Whitening SOA Testing

Cesare Bartolini[1], Antonia Bertolino[1], Sebastian Elbaum[2], Eda Marchetti[1]
[1]ISTI - CNR
Via Moruzzi 1 - 56124 Pisa, Italy
{cesare.bartolini, antonia.bertolino, eda.marchetti}@isti.cnr.it
[2]University of Nebraska
Lincoln, NE, USA
elbaum@cse.unl.edu

ABSTRACT

Service Oriented Architectures (SOAs) are becoming increasingly popular and powerful. Fueling that growth is the availability of independent web services that can be cost-effectively composed with other services to provide richer functionality. The reasons that make these systems easier to build, however, also make them more challenging to test. Independent web services usually provide just an interface, enough to invoke them and develop some general (black-box) tests, but insufficient for a tester to develop an adequate understanding of the integration quality between the application and independent web services. To address this lack we propose a "whitening" approach to make web services more transparent through the addition of an intermediate coverage service. The approach, named Service Oriented Coverage Testing (SOCT), provides a tester with feedback about how a whitened service, called a Testable Service, is exercised. In this paper we introduce the SOCT approach, implement an instance of it, and perform a preliminary study to show its feasibility and potential value. SOCT enables SOA white-box testing, while maintaining SOA flexibility, dynamism and loose coupling.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; H.3.5 [**Information Storage and Retrieval**]: On-line Information Services—*Web-based services*

General Terms

Design

Keywords

White-box testing, coverage adequacy criteria, testing web services, service-oriented architecture

1. INTRODUCTION

Web services are applications with a public interface that can be invoked by other services and applications through the web. Their usage has increased dramatically in the last years [15] as companies develop their web systems and find it cost-effective to integrate their own services with those developed and managed by third parties (this development paradigm is referred to as following a Service Oriented Architecture (SOA) [20], and the integration of distributed web services as an *orchestration*). Failures in web services, however, are common and their impact more obvious as their popularity and interdependencies increase. For example, a recent failure in Amazon's storage web service affected many companies relying on it [2].

For a service orchestrator, building effective tests that can detect failures in the interaction among the composed services is challenging for two reasons. First, even if best practices [26] are followed by the developer to test a service to ensure its quality, nothing guarantees that it will operate smoothly as part of a dynamic distributed system made of multiple orchestrated but autonomous services. Second, the orchestrator can often only access the service interface to derive test cases and determine the extent of the testing activity. This limited visibility means that the orchestrator has to rely heavily upon an interface whose documentation is often limited and possibly inconsistent with the true system behavior, especially with services that undergo frequent updates [12].

Researchers have developed several approaches to address these challenges. In particular, much work has focused on test case generation from improved service interfaces (i.e., more precise behavioral specifications) [24, 25, 27], and on the detection of inconsistencies between a service interface description and its behavior [13]. One trait existing test approaches share is that they treat the web services as *black boxes* [9], focusing on the external behavior but ignoring the internal structure of the services. This trait follows the very nature of web services, which are meant to be *implementation neutral.* From a testing perspective, though, this is a pity. White-box approaches are in fact a well-known valuable complement to black-box ones [23], as coverage information can provide an indication of the thoroughness of the executed test cases, and can help identify additional test cases which might detect more faults.

We have conceived, however, an approach by which services can be made more transparent to an external tester while maintaining the flexibility, dynamism and loose coupling of SOAs. In essence, we propose an approach for "whitening" the current black-box testing approaches for SOA applications through the use of dedicated testing services that blend naturally into the existing paradigm. The approach is thus called Service Oriented Coverage Testing (SOCT). The added transparency from test whitening will increase testability, letting the application developer gain detailed feedback about how a service orchestration is exercised during validation.

This feedback can then be used by developers of orchestrations to design tests that cover more of the space of potential behavior of the third party service as it interacts with their application, to determine whether a coverage adequacy criterion that includes the third party service structure has been reached, or to detect possible updates in the implementation of third party service that may affect their application behavior. On the other end, third party service providers may be enticed to provide such extended testing interface as a way to implement continuous quality assurance checks, or may be required to do so as part of a service quality agreement.

From a broader perspective, whitening of SOA testing relies on lying down a *governance* framework to realize inter-organization testing at the orchestration level [7]. The term governance comprehends the complex set of rules, policies, practices and responsibilities by which a complex system is controlled and administered, and it well applies to the governing of a SOA system, including its integration testing [7]. The improved testability of SOA composite applications would require the contribution and availability of all stakeholders, who should be convinced it constitutes a win-win framework. More specifically, to enable this approach we require for: 1) the developer of a provided service to instrument the code so to enable the monitoring of the execution of target program entities, 2) the testing service provider (who could coincide with the service developer) to track test execution results, and 3) the service integrator to request testing information through a standardized published web service testing interface. Note that the approach fits naturally in the service-oriented model by providing the test information as a service controlled by a service interface.

We briefly introduced SOCT in our previous work [3]. In here we further motivate the problem, define the approach, implement an instance of it, and illustrate its application in a preliminary case study. In the next section we overview the problem domain, the approach, and its main challenges. In Section 3 we define SOCT concepts, its components and possible realization scenarios. A study is illustrated in Section 4. Related work is overviewed in Section 5, while conclusions are drawn in Section 6.

2. MOTIVATION

Let us consider the case of a SOA developer building an Integrated Travel Reservation System (ITRS) for a Travel Agent (TA) customer. ITRS is meant to provide the TA with a single-point access to several common on-line services including flight booking and hotel reservation. The ITRS developer builds a set of services to consolidate travel information according to the TA requirements, and uses trusted Global Distribution System (GDS) service providers such as Sabre[1] and Travelport [2], to obtain flight and hotel information.

[1]http://www.sabre-holdings.com/
[2]http://www.travelport.com/

The developer intends to diligently test the new services according to the recommended best practices for SOA testing (see, e.g., [26]). To validate the functioning of the composite SOA system, i.e., the cooperation between the newly built front-end services and the linked GDS services, the developer can choose between two (extreme and non-exclusive) options, which are referred to as off-line and on-line testing [24]. In the first case, the developer could mock the GDS services. Realizing this solution implies building and maintaining a complex test environment simulating those services. Such a solution has a high realization cost and might not be 100% reliable, not only because there may not be enough information to faithfully reproduce the behavior of the external services, but also because the latter could change without notice, making the developer's stubs obsolete.

