Bringing White-Box Testing to Service Oriented Architectures through a Service Oriented Approach

Cesare Bartolini\textsuperscript{a}, Antonia Bertolino\textsuperscript{a}, Sebastian Elbaum\textsuperscript{b}, Eda Marchetti\textsuperscript{a}

\textsuperscript{a}ISTI - CNR
Via Moruzzi 1
56124 Pisa, Italy
\textsuperscript{b}Computer Science and Engineering Department
University of Nebraska
Lincoln, USA

Abstract

The attractive feature of Service Oriented Architecture (SOA) is that pieces of software conceived and developed by independent organizations can be dynamically composed to provide richer functionality. The same reasons that enable flexible compositions, however, also prevent the application of some traditional testing approaches, making SOA validation challenging and costly. Web services usually expose 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 the independent web services. To address this lack we propose an approach that makes web services more transparent to testers through the addition of an intermediary service that provides coverage information. The approach, named Service Oriented Coverage Testing (SOCT), provides testers with feedback about how much a service is exercised by their tests without revealing the service internals. In SOCT, testing feedback is offered itself as a service, thus preserving SOA founding principles of loose coupling and implementation neutrality. In this paper we motivate and define the SOCT approach, and implement an instance of it. We also perform a study to assess SOCT feasibility and provide a preliminary evaluation of its viability and value.

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

1. Introduction

Usage of web services has increased dramatically in the last years [15]. Different definitions can be found in the literature of what a web service is; the World Wide Web Consortium (W3C) qualifies a web service as [38] a software system designed to support interoperable machine-to-machine interaction over a network and having an interface in a machine-processable format. Such interface descriptions can be published and discovered, thus making it cost-effective
for companies to integrate their own services with those developed and managed by third parties [30]. Of course web services are not necessarily used in cross-organizational way, on the contrary they are also widely used “in-house” and within corporate environments. However, the former is the situation we consider in this paper, because, as we explain below, this exposes the most difficult challenges from the tester’s viewpoint.

An emerging paradigm for organizing and utilizing distributed capabilities that may be under the control of different organizations is the Service Oriented Architecture (SOA) [23], whereas the sequence and conditions in which one web service invokes other web services in order to achieve its goals is referred to as an orchestration [38].

Failures in web service orchestrations, unfortunately, are common and their impact becomes more obvious and detrimental as their popularity and inter-dependencies increase. For example, a recent failure in Amazon’s storage web service affected many companies relying on it [1].

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 [35] are followed by the developer to test an individual service to ensure its quality, nothing guarantees that it will then operate smoothly as part of a dynamic distributed system made of multiple orchestrated but autonomous services. Second, the orchestrator of independently developed services can usually only access their 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) [27, 31, 40], on the detection of inconsistencies between a service interface description and its behavior [13], on defining adequacy criteria based on the web services interactions [17], on procedures to build scaffolding to test services in more controlled settings [32], and on using the availability of multiple services as oracles [36].

One trait shared by existing test approaches is the treatment of web services as black boxes [9], focusing on the external behavior but ignoring the internal structure of the services included in the orchestration. 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 [26], as coverage information can provide an indication of the thoroughness of the executed test cases, and can help maintain an effective and efficient test suite.

To address this limitation we have conceived an approach through which services can be made more transparent to an external tester while maintaining the flexibility, dynamism and loose coupling of SOAs. Our approach enables a service orchestrator to access test coverage measures (or their byproducts) on the third-party services without gaining access to the code of those services. We
refer to this enhancement as “whitening” of SOA testing [3] to reflect a move towards whitebox testing in terms of providing increased feedback about test executions on a service, yet the service provider remains in control of how much of the internal system structure is revealed.

Whitening is achieved through the use of dedicated services built for collecting coverage data; these compute the coverage of the services under test, on behalf of the orchestrator. The loose coupling of the web service paradigm is not lost between the orchestrator and the developer of the provided service, because the orchestrator is still unable to see anything of the service beyond the interface. In particular, the orchestrator is completely unaware of any implementation detail, and simply obtains some cumulative measures (percentages) which only reveal how much of the service the executed tests are actually using. Loss of loose coupling happens, at most, between the provided service and the coverage collecting service (of which it is reasonable to assume it is a trustworthy third party), but, even so, which and how much information is disclosed is under the control of the provider of the service under test. The approach thus blends naturally into the SOA paradigm and is called Service Oriented Coverage Testing (SOCT).

The added transparency from test whitening is clearly far from a “complete” white-box testing; coverage only adds a slight bit of information. However, this improvement in transparency increases testability, letting the service orchestrator gain additional feedback about how a service orchestration is exercised during validation. This feedback can then be used by orchestrators in many ways: to determine whether a coverage adequacy criterion that includes the third-party service structure has been reached; to identify tests that do not contribute as much and can be removed from a suite; or, to drive regression testing to detect possible updates in the implementation of a third-party service that might affect the behavior of their application. On the other end, third-party service providers may be enticed to provide such an extended testing interface as a way to implement continuous quality assurance checks (perhaps in association with the orchestrator), or may be required to do so as part of a service quality agreement.

