8

Web Services Regression Testing

Massimiliano Di Penta, Marcello Bruno, Gianpiero Esposito,
Valentina Mazza and Gerardo Canfora

RCOST — Research Centre on Software Technology — University of Sannio
Palazzo ex Poste, Via Traiano 82100 Benevento, Italy
{dipenta, marcello.bruno, gianpiero.esposito}@unisannio.it,
{valentina.mazza, canfora}@unisannio.it

Abstract. Service-oriented Architectures (SOA) introduce a major shift of perspective in software engineering: in contrast to components, services are used instead of being physically integrated. This leaves the user with no control over changes that can happen in the service itself. When the service evolves, the user may not be aware of the changes, and this can entail unexpected system failures.

When a system integrator discovers a service and starts to use it, she/he may need to periodically re-test it to build confidence that (i) the service delivers over the time the desired functionality and (ii) at the same time it is able to meet Quality of Service requirements. Test cases can be used as a form of contract between a provider and the system integrators. This chapter describes an approach and a tool to allow users to run a test suite against a service to discover if functional and non-functional expectations are maintained over time.

8.1 Introduction

A challenging issue for the verification and validation of service-oriented systems is the lack of control a system integrator has over the services she/he is using. System integrators select services to be integrated in their systems based on a mixture of functional and non-functional requirements. An underlying assumption is that the service will maintain its functional and non-functional characteristics while being used. However, behind any service there is a software system that undergoes maintenance and evolution activities. These can be due to the addition of new features, the evolution of the existing ones, or corrective maintenance to cope with problems that arise during the service usage.

Whilst the service evolution strategy is out of the system integrators control, any changes to a service may have an impact on all the systems using it. This is a relevant difference with respect to component-based development:

when a component evolves, this does not affect systems that use previous versions of the component itself. Component-based systems physically integrate a copy of the component and, despite the improvements or bug fixing performed in the new component release, systems can continue to use an old version.

Several types of changes may entail that a service does not satisfy anymore the requirements of an integrator. When the evolution activity does not require modifying the service interface and/or specification—e.g., because the provider believes this is a *minor* update—the change remains hidden from whoever is using the service. In other words, the system continues to use the service without being aware that its behavior, in correspondence with some inputs, may be different from the one exhibited previously. Evolution cannot only alter the service functional behavior, but can also affect its Quality of Service (QoS). While the current version of a service meets integrator non-functional requirements, future versions may not. Finally, when the service is, on its own, a composition of other services, the scenario may be even more complex. As a matter of fact, changes are propagated between different system integrators, and it happens that the distance between the change and the actor affected by the change makes unlikely that, even if the change is advertised, the integrator will be able to get it and react accordingly. To summarize, functional or non-functional changes can violate the *assumptions* the integrator made when she/he discovered the service and negotiated the Service Level Agreement (SLA).

This chapter describes a regression testing strategy that can be used to test whether or not, during its lifetime, a service is compliant to the behavior specified by test cases and QoS assertions the integrator downloaded when she/he discovered the service and negotiated the SLA. Similarly to what made for components [1, 2], test cases are published together with the service interface as a part of its specification. In addition, they can be complemented by further test cases produced by the system integrator, as well as by monitoring data, to form a sort of *executable contract*, which may or may not be part of the legal contract. During the service lifetime, the integrator can run the test suite against the (possibly new versions of the) service. If some test cases or QoS assertions fail, the contract has been violated.

A relevant issue, that makes service testing different from component testing, is the cost of such a test. Test case execution requires service invocations, that are supposed to have a cost, and a massive testing can consume provider resources or even cause denial of service. Both provider and integrator, therefore, may want to limit actual service execution during testing. To this aim, this chapter explains how monitoring data can be used to reduce the number of service invocations when executing a test suite.

The proposed service regression testing approach is supported by a toolkit, described in the chapter. The toolkit comprises a *Testing Facet Generator*, that analyzes unit test suites (e.g., JUnit test suites) produced by the service developer/provider and generates XML-encoded test suites and QoS

assertions that can be executed by service consumers, who do not have access to the service internal implementation but only to the service interface. Such test suites and QoS assertions will be one of the *facets* composing the whole service specification.[1] Another component of the toolkit, the *Test Suite Runner*, permits the downloading and the execution of the test suites against the service. Finally, the tool manages test logs and provides a capture/replay feature.

The chapter is organized as follows. Section 8.2 motivates the approach, describing the different service evolution scenarios that can result in a need for re-testing the service, discussing the different stakeholders that can be involved in service regression testing, and finally describing the running example used to describe the approach. Section 8.3 presents the regression testing approach through its different phases and the related tool support, also discussing issues and open problems. Section 8.4 presents case studies carried out to evaluate different aspects of the approach. After a discussion of the related literature in Sect. 8.5, Sect. 8.6 concludes the chapter.

8.2 Motivations and Testing Scenarios

This section motivates the need for service regression testing. It firstly describes how services can evolve and to what extent this can have an impact on systems using them. Then, it discusses the different perspectives from which a service can be tested and what makes service regression testing different from component regression testing. Finally, it presents some motivating examples that will be used to explain the testing approach.

8.2.1 Evolution Scenarios in SOA

Let us imagine that a system integrator has discovered the release r_n of a service and, after having negotiated the SLA with the service provider, starts to use it. At release r_{n+k} the service has evolved, and different scenarios may arise:

- Change in the service functional behavior: At release r_{n+k} the service may behave differently from release r_n. If the integrator negotiated the SLA at r_n, the new, unexpected behavior may cause failures in the system. This happens when the service, at release r_{n+k}, replies to a given set of inputs differently from release r_n, or it handles exceptions differently. For example, release r_{n+k} of a search hotel service may return an unbounded list of available hotels, while r_n only returned a single results.

[1] To enrich the service specification available within the WSDL interface, one could hyperlink other files, e.g., specifying the semantics, the QoS, or containing test cases.

- Change in the service non-functional behavior: This can be due to changes in the service implementation—which may or may not alter the service functional behavior—as well as to changes in the provider hardware, in the network configuration, or in any other part of the environment where the service is executed. For example, the *throughput* may decrease or the response time may increase, causing violations of the contract stipulated between the integrator and the provider.
- Change in the service composition/bindings: A service may, on its own, be composed of other services. It may happen that the composite service owner changes some bindings. For example, let us suppose that an image processing service (S_1) uses another service (S_2) for filtering the image. In particular, S_2 is able to ensure a given image resolution. It can happen that, since S_2 is not available anymore, S_1 re-binds its request to an equivalent image filtering service, S_2' which, however, is not able to ensure the same resolution anymore. As a result, S_1 users will obtain an image having a lower resolution without being aware of what actually happened behind S_1 interface.

The aforementioned scenarios may or may not be reflected in changes visible in the service interface/specification. If the interface does not change, the provider may decide to update the service without advertising the changes. In other cases, the interface update does not necessarily reflect changes made to the implementation. For example, the service interface indicates that a new release of the service has been deployed at a given date. However, since nothing has changed in the service specification nor in any operation input/output parameters, the integrators will continue to invoke the service without verifying whether the new release is still compliant with the assumption underlying their system. In summary, even if providers are encouraged to update service specifications/interfaces when the service evolves, there is no guarantee they will actually do it properly whenever needed.

This urges the need to provide system integrators with a way to test the service, either periodically or when they are aware that something has changed. This form of regression testing can be used to ensure that the functional and non-functional behavior is still compliant with the one observed when negotiating the SLA.

8.2.2 Service Testing Perspectives

Whilst the introduction motivates the need for service regression testing from a system integrator's point of view, there are different stakeholder interested to make sure that a service, during its lifetime, preserves its original behavior. Similarly to what Harrold *et al.* [3] defined for components, it is possible to foresee different service testing perspectives [4]:

1. Provider/developer perspective: The service developer would periodically check whether the service, after its maintenance/evolution, is still

compliant to the contract stipulated with the customers. To avoid affecting service performance, testing can be performed off-line, possibly on a separate instance (i.e., not the one deployed) of the service and on a separate machine. The cost of testing is therefore limited (no need for paying service invocation, no waste of resources). On the other hand, developer's inputs may not be representative of system integrator scenarios, and the non-functional testing does not necessarily reflect the environment where the service will be used.

2. System integrator's perspective: On his/her side, the system integrator may periodically want to re-test the service to ensure that its evolution, or even changes in the underlying software/hardware, does not alter the functional and non-functional behavior so to violate the assumptions she/he made when starting to use the service. Testing from this perspective is more realistic, since it better reflects integrator's scenarios and software/hardware configuration. On the other hand, as discussed in Sect. 8.3.5, testing from this side is a cost for the integrator and a waste of resources for the provider, raising the need for countermeasures.
3. Third-party/certifier perspective: A certifier has the responsibility of testing the service on behalf of another stakeholder, which can be either a service provider or one or more system integrators. The provider can rely on a certifier as a mean to guarantee the service reliability and performance to potential service users (e.g., integrators). Testing from this perspective has weaknesses for both the provider and the integrator perspective: the testing scenario and the configuration under which the service is tested may not fully reflect the environment where the service will actually work. As for system integrators, testing from this perspective has a cost and consumes provider's resources, although having a single certifier is certainly an improvement over having each integrator testing the service.
4. User perspective: As described by Canfora and Di Penta [4], the user might also be interested to have a mechanism which periodically re-tests the service his/her application is using. Let us imagine the onboard computer software installed in John's car. Such application communicates with some services (see Sect. 8.2.4) that, over the time, might vary their behavior, causing problems to the onboard computer software. For this reason, an automatic (the user is not a tester and would be unaware of such a detail) regression testing feature from the user side is highly desirable. The limitations and weaknesses are the same as for a service integrator.