Second, the developer could test ITRS on-line, i.e., by directly accessing the GDS services, and exercise the whole integrated system at once. A significant limitation of this alternative is that when the test session is completed, the ITRS developer will not be aware of the extent to which the GDS integration has been tested, and therefore how comprehensive the test cases have been. This lack of awareness can lead to some service elements left untested, and therefore to potential detectable problems going unnoticed (e.g., a branch that was not covered in GDS leads to the return of flight information in a format that ITRS is not currently handling and that makes it fail ungraciously) or to redundant testing effort (e.g., addition of tests that traverse paths in GDS that have been already exercised and do not add new value).

Our proposed approach, SOCT, can mitigate the limitations associated with the on-line testing activity by enabling the application of traditional white-box testing techniques to SOA applications.

2.1 The SOCT Approach

The envisioned testing scenario for SOCT is depicted in Figure 1. The traditional actors in SOA testing [9] are the service provider, who can test their service before deployment, and the service integrator, who has to test the orchestrated services. SOCT introduces a new stakeholder for SOA testing at the orchestration-level, a service provider called `TCov` who sits between the ITRS developer and the GDS service provider. Imagine the `TCov` provider as a trusted provider of services that deliver coverage information on GDS as it is tested by the ITRS developer.

To realize the SOCT scenario, we need for the company that provides the GDS services to instrument them (callout 1 in Figure 1) to enable the collection of coverage data, not differently from how instrumentation is normally performed for traditional white-box testing. We call the instrumented services *Testable Services*. As the ITRS developer invokes the GDS services during on-line testing (callout 2), coverage measures are collected from the Testable Services and are sent (callout 3) to `TCov`, which will then be responsible to process the information and make it available to the ITRS developer as a service (callout 4).

This coverage information can help the ITRS developer to: 1) become aware of when an adequacy criterion is reached, 2) maintain an existing test suite by identifying tests worth adding to cover untested behavior including GDS or to drop

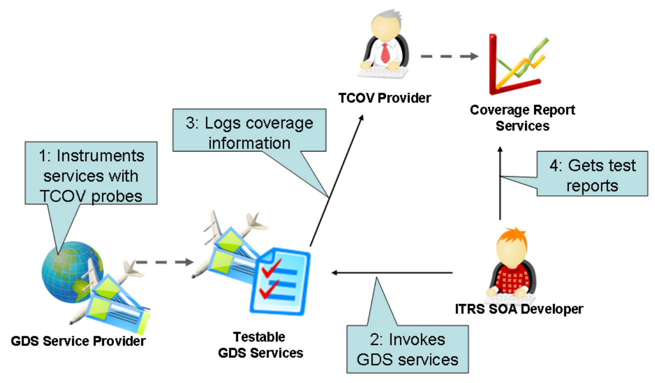


Figure 1: Scenario of SOCT approach.

tests that are exercising the same activity, 3) detect faults that arise from unexercised cases pinpointed by the coverage information, and 4) detect unexpected behavioral changes by collecting coverage information on successive versions of GDS (e.g., coverage differences between v_i and v_{i+1} reveal that the available flights information is partitioned in smaller packages and now requires multiple service requests).

It is obvious that any means to reveal more of the structure of a service will increase testability. What is novel about SOCT is how it achieves that increase while the loose coupling and late binding typical of the technology are kept untouched. Furthermore, by decoupling instrumentation (performed by the GDS service provider before deployment) and testing (carried on subsequently by ITRS-like developers) they can both evolve independently as long as the service interface remains the same. Of course, the coverage information collected on a previous implementation of a service will also become obsolete is the service evolves, but as said above the SOCT approach allows for (re)collecting coverage information in a new test session, without any need for changing the SOCT test infrastructure. Still, putting the approach to work does not come without challenges.

2.2 Challenges

The *first challenge* is how to define a general notion of coverage so that it can be valuable to the ITRS developer. SOCT will be able to work with the wide range of traditional coverage metrics (e.g., control-flow or data-flow, performed at intra-procedural and inter-procedural levels) as long as the Testable Services support its collection. Using those coverage measures to determine the extent of the testing activity, however, may prove fruitless. In our example, we note that the GDS provider under test may offer a variety of services from hotel booking, to flight reservation, to whole tours. If ITRS only invokes a part of the airplane reservation functionality, the coverage measures reported will be precise from the GDS perspective but of limited use for the ITRS developer to determine the degree of completeness of the testing effort (they will underestimate the extent of coverage respective to the functionality utilized by ITRS). Clearly, a definition of an upper bound for the coverage space is needed. In this work we develop the notion of *relative coverage*, that is, coverage that takes into consideration how the service is used by the client. In particular, we propose a relative coverage metric that identifies the potential coverage space by using a reachability analysis rooted in the used functionality (i.e., the relevant service operations).

The *second challenge* is to define an infrastructure that is rich enough to support the testing effort but simple enough to put into practice. For example, the ITRS developer will be interested in obtaining coverage information not just on one test with a service invocation but on a test suite involving multiple service invocations, which implies that the TCov interface must include capabilities to define a *Testing Session*. Such infrastructure should enable the ITRS developer to initiate a Testing Session through TCov, invoke the GDS services, and then close the Testing Session and retrieve the coverage measures for that session from TCov. In this work we define, implement, and assess the TCov service interface with such facilities. Another related infrastructure requirement is a coverage collection mechanism that can support the concurrent execution of *multiple independent clients*. Traditional coverage collection is meant to assess one test suite. However, popular web services are used concurrently by many clients, so there will be a need to compartmentalize the collection of coverage information on a clients basis, which has implications for the Testable Services and TCov.

