Using Test Cases as Contract to Ensure Service Compliance Across Releases

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

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

Abstract. Web Services are entailing a major shift of perspective in software engineering: software is used and not owned, and operation happens on machines that are out of the user control. This means that the user cannot decide the strategy to migrate to a new version of a service, as it happens with COTS. Therefore, a key issue is to provide users with means to build confidence that a service i) delivers over the time the desired function and ii) at the same time it is able to meet Quality of Service requirements.

This paper proposes the use of test cases as a form of contract between the provider and the users of a service, and describes an approach and a tool to allow users running a test suite against a service, to discover if functional or non-functional expectations are maintained over the time. The approach has been evaluated by applying it to two case studies.

Keywords: Service Testing, Evolution of Service–Oriented Systems, Regression Testing, Service Level Agreements.

1 Introduction

Service–oriented architectures are having a relevant impact on the development of today's software systems, and promise to become a major technology to even enable the development of business–critical applications. This, however, requires highly reliable and robust services. To this aim, it is necessary to perform service testing. All in all, a service can be considered very similar to a component, and thus testing approaches developed in Component–Based Software Engineering (CBSE) can be adapted to services. Much in the same way, a complex service–oriented system is a distributed system, thus, again, existing techniques can be reused.

However, service–oriented architectures introduce some important issues that need to be considered when performing software testing. In a service–oriented scenario, users just invoke a service, instead of physically integrating it (as it happens for components). The service provider can decide to maintain the service, and the user could not be aware of that. For example:

B. Benatallah, F. Casati, and P. Traverso (Eds.): ICSOC 2005, LNCS 3826, pp. 87–100, 2005.

- *new features can be added:* despite that, the service provider could decide not to advertise in the service interface the change performed, because the input and output parameters are not affected. However, the change made alters the service behavior, and alters the service non-functional properties (e.g., the *response time*) as well;
- *optimizations (e.g., changes in algorithmic solutions) can be performed:* this will, for sure, cause a variation in the service non-functional properties. As a result, the Service Level Agreement (SLA) stipulated between the user and the provider may or may not be violated. In fact, an optimization could improve a non-functional property while worsening another, or even an improvement of some Quality of Service (QoS) attributes (e.g., the *response time*) may not be desirable since it may cause unwanted effects in the whole system behavior. Last but not least, any optimization could introduce faults, thus varying the service functional behavior as well.

To deal with the aforementioned issues, this paper proposes the use of test cases as a way to stipulate contracts between a service provider and service users[1]. This calls for empowering users to perform regression testing [1] with the aim of discovering if a new version of a given service is still in line with the expectations and assumptions that led to the inclusion of the service in a system.

Test suites are published by the service provider as a part (*facet*) of the service description. When a user acquires a service, s/he can use such test suites to check whether the service behaves as desired. In addition, the user can add a further test suite (this can be particularly important since the user may not completely trust test cases delivered by the provider). If no deviation from the expected behavior is noticed, the contract is stipulated, and the test suite specifies the service behavior required to fulfill the contract. Then, the user can periodically run the test suite to discover if changes made to the service implementation entail the violation of any of the initial assumptions and expectations, either functional or related to QoS.

This paper makes the following contribution:

- it proposes to support service consistency verification through evolution by executing test suites contained in a XML–encoded facet attached to the service;
- it presents a toolkit that allows to generate testing facets from JUnit test suites, combining static and dynamic analysis, and to run them against the service; and
- it discusses empirical data demonstrating the effectiveness of using test cases as a contract between service providers and users.

The rest of the paper is organized as follows. Section 2 describes the proposed approach and tool. Section 3 presents and discusses results from empirical studies

[1] According to service–oriented architecture terminology, the term user is used here to refer to the organization or individual engineer that integrates a service into a service–oriented system, and not to the end user of the system itself, that is not expected to test a service.

performed to assess the approach. Finally, after a discussion of the related work, Section 5 concludes.