8.2.3 What Makes Services Different from Components?

The above section explained the need for regression testing in service-oriented systems, while highlighting several commonalities with component-based software. In the authors' experience, the main differences with components, that need to properly adapt the approach, are, among others

- the lack of control the integrator/user has on the service evolution, and on the way the service, on its own, provides a piece of functionality by using other, dynamically bound, services;
- the testing cost, both for the tester and for the service provider;
- the key role played by the QoS: even if QoS is also relevant for component-based systems, in service-oriented computing it is used to determine binding choices and to assess whether a service provider is able to meet what stipulated with the service consumer in the SLA. Furthermore, the need for QoS testing is also due to the highly distributed nature of service-oriented systems that may cause huge variations in QoS values or even service unavailability.

8.2.4 Regression Testing Scenarios

To better explain the approach, let us consider the scenario described in the Chap. 1. When John searches for a restaurant in a given location, this search is made through a complex system that takes as inputs

1. the current latitude and longitude;
2. the maximum distance allowed;
3. the arrival date and hour;
4. the number of seats requested.

The system, among other services—e.g., services for computing the routing distance between two locations—accesses a third party service, *RestaurantService*, that provides five operations:

1. *getRestaurantID*, which, given the restaurant's name, the city name and the address returns the related ID composed of four decimal digits.
2. *getRestaurantInfo*, which returns an information record of the restaurant (i.e., address, location expressed in GPS coordinates, etc.).
3. *checkRestaurantAvailability*, which, given a list of restaurant IDs, the date and the number of seats requested, returns an array of availabilities.
4. *restaurantReservation*, which reserves a specified number of seats in the restaurant for a given date and hour.
5. *getRestaurantList*, which, given a city name, returns a list of up to three restaurants from that city.

Let us imagine now that the service undergoes a series of maintenance activities. Some of them have been inspired from maintenance/evolution activity actually carried out over the Amazon Web service and documented in its release notes[2]:

[2] http://developer.amazonwebservices.com/ecs/resources

1. The comma-separated list of restaurant IDs as parameter for the *checkRestaurantAvailability* operation is no longer supported. This means that, similar to the Amazon service, whilst the interface does not change, only a single ID is accepted.
2. The list returned by *getRestaurantList* is now unbounded, while in the previous release it contained at most three *restaurantInfo* objects.
3. The restaurant ID format changes, due to the increasing number of restaurants handled. The new ID is composed of five digits, rather than the original four digits.

8.3 The Approach

The previous section identified the need for a system integrator, as well as for other stakeholders, to perform service testing with the aim of ensuring that the service meets his/her functional and non-functional expectations. To this aim, it would be useful that the providers make available to the system integrator test cases she/he can use to regression test the service during its lifetime.

Since test suites are, very often, created during early stages of the software system development, it would be useful to reuse them to permit the testing of services that expose pieces of functionality of the system itself. However, since such test suites access system's internal resources not visible from outside, they must be adequately transformed so that they can be executed from the perspective of a service integrator, which has access only to the service interface.

8.3.1 Service regression testing process

Figure 8.1 describes a possible scenario for the test case publication and regression testing process. The scenario involves both a service provider (*Jim*) and two system integrators (*Alice* and *Jane*), and explains the capabilities of the proposed regression testing approach.

1. At time t_0 *Jim* deploys the *RestaurantService* service together with a test suite.
2. At time t_1 *Alice* discovers the service, negotiates the SLA and downloads the test suite; she can complement the test suite with her own test cases, perform a pre-execution of the test suite, and measure the service non-functional behavior. A SLA is agreed with the provider, and Alice stores both the test suite and the QoS assertions generated during the pre-execution.
3. Then, *Alice* regularly uses the service, until,
4. after a while, Jim updates the service. In the new version the *ID* return value for *getRestaurantID* is composed of five digits instead of four. Also,

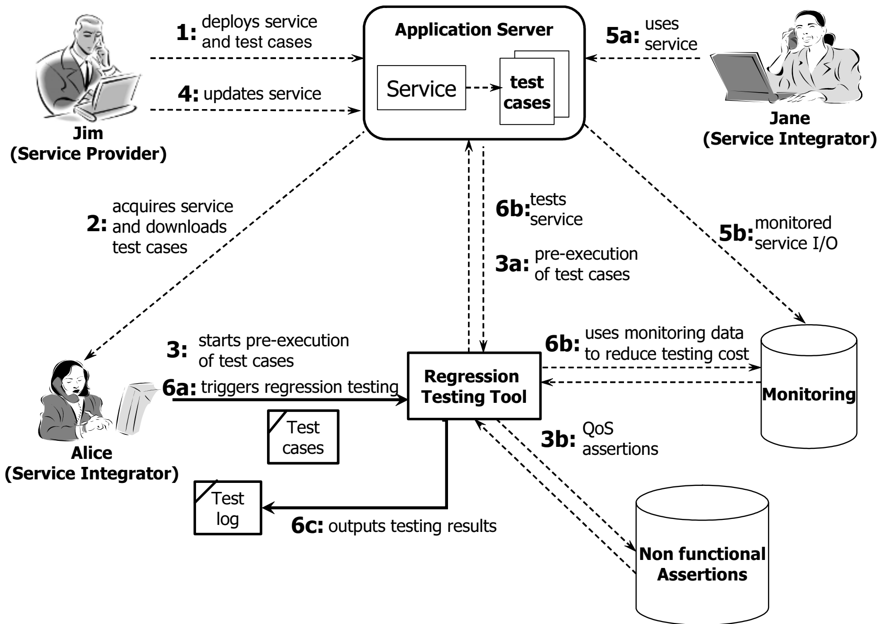


Fig. 8.1. Test generation and execution process

because of some changes in its configuration, the modified service is not able to answer in less than two seconds.

5. *Jane* regularly uses the new service with no problems. In fact, she uses a field that is large enough for visualizing a restaurant ID composed of five digits. Meanwhile, *Jane*'s interactions are monitored.
6. Since the service has changed, *Alice* decides to test it: data monitored from *Jane*'s executions can be used to reduce the number of service invocations during testing. A test log containing successes and failures for both functional test cases and QoS assertions is reported. For example, test cases expecting a restaurant ID composed of four digits will fail. The non-functional assertions that expect a response time less or equal than two seconds for *getRestaurantID* will also fail.

Test case publication and regression testing is supported by a toolkit, developed in Java, comprising two different tools:

1. The *Testing Facet Generator* that generates service test cases from test suites developed for the software system implementing the features exposed by the service. In the current implementation, the tool accepts JUnit[3] test suites, although it can be extended to accept unit test suites

[3] http://www.junit.org/

developed using different frameworks available for other programming languages (such as *SUnit* for Smalltalk or *PHPUnit* for PHP). JUnit supports the development of a unit test suite as a Java class, containing a series of methods that constitute the test cases. Each test case, in its turn, is composed of a sequence of assertions checking properties of the class under test.
2. The *Test Suite Runner* that permits the service consumer to
   - download the testing facet hyperlinked to the service;
   - generate QoS assertions by pre-executing the test suite;
   - execute the test suite and to produce the test log;
   - support capture/replay operations.

The toolkit relies on JavaCC[4] to perform Java source code analysis and transformation, on the Axis Web services framework[5] and on the Xerces[6] XML parser. The toolkit is freely available[7] and distributed under a BSD license.

The next subsections describe the different phases of the test case generation and execution process.

8.3.2 Testing facet generator

As shown in Fig. 8.2, the *Testing Facet Generator* produces a XML-encoded testing facet organized in two levels.

1. The **first level** contains
   - A *Facet Description*, providing general information such as the facet owner and creation date.
   - A *Test Specification Data*, containing information such as the type of assertions contained in the test suites (functional and/or non-functional), the number of test cases composing each test suite and the perspective from which QoS assertions were generated (i.e., provider or integrator).
   - *Links* to XML files containing the test suite itself and QoS assertions.
   - Some *Policies*, i.e., constraints that limit the number of operations that can be invoked during a test in a given period of time. For example, the facet in Fig. 8.2 defines the limitation of three operation invocations per day.
2. The **second level** comprises files containing XML-encoded test suites and QoS-assertions (the file `testRestaurant.xml` in our example).

