Pattern-based Integrability on Service Oriented Applications

Diego Anabalón and Martin Garriga and Andres Flores and Alejandra Cechich, GIISCo Research Group, Facultad de Informática, Universidad Nacional del Comahue, Neuquen, Argentina.
Alejandro Zunino, ISISTAN Research Institute, UNICEN, Tandil, Argentina.

diego.anabalon@fi.uncoma.edu.ar, martin.garriga@fi.uncoma.edu.ar, andres.flores@fi.uncoma.edu.ar, alejandra.cechich@fi.uncoma.edu.ar, azunino@isistan.unicen.edu.ar

Building service-oriented applications implies the selection and integration of adequate services to fulfill a 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 and integration of Web Services. In this paper, we detail an integration model that allows a loosely coupled connection to third-party Web Service. For this, the integration model is based in two design patterns, the Adapter and Proxy patterns, into a specific composition. The Adapter pattern provides the solution upon partial interface compatibilities between a consumer application requirement and the interface offered by a candidate Web Service. The Proxy pattern encapsulates the physical connection aspects required to communicate with the remote Web Service. In our integration model, a consumer application is related to the Adapter patten structure, which in turn makes use of the Proxy pattern structure to invoke the remote service. Therefore, the joint usage of such design patterns helps a consumer application to remain untouched, without modifying its source code upon the integration of a remote Web Service. This also enables a flexible maintenance through likely changes of a service provider without updating and/or re-testing the consumer application. A concise example is included to show the potential of this approach for both selection and integration of a candidate Web Service.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*

General Terms: Design, Verification

Additional Key Words and Phrases: Service-Oriented Computing, Web Services, Service Selection, Design Patterns

ACM Reference Format:
jn V, N, Article (January YY), 16 pages.

1. INTRODUCTION

Service-Oriented Computing (SOC) is a paradigm that promotes the development of rapid, low-cost, interoperable, evolvable, and massively distributed applications through a network of services, which can create dynamic business processes that span organizations and computing platforms [Papazoglou et al., 2008]. Service-oriented applications can be seen as component-based applications that provide a business-oriented solution by assembling both internal and external components. The former implies in-house components or services, while the latter implies consuming services from one or more providers to integrate them into the business process [Sprott and Wilkes, 2004; Rodriguez et al., 2013]. Mostly, the software industry has adopted SOC by using Web Service technologies. A Web Service is a program with a well-defined interface that can be located, published, and invoked by using standard Web protocols [Papazoglou et al., 2008].

However, a broadly use of the SOC paradigm requires efficient approaches to allow service consumption from within applications [McCool, 2005] . Although developers do not need to know the underlying model and rules of a third-party service, its proper reuse still implies a big effort. Currently, developers are required to manually search for suitable services to then provide the adequate *glue-code* for assembling into the application under development. Ideally, the client of such applications can be unaware of physical details to invoke available services, e.g., interaction protocols, data-type formats, location, and so forth. Furthermore, client components usually end

up closely tight to interfaces exposed by the services. This leads to in-house components being subordinated to third-party interfaces, which must be modified and/or re-tested every time such interfaces change. For instance, a common requirement in SOC is to change the provider of a service, which could turn into a hard task. Therefore, specific programming models should be delivered to allow consumer applications being able to exploit services while enabling loosely coupling, as a manner to supply maintainability.

In order to ease the development of service-oriented applications, we presented in previous work [De Renzis et al., 2014; Anabalon et al., 2015a] a proposal to assist developers on the selection of candidate Web Services by evaluating compatibility at interface and behavior levels. In addition, an integration model was defined, which posses a dual utility. It serves as support to the service behavior evaluation and also enables a loose coupled assembly of a selected Web Service. In the present work, we give details of the integration model, which has been defined by applying two design patterns, the Adapter and Proxy patterns [Gamma et al., 1995; Grand, 2002; Cooper, 2000], in a specific composition.

As request to the integration model, a business consumer application should be decoupled from any remote Web Service. On one hand, client-side operations should be properly linked to those offered by a Web Service, without any update on both sides – upon interface mismatchings. Hence, the Adapter pattern provides a suitable solution. On the other hand, a physical connection to a remote Web Service must be settled, without affecting the functional code of a consumer application. Also, likely location changes of service providers should occur in a transparent way. Thus, the Proxy pattern helps to address these issues. All in all, the integration model involves a consumer application to be (thin) related to the Adapter structure, which is in turn related to the Proxy structure, to finally connect to a remote service. Through this paper, we show how the joint application of both design patterns allows the integration of a Web Service into a bussines consumer application in a seamless way.

The rest of the paper is organized as follows. Section 2 presents an overview of the service selection method. Section 3 presents the pattern aplication in our approach. Section 4 presents related work and, 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. However, 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: (1) Interface Compatibility, (2) Behavioral Test Suite, and (3) Behavioral Compatibility.

