Testing services and service-centric systems: Challenges and opportunities

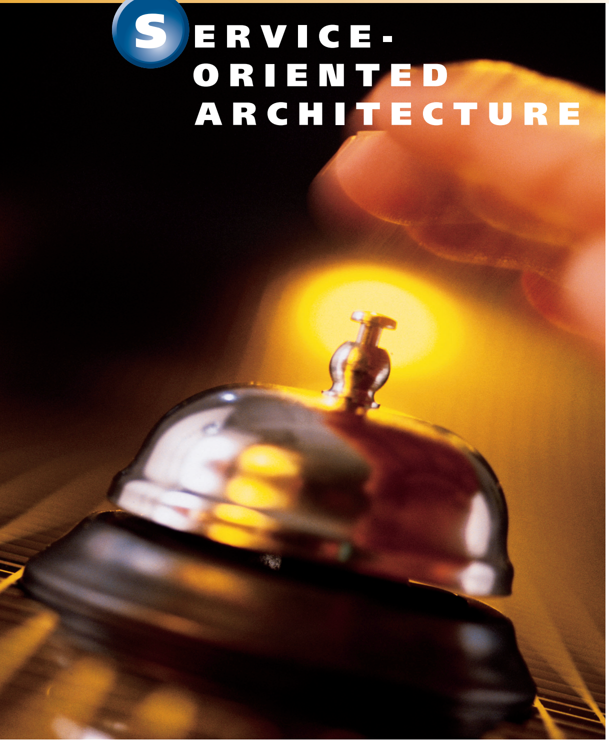
Article *in* IT Professional · April 2006

DOI: 10.1109/MITP.2006.51 · Source: IEEE Xplore

2 authors:

Gerardo Canfora
Università degli Studi del Sannio
298 PUBLICATIONS 7,217 CITATIONS
SEE PROFILE

Massimiliano Di Penta
Università degli Studi del Sannio
312 PUBLICATIONS 7,749 CITATIONS
SEE PROFILE



Testing Services and Service-Centric Systems: Challenges and Opportunities

Gerardo Canfora and Massimiliano Di Penta

Service-oriented architectures' unique features, such as dynamic and ultra-late binding, raise the need for new testing methods and tools.

Inside

With *service-oriented architectures* (SOAs), software is used and not owned, and operation happens on machines that are out of the user's control. Because a lack of trust prevents service computing's mainstream adoption, a key issue is providing users and system integrators the means to build confidence that a service delivers a function with the expected quality of service (QoS).

Unfortunately, many established testing methods and tools don't work with services. For example, to users and systems integrators, services are just interfaces. This hinders white-box testing methods based on code structure and data flow knowledge. Lack of access to source code also prevents classical mutation-testing approaches, which require seeding the code with errors.

In this article, we provide users and system integrators with an overview of SOA testing's fundamental technical issues and solutions, focusing on Web services as a practical implementation of the SOA model. We discuss SOA testing across two dimensions:

- Testing perspectives. Various stakeholders, such as service providers and end users, have different needs and raise different testing requirements.
- Testing level. Each SOA testing level, such as integration and regression testing, poses unique challenges.

TESTING PERSPECTIVES

SOA testing has similarities to commercial off-the shelf (COTS) testing. The provider can test a component only independently of the applications in which it will be used, and the system integrator is not able to access the source code to analyze and retest it. However, COTS are integrated into the user's system deployment infrastructure, while services live in a foreign infrastructure. Thus, the QoS of services can vary over time more sensibly and unpredictably than COTS. This QoS issue calls for specific testing to guarantee the service-level agreements (SLAs) stipulated with consumers.

Different stakeholders might want to test individual services or service-centric systems to ensure or verify SLA adherence. Each perspective has its specific requirements and issues.