The *third challenge* is to understand the tradeoffs associated with the introduction of SOCT at the organization and business levels. Behind the technical implementation, a governance framework for SOCT must be defined. For the service provider, implementing SOCT implies costs for instrumenting a service, running an instrumented service and communicating with TCov, which increases the risk of revealing intellectual property. For the ITRS developer it may imply paying for an additional service or a service overhead to obtain coverage information. But there are several degrees of freedom in these implications. For example, if TCov becomes a part of the service provider itself, or if having Testable Services becomes a competitive advantage, then the tradeoffs change. In this work we start exploring such variations in the context of SOCT, by identifying two alternative business flows.

3. DEFINING SOCT

In this section we first define the concepts supporting SOCT and then explain how we address the above outlined challenges.

3.1 SOCT Conceptual Definition

There are three main SOCT stakeholders: a service provider, called the `Testable Service`; a service integrator, called the `SOCT Tester`; and a coverage services provider, called `TCov`. Figure 2 provides a high level view of the interactions between these stakeholders. The SOCT Tester initiates the process by launching a Testing Session on the Testable Service. The Tester then receives a unique test session identifier SID and can start launching test cases on the Testable Service. The Testable Service can then collect the coverage information for SID and log it by invoking TCov. The Tester can retrieve the coverage information by invoking TCov.

Note that the SOCT approach is independent of the mechanisms used to generate the test cases and it is orthogonal to failure data collection. SOCT is only concerned with providing coverage information to the tester while keeping implementation neutrality.

In the rest of this section we define the concepts in Figure 2 in more detail. We start by describing the stakeholders more precisely.

Definition 1. *A* `Testable Service` *is a web service opportunely instrumented to collect coverage information and accessible through a WSDL interface that includes basic operations to enable testing data collection.*

Definition 2. *A* `SOCT Tester` *is a service developer or integrator providing composite services who launches test cases and specific operations on a Testable Service.*

Definition 3. `TCov` *is an organization that deploys and maintains a set of web services supporting coverage collection and reporting through a predefined WSDL interface. The* `Coverage Collecting Services` *(CCS) in TCov are invoked by the Testable Service to log coverage information. The* `Coverage Reporting Services` *(CRS) in TCov are invoked by the SOCT Tester to obtain coverage measures.*

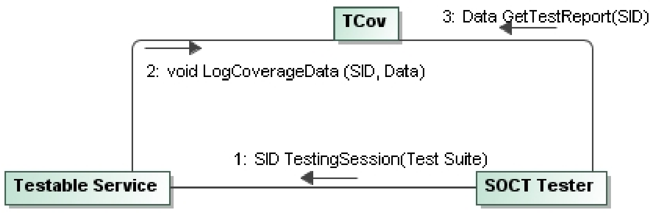

TCov
3: Data GetTestReport(SID)
2: void LogCoverageData (SID, Data)
1: SID TestingSession(Test Suite)
Testable Service
SOCT Tester

Figure 2: The SOCT Collaboration Diagram.

Two other basic but unique concepts that we need to define are *Testing Session* and instrumentation *Probes*.

Definition 4. *A* `(SOCT) Testing Session` *is a bounded set of interactions between the SOCT Tester and the Testable Service for which coverage information is collected. The bounds are defined by two operations: 1)* `start` *sets the environment parameters necessary for measuring coverage, and 2)* `stop` *restores the initial conditions. Each Testing Session is assigned a unique Testing Session identifier,* `SID`*.*

The `start` and `stop` operations are executed by the SOCT Tester. Test cases executed between these two operations belong to one Testing Session. The `SID` of a Testing Session is assigned by either TCov or the Testable Service depending on the implemented scenario of interaction (further discussed in Section 3.4).

As previously alluded to and as with any coverage testing approach, SOCT requires code instrumentation in the form of *probes* in order to monitor the execution of target program entities.

Definition 5. `Probes` *are additional instructions inserted at targeted locations in a Testable Service to enable coverage data collection according to a specific coverage criterion.*

Three aspects make SOCT probes different from traditional probes:

- the ones located at return points will be service invocations to Tcov to log the collected coverage data;
- they can measure coverage separately for concurrent Testing Sessions (initiated by one or multiple testers);
- they can be bounded by the Test Scope (see Section 3.2 below) to tailor coverage measures to a specific client.

3.2 SOCT Challenge 1: Reporting Meaningful Coverage

In conventional coverage measurement the coverage domain is determined by the analysis of the target program. For example, to compute the coverage domain to report statement coverage, the number of program statements is counted. A similar computation of the coverage domain to characterize how a SOCT Tester utilizes a Testable Service will often result in a coarse over-approximation as testers commonly exercise a limited and specific part of a Testable Service. To address this problem we refine the coverage domain in order to produce more meaningful coverage measures.

We start such refinement by explicitly specifying the subset of the service's operations that are relevant to the tester, which we define as the Test Scope. More specifically:

Definition 6. *The* `Test Scope` *of a Testing Session consists of a (sub)set of relevant WSDL operations of the Testable Service.*

We foresee different ways in which a Test Scope can be set, depending on by whom and when it is decided which operations are relevant for coverage testing purposes. A Test Scope could be explicitly defined by the SOCT Tester by communicating to the TCov Provider the list of operations to be monitored. Alternatively, the Test Scope can be dynamically defined at run-time by solely considering the operations actually invoked by the tester during a session.

We can now refine the coverage domain relative to the Test Scope.

Definition 7. *Given a Testable Service with a specified Test Scope, the* `Relative Coverage Domain` *is the set of service entities, associated with a specified coverage criterion, that can be reached through invocations rooted in any of the operations included in the Test Scope.*

For instance, when considering a block coverage criterion, and therefore the target entity as a block, the Relative Coverage Domain consists of the set of blocks in the Testable Service that implement the operations in the Test Scope. Reachability can be computed by analyzing the control and data flow structure of the web service. The choice of analysis will be determined by the selected coverage criteria, the analysis costs, and the precision requirements. Note that, given different test scopes, the relative coverage domain will vary.

With the refined coverage domain we can now produce a more precise measure to represent the extent to which the test suite covers the program entities of interest.