Whitening SOA testing requires the design of an infrastructure that fits naturally in the service-oriented model by providing test coverage information itself as a service accessible only through a service interface. The infrastructure supporting our approach achieves that goal by requiring:

1. for the developer of a provided service, to instrument the code to enable the monitoring of the execution of target program entities, and make the relative usage information publicly available;
2. for the provider of the coverage collecting service, to track test execution results; and
3. for the service orchestrator testing the integrated application, to request testing information through a standardized published web service testing interface.

From a broader perspective, such infrastructure relies on laying down a gov-
ermance framework to realize inter-organization testing at the orchestration level [7]. Such framework encompasses the set of rules, policies, practices and responsibilities by which a complex SOA system is controlled and administered. In this paper we focus on the governance issues more closely associated with the integration testing of the orchestrated application. Of course, governance per se does not prevent malicious or irresponsible behavior on the service provider’s part. The SOCT approach works as far as all involved stakeholders cooperate diligently (which is not different from any other collaborative engineering endeavor). The service provider, in particular, should ensure that the coverage information sent to the collecting service are precise, complete, and timely.

The very idea of SOCT has been proposed for the first time in [4], and elaborated in a conceptual approach in [3]. This paper extends on the latter work by revising the approach’s associated definitions together with its potential applications, providing more detailed explanations of the interactions among the stakeholders, describing a full implemented instance of it, and performing a completely new assessment of its usefulness and performance through a case study. More precisely, in the next section we overview foundational related work and then, in Section 3, we present the problem domain, its motivation and main challenges. In Section 4 we define SOCT concepts and a realization scenario. In particular, the main components of the developed instance are described in Section 4.3. The case study is described in Section 5. Conclusions are drawn in Section 6.

2. Related work

In this section we overview the topic of web service testing, which is currently actively researched, as recently surveyed by Canfora et al. [9]. As said in the Introduction, we focus here on SOA testing at the integration level; in particular 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 by Bucchiarone et al. [8], distinguishing between testing of orchestrations and of choreographies.

Today, the standard for service orchestration is the Business Process Execution Language (BPEL) [24]. Several authors have leveraged the BPEL representation 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., [41, 43]). Others (including some of the authors of this paper) have also proposed to exploit BPEL data-flow information [21, 5].

So far, all existing approaches to SOA testing validate the services invoked in a composition as black-boxes. Indeed, the shared view 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, by revising and adapting a notion of code coverage testing within the service-oriented
paradigm, the proposal of this paper is the very first attempt to circumvent such a vision.

The need to enhance black-box testing with coverage information of tested services is also recognized by Li et al. [18], where “grey-box testing” is introduced. The BPELTester tool presented there extends the above-cited BPEL-based test approach [43]. 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 analyzed against the BPEL paths. The work thus addresses some of the concerns that will be tackled in this paper, even if starting by different assumptions. Indeed, the assumption of BPELTester that the orchestrator can access and analyze service execution traces breaks the loose coupling between service provider and service user. This is a constituent principle of SOA, which we maintain in the approach proposed in this paper.

To the best of our knowledge, few proposals are available for regression testing of web services, and most of them exploit the availability of additional documentation or constraints, such as a Control Flow Graph as in [29], or a UML model as in [28]; more sophisticated approaches include a two-level Timed Labeled Transition Systems model with timing constraints (in [33]), or an Event Dependency Graph combining dependencies with interacting components (in [34]). Hou et al. [16] proposed specific quotas to constrain the number of requests, which could help service orchestrators understand the service behavior and test case coverage information; finally, Mei et al. [22] developed a multi-level coverage model to capture business processes based on XPath and WSDL. Considering regression testing, the work presented here is complementary to the above-mentioned proposals and is meant to prevent misalignments due to unforeseen service changes, and thus to detect the need to re-test (as we describe in Section 5.2.2).

The idea of leveraging service execution traces is pursued by Benharref et al. [6]. Similarly to our proposal, 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 the one proposed in this paper: while our focus is to monitor code coverage, in ESOA services are monitored (passive testing) against a state model. In our proposal we do not assume the availability of additional state-based information for testing purposes.

Finally, our work is aimed at enabling the derivation of coverage measures in testing of service compositions. How the test cases should be selected or derived is not in the scope of this paper, although of course methodologies for test case generation should also be considered. Different strategies have been proposed for test case generation, such as those that rely on Genetic Algorithms [11], the above-cited BPEL-based approaches [41, 14, 10, 21, 5, 19], and others that apply Model-based Testing [27, 31]. All these and others not mentioned here due to space limitations could be seen as complementary to the proposal of this paper, and could indeed be made more effective if enhanced with coverage information.
3. Motivation

In this section we motivate and discuss the key ideas behind SOCT by illustrating some testing scenarios in which SOA test whitening would be valuable. The case study in Section 5 provides an assessment of some of these scenarios.