Service developer

Aiming to release a highly reliable service, the service developer tests the service to detect the maximum possible number of failures. Among the other things, the developer also tries to assess the service's nonfunctional properties and its

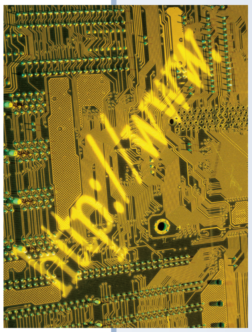
Published by the IEEE Computer Society

1520-9202/06/$20.00 © 2006 IEEE

ability to properly handle exceptions. Although testing costs are limited in this case (the developer does not have to pay when testing his own service), nonfunctional testing isn't realistic because it doesn't account for the provider and consumer infrastructure, and the network configuration or load.

Service provider

The service provider tests the service to ensure it can guarantee the requirements stipulated in the SLA with the consumer. Testing costs are limited. However, the provider can't use white-box techniques, and nonfunctional testing doesn't reflect the consumer infrastructure and network configuration or load.



Service integrator

The service integrator tests to gain confidence that any service to be bound to her own composition fits the functional and nonfunctional assumptions made at design time. Runtime binding can make this more challenging because the bound service is one of many possible or even unknown services. Furthermore, the integrator has no control over the service in use, which is subject to changes during its lifetime. Testing from this perspective requires service invocations and results in costs for the integrator and wasted resources for the provider.

Third-party certifier

The service integrator can use a third-party certifier to assess a service's fault-proneness. From a provider perspective, this reduces the number of stakeholders—and thus resources—involved in testing activities. However, the certifier doesn't test a service within any specific composition (as the integrator does) or from the same network configuration as the service integrator. This raises serious issues about the guaranteed confidence level.

SOA Testing Technologies

The following list of products, by no means comprehensive, provides an initial roadmap to SOA testing technology. The products cover unit, integration, and system testing and include both commercial and open source tools.

- **ANTS Load (http://www.red-gate.com/products/ANTS_Load) supports testing Web services behavior and performance under the stress of a multiple user load.**
- **e-TEST (http://www.empirix.com) suite for Web services provides ways to generate Web services test scripts, validate XML responses, and identify performance bottlenecks by server-side monitoring.**
- **JBlitz (http://www.clanproductions.com/ jblitz) carries out stress, performance, and functionality testing by generating different loading levels and records anomalies as they occur.**
- **SOAPscope (http://www.mindreef.com) supports testing SOAP transactions by monitoring communications among SOAP endpoints and analyzing Web Services Description Language (WSDL) and SOAP messages against industry standards, such as Web Services-Interoperability.**
- **SOA Test (http://www.parasoft.com) supports WSDL validation and functionality, performance, and scalability testing. It features a collaborative workflow in which engineers create test cases that the quality assurance team can leverage into scenario-based testing.**
- **TestMaker (http://www.pushtotest.com) is a free open-source framework for building automated test agents that check Web services for scalability, performance, and functionality. The commercial TestNetwork platform allows test agents to run on multiple test nodes concurrently.**
- **WebServiceTester (http://www.optimyz.com) is an integrated testing suite for functionality, performance, and regression testing Web services. It comprises automated test data generation and testing-service orchestrations based on Business Process Execution Language for Web Services.**