2 The Approach

The basic idea behind our approach is to provide a service with i) a set of test cases and ii) a set of QoS assertions. This idea comes from component–based software testing, where some authors proposed the idea of providing a test suite together with the component [2, 3]. However, the fact that a service is executed on the server machine and evolves independently from the systems using it, and the need for a service to meet non-functional requirements established in the SLA, introduce new issues that the approach has to fulfill.

When the user acquires a service, s/he is able to access the XML–encoded test cases and QoS assertions hyperlinked to the service WSDL. These test cases and assertions constitute a kind of "contract" between the user and the service provider. By executing the test cases, the user can observe the service functional and non-functional behavior. If satisfied, the user 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 will be made to the service implementation. If, during that period, the service evolves (i.e., a new release is deployed), deviations from the agreed behavior would cause a contract violation.

Fig. 1 provides an overview of the whole test case generation and regression testing process. The light gray area indicates what happens when the service

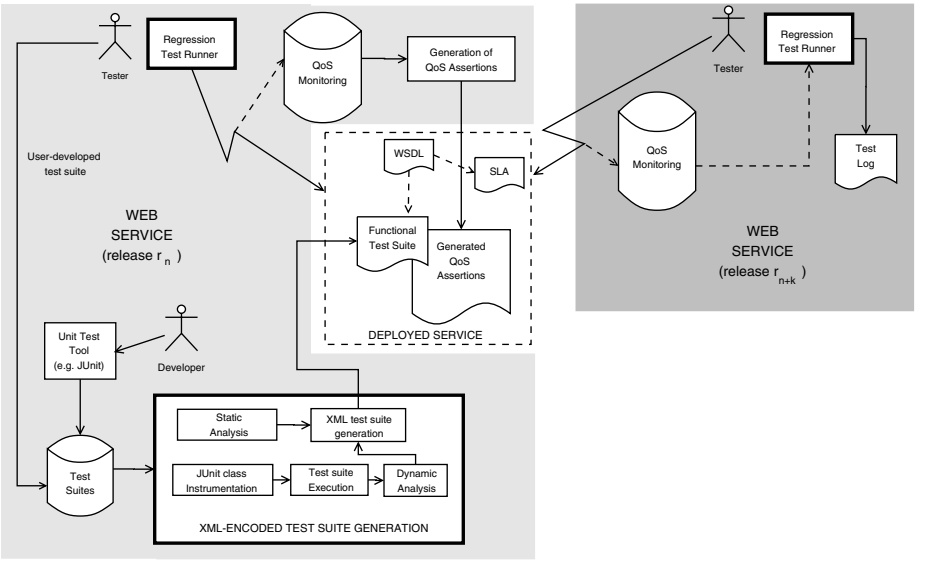

Fig. 1. Test generation and execution

is acquired (release r_n). The dark gray area indicates what happens when the service evolves (to release r_{n+k}) and the user re-tests the service. When the developer implements the service, s/he also provides for it a unit test suite. By proper analysis and transformations, the test suite is XML-encoded and attached to the service interface. The service is therefore published together with the testing facet, comprising test cases (and, eventually, QoS assertions). Subsequently, the service user can periodically re-execute the test suite against the service to verify whether the service still exhibits the functional and non-functional properties held when the service was acquired. The whole approach is supported by a toolkit, developed in Java, composed of two modules:

- the *testing facet generator* that, as described in Section 2.2 supports the generation of the XML-encoded testing facet. In the current implementation, the tool accepts JUnit[2] test suites, although we plan to extend our tool so to be able to accept as input test suites developed using other tools (and for services written using different languages). 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 is composed of a sequence of assertions checking properties of the class under test. The tool relies on JavaCC[3] to perform Java source code analysis and transformation, on the Axis web services framework[4] on the Xerces[5] XML parser.
- the *test suite runner* that permits the service consumer to execute the test suite against the service and produces the test log.

After describing the approach's assumptions, the next subsections thoroughly describe the different phases of the test case generation and execution process.

2.1 Assumptions

In order to let the approach work properly, we need to make some assumptions, and, in case they fail, proper countermeasures should be taken.

