SOA: Testing and Self-Checking

Gerardo Canfora and Massimiliano Di Penta [1]

RCOST - Research Centre on Software Technology - University of Sannio
Palazzo ex Poste, Via Traiano - 82100 Benevento, Italy

Abstract

The dynamic nature of service-oriented architectures poses new challenges to system validation. Traditional testing is unable to cope with certain aspects of a service–oriented system validation, essentially because of the impossibility to test all (often unforeseen) system's configurations. On the other hand, run–time monitoring, while able to deal with the intrinsic dynamism and adaptiveness of a service–oriented system, are unable to provide confidence that a system will behave correctly before it is actually deployed.

In this paper we discuss the role of testing and monitoring to validate a service–oriented system and how they can be combined to increase the confidence and reduce the cost of validation.

Key words: Web Services, Service–Oriented Architecture, Service Testing, Service Monitoring

[1] Emails: {canfora, dipenta}@unisannio.it

1 Introduction

Testing software has long been recognized as a key and challenging activity of system development. Service–Oriented Architectures (SOA), for example as implemented by web services, present unique features, including dynamic and ultra–late binding, that add much complexity to the testing burden. A few notable examples are such features are:

- systems based on web services are intrinsically distributed, and this requires that Quality of Service (QoS) be ensured for different deployment configurations;
- web services in a system change independently from each other;
- systems implement adaptive behaviors, either by replacing individual services or adding new ones;
- ownership over the system parts is shared among different stakeholders.

Many consolidated testing approaches, applied for years over traditional systems, apply over service–oriented systems as well. Primarily, the idea that a combination of unit, integration, system, and regression testing is needed to gain confidence that a system will deliver the expected functionality. Nevertheless, the dynamic and adaptive nature of SOA makes most testing techniques not directly applicable to test services and service–oriented systems. As an example, most traditional testing approaches assume that one is always able to precisely identify the actual piece of code that is invoked at a given call-site. Or, as in the case of object-oriented programming languages, that all the possible (finite) bindings of a polymorphic component be known. These assumptions my not be true anymore for SOA, which exhibit run-time discovery in an open marketplace of services and ultra–late binding.

The adoption of SOA, in addition to changing the architecture of a system, brings changes in the process of building the system and using it, and this has effects on testing too. Services are used, not owned: they are not physically integrated into the systems that use them and run on a provider's infrastructure. This has several implications for testing: code is not available to system integrators; the evolution strategy of a service (that is, of the software that sits behind the service) is not under the control of the system owner; and, system managers cannot use capacity planning to prevent QoS failures.

Key issues that limit the testability of service–oriented systems include [6]:

(i) *lack of observability of the service code and structure:* for users and system integrators services are just interfaces, and this prevents white–box testing approaches that require knowledge of the structure of code and flow of data.

(ii) *dynamicity and adaptiveness:* for traditional systems one is always able to determine the component invoked in a given call–site, or, at least, the set of possible targets [9]. This is not true anymore for SOA, where a system can be described by means of a workflow of abstract services that are automatically bound to concrete services retrieved by one or more registries during the

execution of a workflow instance.

(iii) *lack of control:* while components/libraries are physically integrated in a software system, this is not the case for services, which run on an independent infrastructure and evolve under the sole control of the provider. The combination of these two characteristics implies that system integrators cannot decide the strategy to migrate to a new version of a service and, consequently, to regression testing the system [5].

(iv) *cost of testing:* invoking actual services on the provider's machine has effects on the cost of testing, too, if services are charged on a *per–use* basis. Also, service providers could experience denial–of–service phenomena in case of massive testing, and repeated invocation of a service for testing may not be applicable whenever the service produces side effects other than a response, as in the case of a hotel booking service [6].

An alternative to testing is continuous self-checking of a service–oriented system by monitoring it during execution [2]. Runtime monitors can check both the functional correctness and the satisfaction of QoS expectations, thus realizing the idea of continuous testing [11]. Being performed at run-time, that is, on the actual running configuration of the system, monitoring can naturally accommodate the intrinsic dynamicity and adaptability of SOA. It does not have cost problems, or undesired side effects on real world, as the service invocation issued are those actually needed to run the system. Of course, there is some run–time overhead, but this is acceptable in most cases and, in addition, modern monitoring infrastructures allow for setting the amount of monitoring at run–time [2].