Definition 8. *Given n, the cardinality of a test suite Relative Coverage Domain, and k, the number of unique program entities executed by a test in the suite, the* `Relative Coverage Measure` *is k/n.*

3.3 SOCT Challenge 2: SOCT Infrastructure

In this section we provide a minimal WSDL specification of the interfaces between: Testable Service and SOCT Tester, CCS and Testable Service, and CRS and SOCT Tester.

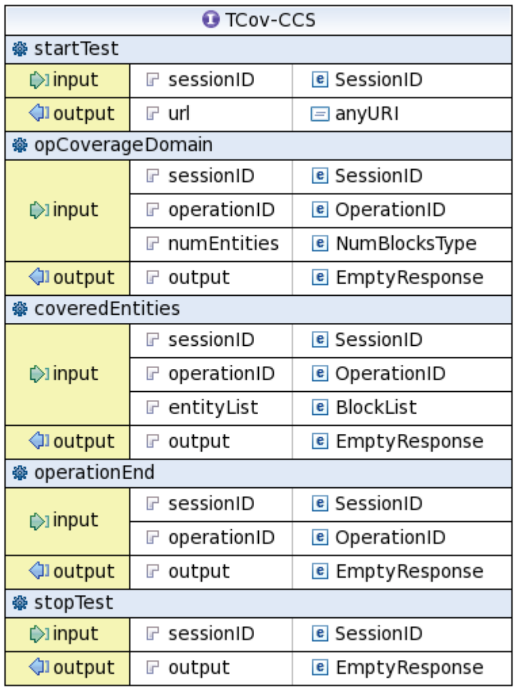
3.3.1 *Testable Service and SOCT Tester*

The SOCT Tester interacts with the Testable Service to start a Testing Session, executing the test cases (which includes invoking Testable Service operations) and terminating the Testing Session. The operations that should be provided by the Testable Service include:

- `SessionData startTest()`: this operation initiates a Testing Session between the SOCT Tester and the Testable Service. The SessionData returned to the SOCT tester contains the unique ID of the opened Testing Session (SID) and the URI of the CRS service;
- `void stopTest()`: this operation ends the current Testing Session, invalidating further use of SID.

3.3.2 *CCS and Testable Service*

The Testable Service interacts with the CCS to notify the opening and closing of a Testing Session, to log the cardinality of the Relative Coverage Domain, and to provide coverage data information. Figure 3 shows the operations that should be included by the CCS service and, for each operation, it shows the input and output messages in the first column, the composing part in the second column, and either the part type (displayed with two dashes in a rectangle) or its element (displayed with an *e* in a square) in the third column.



TCov-CCS		
startTest		
input	sessionID	SessionID
output	url	anyURI
opCoverageDomain		
input	sessionID	SessionID
	operationID	OperationID
	numEntities	NumBlocksType
output	output	EmptyResponse
coveredEntities		
input	sessionID	SessionID
	operationID	OperationID
	entityList	BlockList
output	output	EmptyResponse
operationEnd		
input	sessionID	SessionID
	operationID	OperationID
output	output	EmptyResponse
stopTest		
input	sessionID	SessionID
output	output	EmptyResponse

Figure 3: Basic CCS WSDL.

- `URI startTest(SID)`: this operation communicates the opening of a Testing Session between the SOCT Tester and the Testable Service. The URI returned to the Testable Service is the URI of the CRS service. This piece of information must be sent to the Testable Service so that it can pass it on to the SOCT Tester who will use it to retrieve the coverage information from TCov.
- `void opCoverageDomain(SID, OID, numEntities)`: this operation communicates to the CCS the cardinality of the Relative Coverage Domain for a specific operation (OID is the unique operation identifier) and the number of entities (numEntities) reachable from OID;
- `void coveredEntities(SID, OID, entityList)`: this operation communicates to the CCS the list of entities covered during the execution of the operation OID. In particular, entityList contains the entity identifiers executed during the operation. Note that the entity identifiers are assigned by the Testable Service and are only required to be consistent within a testing session.
- `void operationEnd(SID, OID)`: this operation is the counterpart of `opCoverageDomain`, and is used to communicate to TCov that the current operation has reached its conclusion.
- `void stopTest()`: this operation communicates the end of a Testing Session.

3.3.3 *CRS and SOCT Tester*

The SOCT Tester interacts with the CRS to obtain the coverage information collected. The operation that should be included by the CRS are:

- `void setTestScope(SID, operationList)`: it enables the tester to bound the scope of the Testing Session. It requires the SID and the list of operations (operationList);
- `PercentageData coverageMeasure(SID)`: this operation returns PercentageData which includes the achieved coverage and the list of operations executed during the Testing Session.

3.4 SOCT Challenge 3: Understanding the Tradeoffs

Having settled the main concepts of the SOCT approach, we now describe the potential flow of interactions in two likely scenarios that illustrate the range of applicability of the approach, depending on who is the stakeholder that assigns to a TCov provider the supervising responsibility.

In the first scenario, which we illustrate in Figure 4, a contract is established between the Testable Service (TS) provider and a TCov service provider. The justification may be that the TS provider has to show that they have taken diligent care for making the offered services monitorable. The contract between TS and TCov is long-term, and implies that every system integrator using TS services will interact with the same TCov service, without having to choose between several possible providers. The coverage measures of TS will always be performed by the one and only contracted TCov, and the SOCT Tester will only need to use one TS interface to start the coverage collection. In such a scenario the binding with TCov can be static, and the general procedure works as follows:

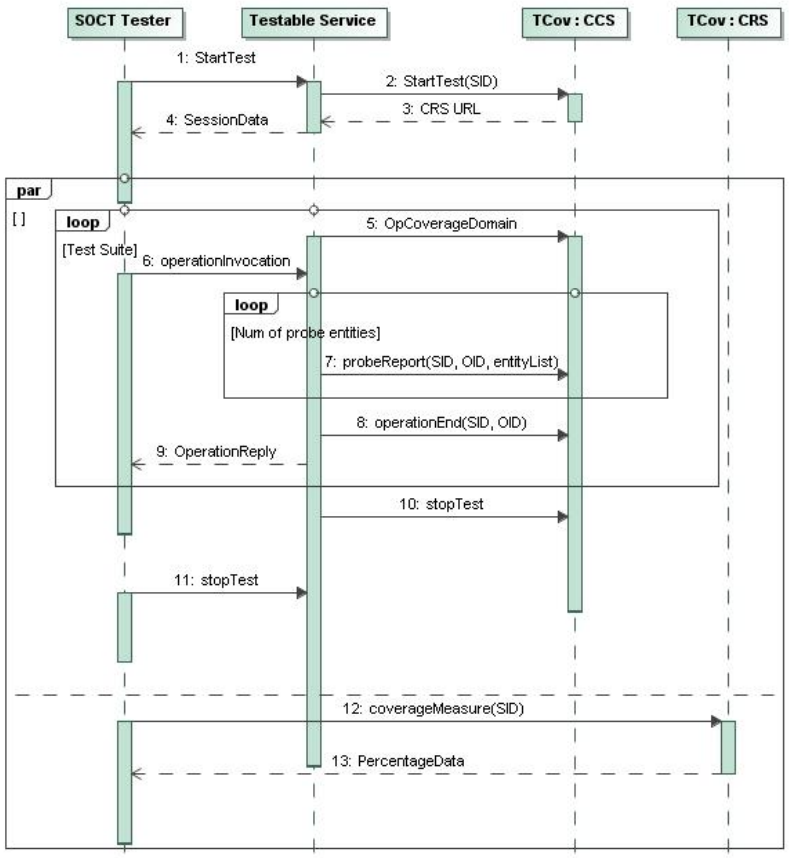


Figure 4: A scenario of interaction among SOCT stakeholders.

1. the SOCT Tester invokes a `startTest` operation on TS;

2. TS generates a *Testing Session identifier* (SID), and sends a reply to the SOCT Tester which contains this information, both as a SOAP variable *and* as a cookie. The former will be used later by the SOCT Tester to query the CRS, while the latter is the cookie which will be used onward for every communication between the SOCT Tester and the Testable Service, until the end of the Testing Session;

3. the SOCT Tester will make invocations to the operations defined in the TS WSDL. Every SOAP invocation (which is contained in the payload of an HTTP request) will contain the cookie in the HTTP header, so the probes can retrieve this information and send it to TCov as part of the logging data;

4. the SOCT Tester can then invoke TCov, sending the SID along, to retrieve the coverage measures related to its Testing Session;

5. when finished, the SOCT Tester will invoke a `stopTest` operation on TS, and TS will reply by invalidating the cookie (which will expire anyway after a given time).

This is relatively simply communication protocol, in which the SID must be used in the communication between the SOCT Tester and TCov only to retrieve the coverage data. Apart from this, the SID is only used in the background (in the HTTP header) as the communication between the SOCT Tester and TS occurs. Also, this solution does not prevent the existence of a contract between TCov and the SOCT Tester, which might be required at the time coverage measures are requested.

The second scenario is slightly more complex because there is no contract, or previous agreement, between TS and TCov. In this scenario there are a number of TCov providers, and the choice of using one or another is up to the SOCT Tester. Therefore, the handshake must not start between TS and the SOCT Tester but between the latter and TCov. This second scenario stresses the point that the interest in obtaining coverage measures rests on the SOCT Tester.

1. the SOCT Tester invokes an `init` operation on TCov, specifying which service will be tested;

2. TCov generates a SID and sends it (as a SOAP response) to the SOCT Tester;

3. the SOCT Tester invokes a `startTest` operation on TS, passing two arguments: the SID and the URL of the selected TCov;

4. TS binds its probes to the URL given by the SOCT Tester; this binding is valid only for this SOCT Tester, since another Testing Session might be bound to another TCov provider. Additionally, it sends a reply to

the SOCT Tester, packing the SID in a cookie associated to its own domain. This allows the SOCT Tester to send the cookie on every subsequent SOAP request to TS;

5. the TCov Tester invokes the normal TS operations, and the coverage data is logged onto the selected TCov service;

6. when finished, the `stopTest` operation is invoked on TS, and TS invalidates the cookie;

7. the SOCT Tester requests the coverage measures using the SID generated by TCov.

These two possible flows expose different business models behind the SOCT approach. Implementing the approach requires in fact not only to address technical issues (as those discussed in Sections 3.2 and 3.3), but as importantly also to agree on organizational and business-oriented tradeoffs, in particular concerning who is willing to push (and pay) for whitening SOA testing. As illustrated by the two alternative flows of interaction, the latter issue can impact the technical implementation as it affects the SOCT process flow. More important, these are just two scenarios of many potential ones involving SOCT.

4. EXPLORATORY STUDY

In this section we share our experience in instantiating and utilizing SOCT to support white-box testing of an independent web service provided by our collaborators. Our instantiation is compliant with the WSDL described in Section 3.3, it follows the protocol described in the first scenario of Section 3.4 (shown in Figure 4), and the testing scope is defined by the Tester through the `setTestScope` operation defined in Section 3.3.1.

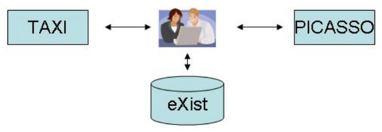
4.1 Study setup

Our collaboration with the Italian company Codices aims to facilitate the testing of their platform, PICASSO[3], which aims to support interoperability among health-related applications by translating to and from HL7-V3 standard [16]. Details of the collaboration work between CNR and Codices can be found in [22].

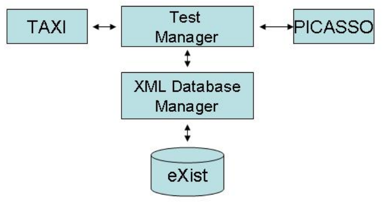
The current testing status, presented in Figure 5(a), for testing PICASSO requires some manual work for coordinating the activities of an input generator tool, called TAXI, and the execution of the interoperability platform. In particular, the TAXI [6] tool is used to automatically derive a set of XML instances conforming to a given XML Schema. Switching to a more automated system such as the one depicted in Figure 5(b) could improve the testing efficiency and reduce costs. In this case the manual activity is replaced with two web services: *Test Manager*, which coordinates the test case generation and execution between PICASSO and TAXI; and *XML Database Manager*, which handles the XML database where testing inputs are stored and queried by the Test Manager.