[4] https://javacc.dev.java.net/
[5] http://xml.apache.org/axis/
[6] http://xml.apache.org/xerces2-j/
[7] http://www.rcost.unisannio.it/mdipenta/Testing.zip

```xml
<LanguageSpecificSpecification>
  <FacetType>Testing</FacetType>
  <ReferencedOntology></ReferencedOntology>
  <ReferencedSIM ></ReferencedSIM>
  <FacetSpecificationLanguage>XML</FacetSpecificationLanguage>
  <FacetSpecificationOwner>Sannio</FacetSpecificationOwner>
  <FacetSpecificationLastEdited>
        Mon Sep 04 12:17:16 CEST 2006
  </FacetSpecificationLastEdited>
  <FacetSpecificationData>
  <TestingSpec>
    <Perspective>Provider</Perspective>
    <Type>NonFunctionalANDFunctional</Type>
        <Description>This facet has been generated</Description>
    <TestSuite>
      <NumberOfTestCases>7</NumberOfTestCases>
      <NonFunctionalANDFunctional>
        <XMLlink>http://127.0.0.1:8080/XMLfile/testRestaurant.xml</XMLlink>
      </NonFunctionalANDFunctional>
      <TestPolicy>
        <NumberOfTestExecutions freq="daily">3</NumberOfTestExecutions>
        <ExceptionFacetViolationLink></ExceptionFacetViolationLink>
      </TestPolicy>
    </TestSuite>
  </TestingSpec>
  </FacetSpecificationData>
</LanguageSpecificSpecification>
```

Fig. 8.2. Structure of testing facet

Generating Service Test Cases from JUnit Test Suites

As mentioned before, the service test cases are obtained by analyzing and transforming test suites — implemented e.g., using JUnit — that the developer has produced for the system implementing the service's features. These test suites are very often available, and many software development methodologies, e.g., test-driven development, strongly encourage developers to produce them, even before implementing the system itself.

However, although these test suites are available, they cannot be directly used by a service integrator to test the service. This because assertions contained in the JUnit test cases can involve expressions composed of variables containing references to local objects and, in general, access to resources that are only visible outside the service interface. Instead, a test suite to be executed from a system integrator can only interact with the service operations. This requires that any expression part of a JUnit assertion, except invocations to service operations and Java static methods (e.g., methods of the `Math` class), needs to be evaluated and translated into a literal, by executing an instrumented version of the JUnit test class from the server-side. The obtained dynamic information is then complemented with test suite static analysis to generate service test cases. Such test cases, as any other piece of information describing the service, are XML-encoded and, to be executed, only require access to service operation, and not to any service internal resource.

The process of generating service test cases from JUnit test suites can be completely automatic, or user-guided. In the first case, the JUnit test suite is translated so that operation invocations are left symbolic, whilst other expressions are evaluated and translated into literals. In the second case, the

tool user can guide the transformation. The tool shows to the user the list of test cases contained in the test suite (*Choose test cases* window in the screenshot of Fig. 8.3). The user can select the JUnit test cases that should be considered to generate service test cases. For the selected test cases, the user can select (from the *Select analysis* window) two different options:

1. Default analysis: The tool automatically translates any expression, except service operation invocations, in literals and generates the service test suite;

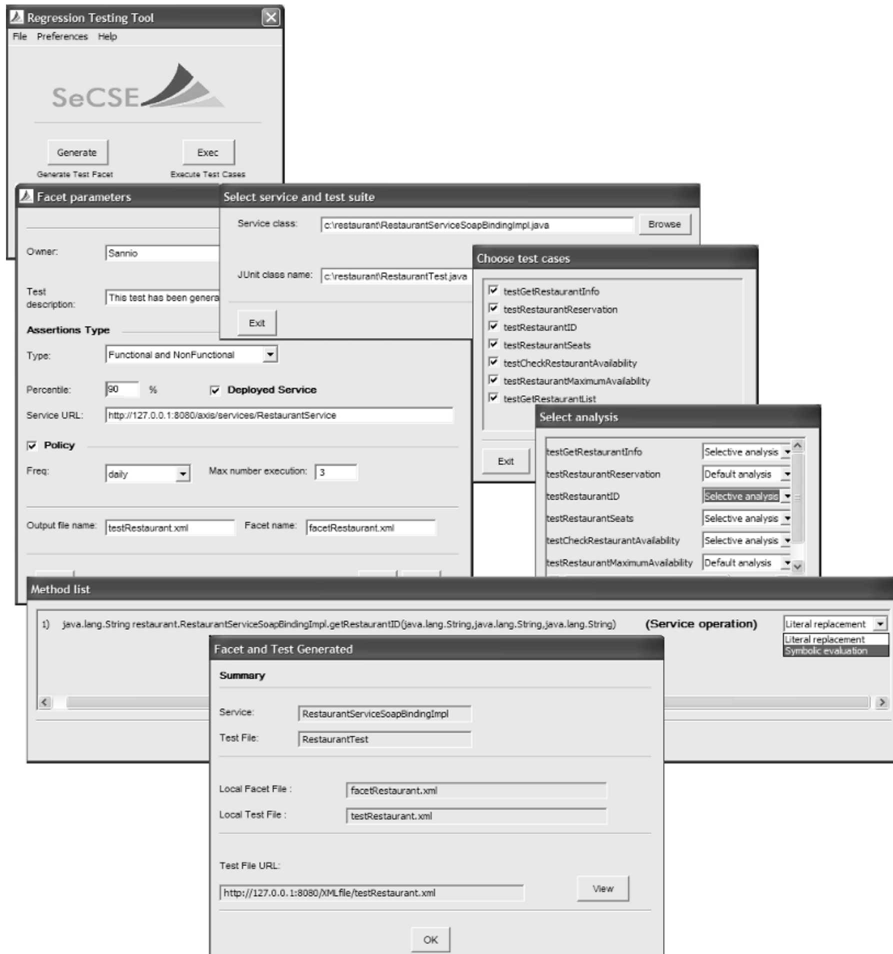


Fig. 8.3. Facet generation tool

2. Selective analysis: The user can select which method invocations, corresponding to service operations, should be evaluated and translated into literals, and which should be left symbolic in the testing facet.

Figure 8.4 shows an example of how a JUnit test case is mapped onto a XML-encoded service test suite. The first assertion checks whether the operation *getRestaurantID* returns a valid ID, i.e., a sequence of four digits. The upper part of the figure shows the JUnit test case, while the lower part shows how the two assertions are mapped onto the service test suite. Note that each functional assertion is followed by a QoS-assertion, which is checked against the QoS values monitored when executing the assertion. As shown, some assertion parameters appear as literals. For the first assertion, they were already literal in the JUnit test suite. However, it can happen that a literal value contained in the service test suite results from the evaluation of an expression contained in the JUnit test case. The service test suite also contains some symbolic parameters. These are Java static methods, e.g., *Pattern.matches*, that can be invoked from the regression testing tool without the need for accessing the service implementation and service operations, e.g., *getRestaurantID*. The second assertion checks whether the *restaurantReservation* returns an error output when someone attempts to book a table in the past.