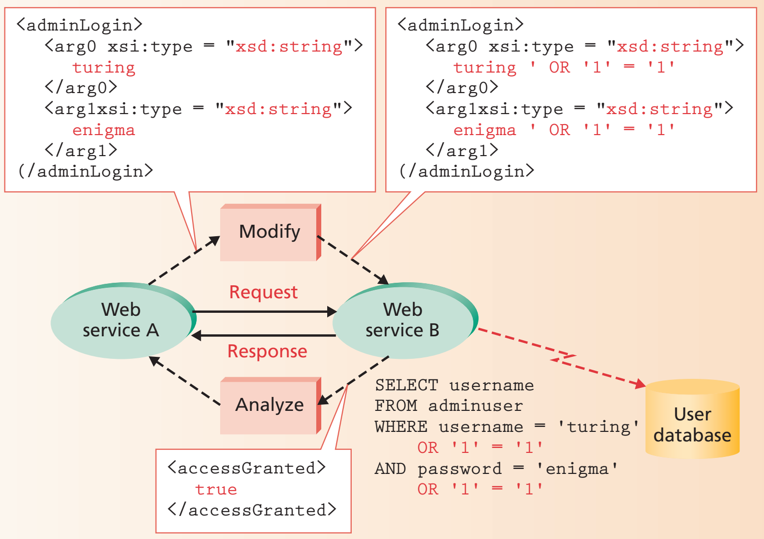
User

The user has no clue about service testing. His only concern is that the application he's using works while he's using it. For the user, SOA's dynamicity represents both a potential advantage—for example, better performance, additional features, or reduced costs—and a potential threat, such as unpredicted response time and availability. Making a service-centric system capable of self-retesting certainly helps reduce such a threat. Once again, however, testing from this perspective incurs costs and wastes resources. Imagine if a service-centric application installed on a PDA and connected to the network through a General Packet Radio Service were to suddenly self-test by invoking several services.

TESTING LEVELS

Testing comprises activities that validate a system's aspects. New challenges arise at each SOA testing level. (See the "SOA Testing Technologies" sidebar for a sampling of commercial and open source testing tools.)

Figure 1. Mutation strategy can help generate test cases



```
<adminLogin>
   <arg0 xsi:type = "xsd:string">
      turing
   </arg0>
   <arg1xsi:type = "xsd:string">
      enigma
   </arg1>
(/adminLogin>
```

```
<adminLogin>
   <arg0 xsi:type = "xsd:string">
      turing ' OR '1' = '1'
   </arg0>
   <arg1xsi:type = "xsd:string">
      enigma ' OR '1' = '1'
   </arg1>
(/adminLogin>
```

```
SELECT username
FROM adminuser
WHERE username = 'turing'
    OR '1' = '1'
AND password = 'enigma'
    OR '1' = '1'
```

```
<accessGranted>
   true
</accessGranted>
```

First, mutate the request message—for example, a SOAP message for Web services—then analyze the service response to detect faults. In this example, the SOAP message contains an XML-encoded username and password. Web service B queries the user database: If the adminuser table contains at least one row with such username and password, access is granted. If the mutations add the test ' OR '1' = '1' to both username and password, the query will always return a nonempty recordset, thus granting access. For more details, see the article by Offutt and Xu in the "Further Reading" sidebar.

Service functional testing

Generally speaking, providers and integrators can perform service-functional testing using techniques common in component or subsystem testing. To enable service discovery, providers publish services with more or less thorough specifications. At minimum, service specifications include a Web Services Description Language interface with an XML schema that specifies I/O types. These I/O types define value domains that providers or integrators can use to generate test cases according to a functional, black-box strategy.

In this context, *mutation strategies* assume an important role. Mutation strategies change, or mutate, inputs. Applying these changes to input messages, we can check whether these mutations produce observable effects in the service outputs, as in the SOAP example in Figure 1.

If a service has a semantically rich specification, we can apply more sophisticated test-generation strategies. For example, we can use preconditions to generate inputs and postconditions to create test oracles—that is, sources of expected results for the test cases.

Service nonfunctional testing

When acquiring a service, the user or service integrator, and the provider agree to an SLA, which stipulates that the provider ensure a specific QoS level to the user. The SLA often results from negotiations over what the provider can offer and what the user needs and can afford.