We consider the case of a SOA orchestrator building an Integrated Travel Reservation System (ITRS) for use by Travel Agency customers. ITRS is meant to provide its clients with a single-point access to several on-line services including flight booking and hotel reservation. In developing ITRS, the SOA orchestrator will both build some new services, for example to tailor the user interface or to reformat the collected travel information according to the customer’s preferences, and use existing ones, for example obtaining flight and hotel information via some trusted Global Distribution System (GDS) service such as Sabre\(^1\) or Travelport\(^2\).

Our goal is to support the orchestrator in testing ITRS, in particular on the functional validation of the composite SOA system. How can the SOA orchestrator test the cooperation between the newly-built front-end services and the third-party ones? One option could be to mock the GDS services into a testbed, and perform the testing \textit{off-line}. Setting up this solution implies building and maintaining a complex test environment simulating the external services. Such a solution has a high development cost and might not be reliable because there may not be enough information to reproduce the behavior of the external services with high fidelity, and also because these could change without notice, making the developer’s testing stubs obsolete.

The orchestrator could also test the integrated ITRS \textit{on-line}, i.e., by directly accessing the GDS services. This is the case we consider here, and is the approach being taken in several current proposals [6, 16, 37], as the solution to tackle dynamism and late binding of SOA systems. Because of the SOA loose coupling principle, though, an intrinsic limitation of this alternative from the tester’s viewpoint is that the GDS services are accessed as black boxes. Hence, when testing is completed, the ITRS orchestrator has no data about the extent to which the GDS integration has been tested and cannot know how comprehensive the test cases have been. This lack of awareness can lead on one side to some service elements left untested, and therefore to potentially 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 makes it fail ungraciously), and on the other side to redundant testing efforts (e.g., addition of tests that traverse GDS paths already exercised and do not add new value).

Our proposed approach, SOCT, can mitigate the limitations associated with lack of information from the on-line testing activity of web services. SOCT enables the ITRS orchestrator to obtain traditional coverage measures for the

\(^1\)http://www.sabre-holdings.com/
\(^2\)http://www.travelport.com/
tested external services without breaking the implementation neutrality of SOA, i.e., the orchestrator performing testing receives the coverage information in the form of percentage of covered entities, but gains no visibility on the GDS internals. With such coverage data the tester could:

- assess test thoroughness. Coverage measures provide an assessment of the adequacy of a test suite in exercising predefined entities of a given program. In the case of SOCT, the ITRS orchestrator can obtain an assessment of the coverage achieved on the orchestrated application, including the GDS external services interactions with ITRS. Note that the measures received by the orchestrator do not have a direct connection with the internal structure of the GDS, but only show the percentage of target entities exercised by the executed test cases;

- maintain a test suite. As software evolves, the associated test suites must evolve as well by adding new tests that exercise new behavior and removing obsolete or duplicated tests. In the case of SOCT, the coverage information could assist an orchestrator, for example, in the identification of tests that do not contribute to increase coverage of the orchestrated application invoking the GDS external service, and hence could be dropped from the ITRS test suite;

- detect implementation changes. Changes in third-party software can happen without notice. In the context of SOCT, the GDS services may suddenly change, potentially affecting the reliability of ITRS. Re-executing a test suite periodically would help detect changes that cause tests to fail, but more subtle changes that are not exposed by a failed test could be detectable through changes in the coverage information provided by SOCT. Obviously SOCT is not able to detect how the implementation has changed, but only if it has changed.

It is worth noting that the usefulness of the approach with regard to test suite maintenance is limited to the frequency of the service updates. SOCT can help build a smaller test suite, but when a change is detected the reduced test suite needs to be found again, by re-running all tests from scratch. If the implementation changes too frequently, the benefit of having a reduced test suite would be neutralized; however, an orchestration which relies on consolidated services with a (mostly) stable implementation can take advantage of this possibility.

Also, the capability of the approach to detect changes in the implementation is bound to the granularity of the instrumentation. For instance, if the service is instrumented to measure block coverage, minor changes which do not add or remove blocks might not be detected solely through coverage, unless they change some values in such a way that different paths are exercised.

In general, it is evident 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 SOA 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), we ensure that they can both evolve independently as long as the service interfaces remain the same.