An user can test someone else's service with the assumption that the test case execution does not produce any side effect, but only a service response. This is reasonable, for example, for services used for distributed computations or in a grid environment. For example, services performing computations (e.g., image compressing, DNA microarray processing, or any scientific calculus) are suitable. This is not the case, however, of services whose execution produces an irreversible effect, such as services for hotel booking or book purchasing.

In the case of services with side effects, the approach is still feasible from the provider's side, after isolating the service from its environment (e.g., databases), or even from the user side if the provider exports operations to allow users to test the service in isolation.

[2] *http://www.junit.org/*
[3] *https://javacc.dev.java.net/*
[4] *http://xml.apache.org/axis/*
[5] *http://xml.apache.org/xerces2-j/*

Testing may become problematic for the provider if it is highly resource–demanding or, for the user, if the service has not a fixed fee (e.g., a monthly–usage fee) but the cost depends on the number of its invocations. These issues are discussed in Section 5.

Finally, as explained in Section 2.2, the approach is able to generate assertions for testing service non-functional properties. However, this is based on monitoring data that can depend on the current configuration (server machine and load, network bandwidth and load, etc.). While averaging on several measures can mitigate the influence of network/server load at a given time, changes in network or machines may lead to completely different QoS values.

2.2 Step 1: Generating the XML–Encoded Test Suite

The *testing facet generator* XML–encodes a unit test suite provided with the service. First, the service provider indicates to the tool the service class and the JUnit class containing the test suite for the service–under–test. Then, the tool starts analyzing both the service and the JUnit class. The translation of the JUnit test suite into the facet is not straightforward: in general, any JUnit assertion involves expressions of variables containing references to local objects, and method invocations related to these objects. However, the XML–encoded test suite needs to be executed from user–side. The service user can only access to service operations. Any other expression needs therefore to be evaluated on server side and thus XML–encoded as a literal. Expression evaluation is performed

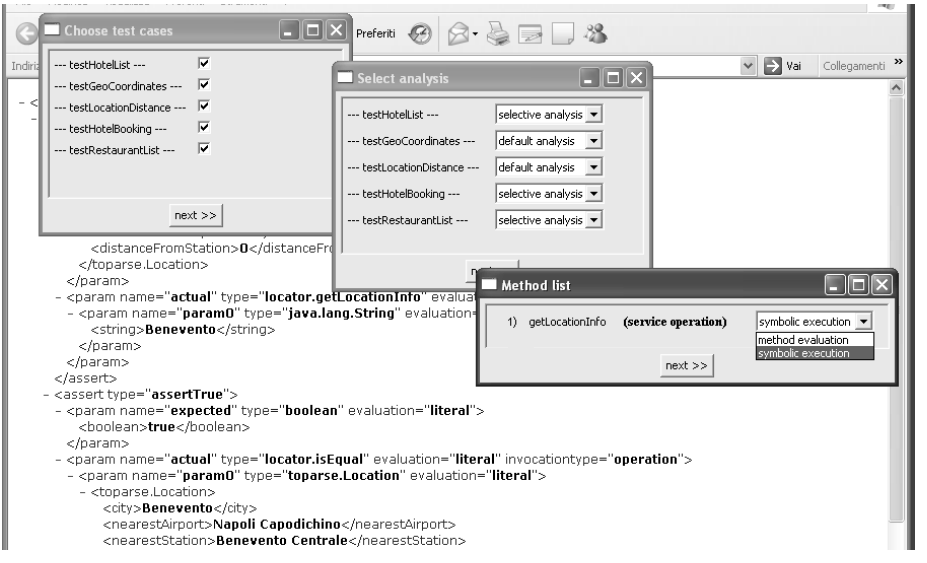

Fig. 2. Test case generation tool

by executing an instrumented version of the JUnit test class from server–side. The obtained dynamic information is then complemented with test suite static analysis to generate the XML testing facet.

The tool shows to the user the list of test cases contained in the test suite ("Choose test case" window in Fig. 2). The user can decide which JUnit test cases should be encoded in XML. For the selected test cases, the user can select (from the "Select analysis" window) two options:

1. *automatic test case transformation:* the tool automatically translates any expression, but service operation invocations, in literals and generates the XML–encoded test suite;
2. *selective translation:* the user can select which method invocations, corresponding to service operations (see the "Method list" window) should be evaluated and which should be left symbolic in the testing facet.