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, and exceptions) at four compatibility levels: *exact*, *near-exact*, *soft*, and *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 [De Renzis et al., 2014]. 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 required interface (I_R) from an application can be fulfilled by a potential candidate Web Service, a *Behavioral Test Suite* (TS) is built (step 2) [Anabalon et al., 2015b]. This TS describes the required messages exchange from/to a third-party service, upon a selected testing coverage criteria [Jaffar-Ur Rehman et al., 2007; Bai et al., 2005] 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, three steps are carried out in which parallely an integration model is built

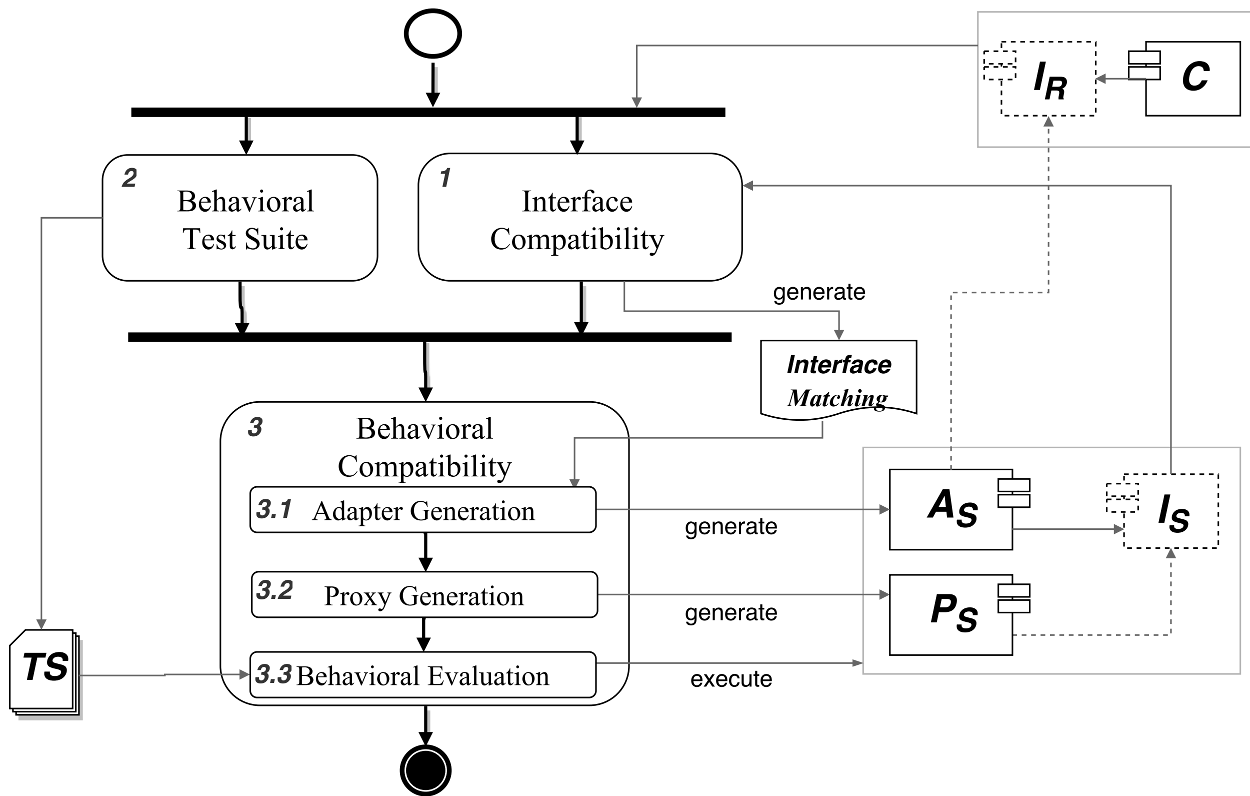


Fig. 1: Service Selection Method

based on the joint application of the Adapter and Proxy patterns. In the *Adapter Generation* (step 3.1), the *Interface Matching* list is processed to generate a set of adapters A_S – based on the identified *conflictive operations* – allowing to run the TS against the candidate service S. In the *Proxy Generation* (step 3.2), a proxy (P_S) for S is generated. Thus, a client C is able to call the operations declared in I_S, where this proxy transparently invokes the remote service S. In the *Behavioral Evaluation* (step 3.3), the TS is exercised against each adapter A_S. In this step, at least one adapter 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 adapter A_S – in composition with the proxy P_S – outlines an integration model allowing a client component C to safely call service S from within the customer application.

Next sections provide a short description of the two initial procedures above. The *Behavioral Compatibility* procedure is explained in detail in Section 3 together with the pattern-based integration model. A simple example will be used to illustrate the usefulness of the Service Selection Method and its integration model.

2.1 Proof-of-Concept

To illustrate our proposal, we assume a simple example of an `Accounting Application` that needs to integrate a calculator service. Figure 2 shows the dependency of the client component C (`Accounting Application`) to the required interface (I_R), to be fulfilled by an external service (`Calculator`).

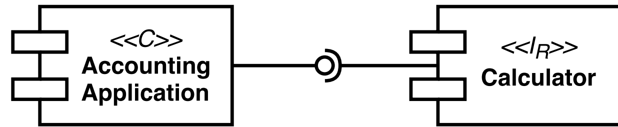
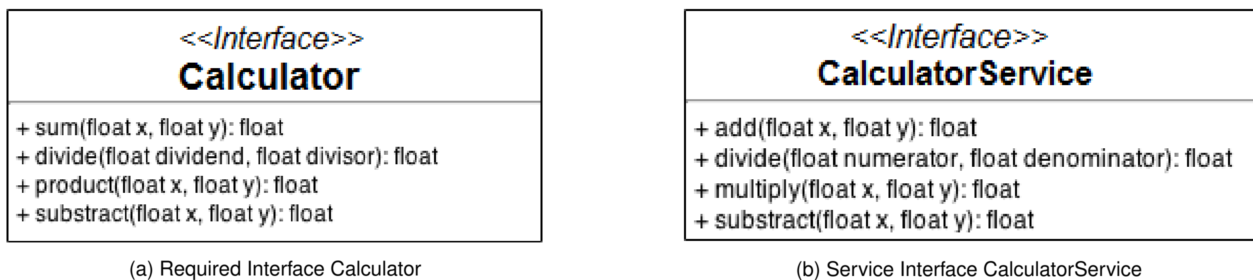


Fig. 2: Dependency of the Accounting Application (client) to the required interface (I_R) to be fulfilled by an external service.