4. The SOCT Approach

With reference to the motivating scenario of the previous section, the SOCT approach is depicted in Figure 1. To the already mentioned ITRS orchestrator and GDS service provider, a new stakeholder that we call TCov is added. TCov is a service provider who sits between ITRS and GDS, delivering coverage information on GDS as the latter is tested by the ITRS orchestrator. To build such a scenario, four activities must take place. First, the company that provides the GDS services must instrument them (callout 1 in Figure 1) to enable the collection of coverage data, not differently from how instrumentation is normally performed in traditional white-box testing. We call such instrumented services the Testable Services. Second, as the ITRS orchestrator (called the SOCT Tester under our approach) invokes the GDS services during on-line testing, coverage measures are collected at the Testable Services (callout 2). Third, the Testable Services send the collected coverage data to TCov (callout 3). Fourth, TCov processes the data, making them available to the ITRS orchestrator as a service (callout 4).

![Figure 1: Scenario of SOCT approach.](image)

In essence, SOCT requires a service-based infrastructure that can enable the collection and reporting of coverage information without exposing the Testable...
Services internals. At the core of that infrastructure is the TCov testing service with an interface for the Testable Services to send the coverage information and another for the SOCT Tester to consume such information. It is worth noting that this simple infrastructure meets the requirements set up front, because: 1) the processes and techniques used by the Testable Services to collect the coverage information remain hidden to the world beyond a service provider interface, maintaining SOA implementation neutrality; 2) the amount and type of coverage information released is defined by the Testable Services, leaving them in control of what they reveal about themselves; and 3) TCov provides a specific interface to report coverage information, making it simpler for SOCT Testers to validate their orchestrations.

The SOCT framework also comes with tradeoffs at the organization and business levels. For a service provider like GDS, implementing SOCT implies costs for instrumenting their services, running the instrumented services, and communicating and sharing coverage information with TCov. For the ITRS orchestrator, obtaining coverage information implies a testing overhead and maybe paying for the TCov additional services. Clearly, there are several degrees of freedom in these implications, and by changing the assumptions behind the scenario the tradeoffs between mutual costs and benefits will change. For example, if TCov is implemented as a part of the same service provider, then the costs and risks for GDS decrease, but the process may lose impartiality and transparency, and hence the approach may be less appealing for the ITRS orchestrator. On the other hand, if having Testable Services becomes a competitive advantage, then service providers would be more willing to make the extra investment. Examining in detail how SOCT would impact SOA business is behind the scope of this work. However, without loss of generality, in Section 4.2 we define a likely scenario on which the TCov implementation presented in this paper is based.

Generally speaking, some effort is needed to set up the SOCT environment, in addition to that required to create the SOA infrastructure. From the point of view of the TCov provider, the TCov service is quite simple, and its only requirements are a web server powered with some database. TCov does not need to do a lot of processing on the coverage information, apart from storing the data and performing some basic queries on the database; the fact that it is not aware of the Testable Service internals implies that the data will be very simple and generic, and coverage reports will only need to perform some basic computation.

Concerning the instrumentation of the Testable Service, this could mean adding many lines of code to the service implementation; however, this activity can be aided if the TCov providers make some facilities available to the Testable Service (e.g., a client library to simplify the introduction of probes). Moreover, there are many tools that can do an automatic instrumentation of the code; this is especially true for the Java language (a major choice for web service), with instrumenters working directly on the bytecode. Instrumenters also greatly reduce the effort of re-instrumenting the code when the service implementation changes, and this aid would be a key speedup of the process if this occurs often.

On the other hand, should the Testable Service change the implementa-
tion without notice to anyone who is currently running a test under the SOCT framework, the coverage information received by the ITRS orchestrator will be badly altered. Therefore, some caution should be taken by the provider of the Testable Service when changing its implementation, for example the change should be deployed only after verifying that no one is performing SOCT testing at the moment, or at least after temporarily inhibiting the SOCT framework. If the orchestrator has reason to believe that the implementation is changing frequently and without notice, then the SOCT approach by itself may not be the optimal solution, and it could be substituted or integrated with more specific solutions, such as [22]. Or, it is also plausible that the orchestrator decides that that service is not stable enough for use in the orchestration, and migrates to some other provider.

4.1. Definitions

The SOCT approach is based on a simple yet powerful idea: obtaining coverage information of third-party services through an independent service provider. In this section we provide definitions for the few concepts on which the approach is based. We start by providing definitions for the key stakeholders.

Definition 1. **A Testable Service** is a web service enhanced to collect coverage information. The enhancements include an extension to the service interface to control the coverage data collection process and the instrumentation to capture coverage data.

Definition 2. **A SOCT Tester** is a person or organization performing validation activities on a composite orchestrated service that includes one or more Testable Service(s).

Definition 3. **TCov** is a set of web services deployed and maintained by a provider, whose purpose is to support the collection and reporting of test coverage information. The Coverage Collecting Services (CCS) in TCov are invoked by any Testable Service to log coverage information. The Coverage Reporting Services (CRS) in TCov are invoked by any SOCT Tester to obtain coverage measures.

