Test Reduction for Easing Web Service Integration*

Diego Anabalon[1,3], Martín Garriga[1,3], Andres Flores[1,3], Alejandra Cechich[1], and Alejandro Zunino[2,3]

Springer-Verlag, Computer Science Editorial,
Tiergartenstr. 17, 69121 Heidelberg, Germany
`{alfred.hofmann,ursula.barth,ingrid.haas,frank.holzwarth,`
`anna.kramer,leonie.kunz,christine.reiss,nicole.sator,`
`erika.siebert-cole,peter.strasser,lncs}@springer.com`
`http://www.springer.com/lncs`

Abstract. Since the irruption of Web Services, in their SOAP and REST flavors, the market has turned from intra-business applications to inter-organizational applications. Nowadays, more organizations have a broad access to the Web and span their frontiers using service-centered applications. In this paper, we review the testing challenges and strategies in Web Services – as the technological weapon-of-choice to implement Business Services. Then we deepen into a possible strategy to address service testing: Test Reduction. Fresh strategies are necessary since Web Services testing is substantially different from legacy systems testing.

Keywords: Web Services, Web Service Testing, Service-oriented Applications, Component-based Software Development

1 Introduction

Service Oriented Computing (SOC) is a computing paradigm whose main objective is the development of distributed applications in heterogeneous environments, which are built by assembling or composing existing functionality called *service*. Services are published through a network and accessed by specific protocols [20, 14]. A service-oriented architecture is composed of three actors: a provider, a consumer and a service registry. The registry is used by the provider to publish the description of their services, and also for consumers looking for services that meet their needs. Once a service is selected, it will be invoked from the client application [21]. In general, service-oriented applications are implemented using Web Services technology. A Web Service is a program with a well-defined interface which can be localized, published and invoked using the standard Web infrastructure [18].

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

SOC brings clear advantages, such as the low coupling between consumer/provider of a particular service and greater reusability. However, a main drawback implies to increase some costs on the development of reliable software. Organizations dealing with load testing, performance testing and integration testing of services face at least two kinds of challenges:

– *Organizational challenges*, such as limited testing budget, restrictive or impossible deadlines and high costs per test.
– *Technical challenges*, such as the high number of test cases, little or inexistent reuse of test cases and geographical distribution of nodes.

Current efforts in industry for selection, test and integration of Web Services involve semi-automatic mechanisms to facilitate the daily task of test engineers [23]. However, such mechanisms provide partial results in the form of sets of candidate services, where the engineer must still perform a manual task to definitively select the adequate candidate service. Candidate services sets may vary both in the required interface and in the expected behavior within a target application. Moreover, a candidate services set can be of a considerable size. Thereby, the task of selection and test may be overwhelming, affecting the development phase of an application as well as total costs.

This paper presents a concrete improvement to the services testing process by means of Test Reduction, in the context of Web Services selection. Candidate services are assessed at interface level and mainly at behavioral level. At interface level, Web Services are analyzed to identify whether their operations are structurally compatible with the required operations of a client application. Behavioral evaluation is based on the execution of a set of test cases or Test Suite (TS) which allows observing the runtime behavior of the service. Considering a reliability factor, the Test Suite usually scale to a high volume, making costly both its generation and execution. However, the definition of a small and proper Test Suite could also be helpful to confirm the compatibility of a candidate service. A specific Test Reduction strategy has been defined, aiming to improve the Web Services testing efficiency.

This paper is structured as follows. Section 2 introduces some foundational concepts in Web Service Testing and Components Testing, along with a review of the current challenges in the field. Section 3 presents our proposal for Web Services behavioral assessment based on Test Reduction. Conclusions and future work are presented afterwards.

2 Background

2.1 Web Service Testing

The following concepts are related to Software Testing and testability, particularly in the context of Component-based Software Development (CSBD) and Web Services – being CBSD the basis for the development and testing of Web Services [16].