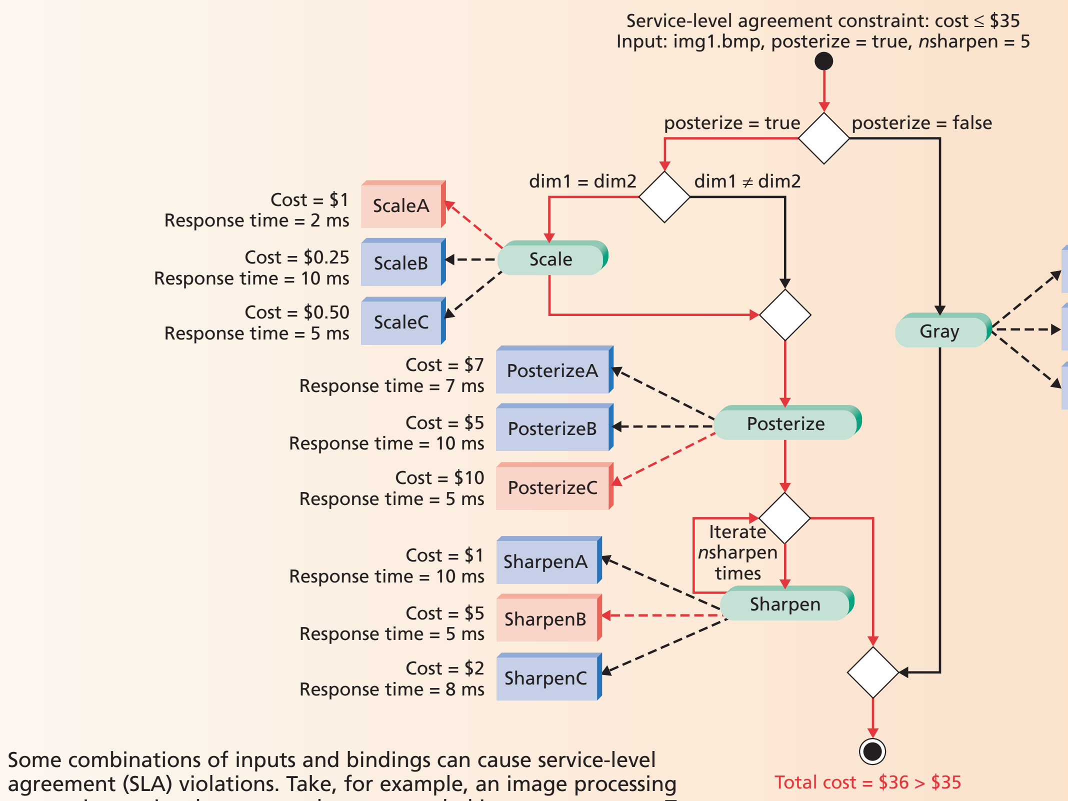
A service that doesn't respond with the expected QoS can be considered in violation of the SLA. External factors, such as heavy network or server load, can affect a service's performance. In some cases, however, an SLA violation can result from particular inputs that neither party considered when stipulating the agreement, as in the image-processing example in Figure 2.

We must therefore stress test SLAs. A possible strategy is to use search-based techniques (such as genetic algorithms) to generate test cases that are likely to violate the SLA. In the past, stress testing for real-time systems have used similar approaches.

Further Reading

- ➤ *Framework to support service testing:* **"Coyote: an XML-based Framework for Web Services Testing," W.T. Tsai and colleagues, *Proc. 7th IEEE Int'l Symp. High Assurance Systems Eng.* (HASE 02), IEEE CS Press, 2002, pp. 173-176.**
- ➤ *Service monitoring:* **"Smart Monitors for Composed Services," L. Baresi, C. Ghezzi, and S. Guinea, *Proc. 2nd Int'l Conf. Service-Oriented Computing* (ICSOC 04), ACM Press, 2004, pp. 193-202.**
- ➤ *Service regression testing:* **"Using Test Cases as Contract to Ensure Service Compliance across Releases," M. Bruno and colleagues, *Proc. 3rd Int'l Conf. Service Oriented Computing* (ICSOC 2005), LNCS 3826, Springer, 2005, pp. 87-100.**
- ➤ *SOAP message mutation:* **"Generating Test Cases for Web Services Using Data Perturbation," J. Offutt and W. Xu, *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 5, 2004, pp. 1-10.**
- ➤ *Use of graph-transformation techniques to generate test cases for Web services:* **"Automatic Conformance Testing of Web Services," R. Heckel and L. Mariani, *Proc. Fundamental Approaches to Software Eng.* (FASE 05), LNCS 3442, Springer, 2005, pp. 34-48.**

Figure 2. Testing quality of service through service compositions.



Some combinations of inputs and bindings can cause service-level agreement (SLA) violations. Take, for example, an image processing composite service that can produce gray-scaled images or posters. To obtain a poster, you scale the image (available only for square images), apply the posterize filter, and sharpen the image outline. You can repeat the sharpen step to make edges more visible. Suppose we have an SLA cost constraint of $35. Generating combinations of inputs and bindings for a given strategy, we discover that making a poster for a scalable image with at least five applications of the sharpening filter violates our cost constraint.

In this context, the QoS measured when executing the service with the generated inputs guides the search.

Integration testing

SOA's peculiar characteristics raise serious integration testing issues. Dynamic binding lets us define service-centric systems as a workflow of invocations to abstract interfaces that are bound to concrete services either just before workflow execution or even during enactment. Thus, classical integration and testing methods fail when they require the precise identification of systems components and their interrelationships.

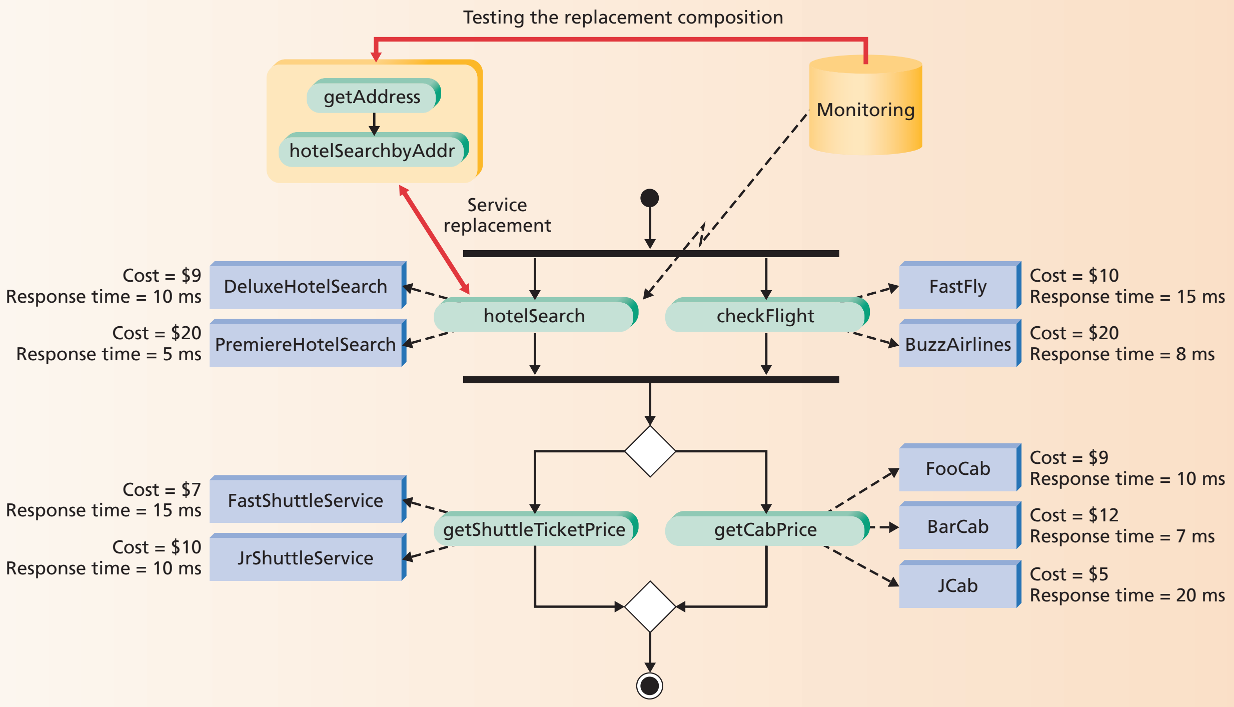
Because of dynamic binding, we must test a composite service partner link for all possible endpoints, as in Figure 3. The problem is similar to testing an object-oriented system because of polymorphism. However, with SOAs the problem is worse. Testing against all possible endpoints could be costly, and endpoints might be unknown at testing time.