So, we set out to build a Test Manager web service which employs a third party web service for data management,

[3]PICASSO is an acronym from the Italian "Piattaforma per l'Interoperabilità e la Cooperazione Applicativa nelle Strutture Sanitarie ed Ospedaliere".



(a) Current testing environment.



(b) Testing environment under development.

Figure 5: The testing environment for PICASSO.

and we use our SOCT approach in the process. For the external service we selected *XMLIndexer*, a publicly available service from the D4Science project [1]. XMLIndexer is a complex entity, made up of three different WSDL interfaces to be used in concert: *XMLIndexerFactory*, *GCUBEDaix*, and *WSDaix*.

A single operation in *XMLIndexerFactory* is the starting point for using the service:

a. `EndpointReference createResource(string collName)` for fetching a data collection. If the collection is already present in the eXist database, the operation will simply return a reference to the collection; otherwise, the collection will be created and the reference returned.

GCUBEDaix provides a set of operations shown in Figure 6. However, due to space limitations, we are not going to describe in detail all the operations, but only those which have actually been used in the next section:

b1. `void addElement(string docName, string content)` for adding an element into a data collection;

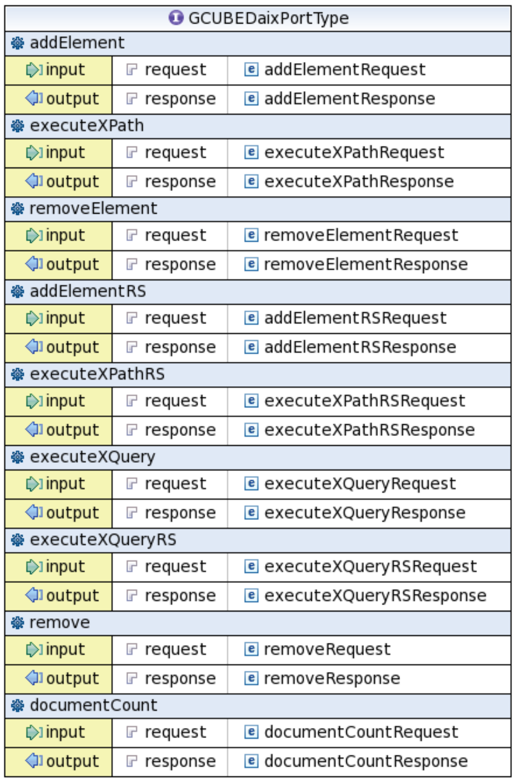
b2. `string[] executeXPath(string xpath)` for selecting elements, or parts thereof, in a collection;

b3. `void remove(void)` completely deletes a collection and its content from the eXist database;

b4. `int documentCount(void)` for counting the elements of a collection.

WSDaix is similar to GCUBEDaix. However, its definition is presently a work in progress, and in the current version a single operation is implemented: `int documentCount(void)`, which counts the elements in a collection.

For the purpose of this study then, we are the SOCT Tester developing the Test Manager, whereas the service provided by D4Science takes the role of the Testable Service. With the consensus and the assistance of the D4Science developers, the operations for communicating with the TCov service, as specified in Section 3.3, have been integrated into the XMLIndexerFactory WSDL. For the study, the implementation of the web service has been properly instrumented to collect *block coverage* information.



GCUBEDaixPortType		
addElement		
input	request	addElementRequest
output	response	addElementResponse
executeXPath		
input	request	executeXPathRequest
output	response	executeXPathResponse
removeElement		
input	request	removeElementRequest
output	response	removeElementResponse
addElementRS		
input	request	addElementRSRequest
output	response	addElementRSResponse
executeXPathRS		
input	request	executeXPathRSRequest
output	response	executeXPathRSResponse
executeXQuery		
input	request	executeXQueryRequest
output	response	executeXQueryResponse
executeXQueryRS		
input	request	executeXQueryRSRequest
output	response	executeXQueryRSResponse
remove		
input	request	removeRequest
output	response	removeResponse
documentCount		
input	request	documentCountRequest
output	response	documentCountResponse

Figure 6: WSDL of the GCUBEDaix service.

4.2 Preliminary results

In the rest of this section we focus exclusively on the point of view of the SOCT Tester. We established a series of activities operating on the XML Database Manager:

- create several XML instance collections corresponding to the three different data formats exchanged between healthcare organizations;
- search within a collection for instances having specific element values. For instance, the selection of all the instances having the value "Pisa" as province within the HL7-V3 collection;
- update a collection adding further XML instances;
- delete a collection and its contents;
- count the elements in a collection.

These actions have been then translated into a test suite, whose scope (with respect to the operations of XMLIndexer) comprises the operations labeled as *a* and the full set of *b* presented in Section 4.1. The test suite consists of the test cases described in Table 1.

We launched a Testing Session among Test Manager and XMLIndexer, and used TCov to collect and report the coverage obtained after the execution of each test case. Table 2 shows the results obtained. The first column lists the tests described above. The second column shows the accumulated number of blocks covered, and the third one reports the corresponding relative coverage. The fourth column displays which XMLIndexer operations are used by each test case. These three pieces of information are provided by TCov. The last column shows the coverage obtained without taking Test Scope analysis into account. The reported absolute coverage is based on the total number of blocks (185) within the whole set of operations in the XMLIndexer service.