The collaboration among these stakeholders is schematized in Figure 2. We observe that for one Testable Service there may be multiple SOCT Testers, each exercising their own operations. The Testable Service collects coverage information for each SOCT Tester and sends it to TCov utilizing a prearranged unique identifier (as later explained in Section 4.2) for each tester. The SOCT Testers can collect their individual coverage measures by invoking the TCov-CRS reporting service.

The communications between these stakeholders are clustered in a Testing Session:

Definition 4. **A (SOCT) Testing Session** is a bounded set of interactions between the SOCT Tester and the Testable Service, over which coverage information is collected.
As with most similar approaches attempting to collect coverage information, SOCT requires code instrumentation in the form of probes to monitor the execution of the targeted program entities:

**Definition 5.** Probes are additional instructions inserted at targeted locations in a Testable Service to enable coverage data collection according to a specified coverage criterion. Testable Service Probes must compartmentalize the coverage separately for each session since there can be multiple parallel testing sessions, and the ones located at a service operation return point will invoke TCov-CCS to log the collected coverage data.

### 4.2. Core Interactions

As expressed before, we anticipate a whole range of possible different instantiations of the SOCT approach, varying according to the degree of independence of TCov, the flexibility provided to the SOCT Tester in terms of choosing coverage data and services, and the stakeholders’ expected returns on investing on a whitened testing process.

In this paper we consider a TCov service that is independent from both the SOCT Tester and the Testable Services. We assume that a contractual agreement has been established between the Testable Services and the independent TCov provider, and that every SOCT Tester exercising an orchestration involving the Testable Services will be directed to interact with that same TCov.

Within this setting, and equipped with the definitions introduced in Section 4.1, our scenario includes the following core interactions (see Figure 3):

1. the SOCT Tester wanting to start a SOCT testing session invokes a `startTest` operation on the Testable Service;
2. the Testable Service obtains from TCov a Testing Session identifier (SID), and sends it to the SOCT Tester. The SID must be persistent throughout the session, and different SIDs (either for different SOCT Testers or for separate sessions opened by a single tester) will produce independent coverage reports;
3. the SOCT Tester will make invocations to the operations defined in the Testable Service interface. Every invocation will contain the SID (the technical solutions are discussed in Section 4.3), so the probes in the testable service can separate the collection of coverage information and send it to TCov-CCS along with the logging data;
4. the SOCT Tester can then invoke TCov-CRS, 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 the Testable Service, and the Testable Service will close the Testing Session, and invalidate the SID.

Figure 3: A scenario of interaction among SOCT stakeholders.

As previously mentioned, there are many degrees of freedom in the application of SOCT. For example, the Testable Service could include TCov as part of its own services to assist service integrators. The proposed infrastructure would
support such a scenario without changes. Still, some scenarios would require further extensions. For example, if our scenario allowed for the SOCT Tester to select a TCov provider among many available, then the Testable Service would have to extend its interface for the SOCT Tester to specify what TCov to use. This is not a challenging technical extension, but it raises the need for additional procedures and policies to manage new interactions with unknown external entities. In this paper we focus on the above detailed scenario, and leave the evaluation of other scenarios to future work.

We now proceed to define more precisely the interfaces involved in these interactions.

4.3. Key Services

To implement the SOCT approach, several technical requirements must be addressed. On one side, a TCov service must be provided, displaying interfaces both for the CCS and the CRS. On the other side, the Testable Service must be modified to embrace the extra coverage functionality. Independently of the implementation of each specific service, the services must support the interfaces introduced next.

4.3.1. The Testable Service

The Testable Service must be expanded to support the following operations to bound a Testing Session:

- **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 identifier of the opened Testing Session (SID) and the URI (Uniform Resource Identifier) of the TCov-CRS service. The SID is generated by TCov-CRS and, as said, is used by the Testable Service to compartmentalize the collected coverage information for that particular Tester as it executes the test cases.

- **void stopTest()**: this operation ends the current Testing Session, invalidating further use of SID.

4.3.2. TCov Coverage Collection Service

The Testable Service interacts with the TCov-CCS to notify the opening and closing of a Testing Session and to log the coverage data information. TCov-CCS operations include:

- **SessionData startTest()**: this operation communicates to the TCov-CCS the intention to set up a Testing Session between the SOCT Tester and the Testable Service. The URI of the TCov-CRS service and a SID is returned to the Testable Service. The latter passes on this piece of information to the SOCT Tester who will use it later on to retrieve the coverage information from TCov.
- **void opCoverageDomain(SID, numEntities)**: this operation communicates to the TCov-CCS the total number of coverage entities, numEntities, in the Testable Service;

- **void coveredEntities(SID, OID, entityList)**: this operation communicates to the TCov-CCS the list of entities covered during the execution of the operation OID. In particular, entityList contains the entity identifiers: these are assigned by the Testable Service and are only required to be consistent within a Testing Session. Ultimately the contractual agreements between the stakeholders will define the level of coverage data accessible by the SOCT Tester. We discuss this in more detail in the next section;

- **void stopTest(SID)**: this operation communicates the end of a Testing Session, invalidating the SID.