Figure 3a shows the required interface (I_R) called `Calculator` with the four basic arithmetic operations. Figure 3b shows the interface (I_S) of a third party candidate Web Service named `CalculatorService`[1].



(a) Required Interface Calculator

(b) Service Interface CalculatorService

Fig. 3: Example of Calculator service

2.2 Interface Compatibility

In the step 1 of *Interface Compatibility* is determined the degree of compatibility between the operations of the interface I_R and the operations of the interface I_S of a candidate service S [De Renzis et al., 2014]. 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[2] 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 degrees: *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, and 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. Only the highest compatibility value is initially considered from the obtained matchings set on each op_R in I_R.

In some cases, certain required operations ($op_R \in I_R$) could obtain multiple matchings (with the same compatibility value) – 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 *"conflictive 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.

[1] http://ws1.parasoft.com/glue/calculator
[2] https://wordnet.princeton.edu/

Example: Calculator-CalculatorService Interface Matching. Table I shows the interface matching results for `Calculator` and `CalculatorService`. In this case, 3 matchings for each operation. 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 compatibility value *near-soft* (*n_soft*). Operations `subtract` and `divide` of `Calculator` are non-conflictive operations. They obtained a unique correspondence of higher compatibility value to their homonyms from `CalculatorService` – i.e., *exact* match for `subtract` operation and *near-exact* (*n_exact*) match for `divide` operation.

Table I. : Interface Compatibility for Calculator-CalculatorService

Calculator	**CalculatorService**		
	Matching 1	Matching 2	Matching 3
float subtract (float x, float y)	[exact, float subtract (float x, float y)] {(x:float-x:float), (y:float-y:float)}	[n_soft, float add (float x, float y)]	[n_soft, float multiply (float x, float y)]
float sum (float x, float y)	[n_soft, float add (float x, float y)] {(x:float-x:float), (y:float-y:float)}	[n_soft, float subtract (float x, float y)] {(x:float-x:float), (y:float-y:float)}	[n_soft, float multiply (float x, float y)] {(x:float-x:float), (y:float-y:float)}
float divide (float dividend, float divisor)	[n_exact, float divide (float numerator, float denominator)] {(dividend:float-numerator:float), (divisor:float-denominator:float)}	[n_soft, float add (float x, float y)]	[n_soft, float subtract (float x, float y)]
float product (float x, float y)	[n_soft, float add (float x, float y)] {(x:float-x:float), (y:float-y:float)}	[n_soft, float subtract (float x, float y)] {(x:float-x:float), (y:float-y:float)}	[n_soft, float multiply (float x, float y)] {(x:float-x:float), (y:float-y:float)}

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 has identical types and equivalent (synonyms) names – $dividend$ with $numerator$ and $divisor$ with $denominator$ – with the operation of `CalculatorService`. For details of the *Interface Compatibility* procedure, we refer the reader to [De Renzis et al., 2014].

2.3 Behavioral Test Suite

In the step 2 of *Behavioral Test Suite* (TS), a TS is built as a behavioral representation of services, specific coverage criteria for component testing has been selected [Anabalon et al., 2015b]. 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 [Jaffar-Ur Rehman et al., 2007; Bai et al., 2005], 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*" [Bozkurt et al., 2013], 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 this 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.

Example: Test Suite 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 a single 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 [μJava Home Page, 2008]. From this combination, 8 test cases were generated in the form of methods into a test file called `MujavaCalculator`. Code Listing 1 shows the test case `testTS_3_1`, which invokes the `sum` operation.