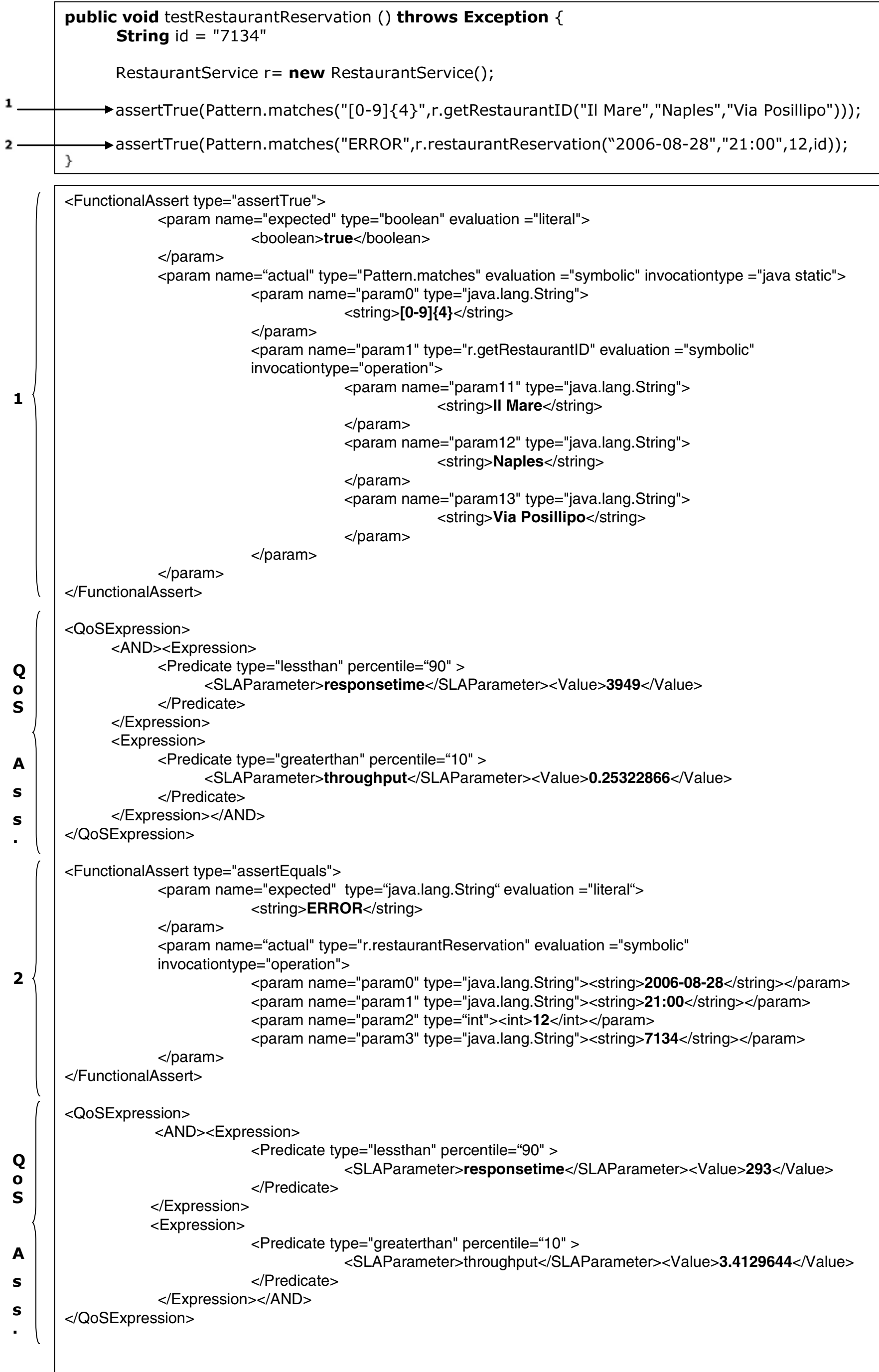
Generating QoS Assertions

Assertions over QoS attributes are used to check whether the service, during its evolution, is able to preserve its non-functional behavior, in compliance with SLA stipulated by service consumers. These assertions are automatically generated by executing test cases against the deployed service, and measuring the QoS attributes by means of a monitor. Test cases are executed against the service for a large, specified number of times and QoS values (e.g., *response time*) measured. Given the QoS value distribution, a constraint can be fixed as

$$ResponseTime < p_i$$

where p_i is the i_{th} percentile of the *response time* distribution as measured when the service was discovered and the SLA negotiated. Both the facet generation tool and the service regression testing tool have a feature for generating the QoS assertions, after having specified how the assertion shall be generated, i.e., how many executions are necessary to compute the average QoS value and which would be the percentile to be used to define the boundary.

QoS assertions are XML-encoded within the test suite using a format similar to those defined by the WSLA schema [5]. An example of QoS assertion for the *getRestaurantID* operation is shown in Fig. 8.4. The example indicates that, when executing the *getRestaurantID* operation (part of the functional assertion), the response time must be less than 3949 ms, which is the 90 percentile of the response time distribution estimated when generating the



```
public void testRestaurantReservation () throws Exception {
        String id = "7134"

        RestaurantService r= new RestaurantService();

1 ——> assertTrue(Pattern.matches("[0-9]{4}",r.getRestaurantID("Il Mare","Naples","Via Posillipo")));

2 ——> assertTrue(Pattern.matches("ERROR",r.restaurantReservation("2006-08-28","21:00",12,id));
}
```

```
1
<FunctionalAssert type="assertTrue">
        <param name="expected" type="boolean" evaluation ="literal">
                <boolean>true</boolean>
        </param>
        <param name="actual" type="Pattern.matches" evaluation ="symbolic" invocationtype ="java static">
                <param name="param0" type="java.lang.String">
                        <string>[0-9]{4}</string>
                </param>
                <param name="param1" type="r.getRestaurantID" evaluation ="symbolic"
                invocationtype="operation">
                        <param name="param11" type="java.lang.String">
                                <string>Il Mare</string>
                        </param>
                        <param name="param12" type="java.lang.String">
                                <string>Naples</string>
                        </param>
                        <param name="param13" type="java.lang.String">
                                <string>Via Posillipo</string>
                        </param>
                </param>
        </param>
</FunctionalAssert>

QoS Ass.
<QoSExpression>
    <AND><Expression>
        <Predicate type="lessthan" percentile="90" >
            <SLAParameter>responsetime</SLAParameter><Value>3949</Value>
        </Predicate>
    </Expression>
    <Expression>
        <Predicate type="greaterthan" percentile="10" >
            <SLAParameter>throughput</SLAParameter><Value>0.25322866</Value>
        </Predicate>
    </Expression></AND>
</QoSExpression>

2
<FunctionalAssert type="assertEquals">
        <param name="expected"  type="java.lang.String" evaluation ="literal">
                <string>ERROR</string>
        </param>
        <param name="actual" type="r.restaurantReservation" evaluation ="symbolic"
        invocationtype="operation">
                <param name="param0" type="java.lang.String"><string>2006-08-28</string></param>
                <param name="param1" type="java.lang.String"><string>21:00</string></param>
                <param name="param2" type="int"><int>12</int></param>
                <param name="param3" type="java.lang.String"><string>7134</string></param>
        </param>
</FunctionalAssert>

QoS Ass.
<QoSExpression>
        <AND><Expression>
                <Predicate type="lessthan" percentile="90" >
                        <SLAParameter>responsetime</SLAParameter><Value>293</Value>
                </Predicate>
        </Expression>
        <Expression>
                <Predicate type="greaterthan" percentile="10" >
                        <SLAParameter>throughput</SLAParameter><Value>3.4129644</Value>
                </Predicate>
        </Expression></AND>
</QoSExpression>
```

Fig. 8.4. Mapping of a JUnit test case onto a XML-encoded service test case

QoS assertion. While a SLA document expresses general QoS constraints[8] (e.g., *"Throughput > 1 Mbps"* or *"Average response time < 1 ms"*), QoS assertions indicate the expected service performance in correspondence with a given set of inputs (specified in the test case). For example, the assertion in figure indicates what is the maximum response time permitted when invoking the *getRestaurantID* operation.

As an alternative to using of QoS assertions, the service non-functional behavior can be checked against the SLA. However, while the assertions are used to check the QoS achieved for each test case, SLA can only be used to check aggregate QoS values (e.g., the average, the minimum, or the maximum against all test case executions).

An important issue to be discussed is who should generate QoS assertions. The provider can generate QoS assertions when deploying the service. These assertions will reflect the service QoS (e.g., response time or throughput) that only depends on the service behavior (e.g., an image compression service will respond slowly when the input is a large image) and on the provider's machine configuration. However, different integrators may experience response times having a large variability from those generated by the provider. To overcome this limitation, a system integrator can generate his/her own assertions, measuring the QoS expected in correspondence with the given inputs (specified by test cases) within a more realistic configuration.

8.3.3 Test Cases as a Contract

Test cases and QoS assertions constitute a kind of *executable contract* between the system integrator and the service provider. When executing the test cases, the integrator can observe the service's functional and non-functional behavior. If satisfied, she/he stipulates the contract. The provider, on his/her own, agrees to guarantee such a behavior over a specified period of time, regardless of any change that would be made to the service implementation in the future. If, during that period, the service evolves — i.e., a new release is deployed — deviations from the agreed behavior would cause a contract violation.

When a service has been found, the system integrator can download the test suite published as a part of the service specification. Since the system integrator may or may not trust the test suite deployed by the provider, she/he can complement it with further test cases, also to better reflect the intended service usage scenario. In a semi-structured environment — e.g., a service registry of a large organization — the system integrator can publish the new test suite, so that other integrators can reuse it. On the contrary, this may not be possible in an open environment, where the additional test suite is stored by the system integrator, and only serves to check whether future service releases still satisfy his/her requirements.

[8] That must hold for any service usage.

The decision on whether the integrator has to add further test cases may be based on the analysis of the provider's test suite (e.g., characterizing the range of inputs covered) and from the test strategy used by the provider to generate such a test suite — e.g., the functional coverage criterion used — also advertised in the testing facet. The trustability level of the provider's test suite can be assessed, for instance, by analyzing the domain in which the service inputs vary and the functional coverage criteria adopted.

8.3.4 Performing Service Regression Testing

Once the system integrator has downloaded the test suite and has generated the QoS assertions, she/he can use them to perform service regression testing. When regression testing starts, service operations contained in the test suite are invoked, and assertions are evaluated. A test log is generated, indicating, for each test case, (i) whether the test case has passed or failed and (ii) the differences between the expected and actual QoS values. Also in this case, the QoS monitoring is used to measure actual QoS values, thus permitting the evaluation of QoS assertions.

Figure 8.5 shows a screenshot of the *test suite runner*. After specifying the service URL and selecting a test facet from a local file or from a remote URL, it is possible to run the test cases against the service, selecting whether someone wants to perform only functional check, only non-functional, or both. Progress bars report the test cases that have been passed and failed, both for the functional and for the non-functional parts. Also, a progress bar indicates the percentage of operation invocations that were *avoided* because reuse was made through monitoring data. This would let the tester figuring out to which extent the use of monitoring data permits to reduce the service testing cost (further details will be provided in Sect. 8.3.5). After the execution has been completed, it is possible to analyze the test execution results from a XML-encoded log, or by browsing a table reporting summary results for each test case executed.

When Regression Testing Needs to be Performed

The lack of control system integrators have over services poses critical issues on when service regression testing should be performed.

- *Triggered by service versioning:* If the service specification provides information on when the service was changed, the integrator can check such an information and launch regression testing. For example, the WSDL can contain service versioning information, or the service deployment date. Nevertheless, this kind of information cannot be completely trusted: the service implementation can, in fact, change without the need for a service re-deployment.
- *Periodic re-testing:* The tool permits to automatically launch regression testing periodically.