As briefly mentioned in Section 4, the SID needs to be persistent throughout a Testing Session. Several SOCT Testers might open a Testing Session on the same Testable Service, or even a single SOCT Tester might want to open many Testing Sessions concurrently. Each of these sessions needs to be tracked independently from the others, and entities covered in one session must not affect the results of queries to TCov-CRS using different SIDs. This requires a way to maintain a persistent SID throughout the Testing Session, which could be achieved in several ways including: 1) as a SOAP variable in the SOAP body, which would require that all the Testable Service operations (including their interface) are modified to accommodate an extra message part; 2) as a SOAP variable in the SOAP header, which would require to parse the SOAP header. The SOAP header, however, may not always be accessible; in particular application servers preventing access to the SOAP header inhibit this solution (for an example of such a case, see the case study we earlier used in [3]); 3) as a cookie exchanged between the SOCT Tester and the Testable Service, thus avoiding any modification to the SOAP communications. Additionally, by setting cookies with an expiration time one can control the validity duration of testing sessions. Our implementation of TCov is based on this last solution.

### 4.3.3. TCov Coverage Report Service

The SOCT Tester interacts with the TCov-CRS to obtain the coverage information collected through:

- **PercentageData coverageMeasure(SID)**: this operation returns PercentageData which includes the achieved coverage and the list of operations executed during the Testing Session.

As previously described, the contractual agreement between the stakeholders will affect what is returned. TCov-CRS may only provide summary coverage percentages as a result, keeping the Testable Service as closed as possible, but limiting the forms of coverage analysis that the SOCT Tester can perform. At
the other extreme, TCov-CRS may reveal the names of the entities covered. This would allow the SOCT Tester to perform a more in-depth analysis of their usage of the Testable Service, but on the other hand might introduce privacy issues on the Testable Service’s side. Clearly there is a yet-unexplored spectrum of choices in between, offering different tradeoffs.

5. Case Study

In this section we start assessing the SOCT support for whitened testing of SOA orchestrations like the ones described in Section 3. We focus on two research questions:

RQ1: SOCT Usefulness: is SOCT useful for improving SOA testing? In particular, we will assess whether it can support test suite assessment and reduction (RQ1.1) and selective regression testing for change detection (RQ1.2);

RQ2: SOCT Viability: is the overhead introduced by the proposed SOCT infrastructure acceptable?

5.1. Infrastructure and Artifacts

To run the case study we developed an instantiation of TCov. The TCov interfaces described in Section 4.3 utilize a small service engine written in 180 lines of PHP using the PHP-SOAP extension. It refers to a MySQL database backend to store the data captured by CCS and to serve the queries of the SOCT Tester submitted through CRS. For each session, our implementation of TCov stores: the type of coverage collected and the total number of entities for that coverage type, the operations called with a timestamp, and the covered entities once the session is finished.

Keeping TCov light-weight is key for a higher efficiency of the setup. First of all, TCov only has to collect data and send them as aggregate after performing some basic calculations; there is no need of a complex service, as TCov does not delve into the meaning of the data it collects. Secondly, the simpler TCov is, the less likely it will be for it to contain errors. The presence of errors introduced by an erroneous implementation of TCov would thwart the benefits of the SOCT approach. In the following, we will assume that TCov is correct in collecting data and in sending the coverage reports.

SOCT requires that the independent services invoked within a SOA orchestration are instrumented, so as to serve as Testable Services. For this reason, our choice fell on the WorldTravel service [39], a system purposely conceived by its developers as a testbed for research and made available under the BSD license. We made some adjustments to the original WorldTravel benchmark for it to work in newer versions of the specified run-time environments. The current version, with our revision, is available for download:\footnote{\url{http://labsewiki.isti.cnr.it/labse/tools/worldtravel/public/main}}.
WorldTravel provides a service to search for flights (the scenario described in Section 3 is in fact inspired by this case study). As illustrated in Figure 4(a), it consists of a Global Distribution System (GDS), which receives requests and acts as a load balancer, forwarding requests to one of the multiple Query Processing Servers (QPSs) available. A QPS receives a flight request in the form of an XML file, extracts all the useful information, generates a SQL query which is then sent to the database, and packs a response which is sent back to the GDS\textsuperscript{4}. The database used by WorldTravel consists of 12 relations but most of the queries on flight rates access a single large table containing 8,273,037 records.

We first played the role of the Testable Service provider and instrumented WorldTravel: we inserted probes for capturing block and branch coverage information. As a test proceeds, each inserted probe updates a coverage vector that is sent to TCov at the end of each web service operation invocation.