Listing 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();
}
```

3. PATTERN-BASED BEHAVIOR COMPATIBILITY EVALUATION

In this section, we explain in detail the step 3 of *Behavior Compatibility* (Figure 1). In addition, we discuss the challenges that were overcomed through the definition of a pattern-based integration model that enables the behavior evaluation.

To carry out the *Behavior Compatibility* evaluation for a candidate service S, we need to execute the *Behavioral TS* and compare its results with those specified for the interface I_R.To do this, first it is necessary to process the *Interface Matching* list – generated in the step 1 of *Interface Compatibility* (Figure 1). It can be the case that multiple potential matchings are found for certain required operations $op_R \in I_R$ with respect to I_S operations. This implies a partial interface compatibility, in which those multiple matchings (*"conflictive operations"*) must be disambiguated so to identify proper univocal matchings.

Thus, we applied the Adapter pattern [Gamma et al., 1995; Grand, 2002; Cooper, 2000] to solve the interface incompatibility, by generating a set of adapters (A_S) based on those conflictive operations. As known, the Adapter pattern allows the interface of an existing class (I_S) to be used as another interface (I_R) so that existing classes work together without modifying their source code.

Besides, to safely invoke remote operations from the candidate service S, it is necessary to build a surrogate of the service. A surrogate is a local object that represents the actual object (Web Service) that resides in a different address space. This representative contains all operations signatures defined in the actual service, along with the connection aspects to allow remote invocations.

To provide a service surrogate, we applied the Proxy pattern [Gamma et al., 1995; Grand, 2002; Cooper, 2000]. Thus, a proxy (P_S) is generated for the candidate service S, from where a client component C will end up calling the operations declared in the service interface I_S through P_S, which transparently invokes the remote service S.

The joint application of such design patterns compose our integration model, in which a consumer application remains loosely coupled of a remote Web Service. An alternate first option could be directly injecting a service proxy (P_S) into the application, which requires modifying the client business logic to make it compatible with the service interface (I_S). Instead, a service adapter (A_S) is placed in between with the necessary logic to match the required client interface (I_R) to the actual interface of a selected Web Service (I_S).

Following sections describe the steps of the *Behavioral Compatibility* step, highlighting the applied design patterns that form the integration model.

3.1 Adapters Generation

As we mentioned above, the adapters are necessary to establish a connection of the client component C to the interface of the candidate service I_S, in a seamless way. This is done according to the *Interface Matching* list, where multiple matchings are identified – both at operation and parameters levels.

Adapters generation can be seen as applying the Interface Mutation technique [Delamaro et al., 2001; Jia, Y. and Harman, M., 2011], by using a mutation operator to change invocations to operations and to change arguments in the parameters list. Thus, each adapter is considered a faulty version (or mutant) regarding the adapter that contains the proper matchings of operations and parameters. A tree structure is built to generate all the adapters (A_S), where each path from the root to a leaf node represents a specific matching between operations of I_R and I_S (i.e., an adapter to be generated). Thus, the number of leaf nodes determines the size of the adapters set A_S. Each *conflictive operation* produces several branches on the tree. On the contrary, a non-conflictive operation (implying a univocal match) does produce a single branch 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.

Example: Adapters for Calculator-CalculatorService. According to the results of *Interface Matching* for `Calculator` and `CalculatorService` (Section 2.2), *conflictive operations* ocurred only at operation level – for every operation a unique matching was found at parameters level. Figure 4 shows the adapter generation tree for `Calculator` and `CalculatorService`. Branches were only produced at operation level according to the *conflictive operations* identified: `sum` and `product` of `Calculator` with regard to `add`, `subtract` and `multiply` of `CalculatorService`.

The total number of adapters (size of A_S) to be generated is 9, which is the number of leaves on the tree.

Listings 2 and 3 show a fragment of the code from *adapter2* (A_2) and *adapter3* (A_3) respectively. Where *adapter2* represents both the tree path down-to the third leaf node and the most appropriate matchings – i.e., sum-add and product-multiply. Likewise, *adapter3* represents the path down-to the fourth leaf node – being a faulty (mutant) version , i.e., sum-subtract and product-add.

Adapter pattern application. Figure 5 shows the conceptual application of the Adapter pattern in our approach. The *client* role is played by C (application client component) that is related to the interface I_R, being the *target* role. In addition, the *Behavioral TS* also plays the *client* role when is used for behavioral evaluation. Since, the client (C/TS) needs a way to call the service interface I_S (playing the *adaptee* role), the component A_S (playing the *adapter* role) is settled in between. Thus, the adapter A_S establishes specific calls from the I_R abstract operations to particular service operations defined in I_S.

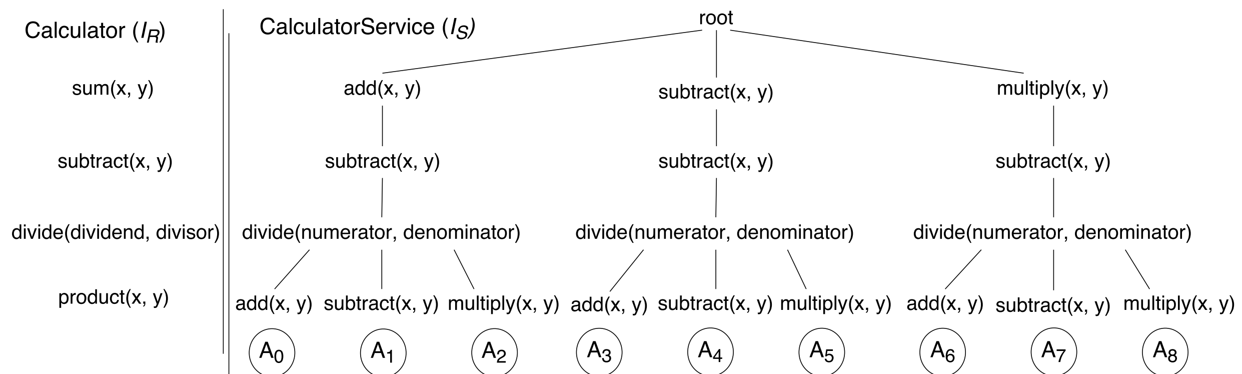


Fig. 4: Adapter generation tree to Calculator Service

Listing 2: Adapter2 for Calculate-CalculateService

```
public class Calculator{
protected glue.calculator.CalculatorService proxy = null;
        public Calculator(){
                this.proxy = new glue.calculator.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;
        }
}
```

Example. The Figure 6 shows the class structure of the Adapter pattern in the context of the `CalculatorService` example. The *client* role of the Adapter pattern in the example is played at one time by the `MujavaCalculator-Service` class (during the evaluation step) and at another time by the `Accounting Application` (to get a service integration). The *adapter* role is instantiated by each of the adapters generated for `Calculator`. Finally the *adaptee* role is played by the `CalculatorService` class that represents the Web Service.

Listing 3: Adapter3 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;
        }
}
```

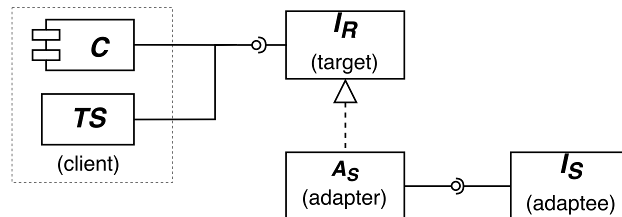


Fig. 5: Adapter design pattern application.