However, self–checking has limitations, too. The system is checked during actual execution, and this may entail that exceptional conditions remain unchecked, including peak usage and scarcely recurrent function or combinations of input data. More importantly, problems are discovered too late, when the system is executing. Whilst several recovery actions are possible [2, 7, 8], it is not always possible to rely solely on these actions, as in the case of dependable applications. There are many cases when a system needs to be validated before deployment, and in these cases monitoring does not apply.

Thus, on the one side traditional testing has limitation to deal with the dynamic nature of SOA, primarily because of its inability to foresee changes, and on the other side self–checking does not help to assess the quality of a system prior of its use. We believe that testing and monitoring are two complementary facets for validating a service–oriented system and, in this paper, we discuss how they complement each other in the process of gaining confidence of the correctness of a system over time.

The rest of this paper is organized as follows: Section 2 discusses the different roles played by testing and monitoring in SOA. Section 3 highlights the need for having both testing and monitoring and discusses how each one can benefit from the other. Finally, Section 4 summarizes the paper and outline directions for future research.

2 The Role of Testing and of Monitoring

As mentioned in the introduction, service functional and non functional characteristics can be either tested before making the service operational, or monitored at run–time. As it will be described below, the roles of testing and monitoring are different.

To better understand how things work in the practice, let us consider a service–oriented system – realized for example as a BPEL process – that performs image processing. The process (see Figure 1) comprises several services realizing different tasks, i.e. image scaling, posterizing, sharpening, or reduction of colors to a gray scale.

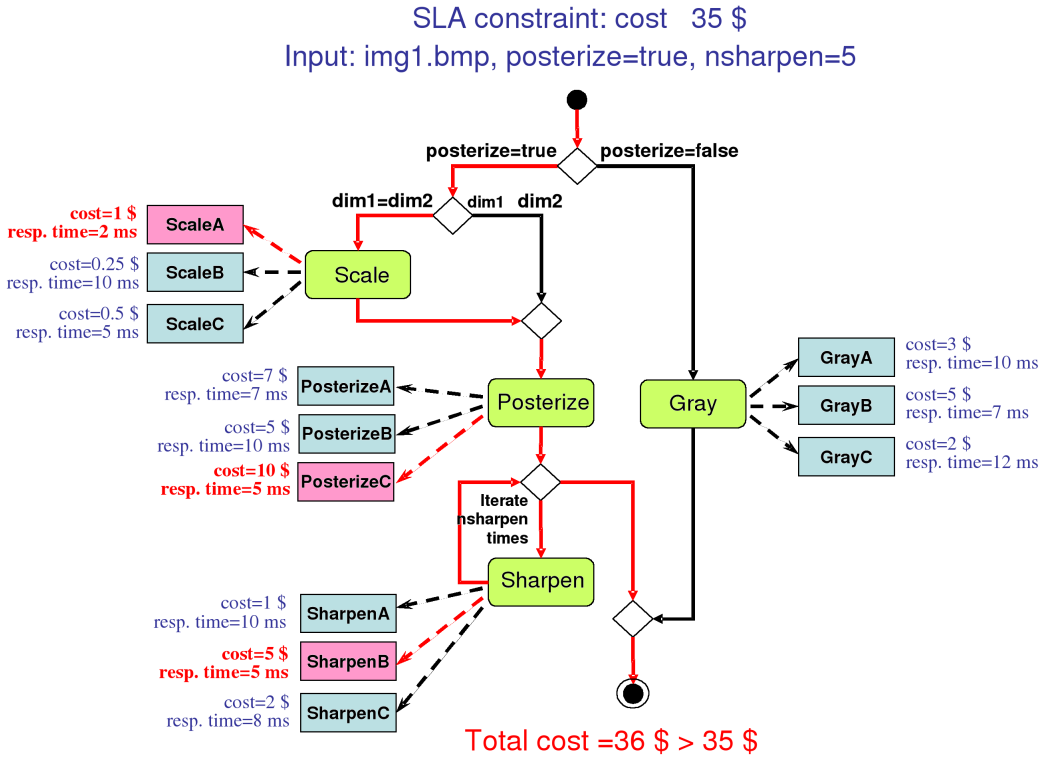

Fig. 1. Running example: image transformation process

The remainder of this section describes how different service characteristics – namely functional aspects, dynamic aspects, and, finally, the way a service evolves – can be tested or monitored, highlighting the role of testing and monitoring.