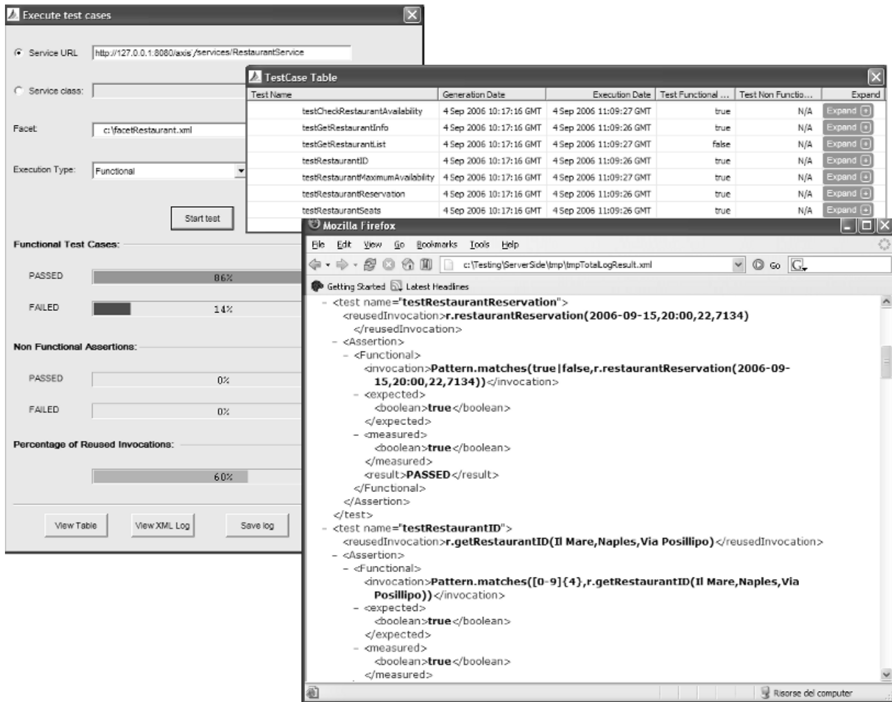


Fig. 8.5. Test suite runner

- *Whenever the service needs to be used:* This option is the most expensive; however, it may be required when high reliability is needed. In this case, the service should be re-tested before each execution. This, however, does not provide an absolute confidence on the fact that, if the test suite does not reveal any failure at time t_x, the same condition will be held at time $t_x + \delta$, where δ is the time interval between the testing and the subsequent service usage.

Finally, it is important to point out that service regression testing is not the only testing activity an integrator should perform. As discussed by Canfora and Di Penta [4], she/he should also perform integration testing between the system and the services being integrated.

8.3.5 Minimizing Service Invocations by Using Monitoring Data

A critical issue of service testing is cost. Test suite execution requires a number of service invocations that, in most cases, have a cost. In other words, the provider charges the service consumer when she/he invokes the service, even if the invocation is done for testing purposes. Also, a testing activity is generally undesired for a provider because it wastes resources reserved for

production service usage. The number of service invocations needed for service regression testing should be, therefore, limited as much as possible. First, the test suite itself needs to be minimized. To this aim, whoever generates a service regression test suite — i.e., both the provider or the integrator — can use one of the several existing regression test suite reduction techniques (see Sect. 8.5). In addition, assuming that service executions are monitored, monitoring data can be used to mimic service behavior and, therefore, avoid (or at least reduce) service invocations during regression testing.

To explain how this can happen, let us recall the scenario explained in Sect. 8.3.1 and depicted in Fig. 8.1. After *Jim* has updated the service at time t_1, *Jane* uses it without experiencing any problem. After a while, *Alice* wants to use the service. She realizes that the service has changed (because, e.g., the versioning info is reported in the service interface) and decides to re-test it. When executing the test suite, some of the inputs can be part of *Jane*'s executions after t_1. For example, if *Alice*'s test suite contains an assertion to check that the *getRestaurantID* operation returns a correct restaurant ID, this result can be reused when *Jane*'s test suite is executed, thus avoiding to actually invoke the *getRestaurantID* operation. In other words, monitoring data can be used to mimic the service behavior.

For security reasons, however, testers should not be allowed to access monitoring data. This, in fact, could issue serious non-disclosure problems. Especially when services are used over the Internet, one would avoid to have other people looking at his/her own service invocation data. To overcome such a risk, in the proposed approach the (server side) monitor supports the possibility to check assertions sent by the client-side testing tool, as a way of mimicking the service behavior.

The usage of monitoring data to reduce the testing cost is feasible if the relationship between service I/O is deterministic, i.e., different service invocations with the same inputs always produce the same output. If this is not the case, it can be possible to overcome such a limitation by checking that the service output matches a pattern or belongs to a given domain, instead of performing an exact match.

A further possibility for reducing the testing cost is to provide the service with a testing interface. Such an interface uses monitoring data (if they are available) to answer a service request, otherwise it directly accesses the service. Whilst this solution still requires service invocation, it will certainly reduce the usage of server resources, due to the execution of the service implementation, on the provider side.

8.3.6 Capture/Replay

A useful feature that the proposed regression testing tool makes available is the possibility to perform capture/replay. Similar approaches have been used for GUI testing [6] and for Web application testing [7]. During the service usage, I/O data is *captured* and stored within a monitoring database. In our

implementation, monitoring is performed by a plug-in installed behind the Axis application server, thus supported by the service provider. Nevertheless, alternative solutions, e.g., sniffing SOAP messages from client side, are also viable and have the advantage of being applicable for any service, even if the provider does not support monitoring.

When a service evolves, the tester can decide to re-test the service by *replaying* the inputs. Test case success or failure can be determined either by doing an exact match between previously monitored outputs and actual outputs or by performing a *weaker* check over the assertions, e.g., by checking that, in correspondence with a given input, the output still belongs to a given domain. The user can select the date interval from which captured data shall be taken. Then, when replay is being performed, the progress bar shows the percentage of test cases that failed. Finally, as for regression testing, it is possible to open a window where a detailed test log can be browsed.

8.3.7 Issues and Limitations

Service testing activities require to perform service invocation. In many cases, this can produce side effects, i.e., a change of state in the service environment. For example, testing a hotel booking service operation (like the *restaurantReservation* in our motivating example) can produce a series of unwanted room reservations, and it can be even worse when testing a book purchasing service. While a component can be tested in isolation, this is not possible for a service when the testing activity is carried out from the system integrator's perspective. In other words, the approach is perfectly viable for services that do not produce side effects in the real world. This is the case, e.g., of services performing computations, e.g., image compressing, DNA microarray processing, or any scientific calculus. For services producing side effects, the approach is still feasible from the provider's perspective, after having isolated the service from its environment (e.g., databases), or even from the integrator's side if the provider exports operations to allow integrators to test the service in isolation.

Despite the effort made to limit it, another important issue from integrator's perspective remains testing cost [4]. This is particularly true if the service has not got a fixed fee (e.g., a monthly usage fee) while the price depends on the actual number of invocations. From a different perspective, testing can have a high cost from the provider, when the service is highly resource-demanding.

The dependency of some service non-functional properties (e.g., response time) from the configuration where the service is used poses issues on the service non-functional testing. To this aim, the integrator can generate some non-functional assertion, by executing the test suite against the service and monitoring the QoS. However, monitoring data depends on the current configuration (server machine and load, network bandwidth and load, etc.). While averaging over several measures can mitigate the effect of network/server load

at a given time, changes in network or machines may lead to completely different QoS values. Clearly, the way our toolkit computes QoS distribution estimates can be biased by network or server loads, although such an effect can be mitigated by sampling the response time over a large set of service executions. More sophisticated QoS estimation approaches are available in the literature, accounting for the server load [8], the HTTP protocol parameters [9] and, in general, to the network and server status [10, 11]. While such kind of QoS estimates are not implemented in our toolkit at the time of writing, their adoption would, in the future, make the QoS testing less dependent on the network/server configuration and load.

Moreover, in case the service to be tested is a composite service and dynamic binding mechanisms hold, it may still happen that the bindings at testing time are different from these that could be determined when using the service. As a consequence, the QoS testing may or may not be able to identify QoS constraint violations due to binding changes.

Finally, as also mentioned in Sect. 8.3.5, non-determinism can limit the possibility of using assertions to check service I/O. Many services do not always produce the same response when invoked different times using the same inputs. This is the case, e.g., of a service returning the temperature of a given city. However, this issue can be addressed by replacing a strong assertion — e.g., $temperature = 12.5^o\ C$ — with a weaker one, e.g., $-40^o\ C < temperature < 50^o\ C$.

8.4 Assessing the Approach

This section presents two studies that have been carried out to assess the usefulness of the approach. The first study aims to investigate to what extent a test suite can be used to check whether the evolution of service would have affected its functional and non-functional behavior. The second study shows how monitoring data has been used to reduce the number of service invocations — and therefore the testing cost — during regression testing.

8.4.1 Study I: Assessing the Service Compliance Across Releases

Due to the lack of availability of multiple releases of real services, we wrapped five releases of an open source system, i.e., *dnsjava*, as Web services. *dnsjava*[9] is a Domain Name System (DNS) client and server; in particular, we focused our attention on *dig* (domain information groper), a utility used to gather information from DNS servers. The service under test is not a real service;

[9] http://www.dnsjava.org/

however, it well reflects the evolution that any DNS existing service[10] could have undergone.