The Testing process consists of the dynamic verification of the behavior of a program against the expected behavior, by means of a finite set of test cases. Test cases are properly selected from a (usually infinite) set of possible executions in a given domain [2]. In general, testing strategies attempt to cover the greatest possible risky or conflictive areas of a system. That is, those more error-prone, or those where defects could significantly affect the data, processes or users. Thus, a TS can become unmanageable when trying to ensure reliability on the behavior of the Service Under Test (SUT), implying an enormous effort to implement an exhaustive TS. To this end, specific strategies have been defined to maximize the value of the TS: minimization, selection and prioritization [23].

– *Minimization* attempts to reduce the size of a TS deleting redundant test cases or retaining essential test cases. Minimization is also called *reduction*, meaning that test cases deletion is permanent. These two concepts are essentially interchangeable.
– *Selection* also tries to reduce the size of a TS, but mostly considering software modifications. Test cases are selected because they are relevant to the modified parts of the SUT, which usually involves a white box static analysis of the program code.
– *Prioritization* attempts to re-order test cases to maximize some desirable properties early, such as fault detection rate. Prioritization finds the optimal permutation of the sequence of test cases. Upon re-ordering of test cases, it is possible to terminate the execution of TS on an arbitrary point.

2.2 Testability in Components and Web Services

According to the Standard Glossary IEEE 610 [2], Testability considers two aspects: (1) the degree to which a system or component facilitates both test criteria establishment and test performance; and (2) the degree to which a requirement allows tests and performance criteria to be established. The software industry requires components and reusable services that can be effectively tested in order to satisfy the high demand for effective and efficient software development. To do this, providers need guidelines and practical and valid methods to develop components and services capable of being tested [15, 4].

Particularly, a set of Testability factors are essential for providers to assess the testability level of their components, and for users to perform components selection and evaluation. The works in [15, 4] summarize Testability factors for software components, which are shown in Table 1.

Such factors are also valid in the context of Web Services. Customers or service requesters must add an additional assessment level that involves runtime availability of services, and continuous and reliable delivery of offered capabilities (functional and non-functional). This implies an additional burden for service providers that must extend the lifecycle of service development towards providing service execution platforms. In particular, such aspect of the SOC paradigm is called "relationship without responsibility" [24]. A service client is neither concerned about the service implementation nor about the actual execution of

Table 1. Testability Factors for Software Components

Factor	Description
Understandability	Information attached to the component (e.g., interfaces documentation, testing documentation)
Observability	Perceiving the output data as a function of the inputs
Controllability	Producing output data from a specific set of input data
Traceability	Comparison of expected behavior against the actual execution of the component (black-box). Checking the internal state of the component in each invocation (white-box)
Testing Support	Mechanisms to speed-up component testing (e.g., to build and execute a TS, to manage the testing process)

the service. Thereby, the reliability challenges in the SOC paradigm significantly expand. This led to a complementary set of factors to support testability in SOC, as summarized in Table 2.

Table 2. Testability factors for Web Services

Factor	Description
Atomic Services	Considers publishing service levels (and their accessible information): source code, binary code, pattern, signature (WSDL).
Data Provenance	Considers the history of data processing: identify their raw origin, how they were derived, its routing in SOA, and the processes applied to the data.
Service Integration	Considers the dynamicity of SOA: interoperability, composition and re-composition, service controller (as orchestrator), controller adaptation.
Service Collaboration	Considers the support of dynamic collaboration protocols: preparation, establishment, execution, completion. Each service determines its collaboration in execution.

2.3 Testing Techniques for Components and Web Services

Following we discuss the suitability of Component-specific testing strategies in the context of Web Services, and new strategies that addressed the inherent complexity of the Web Services environment.

Coverage Criteria for Components and Web Services

The provider/developer of components and services has access to their source code and therefore can apply different white-box testing methods and strategies. However, clients without access to the source code of a component/service require specific testing criteria to assess the correctness of services as units. Following we introduce the coverage criteria for components and Web Services used in this work [15, 22].