2.1 Checking the service functional properties

2.1.1 The role of testing

Except when a service is tested by its developer, that has the source code available (whilst the configuration could be far from those where the service will be actually deployed) black box testing is the only viable solution. A possibility is to perturbate SOAP messages to check whether the service is robust to these perturbations, or if

the effect of the perturbation is observable from the service response [10]. For the image processing service, perturbation of parameters – e.g., different *nsharpen* or *posterize* values – should be observable from the different image produced as output or from the error message possibly generated.

Another possibility is to rely on types defined as XML schema within the WSDL to generate test cases [1, 12]. For example, inputs for our example would lead to equivalence classes proper of image processing option parameters (e.g., `true` or `false` for *posterize*, and six classes for *nsharpen*, i.e. $nsharpen = 0$, $nsharpen = 1, 1 < nsharpen < max\_nsharpen-1, nsharpen = max\_nsharpen-1$, $nsharpen = max\_nsharpen$ and $nsharpen > max\_nsharpen$).

2.1.2 The role of monitoring

Instead of testing the services, some monitoring approaches aims at checking whether some post-conditions are met after the service execution [2, 8]. If this does not happen, proper recovery actions are taken. Monitoring rules are often weaved as crosscutting concerns across the system source code (or across its BPEL process). For example, if a image scaling service is not able to handle an image over a given size, either the process execution must be aborted or a different binding has to be chosen.

2.2 Checking the service QoS

2.2.1 The role of testing

When a service is acquired by a consumer s/he stipulates a Service Level Agreement (SLA) with the provider, which comprises the specification for the QoS level that the provider will ensure to the consumer. At run time, QoS constraint violation can be due to the environment (high network traffic, high number of requests), but also to unexpected inputs. For example, let us suppose that the provider of the *Posterize* service states that images below 2 MB can be processed within 10s. It might happen that, for some posterizing preferences, such a constraint cannot be met. Things get worse when the service under test is composite, and different bindings can lead to different QoS.

The role of testing is therefore to generate combinations of inputs and bindings that cause a violation of QoS constraints. The example in Figure 1 shows how for the image `img1.bmp`, $posterize = true$, $nsharpen = 5$ and the abstract services *Scale*, *Posterize* and *Sharpen* bound to *ScaleA*, *PosterizeC* and *SharpenB* the constraint $cost < 35\$$ is violated.

2.2.2 The role of monitoring

QoS violations can also be handled at run–time using a monitoring mechanism that triggers service re–binding actions. After the (near) optimal set of binding has been determined, the service oriented system starts its execution. New QoS estimates made at run–time, or the lack of availability of a service, can trigger the re–planning of services still to be executed [7]. This will allow to meet QoS

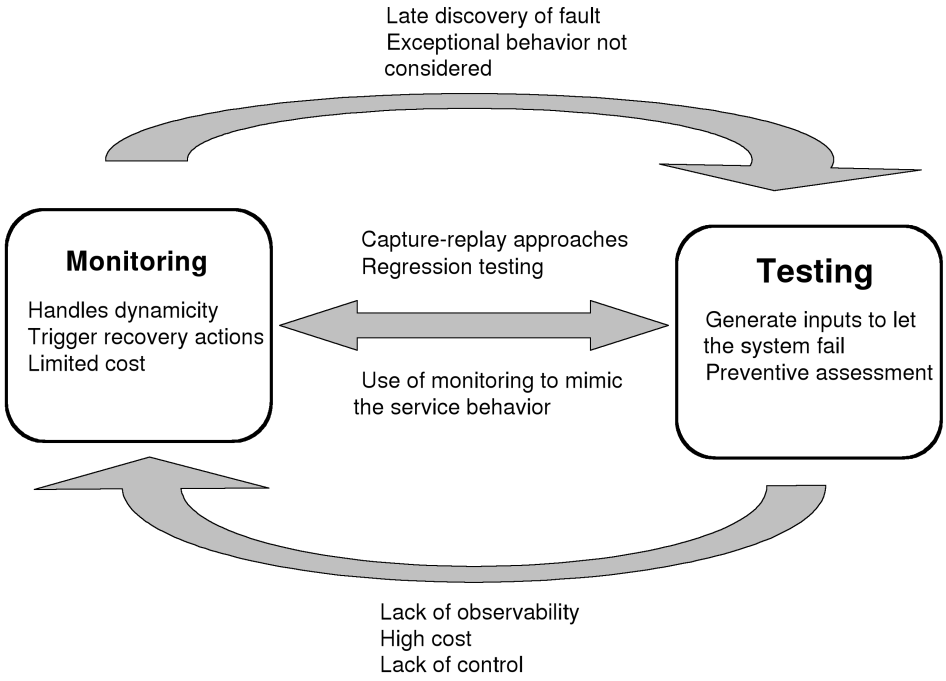

Fig. 2. The role of testing, of monitoring and their interaction

constraints that would have been otherwise violated, and to improve the overall QoS objectives as well (e.g., minimizing the cost or the response time).