The Web service has five input parameters: the domain to be solved (mandatory), the server used to solve the domain, the query type, the query class, and an option switch. The service answers with two strings: the query sent to the DNS and the DNS answer. We carefully checked whether the response message contained values such as timestamps, increasing id, etc. that could have biased the result, i.e., causing a failure for any test case execution. Test case generation was based on equivalence classes for each input parameter. The number of test cases was large enough (1000) to cover any combination of the equivalence classes. Test cases were run against the five service releases.

Service outputs were checked by comparing the output of a reference release, corresponding to the service implementation running when the integrator started to use the service, with the output of future releases. The comparison has been performed using two types of checks:

1. *a strong check*, comparing both *dnsjava* response messages (i.e., the DNS query and answer). This is somewhat representative of a *stronger* functional-contract between the system integrator and the provider, which guarantees an exact match of the whole service response over a set of releases;
2. *a weak check*, comparing only the DNS answer, i.e., the information that often a user needs from a DNS client. This is somewhat representative of a *weaker* functional contract.

Finally, values of two QoS attributes—i.e., the response time and the throughput—were measured. To mitigate the randomness of these measures, the test case execution was repeated 10 times, and average values considered. The following subsections will discuss results related to functional and non-functional testing.

Functional Testing

Table 8.1 reports the percentage of test cases that failed when comparing different *dnsjava* releases, considering the *strong check* contract. Rows represent the releases when the integrator started to use the service, while columns represent the service evolution. It clearly appears that a large percentage of failures (corresponding to contract violations) is reported in correspondence with release 1.4. This is mostly explained by changes in the set of DNS types supported by *dnsjava*.

All the system integrators who started to use the service before release 1.4 could have reported problems in the service usage. Integrator-side testing

[10] Although many DNS services exist, the chapter does not provide any URL for them since they are fairly unstable and likely to change over the time.

Table 8.1. dnsjava: % of failed test cases

	strong check				weak check			
	1.3.0	**1.4.0**	**1.5.0**	**1.6.1**	**1.3.0**	**1.4.0**	**1.5.0**	**1.6.1**
1.2.0	3%	74%	74%	74%	1%	7%	7%	7%
1.3.0		74%	74%	74%		9%	9%	9%
1.4.0			0%	0%			0%	0%
1.5.0				0%				0%

would have been therefore effective to identify the change. If executed from the provider's perspective, testing would have suggested to advertise — e.g., updating the service description — the change made. *Vice versa*, integrators who started to use the service at release 1.4 experienced no problem when the service evolved toward releases 1.5 and 1.6.

Let us now consider the case in which the comparison is limited to the DNS answer (*weak check*). As shown in Table 8.1, in this case the percentage of violations in correspondence with release 1.4 is lower (it decreases from 74% to 7–9%). Such a large difference is due to the fact that only the DNS query (involved in the comparison only when using the *strong check* and not when using the *soft check*) reports DNS types: here the comparison of resolved IP addresses did not produce a large percentage of failures. Where present, failures are mainly due to the different way subsequent releases handle exceptions. While this happens in a few cases, it represents a situation to which both the provider and the system integrators should pay attention carefully.

Non-functional Testing

Figure. 8.6 reports average response time and throughput values measured over the different *dnsjava* releases. A response time increase (or a throughput decrease) may cause a violation in the SLA stipulated between the provider and the integrator. Basically, the figure indicates that

- except for release 1.6, the performance always improves;
- integrators who started to use the service at release 1.5 could have noticed a SLA violation, in case the provider guaranteed, for future releases, at least the same performances exhibited by release 1.5;
- integrators who started to use the service at release 1.4 could have noticed, in correspondence with release 1.6, a slight increase in the *response time*, even if a slight improvement in terms of *throughput*;
- finally, all the integrators who started to use the service before release 1.4 were fully satisfied.

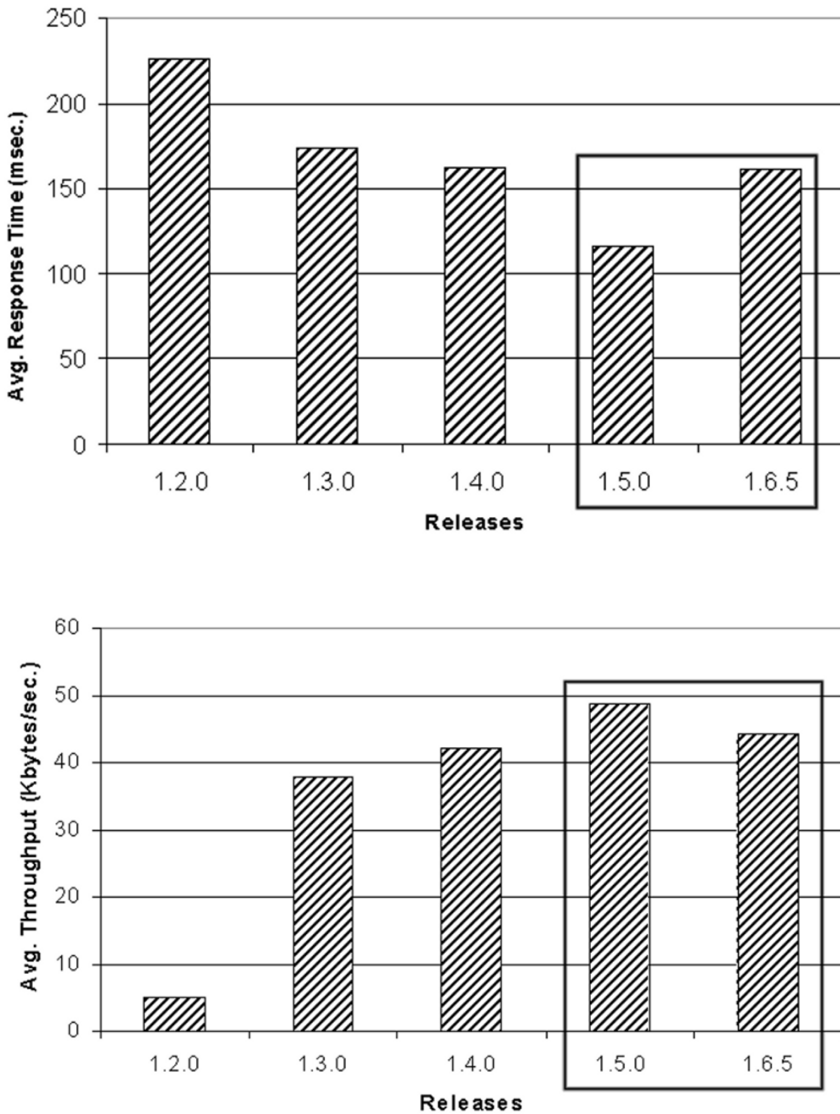


Fig. 8.6. dnsjava measured QoS over different releases

Overall, we noticed that the QoS always improved over its evolution, except for release 1.6.5, where developers decided to add new features at the cost of worsening the performances.

Table 8.2. Characteristics of the services under test

Operation	Inputs	Outputs	# of test Cases
HotelService			
getHotelInfo	HotelID, Arrival Date, # of Nights	# of Rooms Available	13
getHotelListByLocation	City, Location	List of Hotel IDs	
getHotelByLocation	City, Location	Hotel ID	
RestaurantService			
restaurantReservation	Restaurant ID, date, hour, # of seats	Reservation outcome	7
checkRestaurant-Availability	Restaurant ID, date, #of seats	Restaurant availabilities	
getRestaurantList	City, Location	List of Restaurant Info	
getRestaurantInfo	Restaurant ID	Info related to the specified restaurant	
getRestaurantID	restaurant name, city, address	the related ID	

8.4.2 Study II: Usage of Monitoring Data to Reduce Service Invocations During Regression Testing

The second study was performed with the aim of investigating the use of monitoring data to reduce the number of testing invocations. To this aim, we selected two services developed within our organizations and being used as a part of the *test-bed* for an integrated service marketplace developed within a large research project [12]. In particular, we considered a service for searching hotels *HotelService* and a service for searching restaurants *RestaurantService*, also used in Sect. 8.3 to explain the proposed approach and toolkit. Table 8.2 reports characteristics of these services in terms of operations provided and (JUnit) test cases developed for each service (each test case only contains a single service operation invocation).

As shown in Fig. 8.7, the two services underwent three evolution stages. As explained in Sect. 8.2, some of the evolution scenarios stem from the evolution of the Amazon Web service.