WorldTravel is a software written in Java using the J2EE framework; in our experimentation, we used a portion of WorldTravel including a single QPS and totalling 3281 Java LOCs, thus divided:

- 2488 LOCs are for the core component supporting the whole WorldTravel system. This code basically contains shared data structures, factories and so on, without any computation;
- 832 LOCs for the GDS;
- 404 LOCs for the QPS.

Instrumenting the code mostly affected the QPS component, adding to it 63 LOCs for block coverage probes, and 83 more for branch coverage. 35 lines have been added to GDS, mostly to enable the startTest and stopTest operation requests. Additionally, a JAR library providing a ready-to-use TCov client has been linked. The library source consists of 1578 LOCs, but most of it has been automatically created from the WSDL (using the Axis WSDL2Java tool [2]), whereas only 151 LOCs contain the actual service implementation.

Once WorldTravel is transformed into a Testable Service, we take on the role of the SOCT tester and consider to integrate WorldTravel Testable Services into a larger orchestration. We implemented two different orchestrations following the ITRS scenarios as described in Section 4 (Figure 1 and Figure 4(b)). Both are simple web services written in PHP, made up of a total of 271 LOCs and 249 LOCs respectively.

The first one is a flight booking service called ST1: the customers can use it to check flights availability. ST1 receives queries with specific attributes (one-way or round trip, dates, acceptable fare classes, and desired airlines), and finds the best fares among the flights meeting the search criteria. ST1 has the

\textsuperscript{4}The original WorldTravel distribution also includes a frontend servlet, but since we are using WorldTravel as part of an orchestrated service we are not using this component in this study.
The following operation: `FareResponse bestFare(FareRequest)`

- the originating airport (string)
- the destination airport (string)
- the date of the flight to the destination (dateTime)
- [0..1] the date for the return flight (dateTime), if needed for a round-trip flight
- [0..3] the requested fare classes (enum: First, Business, Economy)
- [0..*] the requested airlines

The data returned in the FareResponse object contains zero or one flight that include:
• [1..2] trips, each containing the flight name, date, origin and destination
• the fare class (string)
• the cost of the flight (int)

The second orchestration, called ST2, implements a corporate flight agency. We assumed that the corporation has a number of departments in different cities. To manage frequent travels of personnel between departments, ST2 automatically checks fare updates for the most common routes. ST2 accepts as input an origin city and a number of destination cities, and finds the best first-class fare for all round-trip flights to those cities, for the current day. More formally, the service implements the following operation: 

\[
\text{FindPricesResponse findPrices (FindPricesRequest)}
\]

requires the following input:

• the origin city (string)
• [1..*] the destination cities (string)

The returned data contains:

• [1..*] fares, each with information about the city and the corresponding fare

For this study, WorldTravel and the two frontend services we developed were installed on the same machine, while TCov resides on a remote server. The computer hosting the ITRSs services and WorldTravel is based on a Fedora™ 8 Linux OS running on an AMD Athlon® 64 processor at 1.8 GHz. The machine executing TCov, on the other hand, is equipped with a Fedora™ 9 Linux OS on an Intel® Core™ 2 Duo CPU at 2.2 GHz.

In our role of SOCT Testers, we prepared test suites to validate both ST1 and ST2, following the well-known Category Partition (CP) methodology [25]. This methodology provides a stepwise approach to identify the relevant input parameters and environment conditions for functional testing. In short, CP first partitions the functional specifications of a unit into categories, i.e., the environment conditions and the parameters that are relevant for testing purposes. Then the categories are partitioned into choices: these represent significant values for each category from the tester’s viewpoint. Finally, to prevent the construction of redundant, not meaningful or even contradictory combinations of choices, constraints among choices are determined (either properties or special). A test plan specification is obtained by combining categories, choices and constraints.

The categories, choices, and constraints for ST1 and ST2 are presented in Tables 1 and 2 respectively. As detailed in the tables, the test plans for ST1 and ST2 consider both valid and invalid inputs. For ST1 this setup gave us a total of 31 test cases, where 16 are legitimate tests exercising valid input values, while the remaining 15 are either invalid values or special situations (such as multiple values). Concerning ST2, the CP provided a test suite of only 7 tests where 2 are legitimate and 5 are either invalid values or special situations. As part of the infrastructure, we also built a Test Manager to send these tests as SOAP calls to ST1 and ST2 and collect responses to facilitate the data collection.
Table 1: Test Plan for ST1