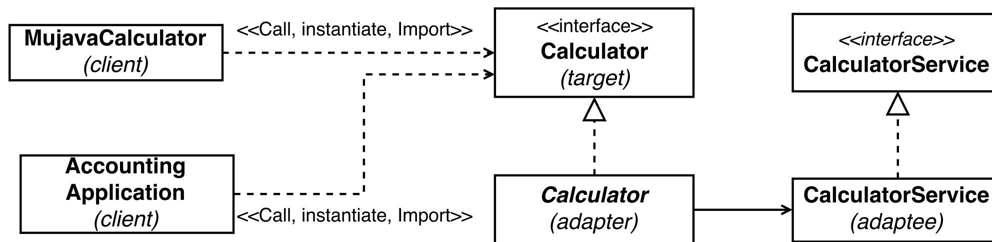


Fig. 6: Adapter pattern application for Calculator

Figure 7 shows the communication between a client, an adapter, and the service interface, in the example. Thus, the client (`Accounting Application`) invokes operations of the adapter (`Calculator`). In turn, the adapter executes a specific service operation mapping (from the *Interface Matching* list). In this case, the `product`

operation from `Calculator` invokes the `add` operation from `CalculatorService` that represents the *adapter3* from Listing 3. This allows a seamless invokation of service operations without modifying the structure or interfaces of the client application.

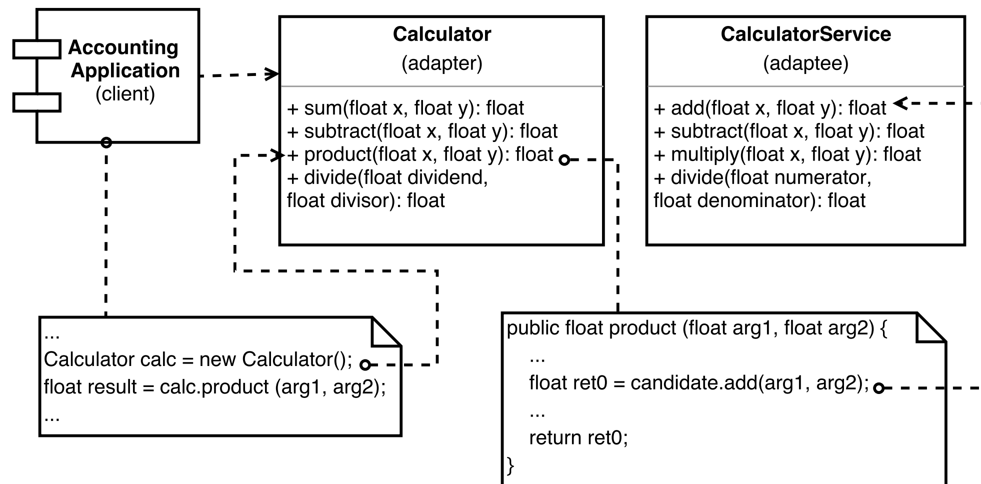


Fig. 7: Adapter pattern comunication structure for CalculatorService

3.2 Proxy Generation

As mentioned earlier, the adapters need to invoke the Web Service S. Then a physical connection to perform the remote calls is necessary, which is managed through a *proxy*. Figure 8 shows the conceptual application of the Proxy pattern in the SOC paradigm, where the functional code of a consumer application is isolated from connection aspects. Those aspects may involve network session/connection, making RPCs (Remote Procedure Calls), XML data (un)marshalling, among others. Thus, the *proxy* is the responsible of such physical connections to a Web Service.

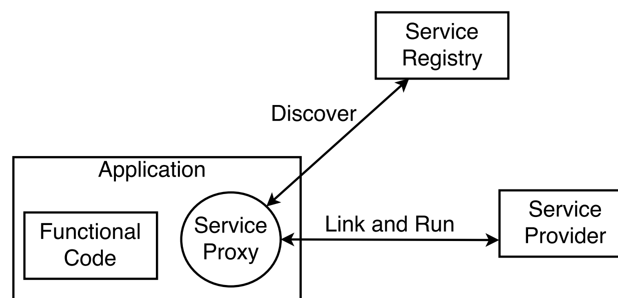


Fig. 8: Proxy application in SOC.

Figure 9 depicts a schematic Proxy pattern application in our approach as a solution to connect with a remote Web Service. The C component needs to interact with the interface (I_S) of Web Service S. However, instead of being C directly related to the *proxy* component (P_S), it is actually in dependence with the adapter component

(A_S) – placed in between. Thus, the *client* role for the Proxy pattern is played by the *adapter* component from the Adapter pattern – described in the previous section. The *subject* role is played by the service interface I_S, because the *proxy* role implements the interface defined by the *subject*. Finally, the *real subject* role is played by the actual Web Service S, from its remote location. Interestingly, this mechanism is not intrusive, since the code of C remains untouched still on dependency with I_R, from where the adapter A_S and the proxy P_S are generated.

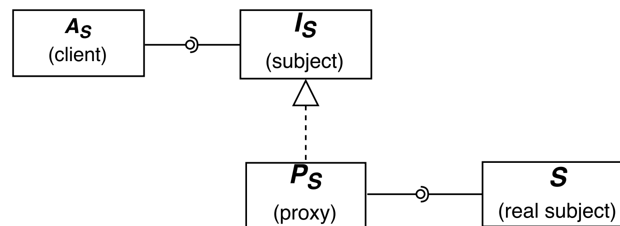


Fig. 9: Proxy design pattern application

In order to automate the generation of the proxy component (P_S), the Web Services Description Language (WSDL) document of a candidate service S is processed through the WSDL2Java[3] library from the Axis Framework. From this, a Java skeleton of the service is generated – a set of *Stub* classes – that manage the remote communication with the Web Service.

Example. Figure 10 particularly shows the *Stub* structure generated for `CalculatorService`. The `CalculatorServiceStub` class encapsulates the remote connection aspects – e.g., network connection and request data marshalling (XML). The `CalculatorServiceCallBackHandler` class receives (unmarshall) the response data from the service operations.