1. *Time* t_0*:* The two services are released.
2. *Time* t_1*:* The *HotelService* input parameter *location* becomes mandatory (while it was optional at time t_0). For *RestaurantService* the operation *getRestaurantList* now returns an unbounded list of Restaurant Info (at time t_0 the list contained three items at most).
3. *Time* t_2*:* For *RestaurantService* the maintenance activity impacted the *checkRestaurantAvailability* and *getRestaurantID* operations. In

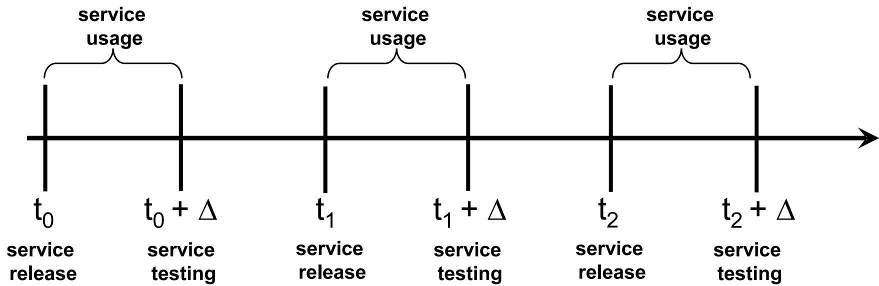


Fig. 8.7. Service evolution and usage timeline

particular, the *checkRestaurantAvailability* operation does not accept a comma-separated list of restaurant IDs anymore, but only a single ID. The *getRestaurantID* operation now returns a restaurant ID composed of five digits instead of four. Finally, the *HotelService* search criteria changed.

Services I/O were monitored. From the beginning of the analysis (t_0) to its completion ($t_2 + \Delta$) we monitored a total of 70 invocations for *HotelService* and 203 for *RestaurantService*. The time between the release time t_x and the testing time $t_x + \Delta$ was about five hours. During these time intervals, we monitored a number of invocations (Table 8.3) reusable to reduce service invocations when performing regression testing.

Figure. 8.8 reports the percentage of test cases that failed at time $t_0 + \Delta$, $t_1 + \Delta$, and $t_2 + \Delta$ respectively. No test case failed at time $t_0 + \Delta$. This is not surprising, since system integrators started to use the services and downloaded the test suites at that time. At time $t_1 + \Delta$, the change made to *HotelService* was not detected by any test case, because the *location* parameter was always specified for all test cases of the *HotelService* test suite. This was not the case of *RestaurantService*, where test runs were able to detect the change: the list returned by the operation *getRestaurantList* contains more elements than the three expected.

When running again the test suite at time $t_2 + \Delta$, it was able to identify the changes made to *HotelService*. In particular, the different outputs produced for the same query were captured when executing the test cases. For *RestaurantService*, the execution of the test suite discovered only the change related to *getRestaurantID* (five digits instead of four), while the change of implementation for *checkRestaurantAvailability* was not detected, since the test cases considered always contained a single ID, instead of a comma-separated list of IDs.

Table 8.3. Number of monitored messages for the services under test

Service	$[t_0, t_0 + \Delta]$	$[t_1, t_1 + \Delta]$	$[t_2, t_2 + \Delta]$
HotelService	11	8	18
RestaurantService	15	19	24

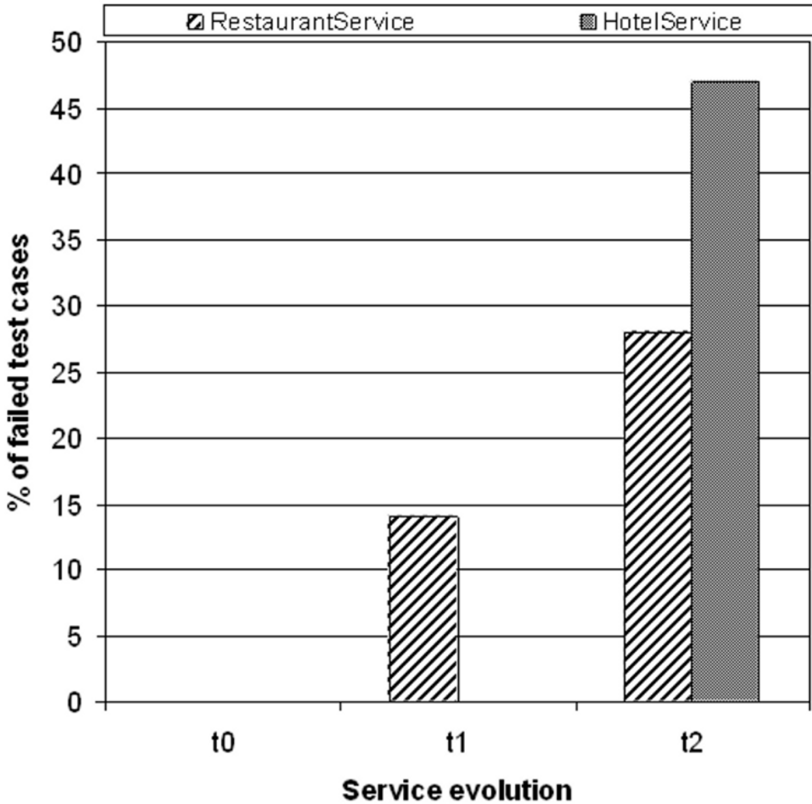


Fig. 8.8. Percentage of failed test cases

Figure. 8.9 shows how data from monitoring were used to reduce the number of operation invocations during the testing activity. Clearly, the percentage of the reused invocations (between 8% and 70% in our case studies) depends on the accesses made by external users during their normal usage of the services and, in particular, during the time interval $[t_x, t_x + \Delta]$ between a new release and the testing activity.

8.5 Related Work

The idea of complementing Web services with a support for testing comes from the testing of component-based systems. As described by Weyuker [2], Bertolino et al. [1], and Orso et al. [13, 14], components can be complemented with a high-level specification, a built-in test suite, and also a traceability map that relates specifications to component interfaces and test cases. Weyuker [2]

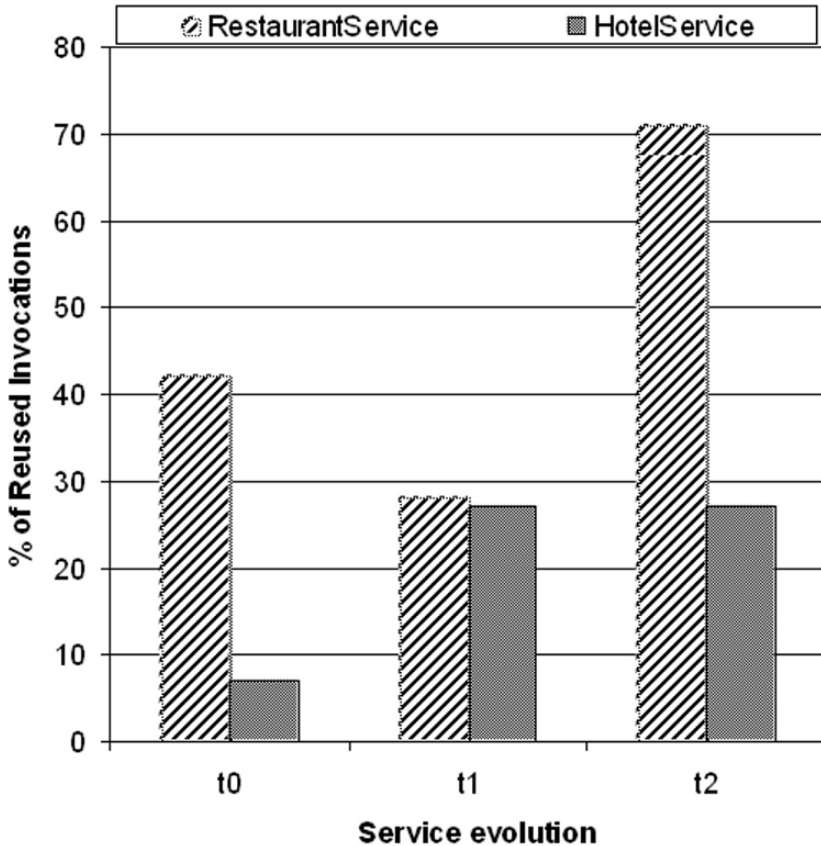


Fig. 8.9. Percentage of reused invocations

indicates that, especially for components developed outside the user organization, the provider might not be able to effectively perform component unit testing, because she/he is not aware of the target usage scenarios. As a consequence, the component integrator is required to perform a more careful re-test inside his/her own scenario. The aforementioned requirements are also true for services and, as discussed in Sect. 8.1 and in Sect. 8.2.3, the shift of perspective services enforces the need for testing during service evolution. In addition, while components are, very often, developed as independent assets for which unit test suites are available, services expose a limited view of complex software systems. However, test suites developed for such systems are not suitable to be executed by the system integrators.

The literature reports several approaches for regression testing. A comprehensive state of the art is presented by Harrold [15], explaining the different techniques and issues related to coverage identification, test-suite minimization and prioritization, testability, etc. Regression test selection [16, 17, 18]

is an important aspect: it aims to reduce the cost of regression testing that largely affects the overall software maintenance cost [19]. Much in the same way, it is important to prioritize test cases that better contribute toward achieving a given goal, such as code coverage or the number of faults revealed [20, 21]. Cost-benefits models for regression testing have also been developed [22, 23, 24]. For services, the issue of cost reduction is particularly relevant, as discussed in Sects. 8.3.5 and 8.3.7. Nevertheless, the aforementioned white-box techniques cannot be applied for services, due to the unavailability of source code from the integrator's perspective.

