S, and others evaluate a group of services that could interact in a composition.

The work in [5] presents an overview on service adaptation, at service interface and business protocol levels. This is required even though the Web Service standardization reduces the heterogeneity and simplifies interaction. At interface level adaptations deal with operation signatures, that implies perform message transformations or data mapping. At business protocol level, services behavior is affected on the order constraints of the message exchange sequences – such as deadlock and non-specified reception.

7 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. On the one side, confirming the suitability of a candidate service by a dynamic behavioral evaluation (execution behavior), in which the applied

testing criteria increase the reliability level. On the other side, effectively building the right adaptation logic for a selected Web Service, while reducing the adaptation and integration effort.

Currently, we are working on service compositions [7]. This is particularly useful when a single service cannot provide all the required functionality. In this context, it is necessary to generate software artifacts (e.g., tests and adapters) according to specifications in business process languages such as BPEL and BPML [14]. Finally, another interesting extension of this work is to automatically derive software artifacts from system models – for example from models described in SoaML [11], a UML profile for modeling Service-oriented Applications.

References

1. Xiaoying Bai, Wenli Dong, W-T Tsai, and Yinong Chen. Wsdl-based automatic test case generation for web services testing. In *Service-Oriented System Engineering, 2005. SOSE 2005. IEEE International Workshop*, pages 207–212. IEEE, 2005.
2. M. Bozkurt, M. Harman, and Y Hassoun. Testing and verification in service-oriented architecture: a survey. *Software Testing, Verification and Reliability*, 23(4):261–313, 2013.
3. A. De Renzis, M. Garriga, A. Flores, A. Zunino, and A. Cechich. Semantic-structural assessment scheme for integrability in service-oriented applications. In *Latin-american Symposium of Enterprise Computing, held during CLEI'2014*, September 2014.
4. M. Delamaro, J. Maidonado, and A. Mathur. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247, 2001.
5. Maryam Eslamichalandar, Kamel Barkaoui, and Hamid Reza Motahari-Nezhad. Service composition adaptation: An overview. *2nd IEEE IWAISE*, page 20À7, 2012.
6. M. Garriga, A. Flores, A. Cechich, and A Zunino. Behavior assessment based selection method for service oriented applications integrability. In *Proceedings of the 41st Argentine Symposium on Software Engineering*, ASSE '12, pages 339–353, La Plata, BA, Argentina, 2012. SADIO.
7. Martin Garriga, Andres Flores, Alejandra Cechich, and Alejandro Zunino. Web services composition mechanisms: A review. *IETE Technical Review*, In press, 2015.
8. M. Jaffar-Ur Rehman, F. Jabeen, A. Bertolino, and A. Polini. Testing Software Components for Integration: a Survey of Issues and Techniques. *Software Testing, Verification and Reliability*, 17(2):95–133, June 2007.
9. Jia, Y. y Harman, M. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.
10. μJava Home Page. Mutation system for Java programs, 2008. http://www.cs.gmu.edu/ offutt/mujava/.
11. OMG. Service oriented architecture modeling language (soaml) specification. Technical report, Object Management Group, Inc., 2012. http://www.omg.org/spec/SoaML/1.0.1/PDF/.
12. M. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: A research roadmap. *International Journal of Cooperative Information Systems*, 17(02):223–255, 2008.
13. D. Sprott and L. Wilkes. Understanding Service-Oriented Architecture. *The Architecture Journal. MSDN Library. Microsoft Corporation*, 1:13, January 2004. http://msdn.microsoft.com/en-us/library/aa480021.aspx.
14. S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, 2005.