Let us assume that, during the execution of the image processing service, the actual, monitored response time for the scaling service is higher than what previously estimated. This causes an increase of the overall service response time, leading to a potential constraint violation. To avoid this, the services still to be executed (i.e., *posterize* and *sharpening*) will be bound to faster (even if more expensive) concretizations that allow response time constraints to be satisfied.

2.3 Checking service interoperability

2.3.1 The role of testing

Interoperability check can vary from the simple check of the compliance to WS-I [2] to the integration testing of services composed in a process or service–oriented system. To test service interoperability, the UDDI registries can change their role from a passive role of service directory to an active role of accredited testing organism [4].

As described in reference [6], dynamic binding makes interoperability issues more and more complex. In such a context, integration testing becomes very expensive, since any service invocation within a process or system should be tested against any possible binding. Problems due to polymorphism in object–oriented systems [9] tend to explode here, even because, very often, bindings are not known

[2] *http://www.ws-i.org*

a priori.

Ideally, the process in our example should be tested for $3 \cdot 3 \cdot 3 \cdot 3 = 81$ possible combinations of concretizations. In the practice, it would suffice to only test those combinations that are compatible with our global QoS constraints.

2.3.2 The role of monitoring

Integration problems can be checked at run–time using monitoring mechanisms. When a binding changes, the new end–point should preserve the post–condition held for the old end–point. If this does not happen, an alternative service should be chosen. For example, if re–binding chooses a new *Posterizing* service that produces an image violating some post–conditions met before – e.g., color depth > 24 bits – then this service should be discarded in favor of an another.

2.4 Checking the service evolution

2.4.1 The role of testing

Regression testing is essential for service–oriented systems, since integrators are out of control of the service being used. When a service evolves, its functionality or QoS can vary, impacting over the systems using it. This raises the need for service regression testing: as proposed by Bruno *et al.* [5], services needs to be accompanied with a facet, containing test suites that the integrator can use for the regression testing of the service functional and non functional properties, either periodically or when a new release of the service has been issued.

For example, the *Sharpen* service implementation can change: either the resulting image can be different, or the response time can be larger than the one experienced when the service was acquired and the SLA negotiated.

2.4.2 The role of monitoring

Monitoring assumes an important role for checking the evolution of a service–oriented system. Once again, a post–condition checking mechanism can be used to check whether the service, while evolving, continues to preserve the functional and non functional properties the integrator is expecting.

3 Combining Testing and Monitoring

The previous section described benefits of different testing and monitoring approaches. One can argue whether it could be possible to avoid testing service–oriented systems and just perform run–time monitoring, followed by proper recovery actions. On the other hand, it can be decided that if a system has been properly tested, monitoring is not needed.

Nevertheless, due to the different roles assumed by testing and monitoring (see Figure 2), we often need both:

(i) Testing is a preventive activity, to be performed before delivering (or before

using) the service. Also, testing exercises the service with the objective of discovering faults. This goes beyond from checking the correctness of the regular service usage, addressed by monitoring.

(ii) Monitoring is performed at a different time, i.e., after the service has been executed. Whilst monitoring can trigger recovery actions, in some cases it may be too late to do anything useful. If the image processing service has just violated its response time constraints, nothing can be done to recover such a violation; the service should have been tested before to check its ability to guarantee a given response time for any allowed input configuration.

As the figure shows, there are many weaknesses of monitoring that suggest to perform testing, and *vice versa* (see the arrows between the monitoring and testing boxes). Also, there are many cases where monitoring strategies can be combined with testing (double arrow). For example, other than using test cases made available with the service, regression testing can be performed using capture–replay strategies. The service inputs and outputs can be monitored and, after the service has evolved, inputs are replayed and outputs observed.

The cost of regression testing can be reduced by using monitoring data for building service stubs that simulate the response to requests equal or similar to those recently made by other users [6].