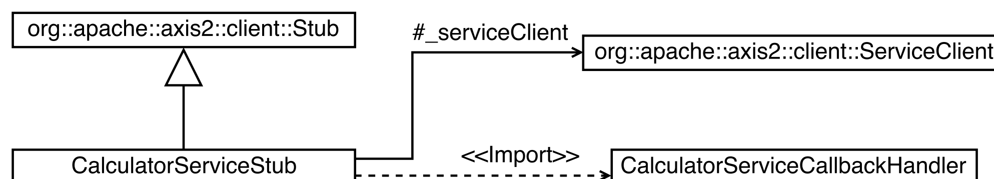


Fig. 10: Proxy Stub structure for CalculatorService

Figure 11 shows the whole class structure of P_S for `CalculatorService`. The *client* role of the Proxy pattern in the example is played by `Calculator`, that invokes service operations. The *proxy* role is played by a set of classes, including the `CalculatorService` class. This class is also in charge of instantiating the remaining classes that involve the *Stub* structure as shown in Figure 10.

3.3 Adapter-Proxy Pattern Composition

Figure 12 shows the final composition of the Adapter and Proxy patterns instanciated in the context of the `Calculator` example. In this composition, the Adapter *client* (`Accounting Application/MujavaCalculator`) invoke operations from the `Calculator` class – *adapter* and *client* in the Adapter and Proxy patterns respectively. In turn, this class invokes operations from `CalculatorService` – *adaptee* and *proxy* in the Adapter and Proxy

[3] http://cxf.apache.org/docs/wsdl-to-java.html

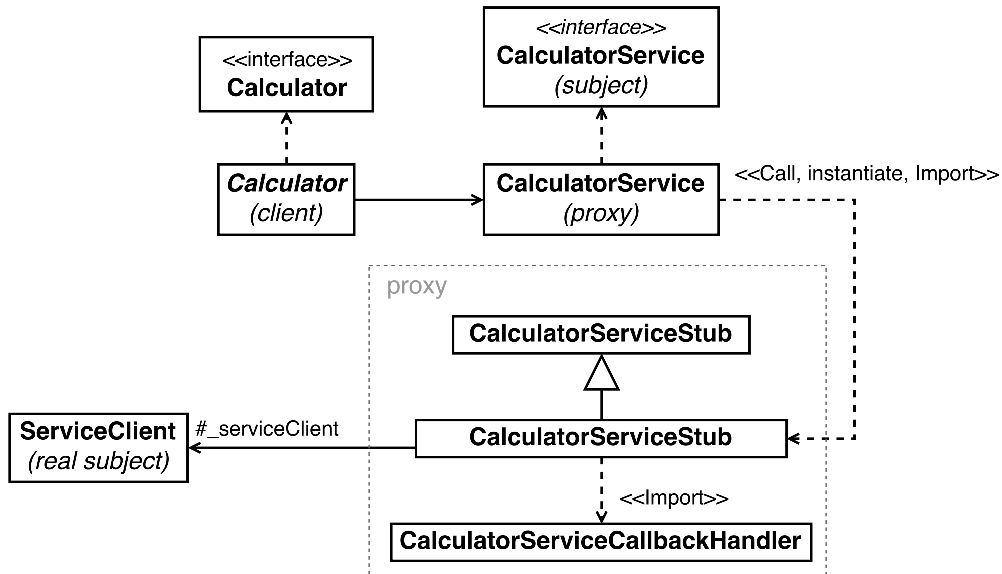


Fig. 11: Proxy pattern application for Calculator

patterns respectively. Finally, this class together with its connected components (*Stub* structure) invokes the actual Web Service.

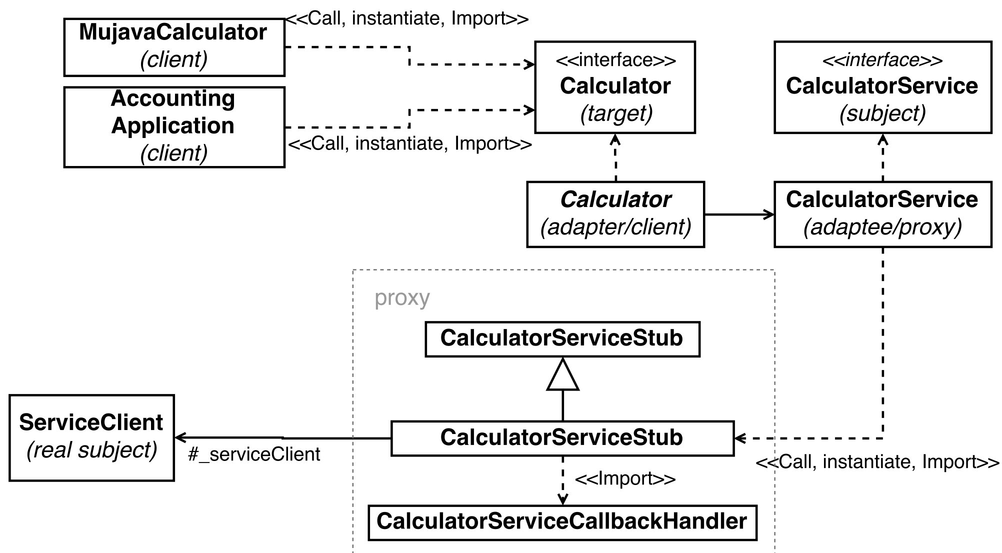


Fig. 12: Proxy+Adapter pattern composition for the Calculator example.