Finally, the service needs to be complemented with QoS assertions, that can be used to verify whether the service is able to preserve its non-functional behavior over the evolution. These assertions can be automatically generated by executing all test cases against the deployed service and measuring each time the QoS attributes by means of a monitoring system. Supposing that each test case contains an assertion involving a service operation, when the test case is executed (and thus the operation invoked) QoS values can be measured. In the current implementation, we are able to measure *response time* and *throughput*; however, with the aid of external monitoring systems, even more complex QoS measures can be used.

The measured values will constitute constraints that should hold in future releases of the service. After obtained this set of constraints and encoded them in XML (see Section 2.2), the service user will send the XML file to the provider. Under some extents, these constraints can be part of the SLA. For example, if the user acquires a service and, invoking one of its operation with a given set of parameters (contained in the test case), observes a *response time* of 20 ms (over a large number of runs), then the generated constraint should be something like:

$$ResponseTime < 20ms + \Delta \tag{1}$$

where Δ represents a tolerance threshold for the expected *response time*, or

$$ResponseTime < p_i \tag{2}$$

where p_i is the i_{th} percentile of the *response time* distribution as measured when acquiring the service.

As an alternative of using QoS assertions, the service non-functional behavior can be checked against any SLA document attached to the service. However, while the assertions allow to check the QoS achieved for each test case, SLA can only be used to check the average QoS values.

XML–Encoding of Test Suites. The *testing facet generator* produces a facet organized in two levels:

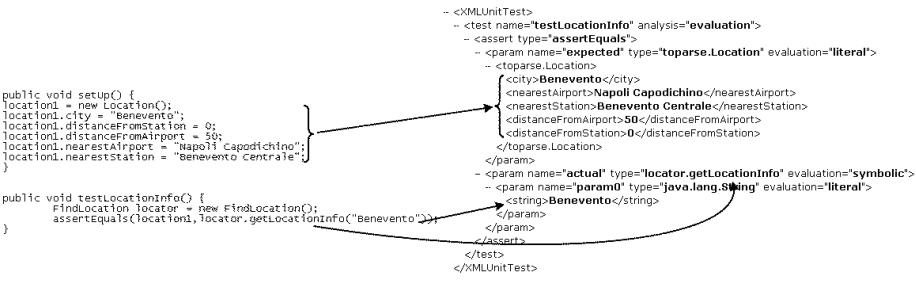

Fig. 3. XML-encoding of a JUnit assertion

1. A **first level**, containing a *Facet Description* and a *Test Specification*. The *Facet Description* contains general information such as the facet owner or the creation date. The *Test Specification Overview* contains, for each test suite enclosed in the facet, information such as the testing strategy adopted (e.g., Functional or Structural) and the tool used to develop the test suite (e.g., JUnit). Then, the *Data* section contains links to XML files containing the test suite itself and QoS assertions.
2. A **second level**, comprising files containing XML–encoded test suites and QoS–assertions.

Fig. 3 shows an example of how a JUnit test case can be mapped to a XML file. The example is related to a service returning travel information for a given location (i.e., closest airport and train station, plus distance to get to the airport and to the station). The left–side of the figure shows a portion of the JUnit test case. The method `setUp()` creates an instance variable containing a *Location* object (`location1`), while the `testLocationInfo()` method asserts that `location1` must match the result of the *getLocationInfo* operation when the passed parameter is "Benevento".

The right–side of Fig. 3 shows the XML mapping. In particular, it is worth noting that:

- objects are serialized in XML (using the XStream[6] serializer);
- the *param* tag has an attribute (*evaluation*) indicating whether the parameter is a literal serialized in the XML, or whether it is symbolic (i.e., it is a service operation that needs to be invoked). For the latter, actual parameters are specified.