Table 1: Test cases in the test suite

T1	create a collection with one element
T2	delete a collection
T3	search for non-existing data within a collection
T4	search in a non-existing collection
T5	create different collections
T6	add an already-existing document to a collection
T7	delete a non-existing collection

Table 2: Incremental coverage results

Test	Blocks	Coverage	Covered ops	Abs. coverage
T1	68	50.37%	a, b1, b4	36.76%
T2	72	53.33%	a, b3	38.92%
T3	78	57.78%	a, b2	42.16%
T4	78	57.78%	a, b2	42.16%
T5	79	58.52%	a, b1, b4	42.70%
T6	79	58.52%	a, b1, b4	42.70%
T7	79	58.52%	a, b3	42.70%
T8	79	58.52%	a, b2	42.70%
T9	79	58.52%	a, b2	42.70%
T10	80	59.26%	a, b2	43.24%
T11	80	59.26%	a, b1	43.24%
T12	80	59.26%	a, b2	43.24%
T13	80	59.26%	a, b3	43.24%

Even in the limited setting where the study is conducted we can make some interesting observations. First and foremost, TCov enabled the collection of coverage information without breaking the underlying principles of SOA. A SOCT Tester looking at these results would be in a position to assess, for example, whether additional testing effort is paying off. In our setting we observe that if testing resources are tight, T4 could be considered a small contributor over T3 as it does not increase coverage and exercises the same operations; the same can be said for other tests that might be assigned a lower priority in the future. When the coverage does not increase, a tester also gets a hint that it is time to vary the input selection.

In our case, we realized after T7 that we had omitted to test the XMLIndexer invocations for robustness against wrong input parameters. Developing these simple additional tests increased the coverage to 59.26% (43.24% absolute). A SOCT Tester would also be able to determine whether sufficient coverage was achieved and the relative coverage measures are clearly useful in that regard. Even though XMLIndexer is a rather small service with many functions shared among the operations, scoping is able to provide a more precise coverage estimate that turns out to be 16% higher at the end of the session.

For completeness, after carrying out the study, we performed an inspection on the XMLIndexer service code with our collaborators to understand the reasons that limited coverage to the current values. The inspection reveals that many blocks are part of *catch* blocks that the Test Manager is unlikely to throw except under very specific cases that will be hard to discover without access to the service source code. This does not take value away from the collected coverage information but rather puts in evidence the need for SOCT and TS to provide further visibility and control in some cases. For example, under our circumstances, it would be helpful if XMLIndexer would allow us to adjust the coverage criteria to ignore exceptional blocks or would provide suggestions for inputs that would reach the missing entities.

5. RELATED WORK

In this paper we have introduced the SOCT approach for enabling white-box SOA testing. The topic of web service testing is actively researched, and several approaches have been proposed, as recently surveyed in [9]. We focus in particular on SOA testing at the integration level, i.e., we address the need of testing a composition of services that might have been developed by independent organizations. The issues encountered in testing a composition of services are investigated in [8], distinguishing between testing of orchestrations and of choreographies.

Today, the *de facto* standard for service orchestration is the Business Process Execution Language (BPEL) [21]. Several authors have leveraged the BPEL code for SOA testing. Although different approaches have been devised, the essential common basis in BPEL-based testing is that variants of a control flow diagram are abstracted and paths over this diagram are used to guide test generation or to assess BPEL coverage (see, e.g., [28, 29]). Others (including authors of this paper) have also proposed to exploit BPEL data-flow information [19, 4].

Anyhow, so far, all existing approaches to SOA testing test the services invoked in a composition as black-boxes. Indeed, the shared view -before SOCT- is that for SOA integrators "a service is just an interface, and this hinders the use of traditional white-box coverage approaches" [9]. To the best of our knowledge, SOCT is the very first attempt to circumvent such a vision by revising and adapting a notion of code coverage testing within the service-oriented paradigm.

The need to enhance black-box testing with coverage information of tested services is also recognized in [17], where "grey-box testing" is introduced. The BPELTester tool is presented, that extends the above cited BPEL-based test approach [29]. Test case design is driven by the BPEL paths; the approach is grey-box in that, after test execution, the produced test traces are collected and analysed against the BPEL paths. Discrepancies are exploited essentially for regression testing purposes, i.e., to prevent misalignment due to unforeseen service changes and detect need for re-test. The work thus addresses some of the concerns tackled by SOCT, however the two approaches make different assumptions. Certainly the assumption of BPELTester that the integrator can access and analyse service execution traces breaks the loose coupling between service provider and service user. This is a constituent principle of SOA, which we seek to maintain through the TCov actor.

The idea of leveraging service execution traces is pursued also in [5]. Similarly to SOCT, this work extends SOA with observation capabilities by introducing an "Observer" stakeholder into the ESOA (Extended SOA) framework. ESOA however does so for a different goal than SOCT: while we introduce TCov to monitor code coverage, in ESOA services are monitored (passive testing) against a state-model. In SOCT we do not assume the availability of additional information for testing purposes.

Finally, SOCT is aimed at enabling the derivation of coverage measures in testing of service compositions, but it is not concerned with how the test cases should be derived. Therefore, the SOCT approach should be integrated with methodologies for test case generation. In literature, different proposals provide strategies for test case generation, such as [11] which relies on Genetic Algorithms, the above cited BPEL-based approaches [28, 14, 10, 19, 4, 18], others that apply Model-based Testing [24, 25]. All these and many others not cited here for space limitations could be seen as complementary to SOCT, and could indeed be made more effective if enhanced with coverage information.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have introduced SOCT, an approach that whitens SOA testing. The approach overcomes what is apparently a contradiction in terms: it empowers a service integrator to obtain coverage information on an invoked external web service without breaking the implementation neutrality of the latter, which is one of SOA founding principles. Such an attainment is possible if the service provider is willing to instrument their services so that users accessing them for testing purposes can monitor the execution of the service program entities. Based on such premises, the approach naturally fits the service-oriented paradigm as test coverage information is collected and retrieved through dedicated service interfaces. We have hereby defined the approach, implemented an instance of it, and performed a preliminary study to show its feasibility and potential value.

We are aware of the commitments required to put SOCT into practice. It requires, for example, for service developers to instrument their services, reveal program information, and incorporate additional services for external testing, all of which imply additional resources. On the other hand, we believe that our approach radically changes the way SOA testing has been conceived so far, opening the way to a whole new range of opportunities and instruments for making services more transparent and, in the end, more trustworthy.