3.4 Behavioral Evaluation

Once generated the set of adapters A_S together with the proxy P_S, the *Behavior TS* is executed against each adapter 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 adapters. After that, results are compared to determine for each adapter the number of test cases that failed – which produced a result different from the one expected. An adapter may survive (as mutation case) when most of the test cases are successful. A successful adapter allows to disambiguate the *conflictive operations*, confirming the proper (behavioral) matching both at operation and parameters levels. In addition, this adapter also represents the suitable artifact to act as a connector between the business application and the candidate Web Service S – providing a seamless integration in which an in-house component C can safely invoke service operations.

Table II shows the execution results, where *adapter2* passed successfully 100% of the test cases allowing to confirm the behavioral compatibility of `CalculatorService`. In addition, this adapter contains the right matchings of operations (`sum-add`, `subtract-subtract`, `divide-divide`, and `product-multiply`). Finally, *adapter2* can be used as an artifact for the safe integration of `CalculatorService` into the client `Accounting Application`.

Table II. : Execution results of TS for Calculator-CalculatorService

Adapters	**Test Cases**		
	successful	*failed*	*success rate*
adapter3, adapter4, adapter6, adapter7	0	4	0
adapter0, adapter1, adapter5, adapter8	2	2	50
adapter2	4	0	100

4. RELATED WORK

This section presents concepts closely related to our approach, such as service selection, adaptation, testing, and integration. Besides, main catalogs of service oriented design patterns are also introduced.

In [Garriga et al., 2015] we surveyed 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. 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 [Rodriguez et al., 2010]: 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. In our approach, services are evaluated at interface level by applying both IR techniques and WordNet (as a lightweight semantic basis). In addition, our approach relies on service contract information that can be gathered from a WSDL specification generated either in a code-first or in a contract-first manner.

Regarding service testing, the work in [Bozkurt et al., 2013] 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. Our approach intends to fulfill a required functionality through a selected candidate service. The expected behavior is described in form of a specific TS, which is then exercised against atomic candidate services. Our TS describes a complex behavior exhibited by operational sequences (instead of testing individual operations), which is more likely on stateful Web Services [Weerawarana et al., 2005].

The work in [Eslamichalandar et al., 2012] 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, it deals with operation signatures that implies to 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.

In [Seguel et al., 2010], it is presented an automated method to construct a minimal protocol adapter with parallelism for two asynchronously communicating business protocols, implemented through services. A minimal adapter only processes messages causing mismatches, introducing as little overhead as possible. Such an adapter ensures that two composed services terminate properly by receiving and reordering messages and delivering them in the expected order and time.

In [Tan et al., 2009], it is proposed a method to analyze compatibility of BPEL-specified services, by converting them into Coloured Petri-Nets (CPNs). A mediator is synthesized in case of partial compatibility. A mediator is, in fact, a kind of adapter. Through a set of formalisms derived from CPNs, such as the Service Workflow Net (SWF-net), the feasibility of mediation can be checked by a state-space method. This is computationally costly.

In [Pastrana et al., 2011], it is proposed an automated approach to build client-side connectors, which are, as mediators, a kind of adapter. Connectors are defined as self-adaptive to react upon different changes – e.g., updates of Web Services, data conversion problems, and non-functional features changes. Their automatic generation is based on server domain OWL ontologies, which implies a drawback. Domain specific ontologies are avoided in practice because of their unavailability and the difficulty to be specified [Crasso et al., 2011; Bouchiha et al., 2012].

In our approach, a client-side adaptation is performed at interface level to solve a partial compatibility – i.e., operations and data mapping – regarding a selected candidate service. Particularly, adapters are generated as interface mutation cases to achieve a suitable adapter that allows a consuming application to safely invoke the selected Web Service.

Regarding design patterns in the context of SOC-based systems some catalogues have been defined. In [Erl, 2008] a vast set of service-oriented desing patterns are provided, which range from business organizational processes to conceptual design of services and service composition. This catalogue is defined for the SOC paradigm without any practical implementation technology – e.g., object-orientation – mainly serving as SOC design principles and heuristics.

In [Monday, 2003], it is presented a catalogue of Java Web Services patterns, which define architectural, business processes, UDDI publish/discovery registry and Web Services implementation solutions in the Java platform. Besides, most solutions involve the Apache Axis Framework. For example, the Architecture Adapter pattern is intended to provide a communication mechanism to connect two heterogenous system architectures. In addition, the Connector design pattern, is intended to connect different tiers abstrations, within a business application, that span physical boundaries. This prevents from changes of remote third-party Web Services, by a proxy-based solution.

In our approach, the integration model is built by composing two design patterns, the adapter and the proxy patterns. The adapter pattern is applied to solve interfaces incompatibility, allowing an internal application component to establish a connection to a selected candidate service in a seamless way. The proxy pattern is applied to provide a surrogate of a remote Web Service. Thus, physical connection aspects of remote invocations – e.g., network session/connection, RPCs, XML data (un)marshalling – become encapsulated into a proxy entity. Therefore, our integration model allows a consumer application to remain loosely coupled of a third-party Web Service – without altering the client business logic.

Table III shows a summary comparing our approach with the related work in terms of selection, adaptation/integration, and testing.

Table III. : Comparative summary of related work

Approaches	*Selection*	*Adaptation/Integration*	*Testing*
Our approach	Evaluated at interface level (IR techniques and WordNet)	Adapter at interface level (mutation cases). Patterns composition (adapter and proxy)	Complex behavior (operational sequences)
[Garriga et al., 2015]	Discovery (IR and semantics)		WSDL, XML, and WSDL-based enriched specifications
[Bozkurt et al., 2013]			Evaluate atomic services, semantic bases (OWL-S)
[Eslamichalandar et al., 2012]		Interface (signatures) and protocol level	
[Seguel et al., 2010]		Minimal protocol adapter	
[Tan et al., 2009]	Compatibility of BPEL-specified services	SWF-net Mediator	
[Pastrana et al., 2011]		Client-side connectors (based on domain OWL ontologies)	
[Erl, 2008]		Bussines organizational process (service composition)	
[Monday, 2003]		Connection of heterogenous system architectures	

5. CONCLUSIONS AND FUTURE WORK

In this paper we have presented an approach to assist developers in the selection and integration of third-party Web Services, when developing a service-oriented application. Particularly, our integration model addresses two main aspects. On the one side, giving support to confirm the suitability of a candidate service by a dynamic behavioral evaluation (execution behavior), in which the applied testing criteria increases the reliability level. On the other side, effectively building the right adaptation and connection logics for a selected Web Service, by patterns application such as Adapter and Proxy patterns. Thus, our automated integration model reduces the effort for developers on the adaptation, connection, and integration of third-party services, while injecting flexibility for maintenability.