All-Methods [13] Also called *all-interfaces* criterion [22]. Components may have multiple interfaces, each one consisting of a set of operations or methods. The criterion requires all operations to be executed at least once. In the context of Web Services, the criterion is called *all-operations* [1], in which WSDL documents describe a single interface comprised of a set of operations.

All-Events [22] An event is an incident resulting in the invocation of an interface. Events can be synchronous (e.g., direct invocation of operations) or asynchronous (e.g., exceptions). The criterion establishes that every event (synchronous or asynchronous) of a component should be covered by a test case.

All-Context Dependences [22] Events can have sequential dependencies with other events, causing different behaviors according to the order in which operations or exceptions are invoked. The criterion requires traversing each valid operational sequence at least once. In the context of Web Services, the criterion is called *operation flows* [1], and considers the sequences of invocations to service's operations.

We consider *inter-component* dependencies, consisting of a component's interface depending upon external events (caused by another component). In the context of Web Services, other dependences are also considered that involve the input and output data (messages) from operations [1]: *Input dependence*, when two operations share the same input messages; *Output dependence*, when two operations share the same output messages; and *Input/output dependence*, when at least one of the output messages from an operation is the input message of another operation.

2.4 Testing strategies for Components and Web Services

The quality and volume of information that a component/service provides about itself allows applying certain testing strategies. The availability of source (or binary) code, or model specifications distinguishes between two types of components. On the one hand, *in-house* and FOSS[1] components, which could also be Web Services (from a provider perspective). On the other hand, components whose only available information is a provided interface, as most of the COTS[2] components and Web Services as well (from a client perspective).

[1] Free and Open Source Software
[2] Commercial off-the-shelf

Some testing techniques assume accessibility to the source code, while others may disregard it by depending on the provider to incorporate testing information that a client can extract from the components/services [15, 23]. In particular, in this work we use specification-based testing [3, 23]. Also known as black-box testing, its aim is either to reveal defects (faults) related to external functionality, communication between modules, and constraints (pre- and post-conditions); or to analyze the operational behavior of a program – i.e., Compliance Testing. The main challenge in testing is to discern the behavior of services from its specification, to then classify services reliably. However, the usually available service specifications are not formal, which hinders consistent test creation. Even though, specification-based criteria can be used in any context (procedural, OO, CBSD and Web Services) without significant adaptation.

We also use *Mutational Testing*, in which slight changes are applied to the source code of a Program Under Test (PUT) generating new versions that should misbehave. Defective versions are called mutants, and are created based on mutation operators: rules that define syntactic changes to the PUT. The purpose is to evaluate a TS in terms of distinguishing between the PUT and its mutants. One problem with the mutation technique is the high cost of running a large number of mutants [23, 5].

Different authors have proposed extensions to Mutational Testing for integration based upon program specifications. For example, Interface Mutation applies the mutation concept to the integration testing phase using a new set of specialized mutation operators [13, 8]. For components that do not provide access to source code, certain defects-based strategies can be applied, depending on the interfaces provided by the components. Particularly, Interface Mutation techniques can be used with minimal adjustments in the context of Web Services and CBSD [5, 4].

3 Test Reduction for Service Selection

In order to ease the development of Service-Oriented Applications, our proposal assist on the selection and test of candidate Web Services – previously retrieved from a service discovery registry. Figure 1 depicts the Testing-based Service Selection Method, which includes our strategy of Test Reduction.

3.1 Testing-based Selection Method Overview

As an initial step, a simple specification is needed, in the form of a required interface I_R (linked to an in-house component C), as input for the four comprising procedures. The Interface Compatibility procedure (step 1) assesses the required interface I_R and the interface (I_S) provided by a candidate service S. Through an structural-semantic analysis, operations signature (return, name, parameters, exceptions) is characterized at four compatibility levels: *exact*, *near-exact*, *soft*, *near-soft*. The outcome is an Interface Matching list where each operation from

I_R may have a matching with one or more operations from I_S [11, 7]. Particularly, operations from I_R with multiple matchings are considered as "conflictive operations" in this approach – i.e., they must be disambiguated yet.

The Behavioral Compatibility procedure is based on a testing framework to explore the required behavior of candidate services [9]. The goal is to fulfill the observability testing metric that observes the operational behavior of a service, by analyzing its functional mapping of data transformations (input/output).

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). This TS exhaustively describes the required messages interchange from/to a third-party service, upon a selected testing coverage criteria [8]. This exhaustive TS might be costly on both its generation and its execution. Thereby, another option is building a Reduced TS (step 3) with focus just on the subset of "conflictive operations" from I_R – identified from the Interface Matching list. In addition, to settle the expected behavior defined by I_R operations, an imperative specification is built as a Shadow Class – describing a black-box relationship between input-output data (functional mapping).

Lastly, for the behavioral evaluation (step 4), the Interface Matching list is processed to generate a set of wrappers W (adapters), which allow running the TS against the candidate service S [10]. After exercising the TS against each wrapper $w \in W$, at least one wrapper must successfully pass most of the tests to confirm the proper (behavioral) matching of "conflictive operations". As an optional final step, the exhaustive TS could be run against the successful wrapper for a more robust confirmation of behavior on the candidate Web Service. The achieved optimization involves a significant reduction in terms of time and effort – that might involve minutes instead of hours or days.

Besides, such successful wrapper can act as a connector between the business application and the candidate Web Service S – allowing an in-house component C to safely invoke service operations. From the whole process, this section will deepen in the procedure to Build a Reduced TS.

3.2 Proof-of-concept

To illustrate the process of generating a reduced TS we will assume a simple example as a calculator application, with the four basic arithmetic operations. The required functionality `Calculator` will be implemented with the candidate Web Service `CalculatorService`. Figure 2 shows the required interface (I_R) of `Calculator` and the interface (I_S) of `CalculatorService`.

3.3 Identifying Conflictive Operations

When the information about requirements conformance is reliable enough, it is possible to reduce the coverage area of the TS to optimize both test case

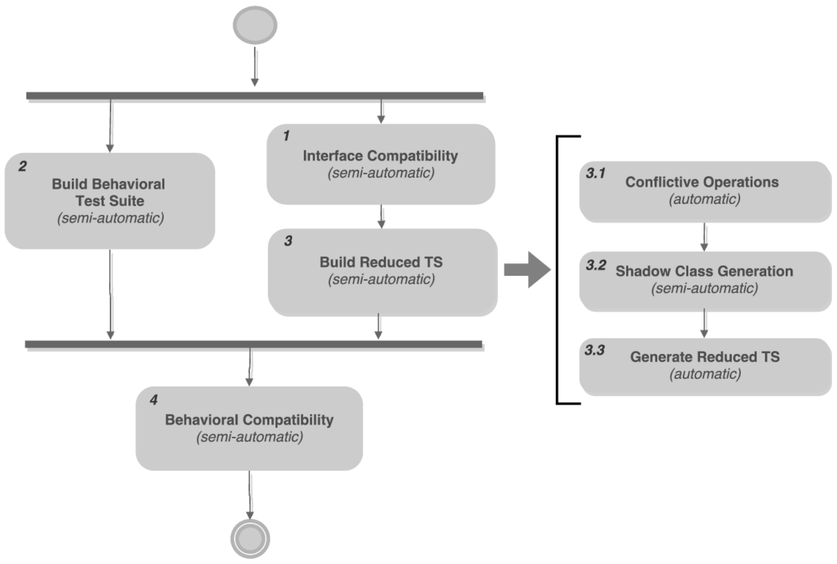
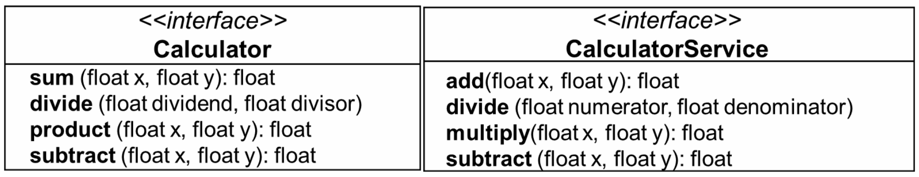


Fig. 1. Testing-based Service Selection Process



<<*interface*>> **Calculator**	<<*interface*>> **CalculatorService**
sum (float x, float y): float	**add**(float x, float y): float
divide (float dividend, float divisor)	**divide** (float numerator, float denominator)
product (float x, float y): float	**multiply**(float x, float y): float
subtract (float x, float y): float	**subtract** (float x, float y): float

Fig. 2. Interfaces of `Calculator` (I_R) and `CalculatorService` (I_S)

generation and execution. In this proposal, the information about requirements conformance comes from the Interface Compatibility evaluation [7, 11]. This evaluation generates an Interface Matching list comprised of recognized matchings among operations op_R of the required interface I_R and operations op_S from the interface I_S of a candidate Web Service S. Ideally, a required operation op_R will obtain a high compatibility with only one operation op_S from the candidate service – i.e., an univocal (unambiguous) matching.

However, some op_R operations may obtain various potential matchings with op_S operations at the same compatibility level – i.e., a multiple matching. For these cases, it becomes difficult to determine the most adequate service operation op_S to match the required operation op_R. Such op_R operations are called "conflictive operations" in this approach and still need a proper disambiguation, becoming relevant for building a Reduced TS.

Example Table 3 summarizes the result from the Interface Compatibility analysis in the `Calculator` example. On the one hand, `sum` and `product` obtained three *near-soft* compatibilities with operations `add`, `subtract` and `multiply` from the candidate service. On the other hand, `subtract` and `divide` obtained only one compatibility (at *exact* and *near-exact* levels respectively), thus they will not be considered as conflictive operations. Then, this example contains two *conflictive operations*: `sum` and `product`, as these two operations present multiple potential matchings with service operations at the same compatibility level.

Table 3. Conflictive operations for the `Calculator` example

I_R	I_S	**Level**	**Conflictive?**
`Sum`	`Add, Subtract, Multiply`	*near-soft*	Yes
`Subtract`	`Subtract`	*exact*	
`Product`	`Add, Subtract, Multiply`	*near-soft*	Yes
`Divide`	`Divide`	*near-exact*	