While the research on service testing is at an initial stage, it is worth comparing a few approaches with ours. Tsai et al. [25] defined a scenario-based testing method and an extension to WSDL to test Web services. The Coyote tool [26] is an XML-based object-oriented testing framework to test Web services. It supports both test execution and test scenario management. The Coyote tool consists of two parts: a test master and a test engine. The test master produces testing scenarios from the WSDL specification. The test engine interacts with the Web service being tested and provides tracing information to the test master. Tsai et al. [27] also proposed to extend the UDDI registry with testing features: the UDDI server stores test scripts in addition to WSDL specifications.

Bertolino and Polini [28] proposed a framework to extend the UDDI registry to support Web service interoperability testing. With their approach, the registry changes its role from a passive role of a service directory toward an active role of an accredited testing organism.

The use of the above approaches and tools is limited to a closed environment, since the tools need to know the scenario in which the available Web services and the user applications are deployed. On the other hand, our tool is usable in a open environment, as it just requires that the provider releases the XML-encoded test suite together with the service. Even if the provider does not release a test suite, it is still possible for the system integrator to develop his/her own test suite and use it against the service.

Heckel and Mariani [29], use graph transformation systems to test single Web services. Like Tsai et al., their method assumes that the registry also stores additional information about the service. Service providers describe their services with an interface descriptor (i.e., WSDL) and some graph transformation rules that specify their behavior.

At the time of writing, some commercial tools supported Web service regression testing. Basically, they generate random test cases starting from input types defined in WSDL interfaces. Such an approach can lead to quite large and expensive test suites. In our case, we can either generate a test suite starting from unit test suites available for the software system which is behind the service interface, or use a capture/replay approach. Moreover, we try to reduce the testing cost further by using monitoring data to reduce the number of service invocations when executing the test. Finally, we combine the check of service functional and non-functional (QoS) properties.

In a companion paper [30] we introduced the idea of service testing as a contract and presented a preliminary description of the approach. This chapter thoroughly describes the service regression testing approach by means of a running example and provides details about the tool support. In addition, it outlines the main open issues for service regression testing and proposes the use of monitoring data to reduce the testing cost.

8.6 Concluding Remarks

The evolution of a service is out of control of whoever is using it: while being used, a service can change its behavior or its non-functional properties, and the integrator may not be aware of such a change. To this aim, regression testing, performed to ensure that an evolving service maintains the functional and QoS assumptions and expectations valid at the time of integration into a system, is a key issue to achieve highly reliable service-oriented systems.

This chapter discussed the idea of using test cases as an executable contract between the service provider and the system integrator. The provider deploys an XML-encoded test suite with the service, while the user can rely on such a test suite, properly complemented if necessary, to test the service during its evolution. The proposed approach is supported by a toolkit composed of (i) a tool that generates the XML-encoded test suite, which can be executed against the service, from JUnit test cases available from the system behind the service interface, and (ii) of a tool that allows the integrator to regression test the service.

Reducing the testing cost has always been an issue for any testing activity. This is particularly true when testing a service, since service invocations have a cost and consume provider's resources. This chapter proposes to exploit previously monitored I/O to reduce the number of service invocations due to the execution of test cases.

Service regression testing still presents a number of open issues. The testability of services that produce a side effect and the dependency of testing results, especially for non-functional testing, from the particular configuration and from factors such as the network workload, are just some of them. Work-in-progress is devoted to enhance the tool, further improving the mechanism for reducing invocations by using monitoring data and adopting more sophisticated mechanisms to model service QoS.

References

1. Bertolino, A., Marchetti, E., Polini, A.: Integration of "components" to test software components. ENTCS **82** (2003)
2. Weyuker, E.: Testing component-based software: A cautionary tale. IEEE Softw. **15** (1998) 54–59

3. Harrold, M.J., Liang, D., Sinha, S.: An approach to analyzing and testing component-based systems. In: First International ICSE Workshop on Testing Distributed Component-Based Systems, Los Angeles, CA (1999) 333–347
4. Canfora, G., Di Penta, M.: Testing services and service-centric systems: Challenges and opportunities. IT Professional **8** (2006) 10–17
5. Ludwig, H., Keller, A., Dan, A., King, R., Franck, R.: Web Service Level Agreement (WSLA) language specification (2005) *http://www.research.ibm.com/wsla/WSLASpecV1-20030128.pdf.*
6. Hicinbothom, J.H., Zachary, W.W.: A tool for automatically generating transcripts of human-computer interaction. In: Proceedings of the Human Factors and Ergonomics Society 37th Annual Meeting. (1993) 1042
7. Elbaum, S.G., Rothermel, G., Karre, S., Fisher, M.I.: Leveraging user-session data to support Web application testing. IEEE Trans. Software Eng. **31** (2005) 187–202
8. Zhang, L., Ardagna, D.: SLA based profit optimization in autonomic computing systems. In: Proceedings of the 2nd ACM International Conference on Service Oriented Computing (ICSOC 2004), ACM Press (2004)
9. Liu, H., Lin, X., Li, M.: Modeling response time of SOAP over HTTP. In: proceedings of the IEEE International Conference on Web Services (ICWS 2005), 11-15 July 2005, Orlando, FL, USA, IEEE Computer Society (2005) 673–679
10. Menasce, D.A.: Qos issues in web services. IEEE Internet Computing **06** (2002) 72–75
11. Menasce, D.A.: Response-time analysis of composite web services. IEEE Internet Computing **08** (2004) 90–92
12. Canfora, G., Corte, P., De Nigro, A., Desideri, D., Di Penta, M., Esposito, R., Falanga, A., Renna, G., Scognamiglio, R., Torelli, F., Villani, M.L., Zampognaro, P.: The C-Cube framework: Developing autonomic applications through web services. In: Proceedings of Design and Evolution of Autonomic Application Software (DEAS 2005), ACM Press (2005)
13. Orso, A., Harrold, M., Rosenblum, D., Rothermel, G., Soffa, M., Do, H.: Using component metacontent to support the regression testing of component-based software. In: Proceedings of IEEE International Conference on Software Maintenance. (2001) 716–725
14. Orso, A. Harrold, M., Rosenblum, D.: Component metadata for software engineering tasks. In: EDO2000. (2000) 129–144
15. Harrold, M.J.: Testing evolving software. J. Syst. Softw. **47** (1999) 173–181
16. Graves, T.L., Harrold, M.J., Kim, J.M., Porter, A., Rothermel, G.: An empirical study of regression test selection techniques. ACM Trans. Softw. Eng. Methodol. **10** (2001) 184–208
17. Harrold, M.J., Rosenblum, D., Rothermel, G., Weyuker, E.: Empirical studies of a prediction model for regression test selection. IEEE Trans. Softw. Eng. **27** (2001) 248–263
18. Rothermel, G., Harrold, M.J.: Empirical studies of a safe regression test selection technique. IEEE Trans. Softw. Eng. **24** (1998) 401–419
19. Leung, H.K.N., White, L.: Insights into regression testing. In: Proceedings of IEEE International Conference on Software Maintenance. (1989) 60–69
20. Elbaum, S., Malishevsky, A.G., Rothermel, G.: Test case prioritization: A family of empirical studies. IEEE Trans. Softw. Eng. **28** (2002) 159–182
21. Rothermel, G., Untch, R.J., Chu, C.: Prioritizing test cases for regression testing. IEEE Trans. Softw. Eng. **27** (2001) 929–948

22. Leung, H.K.N., White, L.: A cost model to compare regression testing strategies. In: Proceedings of IEEE International Conference on Software Maintenance. (1991) 201–208
23. Malishevsky, A., Rothermel, G., Elbaum, S.: Modeling the cost-benefits tradeoffs for regression testing techniques. In: Proceedings of IEEE International Conference on Software Maintenance, IEEE Computer Society (2002) 204
24. Rosenblum, D.S., Weyuker, E.J.: Using coverage information to predict the cost-effectiveness of regression testing strategies. IEEE Trans. Softw. Eng. **23** (1997) 146–156
25. Tsai, W.T., Paul, R.J., Wang, Y., Fan, C., Wang, D.: Extending WSDL to facilitate Web services testing. In: 7th IEEE International Symposium on High-Assurance Systems Engineering (HASE 2002), 23-25 October 2002, Tokyo, Japan. (2002) 171–172
26. Tsai, W.T., Paul, R.J., Song, W., Cao, Z.: Coyote: An XML-based framework for Web services testing. In: 7th IEEE International Symposium on High-Assurance Systems Engineering (HASE 2002), 23-25 October 2002, Tokyo, Japan. (2002) 173–176
27. Tsai, W.T., Paul, R.J., Cao, Z., Yu, L., Saimi, A.: Verification of Web services using an enhanced UDDI server. (2003) 131–138
28. Bertolino, A., Polini, A.: The audition framework for testing Web services interoperability. In: EUROMICRO-SEAA, IEEE Computer Society (2005) 134–142
29. Heckel, R., Mariani, L.: Automatic conformance testing of Web services. In Cerioli, M., ed.: FASE. Volume 3442 of Lecture Notes in Computer Science., Springer (2005) 34–48
30. Bruno, M., Canfora, G., Di Penta, M., Esposito, G., Mazza, V.: Using test cases as contract to ensure service compliance across releases. In Benatallah, B., Casati, F., Traverso, P., eds.: ICSOC. Volume 3826 of Lecture Notes in Computer Science., Springer (2005) 87–100