Currently, we are working on service compositions [Garriga et al., 2015]. This is particularly useful when a single service cannot provide all the required functionality. In this context, it is necessary to widely extend the whole evaluation process, generating software artifacts of greater complexity (e.g., tests and adapters). Such artifacts could be derived from specifications in business process languages such as BPEL and BPML [Weerawarana et al., 2005]. Likewise, software artifacts could be derived from system models – for example from models described in SoaML [OMG, 2012], a UML profile for modeling service-oriented applications.

REFERENCES

Anabalon, D., Garriga, M., Flores, A., Cechich, A., and Zunino, A. (2015a). Adaptability-based service behavioral assessment. *Journal of Computer Science Technology*, 15(2):75–80.

Anabalon, D., Garriga, M., Flores, A., Cechich, A., and Zunino, A. (2015b). Test reduction for web service integration. In *Simposio Argentino de Ingeniería de Software (ASSE 2015)*, pages 115–129.

Bai, X., Dong, W., Tsai, W.-T., and Chen, Y. (2005). 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.

Bouchiha, D., Malki, M., Alghamdi, A., and Alnafjan, K. (2012). Semantic web service engineering: Annotation based approach. *Computing and Informatics*, 31(6):1575–1595.

Bozkurt, M., Harman, M., and Hassoun, Y. (2013). Testing and verification in service-oriented architecture: a survey. *Software Testing, Verification and Reliability*, 23(4):261–313.

Cooper, J. (2000). *Java Design Patterns: A Tutorial*. Addison-Wesley.

Crasso, M., Zunino, A., and Campo, M. (2011). A survey of approaches to web service discovery in service-oriented architectures. *Journal of Database Management (JDM)*, 22(1):102–132.

De Renzis, A., Garriga, M., Flores, A., Zunino, A., and Cechich, A. (2014). Semantic-structural assessment scheme for integrability in service-oriented applications. In *Latin-american Symposium of Enterprise Computing, held during CLEI'2014*. IEEE Computer Society Press.

Delamaro, M., Maldonado, J., and Mathur, A. (2001). Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247.

Erl, T. (2008). *SOA design patterns*. Pearson Education.

Eslamichalandar, M., Barkaoui, K., and Motahari-Nezhad, H. R. (2012). Service composition adaptation: An overview. *2nd IEEE IWAISE*, pages 20–27.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Garriga, M., Flores, A., Cechich, A., and Zunino, A. (2015). Web Services Composition Mechanisms: A Review. *IETE Technical Review*, 32(5):376–383.

Grand, M. (2002). *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML*. Number v. 1 in Patterns in Java. Wiley.

Jaffar-Ur Rehman, M., Jabeen, F., Bertolino, A., and Polini, A. (2007). Testing Software Components for Integration: a Survey of Issues and Techniques. *Software Testing, Verification and Reliability*, 17(2):95–133.

Jia, Y. and Harman, M. (2011). An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37(5):649–678.

McCool, R. (2005). Rethinking the Semantic Web. *IEEE Internet Computing*, 9(6):86–87.

Monday, P. B. (2003). *Web Service Patterns: Java Edition*. Apress.

μJava Home Page (2008). Mutation system for Java programs. http://www.cs.gmu.edu/ offutt/mujava/.

OMG (2012). Service oriented architecture modeling language (soaml) specification. Technical report, Object Management Group, Inc. http://www.omg.org/spec/SoaML/1.0.1/PDF/.

Papazoglou, M., Traverso, P., Dustdar, S., and Leymann, F. (2008). Service-oriented computing: A research roadmap. *International Journal of Cooperative Information Systems*, 17(02):223–255.

Pastrana, J. L., Pimentel, E., and Katrib, M. (2011). Qos-enabled and self-adaptive connectors for web services composition and coordination. *Computer Languages, Systems & Structures*, 37(1):2–23.

Rodriguez, J. M., Crasso, M., Mateos, C., and Zunino, A. (2013). Best practices for describing, consuming, and discovering web services: a comprehensive toolset. *Software: Practice and Experience*, 43(6):613–639.

Rodriguez, J. M., Crasso, M., Zunino, A., and Campo, M. (2010). Improving web service descriptions for effective service discovery. *Science of Computer Programming*, 75(11):1001–1021.

Seguel, R., Eshuis, R., and Grefen, P. (2010). Generating minimal protocol adaptors for loosely coupled services. In *Web Services (ICWS), 2010 IEEE International Conference on*, pages 417–424. IEEE.

Sprott, D. and Wilkes, L. (2004). Understanding service-oriented architecture. *The Architecture Journal*, 1(1):10–17.

Tan, W., Fan, Y., and Zhou, M. (2009). A petri net-based method for compatibility analysis and composition of web services in business process execution language. *IEEE Transactions on Automation Science and Engineering*, 6(1):94–106.

Weerawarana, S., Curbera, F., Leymann, F., Storey, T., and Ferguson, D. (2005). *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR.