Service-Oriented Architectures Testing: A Survey

Chapter · January 2009

DOI: 10.1007/978-3-540-95888-8_4 · Source: DBLP

2 authors:

Gerardo Canfora
Università degli Studi del Sannio
298 PUBLICATIONS **7,218** CITATIONS
SEE PROFILE

Massimiliano Di Penta
Università degli Studi del Sannio
312 PUBLICATIONS **7,751** CITATIONS
SEE PROFILE

Service Oriented Architectures Testing: A Survey

Gerardo Canfora and Massimiliano Di Penta

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

Abstract. Testing of Service Oriented Architectures (SOA) plays a critical role in ensuring a successful deployment in any enterprise. SOA testing must span several levels, from individual services to inter-enterprise federations of systems, and must cover functional and non-functional aspects.
SOA unique combination of features, such as run-time discovery of services, ultra-late binding, QoS aware composition, and SLA automated negotiation, challenge many existing testing techniques. As an example, run-time discovery and ultra-late binding entail that the actual configuration of a system is known only during the execution, and this makes many existing integration testing techniques inadequate. Similarly, QoS aware composition and SLA automated negotiation means that a service may deliver with different performances in different contexts, thus making most existing performance testing techniques to fail.
Whilst SOA testing is a recent area of investigation, the literature presents a number of approaches and techniques that either extend traditional testing or develop novel ideas with the aim of addressing the specific problems of testing service-centric systems. This chapter reports a survey of recent research achievements related to SOA testing. Challenges are analyzed from the viewpoints of different stakeholders and solutions are presented for different levels of testing, including unit, integration, and regression testing. The chapter covers both functional and non-functional testing, and explores ways to improve the testability of SOA.

1 Introduction

Service Oriented Architectures (SOA) are rapidly changing the landscape of today and tomorrow software engineering. SOA allows for flexible and highly dynamic systems through service discovery and composition [1–3], ultra-late binding, Service Level Agreement (SLA) management and automated negotiation [4], and autonomic system reconfiguration [5–7]. More important, SOA is radically changing the development perspective, promoting the separation of the ownership of software (*software as a product*) from its use (*software as a service*) [8].

The increasing adoption of SOA for mission critical systems calls for effective approaches to ensure high reliability. Different strategies can be pursued to increase confidence in SOA: one possibility is to realize fault tolerant SOA by redundancy. For example, Walkerdine *et al.* [9] suggest that each service invoked by a system could be replaced by a container that invokes multiple, equivalent, services, and acts as a voter.

Another possibility is to continuously monitor a service-centric system during its execution [5] and apply corrective actions when needed: whenever an exceptional event is detected (e.g., a failure of a post condition or a violation of a Quality of Service—QoS—constraint), a recovery action is triggered. For example, if a temperature service, part of a whether forecasting system, is not available, an alternative one is located and invoked.

Whilst monitoring is an effective tool to build self-healing and self-repairing service-centric systems [10], it requires that adequate recovery actions be implemented for all possible exceptional events. Thus, testing remains a key process to reduce at a minimum the number of such exceptional events.

Several consolidated testing approaches, applied for years to traditional systems, apply to service-centric 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 of the existing testing techniques not directly applicable to test services and service-centric 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-centric programming languages, that all the possible (finite) bindings of a polymorphic component be known. These assumptions may not be true anymore for SOA, which exhibits run-time discovery in an open marketplace of services and ultra-late binding.

Examples of SOA unique features that add complexity to the testing burden include:

- systems based on services are intrinsically distributed, and this requires that QoS be ensured for different deployment configurations;
- services in a system change independently from each other, and this has impacts on regression testing;
- systems implement adaptive behaviors, either by replacing individual services or adding new ones, and thus integration testing has to deal with changing configurations;
- the limited trust service integrators and users have about the information service providers use to advertise and describe the service makes it complex the task of designing test cases;
- ownership over the system parts is shared among different stakeholders, and thus system testing requires the coordination of these stakeholders.

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 in 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 SLA failures.

This chapter overviews SOA testing, discussing issues and possible solutions for different testing levels and perspectives, introducing ways to improve service testability, and outlining open issues as an agenda for future research.

The chapter is organized as follows. Section 2 identifies key challenges of SOA testing, while Section 3 discusses testing needs, problems, and advantages from the perspectives of different stakeholders. Section 4 focuses on testing levels, namely unit, integration, regression, and non-functional testing, identifying problems and reviewing solutions in the literature. Section 5 discusses ways to increase testability in SOA. Finally, Section 6 concludes the chapter, highlighting open issues and future research directions.

2 Testing Challenges

When a new technology emerges and new software engineering approaches/processes have to be developed for such technology, a typical question is whether it is possible to reuse/adapt existing approaches, previously developed for other kinds of systems. In the context of this chapter, the question is whether testing approaches and techniques developed for traditional monolithic systems, distributed systems, component-based systems, and Web applications, can be reused or adapted to test service-centric systems. Answering this question requires an analysis of the peculiarities that make service-centric systems different from other systems as far as the testing process is concerned.

Key issues that limit the testability of service-centric systems include [11]:

- *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. When further knowledge is needed for testing purpose—e.g., a description of the dependencies between service operations, or a complete behavioral model of the service—either the service provider should provide such information, or the tester has to infer it by observing the service from the outside.
- *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, as it happens in (OO) systems in presence of polymorphism [12]. This is not true for SOA, where a system can be described by means of a workflow of abstract services that are automatically bound to concrete services retrieved from one or more registries during the execution of a workflow instance.
- *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 [13, 14]. In other words, a service may evolve, however this is not notified to integrators/users accessing it. As a consequence, systems making use of the service can unexpectedly change their behavior or, although the functional behavior is preserved, might not meet SLAs anymore.
- *lack of trust:* a service might provide descriptions of its characteristics, for instance a behavioral model, information about the QoS levels ensured, etc. Such information can, in theory, be used by integrators to discover the service, to comprehend

its characteristics, and to select it. However, it is not possible to guarantee that any piece of information a service provides corresponds to the truth. In general, a service provider may lie, providing incorrect/inaccurate description of a service's functional and non-functional behavior, thus influencing decisions integrators may take on the use of the service.
- *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 [11].

Any level of SOA testing needs to develop new strategies, or to adapt existing ones, to deal with these issues. For example, 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 SLAs stipulated with consumers.

3 Testing Perspectives

SOA introduces different needs and challenges for different stakeholders involved in testing activities, i.e. developers, providers, integrators, certifiers, and end-users. Table 1 provides a summary of *pros* and *cons* stakeholders experience in different testing levels.

Service developer. Aiming to release a highly reliable service, the service developer tests the service to detect the maximum possible number of failures. The developer owns the service implementation, thus s/he is able to perform white-box testing. Among the other things, the developer also tries to assess the service's non-functional properties and its 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), non-functional testing is not realistic because it does not account for the provider and consumer infrastructure, and the network configuration or load. In general, test cases devised by the developer may not reflect a real usage scenario.

Service provider. The service provider tests the service to ensure it guarantees the requirements stipulated in the SLA with the consumer. Testing costs are limited. However, the provider might not use white-box techniques, since s/he may be an organization/person different from who developed the service. Finally, non-functional testing does not reflect the consumer infrastructure and network configuration or load and, once again, test cases do not reflect the service real usage scenario.

Testing levels	**Testing Perspectives**				
	Developer	**Provider**	**Integrator**	**Third–party**	**End–User**
Functional testing	+White-box testing possible +Limited cost +Service specification available to generate test cases -Non-representative inputs	+Limited cost -Needs service specification to generate test cases -Black-box testing only -Non-representative inputs	+Tests the service in the context where it is used +White-box testing for the service composition -Black-box testing for services -High cost	+Small resource use for provider (one certifier tests the service instead of many integrators) +(Possible) impartial assessment -Assesses only selected services and functionality on behalf of someone else -Non-representative inputs -High cost	*Service-centric application self-testing to check that it ensures functionality at runtime* -Services have no interface to allow user testing
Integration testing	*Can be service integrator on his/her own*	NA	*Must regression test a composition after reconfiguration or rebinding* -Test service call coupling challenging because of dynamic binding	NA	NA
Regression testing	*Performs regression testing after maintaining/evolving a service* +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 s/he is using changed -High cost	+Lower-bandwidth usage than having many integrators -Re-tests the service during its lifetime only on behalf of integrators, not of other stakeholders -Nonrealistic regression test suite	*Service-centric application self-testing to check that it works properly after evolution*
Non-functional testing	*Needed to provide accurate non-functional specifications to provider and consumers* +Limited cost -Nonrealistic testing environment	*Necessary to check the ability to meet SLAs negotiated with service consumers* +Limited cost -The testing environment might not be realistic	-SLA testing must consider all possible bindings -High cost -Results might depend on network configuration	*Assesses performance on behalf of someone else* +Reduced resource usage for provider (one certifier tests the service instead of many integrators) -Nonrealistic testing environment	*Service-centric application self-testing to check that it ensures performance at runtime*

Table 1. Testing stakeholders: Needs and opportunities [11]. Roles/responsibilities are reported in italic, advantages with a (+), issues with a(-)

Service integrator. The service integrator tests to gain confidence that any service to be bound to her own composition fits the functional and non-functional assumptions made at design time. Runtime discovery and ultra-late 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. To ensure fault tolerance, the certifier can test the service on behalf of someone else, from the same perspective of a service integrator. However, the certifier does not test a service within any specific composition (as the integrator does), neither s/he performs testing from the same network configuration as the service integrator.

End-User. The user has no clue about service testing. His only concern is that the application s/he is using works while s/he is using it. For the user, SOA dynamicity represents both a potential advantage—for example, better performance, additional features, or reduced costs—and a potential threat. Making a service-centric system capable of self-retesting certainly helps reducing such a threat. Once again, however, testing from this perspective entails costs and wastes resources. Imagine if a service-centric application installed on a smart-phone and connected to the network through a wireless network suddenly starts a self-test by invoking several services, while the network usage is charged based on the bandwidth usage.

4 Testing Levels

This section details problems and existing solutions for different levels of testing, namely (i) unit testing of atomic services and service compositions, (ii) integration/interoperability testing, (iii) regression testing, and (iv) testing of non-functional properties.

4.1 Unit Testing

Testing a single atomic services might, in principle, be considered equivalent to component testing. As a matter of fact, there are similarities, but also differences:

– *observability:* unless when the service is tested by its developer, source code is not available. This prevents the possibility of using white-box testing techniques (e.g., code coverage based approaches). For stateful services, it would be useful to have models—e.g., state machines—describing how the state evolves. Such models could support, in addition to testing, service discovery [1, 2], and composition [3]. Unfortunately, these models are rarely made available by service developers and providers, due to the lack of proper skills to produce them, or simply to the lack

of time and resources. Nevertheless, approaches for a black-box reverse engineering of service specifications—for example inspired by likely invariant detection approaches [15]—are being developed [16, 17].
- *test data generation:* with respect to test data generation techniques developed so far—see for instance search-based test data generation [18]—the lack of source code observability makes test data generation harder. With the source code available, the generation heuristic is guided by paths covered when executing test cases and by distances from meeting control-flow conditions [19]; in a black-box testing scenario, test data generation can only be based on input types-ranges and output values.
- *complex input/output types:* with respect to existing test data generation techniques, most of which only handle simple input data generations, many real-world services have complex inputs defined by means of XML schema. Test data generation should therefore be able to generate test data according to these XML schema. Thus, operators of test data generation algorithms for service operations should be able to produce forests of trees, corresponding to XML representations or operation parameter. To this aim genetic programming can be used, as it was done by Di Penta *et al.* [20].
- *input/output types not fully specified:* to apply functional testing techniques, e.g., category partition, it is necessary to know boundaries or admissible values of each input datum. In principle, XML schema defining service operation input parameters can provide such an information (e.g., defining ranges by means of *xs:minInclusive* and *xs:maxInclusive* XML Schema Definition—XSD—tags, or defining occurrences by means of the *xs:maxoccurs*) tag. However, in the practice, this is almost never done, thus the tester has to specify ranges/admissible values manually.
- *testing cost and side effects:* as explained in Section 2, this is a crosscutting problem for all testing activities concerning services. Unit testing should either be designed to limit the number of service invocations—making test suite minimization issues fundamental—or services should provide a "testing mode" interface to allow testers to invoke a service without being charged a fee, without occupying resources allocated for the service production version, and above all without producing side effects in the real world.

Bertolino *et al.* [21] propose an approach and a tool, named *TAXI* (Testing by Automatically generated XML Instances) [22], for the generation of test cases from XML schema. Although not specifically designed for web services, this tool is well-suited for performing service black-box testing. The tool partitions domains into sub-domains and generates test cases using category-partition [23].

Bai *et al.* [24] proposes a framework to deal with the generation of test data for operations of both simple and composite services, and with the test of operation sequences. The framework analyzes services WSDLs (Web Service Description Language) interfaces to produce test data. As known, service operation parameters can be simple XSD types, aggregate types (e.g., arrays) or user defined types. For simple types the framework foresees a database where, for each type and for each testing strategy a tester might want to adopt, a facet defines default values, minimum and maximum length (e.g., for strings), minimum and maximum values (e.g. numeric types), lists of admissible val-

ues (strings) etc. This information is used to generate test data for simple parameter types, while for complex and user defined types the structure is recursively analyzed until leaves defined in terms of simple types are encountered.

Other than generating test data, the approach of Bai *et al.* [24] also generates operation sequences to be tested, based on dependencies existing between operations. Bai *et al.* infer operation dependencies from the WSDL, based on the following assumptions:

1. two operations are input dependent if they have common input parameters;
2. two operations are output dependent if they have common output parameters;
3. two operations are input/output dependent if there exist at least one output produced by an operation that is input for the other.

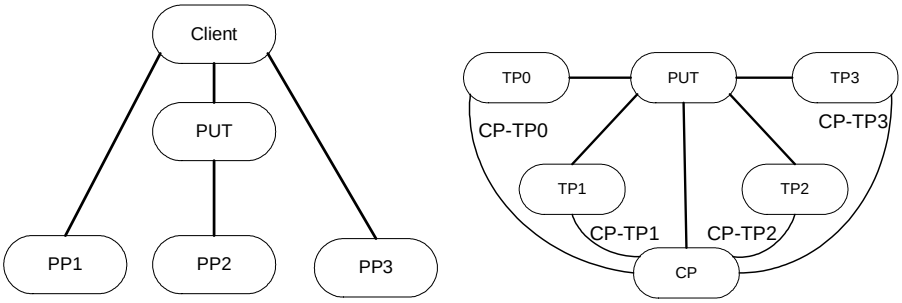
Conroy *et al.* [25] exploit user interfaces of legacy applications—hereby referred as Reference APplications (RAP)—to generate test data for web services—hereby referred as Target APplications (TAP). Their approach is based on the following key ideas:

1. GUI element may be (i) input data acceptors (e.g., input fields), (ii) action producers (e.g., buttons), (iii) output data retrievers (e.g., combo-boxes, tables), or (iv) state checkpoints (i.e., any GUI element that appear in a particular state in order to make the application to work correctly).
2. Data can be captured from the GUI using technologies conceived for accessibility purposes, e.g., to support screen reading for visually impaired.
3. Once data has been captured, the tester can map such data to inputs of the target application/service, and then enact the replay.

Unit testing of service compositions. Other than atomic services, WS-BPEL processes also need unit testing. In this section, we analyze WS-BPEL testing issues from the perspective of a developer, who has access to the WS-BPEL process itself, and thus can test it using white-box testing strategies. We recall that black-box testing of a WS-BPEL process is equivalent to service testing, discussed in the previous section, since a WS-BPEL process is viewed, from an external point of view, as a service exposing a WSDL interface. It is very unlikely that WS-BPEL processes are tested in isolation, since it would require to produce stubs for all the process partners. On the other hand, the possibility of testing a WS-BPEL process without requiring the availability of partners would allows for testing it even before partners have been developed, or in general without involving partners in the testing phase.

Li *et al.* [26] define a framework for WS-BPEL Unit Testing. Such a framework is, to some extent, similar to other unit testing frameworks, such as JUnit. The framework comprises four components:

1. *the WS-BPEL process composition model* (Fig. 1-a), which includes the WS-BPEL Process Under Testing (PUT) and its Partner Processes (PPs).
2. *the test architecture* (Fig. 1-b), where PP are replaced by stubs, named test processes (TPs), coordinated by a Control Process (CP). Thus, TPs simulate PP, plus they contain testing control structure and behavior (e.g., error-detecting logic).



(a) WS-BPEL process composition mode (b) Test architecture

Fig. 1. Li et al. [26] WS-BPEL Unit testing

3. *lifecycle management*, that starts-up or stops CPs and TPs by means of a User Interface (UI). To this aim, the CP provides a *beginTest* and *endTest* interface, and the TPs have *startTest* and *stopTest* interface. When the tester starts the testing activity by invoking *beginTest* through the CP interface, the CP on its own is responsible to start the TP it is coordinating. When all TPs complete their activity, the CP terminates as well.

A constraint-solver based approach for test case generation for WS-BPEL processes has been proposed by Yuan *et al.* [27]. Test case generation is performed in four steps:

1. WS-BPEL processes are represented as BPEL-Flow-Graphs (BFG), a variation of Control Flow Graphs (CFG) with fork, merge (the outgoing edge is activated by any incoming edge) and join (the outgoing edge is activated after all the incoming edges have been activated) nodes. Exception edges are made explicit, loop unfolded, and different control flow structures (e.g., switch and pick) are brought back to a single structure.
2. the BFG is traversed to identify test paths, defined as a partially-ordered list of WS-BPEL activities;
3. Infeasible paths are filtered out by means of constraint solving. Constraint solving is used to solve inequalities for path conditions: if no solution is found then the path is considered unfeasible. Test data produced by constraint solver can be complemented by means of randomly generated test data and manually-produced test cases.
4. Test cases are generated by combining paths and test data. To this aim, only input (e.g., message *receive*) and output (e.g., message *reply*) activities are considered, ignoring data handling activities such as assignments. Test case generation also requires to manually produce expected outputs, i.e., test oracles.

Tsai *et al.* [28] proposes a shift of perspective in service testing, moving from Individual Verification & Validation to Collaborative Verification & Validation. They propose a technique, inspired from blood testing, that overcomes the need for manually

defining oracles. Testing is performed by invoking multiple, equivalent services and then using a voter to detect whether a service exhibited a failure; in other words, if voting is performed upon three equivalent services, and one out of the three provides a different output, then it is assumed to contain a fault.

4.2 Integration Testing

SOA shifts the development perspective from monolithical applications to applications composed of services, distributed over the network, developed and hosted by different organizations, and cooperating together according to a centralized orchestration (e.g., described using WS-BPEL) or in a totally-distributed way (as it happens in a peer-to-peer architectures). Moreover, service compositions can dynamically change to fulfill varying conditions, e.g., a service unavailability, or the publication of a new service with better QoS. In such a scenario, it becomes crucial to test services for interoperability, i.e., to perform service integration testing. This need is also coped by industrial initiative such as WS-Interoperability[1], aimed at suggesting best practices to ensure high service interoperability.

The problem of integration testing for service compositions is discussed by Bucchiarone *et al.* [29]. With respect to traditional integration testing, the following issues have to be handled:

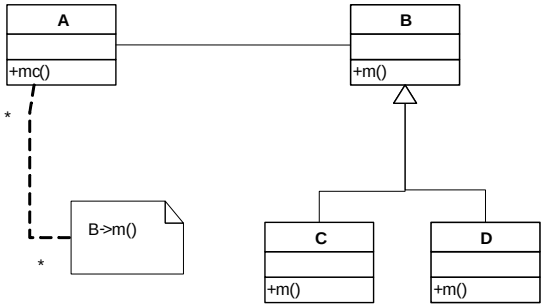
- *The lack of information about the integrated components*, which makes the production of stubs very difficult.
- *The impossibility of executing the integrated components in testing mode*: this requires to limit the involvement of services in testing activities, since it may cause costs, resource consumption, and even undesired side effects, as discussed earlier in this chapter.

As highlighted in our previous work [11], dynamic binding between service composition and partners make integration testing of service composition even more difficult. An invocation (hereby referred as "*abstract service*") in a service composition (e.g., in a WS-BPEL process) can be dynamically bound to different end points.

Dynamic binding can be based on QoS constraints and objectives [30, 31] or else on constraints defined upon service functional and non-functional properties (e.g., one might want to avoid services from a specific provider). This requires to perform integration testing with all possible bindings, as it happens when performing integration testing of OO systems [32]. For example, in Fig. 2-a the invocation from method *mc()* of class *A* to method *m()* of class B can be bound to *B::m()*, *C::m()* or *D::m()*. Regarding data-flow, for OO systems there is a *call coupling* between a method *mA* and a method *mB* when *mA* invokes *mB*, and a *parameter coupling* when *mB* has an input parameter that *mA* defines or an output parameter that *mA* uses. For such couplings, it can be possible to define:

- *call coupling paths*, beginning from the call site where (*mA* invokes *mB*) and finishing with a return site.

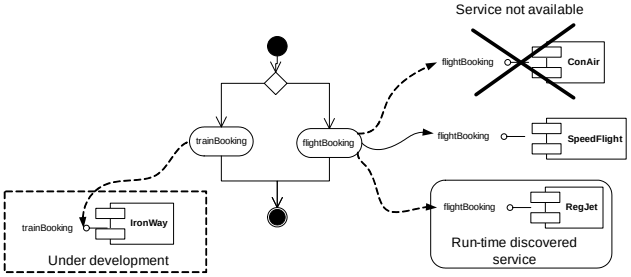
[1] *http://www.ws-i.org/*



(a) Dynamic binding in OO

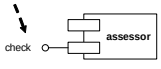
```
public class A{                          public class B{
  int j;                                 …
  B b1;                                  public int mb(int k){
  …                                        …
  public int ma(){                       int y=k-1; // First use in callee
    …                                    …
    int x=10;  //Last def before call    int j=y++; //Last def before return
    if (j > 0) {                         …
      z=b1.mb(x); //Call site            return j;  }
      …                                  }
      int v=x*z; //First use after return
     }
    …
   }
…}
```

(b) Call coupling in OO



(c) Dynamic binding in SOA

```
<assign><copy>
   <from variable="a"/>       Last def before call
   <to variable="request"/>
</copy></assign>
...
<invoke name="invokeAssessor" partner="assessor"   Call site
        portType="asns:riskAssessmentPT"
        operation="check"
        inputVariable="request"
        outputVariable="riskAssessment">
   <target linkName="receive-to-assess"/>
   <source linkName="assess-to-setMessage"
           transitionCondition=
           "bpws:getVariableData('riskAssessment',
           'risk')='low'"/>
   <source linkName="assess-to-approval"
           transitionCondition="
           bpws:getVariableData('riskAssessment',
           'risk')!='low'"/>
</invoke>
...
<assign><copy>
   <from variable="riskAssessment"/>       First use after return
   <to variable="bar"/>
</copy></assign>
```



(d) Call coupling in SOA

Fig. 2. Dynamic binding and call-coupling in OO and SOA

– *parameter coupling paths:* (see Fig. 2-b) for input parameters, it starts from the last definition of the parameter before the call, continues through the call site, and ends with the first use of the parameter in the callee. Similarly, for output parameters, it starts from the last definition of output parameter in the callee before the return, and ands with the first use after call in the caller.

Given the above paths, coverage criteria for OO systems include:

1. *all call sites:* test cases must cover all call-sites in the caller;
2. *all coupling definitions:* test cases must cover, for each coupling definition, at least one coupling path to at least one reachable coupling use;
3. *all coupling uses:* test cases must cover, for each coupling definition, at least one coupling path to each reachable coupling use;
4. *all coupling paths:* test cases must cover all coupling paths from the coupling definition to all coupling uses.

Ideally, such criteria could apply to test service compositions in the presence of dynamic binding. However:

– parameter coupling criteria cannot be applied as they are above defined, since the partners are invoked on providers' servers and thus viewed as black box entities. In other words, for parameters defined in the caller, the only visible use is the partner invocation, while definitions in the callee are only visible from return points (see Fig. 2-d).
– the set of all possible concrete services for a given abstract service might not be known *a-priori*. In fact, some services available at design time might not be available at run-time, or else, if dynamic discovery in an open marketplace is used, new services can be made available at runtime (see Fig. 2-c).
– achieving the above coverage criteria for all possible bindings can be overly expensive and, in general, not possible when, as highlighted by Bucchiarone *et al.* [29], services cannot be invoked in testing mode.

Tsai *et al.* defines a framework for service integration testing, named Coyote [33], supporting 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 specifications, while the test engine interacts with the web service being tested and provides tracing information to the test master.

Bertolino and Polini [34] propose an approach to ensure the interoperability between a service being registered in a Universal Description, Discovery and Integration (UDDI) registry and any other service, already registered, that can potentially cooperate with the service under registration. As Bertolino and Polini suggest, an UDDI registry should change its role from a passive service directory towards the role of an active "audition" authority.

SOA calls for integration testing aimed at SLA, too. Since each abstract service in a workflow can be bound to a set of possible concrete services (equivalent from functional point-of-view, but with different QoS), there might be particular combinations of bindings that can cause SLA violations. This point is further discussed in Section 4.4.

An important issue, dicussed by Mei *et al.* [35], concerns the use WS-BPEL makes of XPath to integrate workflow steps. Services take as input, and produces as output, XML documents. Thus problems can arise when information is extracted from an XML document using an XPath expression and then used as input for a service. The authors indicate the case where a service looks for the availability of DSL lines in a given city. If the city is provided using a XML document like such as:

```
<address>
  <state>
    <name>Beijing</name>
    <city>Beijing</city>
  </state>
</address>
```

or

```
<address>
  <state />
  <city>Beijing</city>
</address>
```

Different XPaths can return different values in the two cases. For example, both `/city/` and `/state/city/` return Beijing for the first document, while for the second `/state/city/` returns an empty document. To perform BPEL data-flow testing, Mei *et al.* rewrite XPaths using graphs, named XPath Rewriting Graphs (XRG), that make explicit different paths through a XML schema. For example, the XPath expression `//city/` can be considered as `*//city/*` or just `*//city/`. An XRG is composed of *rewriting nodes*, containing the original expression (`//city/` in our case), having edges directed to *rewritten nodes*, representing the different forms of an XPath (`*//city/*` and `*//city/` in our case). Then, Mei *et al.* build models named X-WSBPEL, that combine CFG extracted from WS-BPEL with the XRG. To perform data-flow testing on the X-WSBPEL models, Mei *et al.* define a set of data-flow testing def-use criteria, based on variable definition and usages over XPaths.

An approach for data-flow testing of service compositions is proposed by Bartolini *et al.* [36], who essentially adapts data-flow criteria conceived for traditional CFGs to WS-BPEL processes to build, from a WS-BPEL process, an annotated CFG, and then generate test cases achieving different data-flow coverage criteria.

4.3 Regression Testing

An issue that makes service-centric systems very different from traditional applications is the lack of control a system integrator has over the services s/he is using. System integrators select services to be integrated in their systems and assume that such services will maintain their functional and non-functional characteristics while being used. However, this is not the case: a system exposed as a service undergoes—as any other system—maintenance and evolution activities. Maintenance and evolution strategies are out of the system integrators control, and any changes to a service may impact all the

systems using it. This makes service-centric systems different from component-based systems: 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. In such a context, several evolution scenarios may arise:

- Changes that do not require modifying the service interface and/or specification, e.g., because the provider believes this is a *minor* update, As a consequence, the changes remain hidden from whoever is using the service.
- Changes that do not affect the service functional behavior, but affect its QoS. Once again, these are not always documented by the service provider and, as a consequence, the QoS of the system/composition using such a service can be affected.
- A service can be, on its own, a composition of other services. 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.

The above scenarios raises the need for an integrator to periodically re-test the service she/he is using, to ensure that they still meet functional and non functional expectations. To this aim, Di Penta *et al.* [13, 14] propose to complement service descriptions with a *facet* providing test cases, in the form of XML-based functional and non functional assertions. A facet is a (XML) document describing a particular property of a service, such as its interface (in this case the facet can correspond to the WSDL interface), its QoS, a SLA template that can be negotiated with the potential service users [4]. A faceted approach to describe services [37], extends the UDDI registry with an XML-based registry containing different kinds of facets. However, additional facets could simply be linked to the service WSDL without the need for requiring a customized, proprietary registry. Fig. 3 shows an excerpt of the SOA conceptual model [38] related to facets describing *properties* of a service. A facet is *specified* by means of a facet specification expressed using a language, e.g., WSDL for interfaces, or WS-Agreement for SLA. As shown in the figure, a service can be described by a number of facets related to signature, operational semantics, QoS, test cases, etc.

When a service integrator discovers a service and wants to use it, s/he downloads the testing facet and uses it to check whether the service exhibits the functional and non-functional properties s/he expects. As a matter of fact, a test suite can be used to support developers' comprehension of software artifacts: this has been investigated by Ricca *et al.* [39], who empirically assessed the usefulness of acceptance test cases expressed as Fit (Framework for Integrated Testing) [40] tables in requirement comprehension tasks. Then, the test suite is used to periodically re-test the service to check whether it still meets the integrator functional and non-functional needs.

Fig. 4 describes a possible scenario for the test case publication and regression testing process taken from [14]. 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.

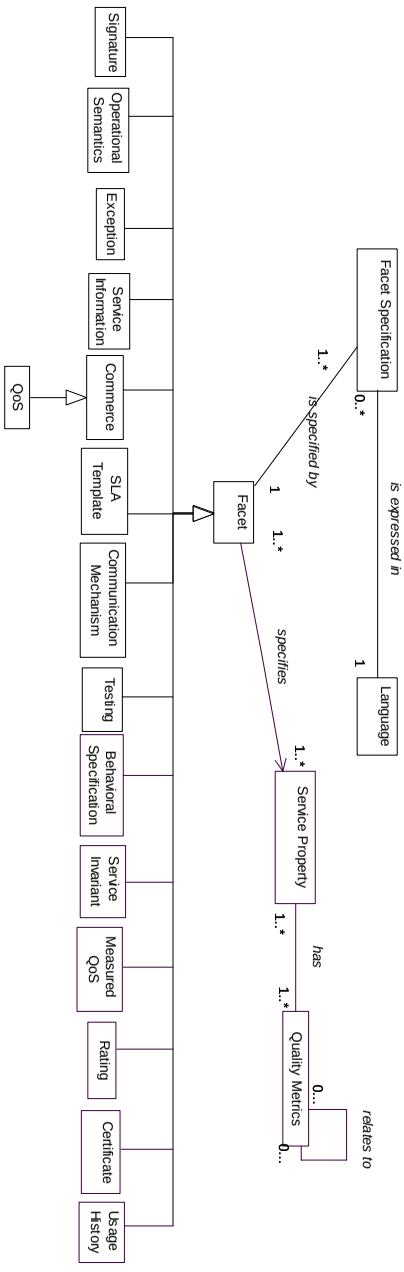


Fig. 3. Faceted specification of a service

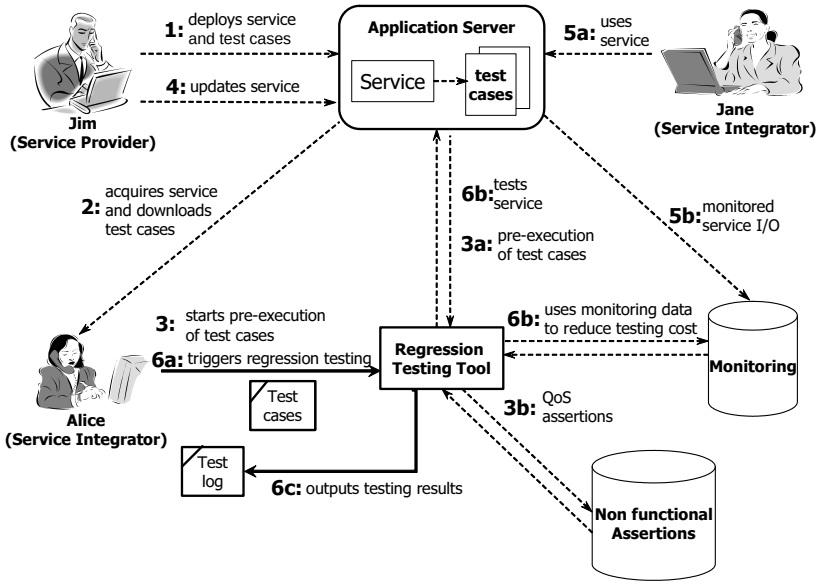


Fig. 4. Service regression testing: test generation and execution process [14]

1. At time t_0 *Jim* deploys a service, e.g., a *RestaurantService* that allows an user to search for restaurants, gets restaurants info and check for availability. The service is deployed together with its test suite (facet).
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, performs a pre-execution of the test suite, and measures 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, 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.

According to Di Penta *et al.* [14], facets to support service regression testing can either be produced manually by the service provider or by the tester, or can be generated from unit test cases of the system exposed as a service, as described in Section 5.

The idea of using test cases as a form of contract between service providers and service consumers [13, 14] also inspired the work of Dai *et al.* [41]. They foresee a contract-based testing of web services. Service contracts are produced, in a Design by Contract [42] scenario, using OWL-S models. Then, Petri nets are used for test data generation. As for Di Penta *et al.*, test cases are then used to check whether during the time a service preserves the behavior specified in the contract.

Regression test suite reduction is particularly relevant in SOA, given the high cost of repeated invocations to services. Ruth and Tu [43, 44] defines a safe regression test suite selection technique largely based on the algorithm defined by Rothermel and Harrold [45] for monolithic application. The proposed technique requires the availability of CFGs (rather than source code) for service involved in the regression testing activity. The idea is that CFGs should be able to highlight the changes that can trigger regression testing, while shielding the source code. Unfortunately, such assumption is, in most cases, pretty stronger in that service providers are unlikely to expose service CFGs.

4.4 Non-Functional Testing

Testing non-functional properties is crucial in SOA, for a number of reasons:

- service providers and consumers stipulate a SLA, in which the provider guarantees to consumers certain pieces of functionality with a given level of QoS. However, under certain execution conditions, caused by unexpected consumer inputs or unanticipated service load, such QoS level could not be met;
- the lack of service robustness, or the lack of proper recovery actions for unexpected behaviors can cause undesired side effects even on the service side or on the integrator side;
- services are often exposed over the Internet, thus they can be subject to security attacks, for instance by means of SQL injection.

Robustness Testing. Different approaches have been proposed in the literature to deal with different aspects of service non-functional testing. Martin *et al.* [46, 47] presents a tool for automatic service robustness testing. First, the tool automatically generates a service client from the WSDL. Then, any test generation tool for OO programs could be used to perform service robustness testing. In particular, the authors use JCrasher[2] [48], a tool that generates JUnit tests. The authors applied their tool on services such as Google search and Amazon; although no major failures were detected, the authors indicated that, sometimes, services hanged up, suggesting a possible presence of bugs.

Ensuring robustness also means ensuring that exceptional behavior is properly handled and that the service reacts properly to such behavior. However, very often the error recovery code is not properly tested. Fu *et al.* [49] proposes an approach for exception code data-flow testing, suited to web services developed in Java, but that can be applied to any Java program. In particular, they define the concept of *exception-catch* (e-c) link, i.e., the link between a fault-sensitive operation and a *catch* block in the program to be tested, and define coverage criteria for such links.

[2] *www.cc.gatech.edu/jcrasher/*

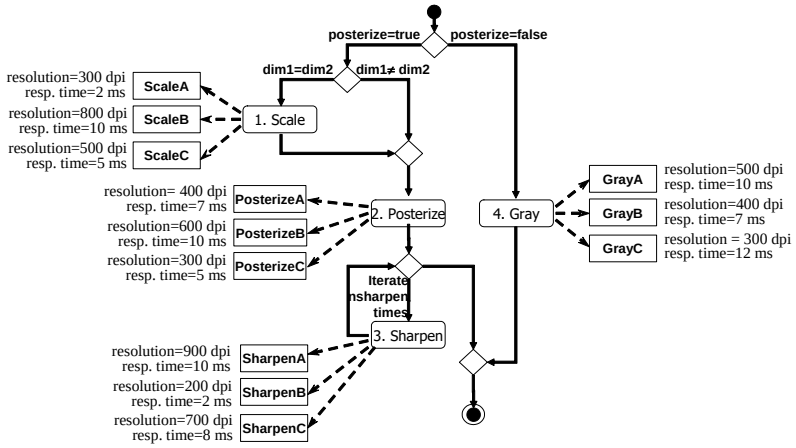


Fig. 5. SLA testing of composite service: example [50]

SLA Testing. SLA testing deals with the need for identifying conditions for which a service cannot be able to provide its functionality with a desired level of SLA. Before offering a SLA to a service consumer, the service provider would limit the possibility that it can be violated during service usage. Di Penta *et al.* [20] proposes an approach for SLA testing of atomic and composite services using Genetic Algorithms (GA). For a service-centric system such violations can be due to the combination of different factors, i.e., (i) inputs, (ii) bindings between abstract and concrete services, and (iii) network configuration and server load. In the proposed approach, GAs generate combinations of inputs and bindings for the service-centric system causing SLA violations. For atomic services, the approach generates test data as inputs for the service, and monitors the QoS exhibited by the service during test invocations. At minimum, the approach is able to monitor properties such as response time, throughput, and reliability. However, also domain specific QoS attributes [50] can be monitored. Monitored QoS properties are used as fitness function to drive the test data generation (the fitness is used to select the individuals, i.e., test cases, that should "reproduce" by means of GA crossover and mutation operators). In other words, the closer a test case is to produce a QoS constraint violation, the better it is. To allow for test data generation using GA, test inputs are represented as a forest, where each tree represents a service input parameter according to its XML schema definition. Proper crossover and mutation operators are defined to generate new test data from existing one, i.e., evolving the population of GA solutions.

For service compositions, in particular where dynamic binding between *abstract services* in the workflow and *concrete services* is possible, GA aims at generating combinations of bindings and inputs that cause SLA violations. In fact, for the same inputs, different bindings might result in a different QoS. Fig. 5 shows an image processing workflow. Let us make the following assumptions:

– the service provider guarantees to consumers a response time less than 30 ms and a resolution greater or equal to 300 dots per inches (dpi);

– a user provides as inputs an image having a size smaller than 20 Mb (which constitutes a precondition for our SLA), $posterize = true$, $dim1 = dim2$, and $nsharpen = 2$;
– the abstract services are bound to *ScaleC*, *PosterizeC*, *SharpenB*, and *GrayA*, respectively.

In this case, while the response time will be lower-bounded by 24 ms, and therefore the constraint would be met, the resolution of the image produced by the composite service would be of 200 dpi, corresponding to the minimum resolution guaranteed by the invoked services. In other cases, the scenario can be much more complex, in particular when combinations of inputs for each service invoked in the workflow can, on its own, contribute to the overall SLA violation.

Testing for Dependability. The approach above described has a specific purpose, i.e., generating service inputs (and bindings) that violate the SLA. More generally, for services used in a business-critical scenario, it would be useful to generate service inputs that cause:

– security problems, e.g., allowing unauthorized access to sensible data;
– service hang up;
– unexpected, unhandled exceptions;
– a behavior not observable in the service response.

Many of the existing approaches rely on SOAP message perturbation. According to Voas [51], perturbation allows for altering an application internal state without modifying its source code.

To this aim, Offutt and Xu [52] foresee an approach to generate web service test cases through Data Perturbation (DP). Basically, the process consists in modifying the request messages, resending them to the web service and analyzing how the response message changed with respect to the original message. Specifically, the mechanism foresees two types of perturbation: Data Value Perturbation (DVP), Remote Procedure Call (RPC) Communication Perturbation (RCP), and Data Communication Perturbation (DCP). DVP produces data input modifications according to the types described in the service parameter XML schema, and is largely inspired to the concepts of boundary value testing [53]. For example, for numeric values minimum, maximum and zero are considered, while for strings the perturbation consists of generating minimum length or maximum length strings, and upper-casing or lower-casing the string. RCP modifies message in RPC and data communication, in particular considering data used within programs and data used as inputs to database queries. For data used within programs, mutation mechanisms are defined for different data types; for example a numeric datum n can be changed to to $1/n$ (*Divide*), $n \times n$ (*Multiply*), $-n$ (*Negative*), $|n|$ (*Absolute*). Another perturbation exchanges the order of arguments. When data is used to query a database, perturbation aims at SQL injection, which occurs when an user input, incorrectly filtered for string literal, escapes characters embedded in SQL statements, causing the execution of unauthorized queries.

Offutt and Xu provide the following example: if a service expects 2 input parameters, *username* and *password*, and then checks for the presence of username and password in a database table using the query:

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

then, the *Unhautorized* perturbation mechanism appends to both username and password the string `' OR '1'='1`. As a result, the query becomes:

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

always providing authentication/access. They also provide an example of data perturbation: given a SOAP message containing a data structure describing, for instance, a book:

```
<book>
  <ISBN>0-781-44371-2</ISBN>
  <price>69.99</price>
  <year>2003</year>
</book>
```

Data perturbation entails sending (i) empty instances of the data structure, (ii) an allowable number of instances, (iii) duplicating instances from the message and (iv) removing an instance from the message. For example, the above message can be mutated by means of a duplication, as follows:

```
<book>
  <ISBN>0-781-44371-2</ISBN>
  <price>69.99</price>
  <year>2003</year>
</book>
<book>
  <ISBN>0-781-44371-2</ISBN>
  <price>69.99</price>
  <year>2003</year>
</book>
```

Such a perturbation test would be useful, for instance, to check whether the different behavior the service exhibits in presence of such a duplicate record is observable from the response message. A response like “true” or “false”—just indicating whether the insertion was successful or not—is not sufficient to see how many records have been inserted.

Looker *et al.* [54] propose to assess service dependability by means of fault injection techniques. They propose an approach and a tool named WS-FIT (Web Service Fault Injection Technology) inspired from network fault injection. Basically, they decode network messages based on SOAP and inject errors in these messages. They define

Level	Reference	Description
Functional	Bertolino *et al.* [21, 22]	Test data generation from XML schema using the category partition strategy
	Bai *et al.* [24]	WSDL test data generation from XSD types
	Conroy *et al.* [25]	Exploit user interfaces of legacy systems to perform capture-replay over web services
	Li *et al.* [26]	Define a framework for BPEL unit testing
	Yuan *et al.* [27]	Constraint-solver based approach for WS-BPEL test case generation
	Tsai *et al.* [28]	Builds automatic oracles by invoking multiple equivalent services and comparing results through a voter
Integration	Tsai *et al.* [33]	Coyote: a framework for service test integration testing
	Bertolino and Polini [34]	The UDDI registry plays the role of audition authority to ensure service interoperability
	Bucchiarone *et al.* [29]	Discusses problems related to service integration testing
	Mei *et al.* [35]	Data-flow testing of WS-BPEL process with focus on XPath expressions
	Bartolini *et al.* [36]	Data-flow testing of service composition
Regression	Di Penta *et al.* [13, 14]	Generation of service testing facet from system JUnit test suite. Regression testing of service from the integrator side
	Dai *et al.* [41]	Services accompanied with a contract; Petri nets for generating test cases
	Ruth and Tu [43, 44]	Applies the safe regression technique of Rothermel and Harrold [45] to web services where source code is unavailable.
Non-functional	Martin *et al.* [46, 47]	Service robustness testing based on Axis and JCrasher
	Fu *et al.* [49]	Framework and criteria to perform exception code coverage
	Di Penta *et al.* [20]	Search-based SLA testing of service compositions
	Offutt and Xu [52]	Perturbation of SOAP messages
	Looker *et al.* [54]	WS-FIT: Service fault injection approach and tool

Table 2. Summary of service testing approaches

a fault model for web services, considering different kinds of faults, namely (i) physical faults, (ii) software faults (programming or design errors), (iii) resource management faults (e.g., memory leakage), and (iv) communication faults. In particular, Looker *et al.* provide a detailed fault model for communication, considering message duplication, message omission, the presence of extra (attack) messages, change of message ordering, or the presence of delays in messages.

When injecting a fault, the service may react in different ways: (i) the service may crash, (ii) the web server may crash, (iii) the service may hang, (iv) the service may produce corrupted data, or (v) the response can be characterized message omission, duplication, or delay. Thus, the aim of the fault injection is to determine to what extent the service is able to properly react to faults seed in input messages, with proper exception handling and recovery actions, without exhibiting any of the above failures.

Reference	Description
Tsai *et al.* [56]	Complement WSDL with I/O dependencies, external dependencies, admissible invocation sequences, functional descriptions
Tsai *et al.* [57]	Extension of UDDI registry with testing features to test service I/O behavior
Heckel and Mariani [58]	Extension of UDDI registry with testing information; use of graph transformation systems to test single web services
Heckel and Lohmann [59]	Complement services with contracts at different (model, XML, and implementation) levels
Di Penta *et al.* [13, 14]	Generate service test cases ("facet") to be used as contracts for regression testing purposed from system test cases
Bai *et al.* [60]	Framework for testing highly dynamic service-centric environments
Bertolino *et al.* [61]	Puppet: QoS test-bed generator for evaluating the QoS properties of a service under development
Bertolino *et al* [62]	Test-bed generator from extra-functional contracts (SLA) and functional contracts (modeled as state machines)

Table 3. Approaches for improving service testability

4.5 Summary Table

Table 2 briefly summarizes the approaches described in this section, providing, for each work, the reference, the testing level, and a short description.

5 Improving Testability

The intrinsic characteristics of SOA are a strong limit for service testability. According to Tsai *et al.* [55], service testability should account for several factors, such as:

- the service accessibility, i.e., source code accessibility, binary accessibility, model accessibility, signature accessibility;
- the "pedigree" of data describing a service. The origin of such information can constitute a crucial aspect since integrators might not trust it;
- the level of dynamicity of the SOA.

Below, we describe some approaches, summarized in Table 3, that deal with service testability.

Tsai *et al.* [56] propose that information to enhance service testability should be provided by extending WSDL. Specifically, the following information is proposed:

1. *Input-Output-Dependency:* they introduce a new complex type in the WSDL schema, named *WSInputOutputDependenceType*, to account for dependencies between service operation inputs and outputs;
2. *Invocation Sequences:* this represents a crucial issue when testing services, since a service might delegate to another service the execution of a particular task. The tester might want to be aware of such dependencies. A WSDL schema type *WSInvocationDependenceType* is defined to represent caller-callee relationships between services;

3. *Functional Description:* a service functional description can also be included as a form of hierarchy, to enhance black-box testing. To this aim Tsai *et al.* define two sub elements, *WSFParents* and *WSFChildren*, that permit the definition of functional structures. However, other than defining a hierarchical form for functional descriptions, the authors did not indicate which information such description must contain;
4. *Sequence Specification:* licit operation calling sequences are described using regular expressions, e.g., $OpenFile \cdot (ReadLine|WriteLine)* \cdot Close$ indicates that a file opening must be followed by zero or more read or write, then followed by a file closing.

Tsai *et al.* [57] also propose to extend the UDDI registry with testing features: the UDDI server stores test scripts in addition to WSDL specifications.

Heckel and Mariani [58] use graph transformation systems to test individual web services. Like Tsai *et al.*, their method assumes that the registry 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 the behavior.

Heckel and Lohmann [59] propose to enable web service testing by using Design by Contract [42]. In a scenario where a service provider offers a piece of functionality through a service and a service consumer requests it, provider-consumer contracts describe the offered and required functionality. Heckel and Lohmann foresee service contract at different levels: (i) at model level, understandable by humans, (ii) at XML level, to be integrated into existing service standards like WSDL, and (iii) at implementation level, realized for example using JContract, and used by tools such as Parasoft Jtest[3] to generate test cases. Clearly, a mapping among different levels is required.

As described in Section 4.3, it would be useful to complement a service with facets containing test cases that can be used as a "contract" to ensure that the service, during the time, preserves its functional and non-functional behavior. To this aim, Di Penta *et al.* [13, 14] propose to generate service test cases from test suites developed for the software system implementing the features exposed by the service. This is useful since, in most cases, developers are reluctant to put effort in producing a service test suite. On the other hand, legacy system unit test suites, such as Junit, are very often in use. Many software development methodologies, e.g., test-driven development, strongly encourage developers to produce unit test suites 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 by 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. Such a translation is supported by the tester, that selectively specifies the operation invocation within the JUnit test suite

[3] *http://www.parasoft.com*

that should be left in the testing facet and those that should be evaluated and translated in literals, since it regards operations not accessible from the service interface or operations not involved in the testing activity.

Testing activities in highly-dynamic environments, such as SOA with run-time discovery binding, require the presence of a test broker able to decouple test case definition from their implementation, and the testing environment from the system under test [60]. This allows for a run-time binding and reconfiguration of test agents in case the service under test change (i) interface operations, (ii) its bindings, or (iii) steps of its underlying process.

An important issue when testing service-centric systems is the need for test-beds that can be used to evaluate QoS properties of a service under development, but also, for instance, to evaluate/test the QoS of a service composition—as described in Section 4.4—without having component services available and, even if they are available, avoiding to invoke them to limit side-effects and reduce the testing costs. Bertolino *et al.* [61] proposed *Puppet* (Pick UP Performance Evaluation Test-bed), a test bed generator used to evaluate QoS properties of services uder development. Also, they propose an approach and a tool [62] to generate stubs from extra functional contracts expressed as SLA and functional contracts expressed as state machines. Further details can be found in Chapter XX of this book.

6 Summary

Software testing has long been recognized as one of the most costly and risky activities in the life-cycle of any software system. With SOA, the difficulty of thoroughly testing a system increases because of the profound changes that this architectural style induces on both the system and the software business/organization models. Testing challenges derive primarily from the intrinsic dynamic nature of SOA and the clear separation of roles between the users, the providers, the owners, and the developers of a service and the piece of software behind it. Thus, automated service discovery and ultra-late binding mean that the complete configuration of a system is known only at execution time, and this hinder integration testing, while per-use-charge of a service affects unit testing and QoS testing of services and their compositions. Whilst SOA testing is a recent area of investigation, numerous contributions have been presented in the literature, primarily in the areas of unit testing of services and orchestrations, integration testing, regression testing, and testing of non-functional properties. The literature also reports several investigations into means to improve the testability of services and service-centric systems. Nevertheless, several problems remain open, calling for additional research work:

– *Combining testing and run-time verification*. The dynamic nature of SOA entails that testing-based validation needs to be complemented with runtime verification. On the one hand, testing is unable to cope with certain aspects of a service-centric system validation, primarily because of the impossibility to test all—often unforeseen—system configurations. On the other hand, run-time monitoring, while able to deal with the intrinsic dynamism and adaptiveness of SOA, is unable to provide confidence that a system will behave correctly before it is actually deployed. Thus,

additional research is needed to fully comprehend the role of testing and monitoring in the validation of a service-centric system and to devise systematic ways to combine them with the aim of increasing the confidence and reducing the cost of validation [63].
- *Improving testability*. For system integrators and, more in general, users a service is just an interface, and this hinders the use of traditional white-box coverage approaches. Service usage may be charged based on the number of invocations; even worst, many services have permanent effects in the real world—e.g. booking a restaurant table—and this makes stress testing approaches infeasible. Lack of observability and cost of repeated invocations could be addressed by publishing a (state-full) model of a service and providing testing interface to query and change the state of a service without affecting the real world. Additional research work is needed to devise the right formalisms to express the models and to help standardizing the interfaces. As developing the models is costly and error prone, means to reverse engineering them from the observation of a service behavior are to be devised [16, 17].
- *Validating fully decentralized systems*. Currently, the most widespread approach to service composition is orchestration, which entails the use of an engine that executes a process description and coordinates the invocations of services. With this approach, systems are intrinsically distributed —services run in independent administrative domains— but control remains centralized. Nowadays, different forms of compositions are emerging, such as peer-to-peer choreography, which makes control, in addition to services, completely decentralized. Fully decentralized compositions opens new scenarios, for example, propagation of a query in a network of active services that may subscribe it based on an introspective knowledge of their capabilities [64], which poses new challenges to testing and monitoring.

SOA has great potentials for reducing costs and risks of enterprise systems, improving efficiency and agility of organizations, and mitigating the effects of changing technology. However, many of the benefits of SOA become challenges to testing services and service-centric systems. Addressing these challenges requires a coherent combination of testing, run-time monitoring, and exception management.

7 Acknowledgements

This work is partially founded by the European Commission VI Framework IP Project SeCSE (Service Centric System Engineering) (*http://secse.eng.it*), Contract No. 511680, and by the Italian Department of University and Research (MIUR) FIRB Project ART-DECO.

References

1. Paolucci, M., Kawamura, T., Payne, T.R., Sycara, K.: Semantic matching of web services capabilities. In: First International Semantic web Conference (ISWC 2002). Volume 2348., Springer-Verlag (2002) 333–347

2. Bromberg, Y.D., Issarny, V.: INDISS: interoperable discovery system for networked services. In: Middleware 2005, ACM/IFIP/USENIX, 6th International Middleware Conference, Grenoble, France, November 28 - December 2, 2005, Proceedings. (2005) 164–183
3. Pistore, M., Traverso, P.: Assumption-based composition and monitoring of web services. In: Test and Analysis of web Services. Springer (2007) 307–335
4. Di Nitto, E., Di Penta, M., Gambi, A., Ripa, G., Villani, M.L.: Negotiation of Service Level Agreements: An architecture and a search-based approach. In: Service-Oriented Computing - ICSOC 2007, Fifth International Conference, Vienna, Austria, September 17-20, 2007, Proceedings. (2007) 295–306
5. Baresi, L., Guinea, S.: Towards dynamic monitoring of WS-BPEL processes. In: Service-Oriented Computing - ICSOC 2005, Third International Conference, Amsterdam, The Netherlands, December 12-15, 2005, Proceedings. (2005) 269–282
6. Kephart, J., Chess, D.: The vision of autonomic computing. IEEE Computer (2003)
7. Hinchey, M.G., Sterritt, R.: Self-managing software. Computer **39** (2006) 107–109
8. Turner, M., Budgen, D., Brereton, P.: Turning software into a service. IEEE Computer **36** (2003) 38–44
9. Walkerdine, J., Melville, L., Sommerville, I.: Dependability properties of P2P architectures. In: 2nd International Conference on Peer-to-Peer Computing (P2P 2002), 5-7 September 2002, Linköping, Sweden. (2002) 173–174
10. Baresi, L., Ghezzi, C., Guinea, S.: Smart Monitors for Composed Services. In: Proc. 2nd International Conference on Service Oriented Computing (ICSOC'04), New York, USA, ACM (2004) 193–202
11. Canfora, G., Di Penta, M.: Testing services and service-centric systems: Challenges and opportunities. IT Professional **8** (2006) 10–17
12. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for Java. ACM Trans. Softw. Eng. Methodol. **14** (2005) 1–41
13. Bruno, M., Canfora, G., Di Penta, M., Esposito, G., Mazza, V.: Using test cases as contract to ensure service compliance across releases. In: Service-Oriented Computing - ICSOC 2005, Third International Conference, Amsterdam, The Netherlands, December 12-15, 2005, Proceedings. (2005) 87–100
14. Di Penta, M., Bruno, M., Esposito, G., Mazza, V., Canfora, G.: web services regression testing. In Baresi, L., Nitto, E.D., eds.: Test and Analysis of web Services. Springer (2007) 205–234
15. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Trans. Software Eng. **27** (2001) 99–123
16. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008. (2008) 501–510
17. Ghezzi, C., Mocci, A., Monga, M.: Efficient recovery of algebraic specifications for stateful components. In: IWPSE '07: Ninth international workshop on Principles of software evolution, New York, NY, USA, ACM (2007) 98–105
18. McMinn, P.: Search-based software test data generation: a survey. Softw. Test., Verif. Reliab. **14** (2004) 105–156
19. Wegener, J., Baresel, A., Sthamer, H.: Evolutionary test environment for automatic structural testing. Information & Software Technology **43** (2001) 841–854
20. Di Penta, M., Canfora, G., Esposito, G., Mazza, V., Bruno, M.: Search-based testing of service level agreements. In: Genetic and Evolutionary Computation Conference, GECCO 2007, Proceedings, London, England, UK, July 7-11, 2007. (2007) 1090–1097
21. Bertolino, A., Gao, J., Marchetti, E., Polini, A.: Systematic generation of XML instances to test complex software applications. In: Rapid Integration of Software Engineering Tech-

niques, Third International Workshop, RISE 2006, Geneva, Switzerland, September 13-15, 2006. Revised Selected Papers. (2006) 114–129
22. Bertolino, A., Gao, J., Marchetti, E., Polini, A.: TAXI - a tool for XML-based testing. In: 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007, Companion Volume. (2007) 53–54
23. Ostrand, T., Balcer, M.: The category-partition method for specifying and generating functional tests. Communications of the Association for Computing Machinery **31** (1988)
24. Bai, X., Dong, W., Tsai, W.T., Chen, Y.: Wsdl-based automatic test case generation for web services testing. In: IEEE International Workshop on Service-Oriented System Engineering (SOSE), Los Alamitos, CA, USA, IEEE Computer Society (2005) 215–220
25. Conroy, K., Grechanik, M., Hellige, M., Liongosari, E., Xie, Q.: Automatic test generation from GUI applications for testing Web services. In: Software Maintenance, 2007. ICSM 2007. IEEE International Conference on. (2007) 345–354
26. Li, Z., Sun, W., Jiang, Z.B., Zhang, X.: BPEL4WS unit testing: Framework and implementation. In: 2005 IEEE International Conference on web Services (ICWS 2005), 11-15 July 2005, Orlando, FL, USA. (2005) 103–110
27. Yuan, Y., Li, Z., Sun, W.: A graph-search based approach to BPEL4WS test generation. In: Proceedings of the International Conference on Software Engineering Advances (ICSEA 2006), October 28 - November 2, 2006, Papeete, Tahiti, French Polynesia. (2006) 14
28. Tsai, W.T., Chen, Y., Paul, R.A., Liao, N., Huang, H.: Cooperative and group testing in verification of dynamic composite web services. In: 28th International Computer Software and Applications Conference (COMPSAC 2004), Design and Assessment of Trustworthy Software-Based Systems, 27-30 September 2004, Hong Kong, China, Proceedings. (2004) 170–173
29. Bucchiarone, A., Melgratti, H., Severoni, F.: Testing service composition. In: In Proceedings of the 8th Argentine Symposium on Software Engineering (ASSE'07). (2007)
30. Canfora, G., Di Penta, M., Esposito, R., Villani, M.L.: A framework for QoS-aware binding and re-binding of composite Web services. Journal of Systems and Software (2008 – in press)
31. Canfora, G., Di Penta, M., Esposito, R., Villani, M.L.: An approach for QoS-aware service composition based on genetic algorithms. In: Genetic and Evolutionary Computation Conference, GECCO 2005, Proceedings, Washington DC, USA, June 25-29, 2005, ACM (2005) 1069–1075
32. Binder, R.V.: Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Publishing Company (2000)
33. 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
34. Bertolino, A., Polini, A.: The audition framework for testing Web services interoperability. In: 31st EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA 2005), 30 August - 3 September 2005, Porto, Portugal, IEEE Computer Society (2005) 134–142
35. Mei, L., Chan, W.K., Tse, T.H.: Data flow testing of service-oriented workflow applications. In: 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008. (2008) 371–380
36. Bartolini, C., Bertolino, A., Marchetti, E., Parissis, I.: Data flow-based validation of web services compositions: Perspectives and examples. In V, R.d., Di Giandomenico, F., Muccini, H., C. Gacek, M.V., eds.: Architecting Dependable Systems. Springer (2008)
37. Walkerdine, J., Hutchinson, J., Sawyer, P., Dobson, G., Onditi, V.: A faceted approach to service specification. In: International Conference on Internet and web Applications and Services (ICIW 2007), May 13-19, 2007, Le Morne, Mauritius. (2007) 20

38. Colombo, M., Di Nitto, E., Di Penta, M., Distante, D., Zuccalà, M.: Speaking a common language: A conceptual model for describing service-oriented systems. In: Service-Oriented Computing - ICSOC 2005, Third International Conference, Amsterdam, The Netherlands, December 12-15, 2005, Proceedings. (2005) 48–60
39. Ricca, F., Torchiano, M., Massimiliano Di Penta, Mariano Ceccato, P.T.: Using acceptance tests as a support for clarifying requirements: A series of experiments. Information and Software Technology (2008 – in press)
40. Mugridge, R., Cunningham, W.: Fit for Developing Software: Framework for Integrated Tests. Prentice Hall (2005)
41. Dai, G., Bai, X., Wang, Y., Dai, F.: Contract-based testing for web services. In: 31st Annual International Computer Software and Applications Conference (COMPSAC 2007), 24-27 July 2007, Beijing, China. (2007) 517–526
42. Meyer, B.: Object-Oriented Software Construction, 2nd Ed. Prentice-Hall, Englewood Cliffs, NJ (1997)
43. Ruth, M., Tu, S.: Towards automating regression test selection for Web services. In: Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007. (2007) 1265–1266
44. Ruth, M., Oh, S., Loup, A., Horton, B., Gallet, O., Mata, M., Tu, S.: Towards automatic regression test selection for Web services. In: 31st Annual International Computer Software and Applications Conference (COMPSAC 2007), 24-27 July 2007, Beijing, China. (2007) 729–736
45. Rothermel, G., Harrold, M.J.: A safe, efficient regression test selection technique. ACM Trans. Softw. Eng. Methodol. **6** (1997) 173–210
46. Martin, E., Basu, S., Xie, T.: WebSob: A tool for robustness testing of web services. In: 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007, Companion Volume. (2007) 65–66
47. Martin, E., Basu, S., Xie, T.: Automated testing and response analysis ofweb services. In: 2007 IEEE International Conference on web Services (ICWS 2007), July 9-13, 2007, Salt Lake City, Utah, USA. (2007) 647–654
48. Csallner, C., Smaragdakis, Y.: JCrasher: an automatic robustness tester for Java. Softw., Pract. Exper. **34** (2004) 1025–1050
49. Fu, C., Ryder, B.G., Milanova, A., Wonnacott, D.: Testing of Java Web services for robustness. In: Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004. (2004) 23–34
50. Canfora, G., Di Penta, M., Esposito, R., Perfetto, F., Villani, M.L.: Service composition (re)binding driven by application-specific QoS. In: Service-Oriented Computing - ICSOC 2006, 4th International Conference, Chicago, IL, USA, December 4-7, 2006, Proceedings. (2006) 141–152
51. Voas, J.M.: Fault injection for the masses. IEEE Computer **30** (1997) 129–130
52. Offutt, J., Xu, W.: 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** (2004) 1–10
53. Beizer, B.: Software Testing Techniques 2nd edition. International Thomson Computer Press (1990)
54. Looker, N., Munro, M., Xu, J.: Ws-fit: A tool for dependability analysis of Web services. In: 28th International Computer Software and Applications Conference (COMPSAC 2004), Design and Assessment of Trustworthy Software-Based Systems, 27-30 September 2004, Hong Kong, China, Proceedings. (2004) 120–123
55. Tsai, W.T., Gao, J., Wei, X., Chen, Y.: Testability of software in service-oriented architecture. In: 30th Annual International Computer Software and Applications Conference (COMPSAC 2006), 17-21 September 2006, Chicago, Illinois, USA. (2006) 163–170

56. 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
57. Tsai, W.T., Paul, R.J., Cao, Z., Yu, L., Saimi, A.: Verification of Web services using an enhanced UDDI server. In: Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems. (2003) 131–138
58. Heckel, R., Mariani, L.: Automatic conformance testing of web services. In: Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings. (2005) 34–48
59. Heckel, R., Lohmann, M.: Towards contract-based testing of web services. Electr. Notes Theor. Comput. Sci. **116** (2005) 145–156
60. Bai, X., Xu, D., Dai, G.: Dynamic reconfigurable testing of service-oriented architecture. In: 31st Annual International Computer Software and Applications Conference (COMPSAC 2007), 24-27 July 2007, Beijing, China. (2007) 368–378
61. Bertolino, A., De Angelis, G., Polini, A.: A QoS test-bed generator for web services. In: Web Engineering, 7th International Conference, ICWE 2007, Como, Italy, July 16-20, 2007, Proceedings, Springer (2007) 17–31
62. Bertolino, A., De Angelis, G., Frantzen, L., Polini, A.: Model-Based Generation of Testbeds for Web Services. In: Testing of Communicating Systems and Formal Approaches to Software Testing – TESTCOM/FATES 2008. Number 5047 in Lecture Notes in Computer Science, Springer (2008) 266–282
63. Canfora, G., Di Penta, M.: SOA: Testing and self-checking. In: keynote speech at the International Workshop on Web Services - Modeling and Testing (WS-MATE 2006). (2006)
64. Forestiero, A., Mastroianni, C., Papadakis, H., Fragopoulou, P., Troisi, A., Zimeo, E.: A scalable architecture for discovery and composition in P2P service networks. In: Grid Computing: Achievements and Prospects. Springer (2008)