Service QoS testing strongly relies on monitoring mechanisms that, at minimum, are used to measure the QoS related to a service execution during testing activities. Once again, stubs built upon monitoring data can be used to limit the number of service invocations required during the testing phase.

When replacing the end–point of an abstract service, data monitored from the previous end–point can be used to test the new one, to ensure that it properly interoperates with our system. For example, if the *PosterizeA* end point is replaced by *PosterizeB*, it is worth using monitoring data from *PosterizeA* to test *PosterizeB* and check whether the new end point preserves the current behavior.

Finally, service functional testing can benefit of monitoring mechanisms. Monitors can be used to implement oracles (i.e., by checking post–conditions).

4 Summary

The paper has discussed the complementary roles of testing and run–time monitoring in the process of validating service–oriented systems. On the one hand, run–time monitoring is needed to deals with systems configurations that change in an unforeseen way; on the other hand, testing is still indispensable to gain confidence on the correctness of service–oriented systems before they are actually deployed for use.

Of course, both testing and monitoring of service–oriented systems present open problems that need further research. A key issue of monitoring is to balance the degree of run-time checking and the impact on the performances of the system. Ideally, monitors should allow for setting the amount of monitoring at run–time

based on the needs of single users [2]. Also, it is needed that monitors be able to work with existing standard technologies, such us standard BPEL engines.

The idea itself of integration testing is challenged by SOA unique features, primarily automated search on services in an open space and ultra–late binding. These features implies that the actual configuration of services involved in a system's run for a given user be known only at execution time. In many cases, however, however, searching is limited to a bounded space, as for example when it is required that a contract be signed before a service can be used. In these cases, conservative approaches that test possible system configurations while minimising the number of test runs are needed.

QoS testing poses new challenge, too. Services run on the (foreign) infrastructures of providers and payment may be on a per-use basis, which makes stress testing prohibitively costly. Monitoring data from past execution could help reducing the cost of testing by minimising the needs for actual calls to services.

5 Acknowledgments

This work is framed within the European Commission VI Framework IP Project SeCSE (Service Centric System Engineering) (*http://secse.eng.it*), Contract No. 511680.

References

[1] X. Bai, W. Dong, W.-T. Tsai, and Y. Chen. In *WSDL-Based Automatic Test Case Generation for Web Services Testing*, pages 215–220, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[2] L. Baresi and S. Guinea. Towards dynamic monitoring of WS-BPEL processes. In Benatallah et al. [3], pages 269–282.

[3] B. Benatallah, F. Casati, and P. Traverso, editors. *Service-Oriented Computing - ICSOC 2005, Third International Conference, Amsterdam, The Netherlands, December 12-15, 2005, Proceedings*, volume 3826 of *Lecture Notes in Computer Science*. Springer, 2005.

[4] A. Bertolino and A. Polini. The audition framework for testing Web services interoperability. In *EUROMICRO-SEAA*, pages 134–142. IEEE Computer Society, 2005.

[5] M. Bruno, G. Canfora, M. Di Penta, G. Esposito, and V. Mazza. Using test cases as contract to ensure service compliance across releases. In Benatallah et al. [3], pages 87–100.

[6] G. Canfora and M. Di Penta. Testing services and service-centric systems: Challenges and opportunities. *IT Professional*, 8(2):10–17, 2006.

[7] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. QoS-Aware Replanning of Composite Web Services. In *in Proc. of the International Conference on Web Services (ICWS 2005)*, Orlando, FL, USA, July 2005.

[8] K. Mahbub and G. Spanoudakis. A framework for requirents monitoring of service based systems. In M. Aiello, M. Aoyama, F. Curbera, and M. P. Papazoglou, editors, *ICSOC*, pages 84–93. ACM, 2004.

[9] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[10] J. Offutt and W. Xu. Generating test cases for web services using data perturbation. *SIGSOFT Softw. Eng. Notes - SECTION: Workshop on testing, analysis and verification of web services (TAV-WEB)*, 29(5):1–10, 2004.

[11] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for java. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 114–123, New York, NY, USA, 2005. ACM Press.

[12] W. T. Tsai, X. Wei, Y. Chen, and R. Paul. A robust testing framework for verifying web services by completeness and consistency analysis. In *IEEE International Workshop on Service-Oriented System Engineering (SOSE)*, pages 159–166, Los Alamitos, CA, USA, 2005. IEEE Computer Society.