QoS assertions are XML–encoded using the WSLA schema [4]. While the SLA poses general QoS constraints[7] (e.g., *"Throughput > 1 Mbps"* or *"Average response time < 1 ms."*), QoS assertions indicate which will be the expected service performance in correspondence of a given set of inputs (specified in the test case). For example, when the input (as specified in the test case) of a MP3

[6] *http://xstream.codehaus.org/*

[7] That must hold for any service usage.

compression service is a 5 MBytes file, the QoS assertion may indicate a *response time* of 30 s (that will clearly be different in case the input file is smaller or bigger).

2.3 Step 2: Running the Test Suite

Once the test suite has been published together with the service, the tester (either a user, a third-party or the provider) can:

1. *download the test suite and the QoS assertions*, hyperlinked to the service WSDL interface;
2. *run the test suite and get the test log:* service operations contained in the test suite are invoked, and assertions 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.

The service user can also provide, on his/her side, further test cases. This is particularly important: the user might not trust the developer's test suite; on the contrary, s/he wants to develop an additional test suite as a contract reflecting the intended service usage (that might have not been contemplated by the provider). In a semi–structured environment (e.g., a service registry of a big organization) the user can therefore publish this new test suite to the service, and other users can eventually reuse it. On the contrary, this may not

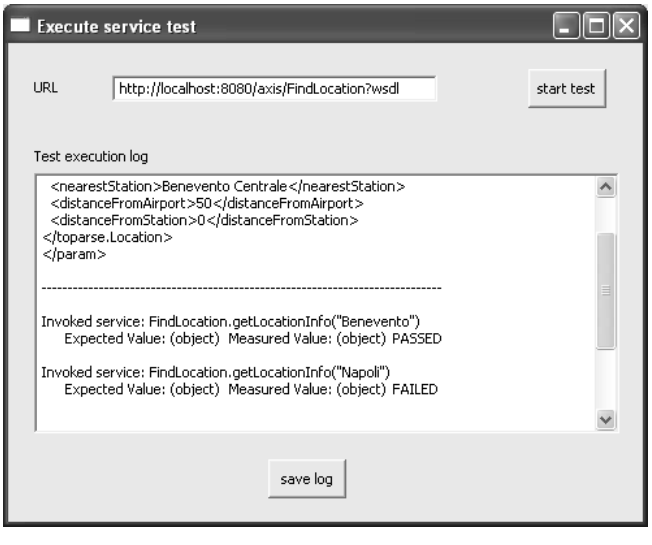

Fig. 4. Test suite runner

be possible in an open environment, where the additional test suite is stored by the user, and only serves to check whether future service releases still satisfy the user requirements.

The decision on whether the user has to add further test case 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 test facet.

The trustability level of the provider's test suite can be assessed, for instance, by analyzing the domain in which the service inputs varies, and the functional coverage criteria adopted.

Fig. 4 shows a screenshot of the *test suite runner*. As shown, after specifying the service URL, it is possible to run the test cases against the service. The window reports the log indicating whether the different assertions passed or failed, and, each time an assertion uses a XML-serialized object, such an object is shown in the window as well.

3 Empirical Study

To validate the proposed approach, we need to generate test cases and related QoS assertions on a service release. Then, test cases need to be run again against subsequent releases, to check whether the service "violates the contract".

We published as web services five releases of two open source systems, *dnsjava* and *InetAddressLocator*. *dnsjava*[8] is a Domain Name System (DNS) client and server; in particular, for this study's purpose, the *dig* (domain information groper) utility has been wrapped with a web service. *dig* is used to gather information from DNS servers, while *InetAddressLocator*[9] is a utility that, given an IP address, returns its geographical location.

The *dig* web service has five parameters: the domain to be solved (compulsory), 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. The *InetAddressLocator* service interface is quite simple: the input parameter is just the IP address to be located, while the output is a string specifying the geographic location. For both services, 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 guided by determining, for each input parameter, equivalence classes. The number of test cases was large enough (1000 for *dnsjava* and 2500 for *InetAddressLocator*) to cover any combination of the equivalence classes. After services have been deployed, test cases are run against all the releases of each system. For *dnsjava*, two different analyses of the test execution logs have been performed:

[8] *http://www.dnsjava.org/*
[9] *http://javainetlocator.sourceforge.net/*

Table 1. dnsjava: % of failed test cases

	strong check				**soft 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%

1. *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 service user and the provider, that guarantees an exact match of the whole service response over a set of releases;
2. *soft 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.

For *InetAddressLocator*, we simply compared the (single) response message. Finally, for *dnsjava* we also measured two QoS attributes, i.e., the *response time* and the *throughput*. To mitigate the randomness of these measures, the test case execution was replicated 10 times, and average values considered[10].

This section reports and discusses the results obtained analyzing test case executions. The following subsections will discuss results related to functional and non-functional testing.

Functional Testing. Table 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 user could have acquired 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 of release 1.4. This is mostly explained by changes in the set of DNS types supported by *dnsjava*.

All the users who bought the service before could have reported problems in the service usage. User–side testing would have therefore noticed the user of the change, while provider–side testing would have suggested to advertise (e.g., updating the service description, although leaving the service interface unaltered) the change made. Vice–versa, users who bought the services at release 1.4 experienced no problem when the service evolved towards releases 1.5 and 1.6.

Let us now consider the case in which the comparison is limited to the DNS answer (*soft check*). As shown in Table 1, in this case the percentage of violations in correspondence of release 1.4 is lower (it decreases from 74% to 7–9%). This large difference is due to the fact that only the DNS query (compared with the strong check) reports DNS types: here the comparison of just resolved IP addresses did not produce a large percentage of failures. Where present, failures

[10] According to what we verified, the standard deviation was below 10% of the average value.

Table 2. InetAddressLocator: % of failed test cases

	2.12	**2.14**	**2.16**	**2.18**
2.10	0%	1%	1%	5%
2.12		1%	1%	5%
2.14			0%	4%
2.16				4%

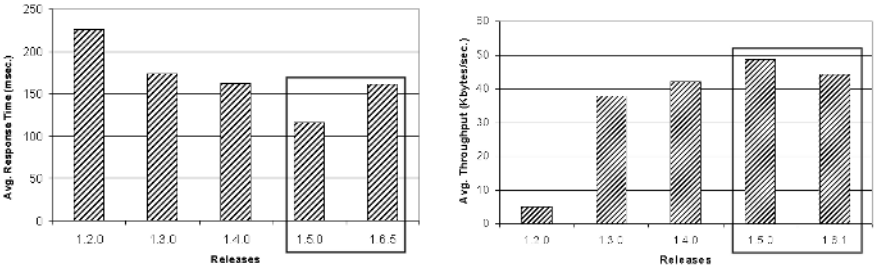

Fig. 5. dnsjava measured QoS over different releases

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 provider and service users should pay careful attention.

Finally, Table 2 shows results for the *InetAddressLocator* software system. Here the differences, mainly appearing in the last release (2.18) are mainly due to updates in the location database. While in this case a different behavior may be considered as a service improvement, it is worth noticing that this could still lead to undesired behaviors from user's side. For example, if the user expects that the *InetAddressLocator* replies with the string *Europe* to a given set of IP addresses, while the new release (more precise) returns the string *Italy*, then the behavior of the system using the service may be affected.

Non-functional Testing. Fig. 5 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 user. Basically, the figure indicates that:

- except for release 1.6, the performance always improved;
- users who acquired 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;
- users who acquired the service at release 1.4 could have noticed, in correspondence of release 1.6, a (slight) decrease of the *response time*, even if a (slight) improvement in terms of *throughput*; and
- finally, all users who acquired the service before release 1.4 were fully satisfied.

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

4 Related Work

As stated in Section 2, the idea of complementing web services with a support for testing comes from the testing of component–based systems. As described by Weyuker [3], Bertolino *et al.* [2] and Orso *et al.* [5, 6], components can be complemented with a high–level specification, a built-in test suite, and also a traceability map able to relate specifications to component interfaces and to test cases. Weyuker [3] indicates that, especially for components developed outside the user organization, the provider might not be able to effectively perform component unit testing, because s/he is not aware of the target usage scenarios. As a consequence, the component user is required to perform a more careful re-test inside his/her own scenario. As discussed in Section 5, this is particularly true for services. For this reason, developer's test cases need to be complemented with user's test cases.

In literature there are plenty of approaches for regression testing. The state of the art is presented by Harrold [7], explaining the different techniques and issues related to coverage identification, test–suite minimization and prioritization, testability etc. Regression test selection [8, 9, 10] constitutes an important aspect aiming to reduce the cost of regression testing, that largely affects the overall software maintenance cost [1]. Much in the same way, it is important to prioritize test cases that better contribute to achieve a given goal, such as code coverage or the number of faults revealed [11, 12].

Cost–benefits models for regression testing have also been developed [13, 14, 15]. Although this is out of scope of the present paper, the issue of modeling, predicting and trying to reduce the testing cost is particularly important for web service testing. Even when test cases are available, service testing consumes network resources, and the provider might want to limit it (see Section 5).

5 Concluding Remarks

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. We have proposed the idea of test cases as a form of contract between a service provider and a service user, and have shown an approach to publish test cases as a facet of the service description, and using such a facet to regression test a service over the time. Whilst the focus of the paper is on the user-side testing, the approach proposed can also be useful for third-party-side testing and provider-side testing, which, similarly to what happens for components [16], constitute the three main perspectives when testing a service–oriented system:

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;
2. *user perspective:* on his/her side, the user may periodically want to re-test the service to ensure that its evolution, or even changes in the underlying software/hardware do not affect the functional and non-functional behavior. Particular attention needs to be paid from the provider's side: service invocation is supposed to have a cost and to consume resources. High–frequency, massive testing of the service from many users would lead to a denial–of–service. Proper countermeasures need therefore to be set from provider's side, limiting the number of service invocations per period of time, and maybe allowing access during periods when the service workload is low;
3. *certifier perspective:* a certifier acts similarly to a user, with the aim of repeatedly testing the service, possibly on behalf of a user, to check whether it is compliant to some functional and non-functional behavior specified in the test suite.

Work–in–progress is devoted to enhance the tool and to integrate it in a complex service–oriented development environment. We are also tackling issues such as the automatic generation of test cases, starting from a service specification or interface, with the aim of violating functional or non-functional contracts. Also, supporting test case reuse and performing cost–benefits analysis are important issues to be considered. Finally, the preliminary empirical studies performed need to be replicated with larger, industrial service–oriented systems.

Acknowledgments

This work is framed within the IST European Integrated Project *SeCSE (Service Centric Systems Engineering) – http://secse.eng.it*, 6th Framework Programme, Contract No. 511680. Authors would like to thank Alberto Troisi for his work the service regression testing tool.

References

1. Leung, H.K.N., White, L.: Insights into regression testing. In: Proceedings of IEEE International Conference on Software Maintenance. (1989) 60–69
2. Bertolino, A., Marchetti, E., Polini, A.: Integration of "components" to test software components. ENTCS **82** (2003)
3. Weyuker, E.: Testing component-based software: A cautionary tale. IEEE Softw. **15** (1998) 54–59
4. 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.*

5. 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
6. Orso, A. Harrold, M., Rosenblum, D.: Component metadata for software engineering tasks. In: EDO2000. (2000) 129–144
7. Harrold, M.J.: Testing evolving software. J. Syst. Softw. **47** (1999) 173–181
8. 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
9. 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
10. Rothermel, G., Harrold, M.J.: Empirical studies of a safe regression test selection technique. IEEE Trans. Softw. Eng. **24** (1998) 401–419
11. Elbaum, S., Malishevsky, A.G., Rothermel, G.: Test case prioritization: A family of empirical studies. IEEE Trans. Softw. Eng. **28** (2002) 159–182
12. Rothermel, G., Untch, R.J., Chu, C.: Prioritizing test cases for regression testing. IEEE Trans. Softw. Eng. **27** (2001) 929–948
13. 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
14. 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
15. 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
16. 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