In the future, we plan to extend this work in several ways. First, we plan to perform a larger scale study in collaboration with the D4Science community and other industrial collaborators to obtain a better understanding of the cost, performance and overall impact on the quality of the service, and potential benefits of SOCT. The setting of such study may include more and larger orchestrated services, tests prepared by engineers using their process and tools, and multiple testers working concurrently to validate a service. We note that the results obtained in this paper set the stage for such costly study. Second, we want to refine several aspects of the SOCT instantiation such as those associated with the estimation of the coverage domain, the realization of the relative coverage notion through various reachability analyses exposing different tradeoffs, and the efficient collection of coverage information where we can leverage a large body of existing work on minimizing the overhead caused by instrumentation probes.

Acknowledgements

This work was partially supported by the TAS^3 Project (EU FP7 IP No. 216287), and by the Italian MIUR Project D-

ASAP (Prin 2007). The authors wish to thank the colleagues of Codices and of the D4Science project for their collaboration in the case study development.

7. REFERENCES

[1] DIstributed colLaboratories Infrastructure on Grid ENabled Technology 4 Science. `http://www.d4science.eu/`.

[2] Amazon Discussion Forum. Thread: Massive (500) Internal Server Error.outage. `http://developer.amazonwebservices.com/connect/thread.jspa?threadID=19714`.

[3] C. Bartolini, A. Bertolino, and E. Marchetti. Introducing service-oriented coverage testing. In *Workshop on Automated engineeRing of Autonomous and run-tiMe evolvIng Systems*, pages 57–64, 2008.

[4] C. Bartolini, A. Bertolino, E. Marchetti, and I. Parissis. *Data Flow-Based Validation of Web Services Compositions: Perspectives and Examples*, pages 298–325. Architecting Dependable Systems V. Springer-Verlag, 2008.

[5] A. Benharref, R. Dssouli, M. A. Serhani, and R. Glitho. Efficient traces' collection mechanisms for passive testing of web services. *Information Software Technology Journal*, 51(2):362–374, 2009.

[6] A. Bertolino, J. Gao, E. Marchetti, and A. Polini. Systematic generation of XML instances to test complex software applications. In *Rapid Integration in Software Engineering*. LNCS 4401, September 2006. Geneve, Switzerland.

[7] A. Bertolino and A. Polini. SOA test governance: enabling service integration testing across organization and technology borders. In *Workshop on Web Testing*, pages 277–286, 2009.

[8] A. Bucchiarone, H. Melgratti, and F. Severoni. Testing service composition. In *Argentine Symposium on Software Engineering*, 2007.

[9] G. Canfora and M. Di Penta. *Service Oriented Architecture Testing : A Survey*, pages 78–105. Number 5413 in LNCS. Springer, 2009.

[10] H. Cao, S. Ying, and D. Du. Towards model-based verification of BPEL with model checking. In *International Conference on Computer and Information Technology*, pages 190–194, 2006.

[11] M. Di Penta, G. Canfora, G. Esposito, V. Mazza, and M. Bruno. Search-based testing of service level agreements. In *Conference on Genetic and Evolutionary Computation*, pages 1090–1097, 2007.

[12] M. Fisher II, S. Elbaum, and G. Rothermel. Automated refinement and augmentation of web service description files. Technical Report 0026, University of Nebraska, Lincoln, Computer Science and Engineering Department, December 2007.

[13] M. Fisher II, S. Elbaum, and G. Rothermel. Dynamic characterization of web application interfaces. In M. B. Dwyer and A. Lopes, editors, *Fundamental Approaches to Software Engineering*, volume 4422 of *Lecture Notes in Computer Science*, pages 260–275. Springer, 2007.

[14] J. García-Fanjul, J. Tuya, and C. de la Riva. Generating test cases specifications for BPEL compositions of web services using SPIN. In *International Workshop on Web Services Modeling and Testing*, 2006.

[15] Gartner and Forrester: Use of Web services skyrocketing, 2003. `www.utilitycomputing.com/news/404.asp`.

[16] Health Level Seven. `http://www.hl7.org/`, accessed Oct. 9, 2008.

[17] Z. J. Li, H. F. Tan, H. H. Liu, J. Zhu, and N. M. Mitsumori. Business-process-driven gray-box soa testing. *IBM Syst. J.*, 47(3):457–472, 2008.

[18] H. Lu, W. Chan, and T. Tse. Testing context-aware middleware-centric programs: a data flow approach and an RFID-based experimentation. *Symposium on Foundations of Software Engineering*, pages 242–252, 2006.

[19] L. Mei, W. Chan, and T. Tse. Data Flow Testing of Service-Oriented Workflow Applications. In *TestCom 2008*, volume 5047 of *LNCS*, pages 371–380. Springer, 2008.

[20] OASIS Reference Model for Service Oriented Architecture 1.0, Official OASIS Standard, Oct. 12, 2006. `http://www.oasis-open.org`.

[21] OASIS WSBPEL Technical Committee. Web services business process execution language version 2.0. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf`, 2007.

[22] M. Pascale, M. Roselli, U. Rugani, C. Bartolini, A. Bertolino, F. Lonetti, E. Marchetti, and A. Polini. Automated testing of healthcare document transformations in the PICASSO interoperability platform. In *Software Engineering in Practice*, 2009.

[23] M. Pezzè and M. Young. *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, 2007.

[24] PLASTIC Validation Framework Tools homepage. `http://plastic.isti.cnr.it/wiki/doku.php/tools`.

[25] A. Sinha and A. Paradkar. Model-based functional conformance testing of web services operating on persistent data. In *Workshop on Testing, analysis, and verification of web services and applications*, pages 17–22, 2006.

[26] Torry Harris Business Solutions. White Paper. SOA test methodology.

[27] W. Xu, J. Offutt, and J. Luo. Testing web services by xml perturbation. In *International Symposium on Software Reliability Engineering*, pages 257–266, 2005.

[28] J. Yan, Z. Li, Y. Yuan, W. Sun, and J. Zhang. BPEL4WS unit testing: Test case generation using a concurrent path analysis approach. In *International Symposium on Software Reliability Engineering*, pages 75–84, 2006.

[29] Y. Yuan, Z. Li, and W. Sun. A graph-search based approach to BPEL4WS test generation. In *International Conference on Software Engineering Advances*, 2006.