We need heuristics that reduce the number of possible endpoints for testing. For example, if our binding strategy won't consider endpoints leading to a violation of a QoS constraint, it might not be useful to perform integration testing against them.

Nonfunctional testing also becomes more complex for service compositions that allow dynamic binding. The resulting QoS depends not only on the inputs but also on the combination of actual bindings. Test case generation is more complex and expensive for generating a combination of bindings and inputs that can cause SLA violations.

More complex scenarios uncover further integration testing needs. If a service is unavailable at runtime or if it must be replaced—for example, if it can't guarantee a given QoS—the replacement can be a newly discovered

Figure 3. Dynamic binding can exponentially increase the cost of service integration testing.



In this workflow, each abstract interface (hotelSearch, checkFlight, getShuttleTicketPrice, getCabPrice) can be bound to several possible concrete services. Each call site requires testing all possible concretizations, assuming they're known. Stronger testing criteria might require testing the workflow with all possible combinations of bindings (24 in this case). However, constraining bindings can reduce this number—for example, it's reduced to six if the cost must be within $15. Service replanning is also an issue. When a new composition replaces part of the workflow (because, for example, a service isn't available), the composition requires testing to ensure it delivers the same functionality. We can exploit monitored I/O from the replaced service for this purpose.

service or even a composition of services when no single service can fulfill the task. Despite the complex automatic discovery-and-composition mechanisms available, the integrator must adequately test the service or composition before using it. In this case, test time must be minimized because it affects runtime performance.

The Web Services-Interoperability (WS-I, http://www.ws-i.org) organization is working to solve such issues as well as broader interoperability concerns. See the "Web Services-Interoperability" sidebar for more information.

Regression testing

The provider controls the evolution strategy of the software behind a service, thereby further complicating testing. *Regression testing*—retesting a piece of software after a round of changes to ensure the changes don't adversely affect the delivered service—requires that service users and systems integrators know the service release strategy. However, the service provider might not be aware of who's using the service, so the provider can't notify those users about changes to the service's interface or implementation.

Integrators' lack of control over services truly differentiates SOAs from component-based systems and from most distributed systems. In those systems, one organization is often responsible for the distributed components it uses. During a service-centric system's lifetime, services can change their functionality or QoS, without necessarily exposing those changes through their interfaces. This affects the application's behavior and, from legal point of view, could be considered an SLA violation.

For example, imagine a service integrator using a service that computes a person's social security number (SSN). If the way the SSN is computed changed, the system's behavior would also change. The results would be similar if, for example, a hotel booking service stopped accepting bookings without a credit card.

Any service integrated into a composition requires regression testing to ensure its compliance with the integrator's assumptions and with the provider-integrator contract. Such a test can be triggered by periodic invocations or by notification of a service update, if the application server where the service is deployed supports service versioning and advertises it through the interface. To accomplish this, the provider can publish test suites as a part of the service specification, as Figure 4 illustrates. The integrator can complement them with additional test suites and a capture-replay strategy, monitoring and replaying previous service executions.

TESTING COSTS AND THE ROLE OF MONITORING

Regardless of the test method, testing a service-centric system requires the invocation of actual services on the provider's machine. This has several drawbacks. From the system integrator viewpoint, costs can be prohibitive if they must pay for services on a per-use basis. At the same time, massive testing could cause a denial-of-service phenomena for service providers. You might not be able to repeatedly invoke a service for testing when the service invocation affects the real world, as in the case of flight booking or payment services.

In most cases, service testing implies several service invocations, leading to unacceptably high costs and bandwidth use. Several cost-cutting approaches aim to reduce a test suite's size, but this might be not enough.

Service monitoring can play an important role in cutting testing costs. Monitoring is useful for such tasks as

- preventing failures—for example, by replacing a soon to be unavailable service with another equivalent;
- verifying that a service invocation meets given pre- and postconditions; and
- triggering recovery actions when needed.

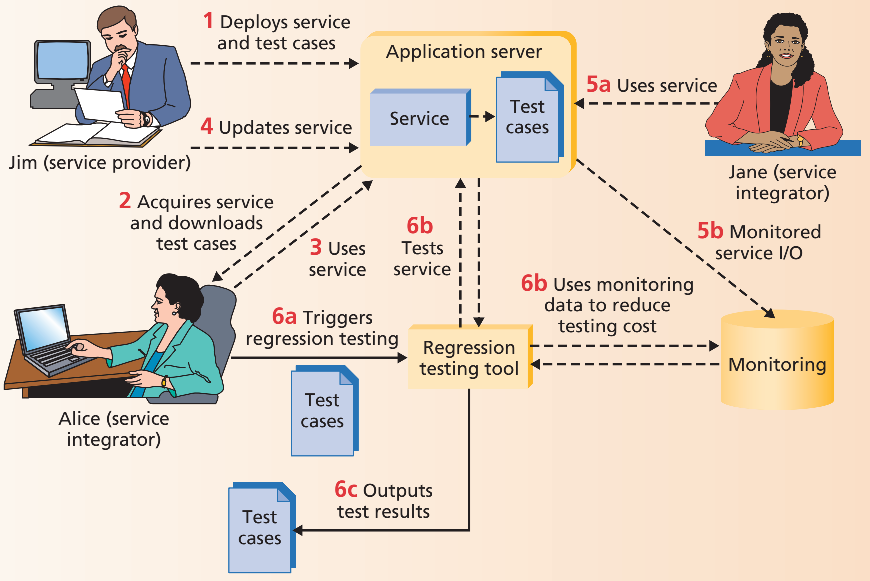
Web Services-Interoperability

Web Services-Interoperability (WS-I, http://www.ws-i.org) is an open industrial organization that promotes Web services interoperability across platforms, operating systems, and programming languages. WS-I helps define protocols for the interoperable exchange of messages between Web services. More specifically, WS-I delivers

- ***profiles*** **that provide implementation guidelines on using related Web services specifications together for optimal interoperability,**
- ***sample applications*** **that demonstrate Web services applications complying with WS-I guidelines, and**
- ***testing tools*** **that help determine whether the messages exchanged with a Web service conform to WS-I guidelines.**

Two notable WS-I testing tools are the *monitor* and the *analyzer*. The monitor provides an unobtrusive way to log Web service messages using a man-in-the-middle approach. The analyzer determines if a set of Web-service-related artifacts—messages, service descriptions, and UDDI universal description, discovery, and integration) entries—conform to the requirements in the WS-1 Basic Profile 1.0.

Figure 4. Service regression testing.



(1) Jim (a service provider) deploys a service with a test suite. (2) Alice (a service integrator) acquires the service and test suite, which she can complement with her own test cases. (3) She then regularly uses the service. After a while, (4) Jim updates the service, and (5a) Jane (another service integrator) regularly uses the new service with no problems. Meanwhile, (5b) a monitoring system monitors Jane's interaction. (6a) Alice tests the service. (6b) She can use data monitored from Jane's executions to reduce the number of service invocations during testing. For more details, see the article by Bruno and colleagues in the "Further Reading" sidebar.

Table 1. Highlights per testing dimension. Needs and responsibilities of each stakeholder are in black, advantages in green, issues and problems in red.

Testing levels	Testing perspectives				
	Developer	Provider	Integrator	Third-party	User
Service functional testing	White-box testing available Service specification available to generate test cases Nonrepresentative inputs	Needs service specification to generate test cases Limited cost Black-box testing only Nonrepresentative inputs	Needs service specification Needs test suite deployed by service provider Black-box testing only High cost	Assesses only selected services and functionality on behalf of someone else (should be impartial assessment of all services) Small resource use for provider (one certifier tests the service instead of many ntegrators) Only nonrepresentative inputs High cost	Service-centric application self-testing to check that it ensures functionality during runtime Services have no interface to allow user testing
Service nonfunctional testing	Necessary to provide nonfunctional specifications to provider and consumers Limited cost Nonrealistic testing environment	Necessary to check own ability to meet SLA stipulated with user Limited cost Testing environment might not be realistic	High cost Might depend on network configuration Difficult to check whether SLA is met	Assesses performance on behalf of someone else Small resource use for provider (one certifier tests the service instead of many integrators) Nonrealistic testing environment High cost	Service-centric application self-testing to check that it ensures performance during runtime
Integration testing	Can be service integrator on his own	NA	Service call coupling increases because of dynamic binding Must regression test a composition after reconfiguration or rebinding Quality-of-service testing must consider all possible bindings	NA	NA
Regression testing	Limited cost (service can be tested off-line) Unaware of who uses the service	Limited cost (service can be tested off-line) Can be aware that the service has changed but unaware of how it changed	Might be unaware that the service has changed High cost	Retests service during its lifetime only on behalf of integrators, not other stakeholders High cost Lower-bandwidth use than having many integrators Nonrealistic regression test suite	Service-centric application self-testing to check that it works during evolution

Monitoring can also help testing in various ways. We can use monitoring I/Os in a capture-replay scenario. Additionally, we can use monitoring to mimic service behavior during testing activities. This use reduces the number of invocations needed, as in Figure 4.

In a closed corporate environment, testing tools can directly access monitoring data. In an open environment, such information can't always be disclosed. However, the provider can publish services with a testing interface. Integrators execute test cases against this interface, which actually invokes the services only when necessary. In all the other cases, it forwards the innovation to a service simulator (or stub) that uses monitoring data to mimic the service behavior.

SOAs promise loosely coupled and dynamic connections that let applications take advantage of ever-expanding service capabilities, or even services unknown and unforeseen at design time. This is, of course, appealing for architects called to design systems in today's competitive, highly uncertain business landscape.

Table 1 summarizes SOA testing's main highlights for different stakeholders and testing levels. Central to all of these issues is that SOAs, and in particular Web services, aim to build systems comprising services that were developed by providers outside the enterprise. Thus, testing is boundless. Of course, users can always test services and their compositions independently when using the final service-centric system. However, testing would happen too late in the development cycle, thus making it difficult to identify a known error's source.

Devising testing strategies and methods for SOAs is still a young research area. The foreseeable diffusion of this architectural style depends on developing a trustable means of functional and nonfunctional testing of services and their compositions. ■

Gerardo Canfora *is a full professor of computer science in the Faculty of Engineering and the director of the Research Centre on Software Technology (RCOST) of the University of Sannio in Benevento, Italy. Contact him at canfora@unisannio.it.*

Massimiliano Di Penta *is assistant professor at the University of Sannio in Benevento, Italy and lead researcher at the Research Centre On Software Technology (RCOST). Contact him at dipenta@unisannio.it.*

Acknowledgments

This work is framed in the European Commission VI Framework IP Project SeCSE (Service-centric System Engineering; *http://secse.eng.it*), contract no. 511680.

For further information on this or any other computing topic, visit our Digital Library at http://www.computer.org/publications/dlib.