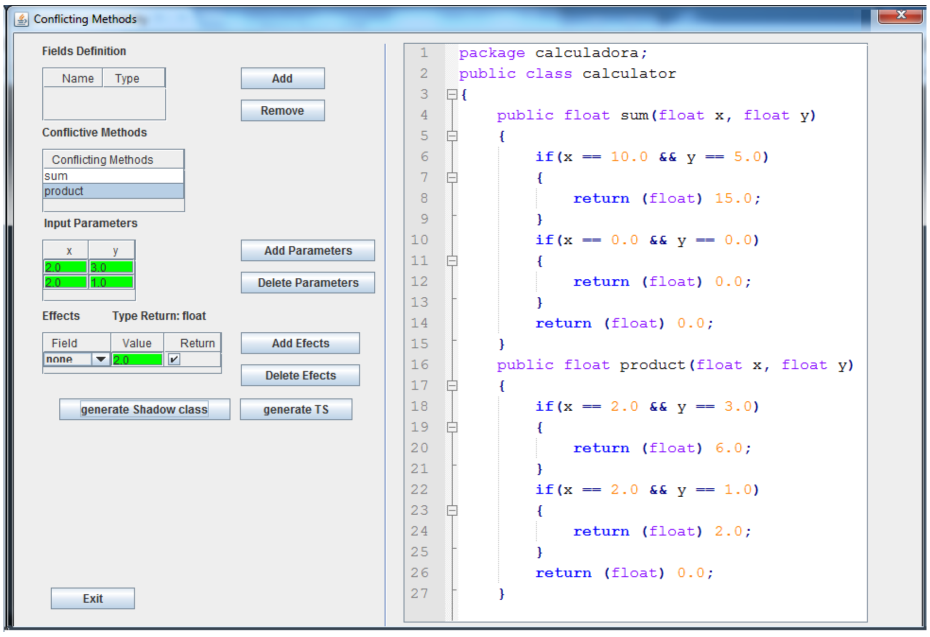
3.4 Shadow Class Generation

Through the shadow class, the test engineer can specify the expected behavior defined by each operation in the required interface I_R. The goal is to determine the validity of the results obtained during the execution of each test case into the Reduced TS. The generated shadow class is named as the required interface I_R, to act as its fair representative. For every *conflictive operation* identified from the required interface I_R, an operation with the same name is shaped into the shadow class.

The expected behavior for an operation is defined in terms of *causes* and *effects* that help to describe the input/output transformation or *functional mapping*. The cause-effect relationship raises four possible cases:

1. An operation may have an explicit cause defined by input parameters;
2. An operation may have an implicit cause without input parameters;
3. An operation may have an explicit effect defined by its return values;
4. An operation may have an implicit effect without return values. The effect could be changing a state variable or a constant value. For this, an *additional operation* with explicit effect may show such changing effect.

Example In the Calculator example (Figure 2) – all operations have an explicit cause. Figure 3 shows a facility from our tool support that assists a test engineer to define the values for parameters, return and fields of conflictive operations. Figure 3 also shows (on the right) the generated code of the shadow class according to the input values.



Name	Type

Conflicting Methods
sum
product

x	y
2.0	3.0
2.0	1.0

Field	Value	Return
none	2.0	✓

```java
package calculadora;
public class calculator
{
    public float sum(float x, float y)
    {
        if(x == 10.0 && y == 5.0)
        {
            return (float) 15.0;
        }
        if(x == 0.0 && y == 0.0)
        {
            return (float) 0.0;
        }
        return (float) 0.0;
    }
    public float product(float x, float y)
    {
        if(x == 2.0 && y == 3.0)
        {
            return (float) 6.0;
        }
        if(x == 2.0 && y == 1.0)
        {
            return (float) 2.0;
        }
        return (float) 0.0;
    }
```

Fig. 3. Test data and shadow class for the `Calculator` example

3.5 Reduced Test Generation

The exhaustive Behavioral TS has been particularly designed upon the *all-context dependence coverage criterion* – introduced in Section 2.3. It describes the interdependences among operations of the required interface I_R as operational sequences. In turn, operational sequences define an interaction protocol (or usage protocol), represented by regular expressions. In the context of test reduction, the interaction protocol is as a regular expression comprising only *conflictive operations* – i.e., the shadow class interface. Each sentence that can be derived from this regular expression corresponds to a *test template* describing a valid operational sequence to be tested. First, the class to be executed must be instantiated, so always the first operation of the operational sequence will be the constructor method of the required interface. Thus, test templates always begin with a call to the default constructor of the class. Then, the test template should call a conflictive operation and return its value.

Test templates and test data are then combined to generate test cases. *Test data* are representative input/output values for parameters and return of the required operations op_R that were defined by a test engineer to settle the *shadow* class. Test cases are generated in the specific format of MuJava [17], a framework for mutation testing. According to this format, test cases return a `String` value that will be compared against the expected result stored in the shadow class.

Example The `Calculator` example (Figure 2), presented the conflictive operations `sum` and `product`. Thus, a test engineer must define input data for those operations – in the form of float values to instantiate parameters – and output data for the given input – in the form of float values to instantiate the return.

The regular expression is automatically generated as:

`Calculator (sum|product)`

From where the test templates to derive are:

1. `Calculator sum`
2. `Calculator product`

Then, for the `sum` operation the *causes* define conditional compound predicates (if-then sentences), through the input data pairs (10.0, 5.0) and (0.0,0.0), while *effects* define expected return as output data 15.0 and 0.0 respectively. For the `product` operation, input data (causes) are pairs (2.0, 3.0) and (2.0,1.0), while expected results (effects) are 6.0 and 2.0 respectively.

The combination of the 2 *test templates* and 2 sets of *test data* generates 4 *test cases* into a test file called `MujavaCalculator` – being the Reduced Behavioral TS. Code Listing 1.1 shows one of the 4 test cases generated for the `Calculator` example. In particular, the shown test case instantiates the class under test using the constructor (line 3) and calls the conflictive operation `sum` with values 10.0 and 5.0 for the parameters (lines 4, 5 and 6). The result of the operation (line 7) will be compared against the expected result in the shadow class – 15.0 for this example (Figure 3, line 8 from the *shadow* class).

Listing 1.1. Sample Test Case for the `Calculator` example

```
1 public String testTS_0_1(){
2       calculadora.calculator obtained = null;
3       obtained = new calculadora.calculator();
4       float arg1 = (float) 10.0;5
5       float arg2 = (float) 5.0;
6       float result0 = obtained.sum(arg1, arg2);
7       return new Float(result0).toString();
8 }
```

3.6 Proof-of-concept: Behavioral Compatibility

As a final step, the Behavioral TS must be executed for a candidate Web Service S to then compare the results against those enclosed in the *shadow* class. As explained in Section 3.1, the execution is performed through a set of wrappers W that act as adapters to allow a seamless communication with the candidate Web Service. Then, the proper execution of the TS is performed using our tool support based upon the Mujava framework. Table 4 shows the results of executing the TS for each wrapper $w \in W$ where the *wrapper2* successfuly passed the

100% of the tests. This successful wrapper helps to confirm the compatibility of the candidate service `CalculatorService` as well as confirming the most adequate operation matching for conflictive operations – `sum` to `add` and `product` to `multiply`. Additionally, the *wrapper2* can be selected as an adapter for the safe integration of the candidate `CalculatorService` into the client application.

Table 4. Results of executing the Reduced TS for the `Calculator` example

Wrapper	**Test Cases**		
	Success	**Failure**	**Success %**
wrapper0	2	2	50
wrapper1	2	2	50
wrapper2	4	0	100
wrapper3	0	4	0
wrapper4	0	4	0
wrapper5	2	2	50
wrapper6	0	4	0
wrapper7	0	4	0
wrapper8	2	2	50

3.7 Discussion

As explained in Section 3.5, the exhaustive TS has been designed to satisfy certain coverage criteria for Integration Testing. However, the test reduction strategies directly affect the satisfiability of the different coverage criteria as follows.

– *All-Methods*: as the Reduced TS is narrowed to the conflictive operations subset, it is not possible to satisfy this criterion considering all the required operations. However, considering only operations included into the *shadow* class, then the criterion is satisfied, as every operation from the shadow class is covered by, at least, one test case.
– *All-Events*: requires executing both synchronous events (operations invocations) and asynchronous events (exceptions) – which covers the *exceptions criterion.* The reduced TS is not intended to test asynchronous events, particularly considering the scarce adoption of exceptions in the context of Web Services [6]. Finally, the events criterion is partially satisfied considering only operations in the *shadow* class.

– *Context dependence criterion*: this criterion is not satisfiable as the regular expression generated to build the Reduced TS does not include all operational sequences. However, it is satisfiable considering only *conflictive operations.*

Conclusively, although the Reduced TS satisfies only partially the coverage criterion defined for the exhaustive TS, it is possible to assume certain confidence in the obtained results. Moreover, the Reduced TS is optimized in terms of efficiency, reducing the number of software artifacts and executions needed, and thus reducing test times and costs. Particularly, for the `Calculator` example, the exhaustive TS comprised 33 test cases, while the Reduced TS comprises only 4 test cases. Both tests successfully identify the adequate wrapper that confirms the behavioral compatibility, also allowing to safely integrate the candidate Web Service in the client application.

4 Conclusion

In this paper we detailed specific test reduction techniques for Web Services selection and integration in organizational systems. Particularly, this work concerned compliance testing, which is intended not to find errors in services but to dynamically verify the behavior of a program (service) against the expected behavior. This is accomplished through a Behavioral Testing framework, in which we apply test reduction techniques. A dynamic and efficient evaluation of services is crucial to integrate them successfully at organizational level. Moreover, the adoption of the detailed techniques and processes facilitates:

– Monitoring and adaptation against service evolution
– Monitoring the quality of services offered by providers
– Detecting violations to Service Level Agreements (SLAs)
– Understanding the dynamics and elasticity of service providers

As for the daily work of testers in-the-trenches, test reduction expedite on-demand testing of new or updated Web Services, ensuring a similar coverage to any exhaustive test. These advantages can imply, in the middle or long term, the association or contract of service providers as operating partners, particularly in cases where the client organization produces commercial software for third-parties (COTS). This can also be of great interest to the service provider, if the client organization is ready to be used as a case study, mitigating the costs of hiring the services.

4.1 Future Work

Currently, we are working on testing service compositions [12]. This is particularly useful when a single service cannot provide all the required functionality. In this context, it is necessary to generate tests according to specifications in business process languages such as BPEL and BPML [21]. Finally, another interesting extension of this work is to derive automatically the reduced tests from system models – for example from models described in SoaML [19], an 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. A. Bertolino. *Guide to the software engineering body of knowledge-SWEBOK*, chapter 5. IEEE Press, 2001.
3. Robert Binder. *Testing object-oriented systems: models, patterns, and tools.* Addison-Wesley Professional, 2000.
4. G. Canfora and M. Di Penta. Testing Services and Service-Centric Systems: Challenges and Opportunities. *IT Professional*, 8(2):10–17, Mar./Apr. 2006.
5. Alejandra Cechich, Mario Piattini, and Antonio Vallecillo. *Component-based software quality: methods and techniques.* Springer Science & Business Media, 2003.
6. M. Crasso, J. M. Rodriguez, A. Zunino, and M. Campo. Revising WSDL Documents: Why and How. *IEEE Internet Computing*, 14(5):48–56, 2010.
7. 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.
8. M. Delamaro, J. Maidonado, and A. Mathur. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247, 2001.
9. M. Garriga, A. Flores, A. Cechich, and A. Zunino. Testing-based process for service-oriented applications. In *30th International Conference of the Chilean Computer Science Society (SCCC)*, pages 64–73, Nov 2011.
10. 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.
11. M. Garriga, A. Flores, C. Mateos, A. Zunino, and A. Cechich. Service selection based on a practical interface assessment scheme. *International Journal of Web and Grid Services*, 9(4):369–393, October 2013.
12. Martin Garriga, Andres Flores, Alejandra Cechich, and Alejandro Zunino. Web services composition mechanisms: A review. *IETE Technical Review*, In press, 2015.
13. S. Gosh and A. P. Mathur. Interface mutation. *Software Testing, Verification and Reliability*, 11:227–247, 2001.
14. M. Huhns and M. Singh. Service-Oriented Computing: Key Concepts and Principles. *IEEE Internet Computing*, 9(1):75–81, January-February 2005.
15. 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.
16. L. Kung-Kiu and W. Zheng. Software Component Models. *IEEE Transactions on Software Engineering*, 33(10):709–724, October 2007.

17. μJava Home Page. Mutation system for Java programs, 2008. http://www.cs.gmu.edu/ offutt/mujava/.
18. Ramesh Nagappan, Robert Skoczylas, and Rima Patel Sriganesh. *Developing Java web services: architecting and developing secure web services using Java*. John Wiley & Sons, 2003.
19. 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/.
20. M. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *IEEE Computer*, 40(11):38–45, November 2007.
21. 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.
22. Ye Wu, Dai Pan, and Mei-Hwa Chen. Techniques of maintaining evolving component-based software. In *Software Maintenance, 2000. Proceedings. International Conference on*, pages 236–246. IEEE, 2000.
23. Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.
24. O. Zimmerman, M. Tomlinson, and S. Peuser. *Perspectives on Web Services – Applying SOAP, WSDL and UDDI to Real-World Projects*. Springer-Verlag, 2005.