<table>
<thead>
<tr>
<th>Categories</th>
<th>Choices</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>origin</td>
<td>city existing in database</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>non-existing error</td>
<td>error</td>
</tr>
<tr>
<td></td>
<td>null error</td>
<td>error</td>
</tr>
<tr>
<td>destination</td>
<td>city existing in database</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>non-existing error</td>
<td>error</td>
</tr>
<tr>
<td></td>
<td>null error</td>
<td>error</td>
</tr>
<tr>
<td>dateTo</td>
<td>valid date</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>invalid date (wrong dd) error</td>
<td>error</td>
</tr>
<tr>
<td></td>
<td>invalid date (wrong mm) error</td>
<td>error</td>
</tr>
<tr>
<td></td>
<td>invalid date (wrong yyyy) error</td>
<td>error</td>
</tr>
<tr>
<td></td>
<td>null error</td>
<td>error</td>
</tr>
<tr>
<td>dateFrom</td>
<td>valid date</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>field not present</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>invalid date (wrong format) error</td>
<td>error</td>
</tr>
<tr>
<td></td>
<td>date prior to dateTo error</td>
<td>error</td>
</tr>
<tr>
<td></td>
<td>null error</td>
<td>error</td>
</tr>
<tr>
<td>classes</td>
<td>First</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>Business</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>Economy</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>field not present</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>all three classes single</td>
<td>single</td>
</tr>
<tr>
<td></td>
<td>null error</td>
<td>error</td>
</tr>
<tr>
<td>airlines</td>
<td>existing airline</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>field not present</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>two airlines</td>
<td>single</td>
</tr>
<tr>
<td></td>
<td>non-existing error</td>
<td>error</td>
</tr>
<tr>
<td></td>
<td>null error</td>
<td>error</td>
</tr>
</tbody>
</table>

Resulting in 31 Tests

5.2. RQ1: SOCT Usefulness

In this section we investigate the usefulness of SOCT in terms of:

RQ1.1: Test suite reduction: we study whether TCov reports can be useful to guide test suite reduction in the testing of SOA orchestrations;

RQ1.2: Test suite for regression detection: we investigate the use of TCov reports for detecting regression changes involving the external services, in particular we evaluate the effectiveness of the reduced test suite in detecting changes to the Testable Services.
Table 2: Test Plan for ST2

<table>
<thead>
<tr>
<th>Categories</th>
<th>Choices</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>origin</td>
<td>city existing in database</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>non-existing</td>
<td>error</td>
</tr>
<tr>
<td></td>
<td>null</td>
<td>error</td>
</tr>
<tr>
<td>destination</td>
<td>one city existing in database</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>two cities existing in database</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>no city</td>
<td>single</td>
</tr>
<tr>
<td></td>
<td>non-existing</td>
<td>error</td>
</tr>
<tr>
<td></td>
<td>null</td>
<td>error</td>
</tr>
</tbody>
</table>

Resulting in 7 Tests

5.2.1. Test suite reduction

As a program evolves, so does its test suite. Over time a test suite can get bloated with obsolete and duplicated test cases, leading to inefficiencies and correspondingly higher test execution costs. Test suite reduction techniques are meant to address this problem. Given a test suite $TS$ and a test adequacy criterion $A$, a test suite reduction consists of selecting a subset of tests in $TS$ resulting in a test suite $TS'$ that still satisfies $A$.

Consider the test plans for the flight booking service orchestration ST1. Running all the 31 test cases in one random sequence results in the data plotted in Figure 5, which depicts the cumulative coverage increments for block and branch coverage reported by TCov. Running the whole test suite reached a 82.54% of the blocks and 72.29% of branches, taking approximately 227.12 seconds. As it can be noticed in Figure 5, there are test cases that do not contribute to increase the overall coverage, while others attain a big leap in the coverage percentage. By comparison, a reduced test suite made up of only tests $T_1$, $T_{29}$ and $T_{31}$ can reach the same total block coverage in 9.68 seconds. Clearly, larger orchestrations may have much larger test suites, increasing the opportunities for test suite reduction in decreasing testing time, and more important, reducing the number of tests that require effort to validate their outcome against an oracle.

Techniques for test suite reduction based on code coverage measures have been actively investigated [42]. However, the SOCT setting is unique in that TCov does not provide the SOCT Tester with information on which are the entities that are covered, it just provides the (increments in the) coverage measure. So, we need to conceive a heuristic for reduction solely based on the coverage measures that can be reported by TCov. We built a simple Test Reduction Heuristic (TRH) as follows: given a coverage report listing the cumulative coverage measure for each executed test case, the reduced test suite includes all and only those test cases that provide a coverage increment.

Applying TRH to the results of Figure 5, for instance, we would then choose
Test Cases $T_1$, $T_2$, $T_3$, $T_5$, $T_7$, $T_9$, $T_{29}$ and $T_{31}$. Now, the reduced test suite obtained with this heuristic varies depending on the order in which the test cases are executed within the initial test suite. To assess the effectiveness of the proposed heuristic then, we repeatedly launched several Testing Sessions which executed all test cases randomly shuffled.

For each shuffled Testing Session, we applied TRH and extracted a reduced set of test cases. This reduced set has a number of tests which varies according to the order in which tests were run. We executed the reduced test suites and measured their cumulative coverage. The results relative to block coverage are displayed in Table 3 and show that this simple heuristic can reduce the test suite to approximately one fifth of the original size (a mean of 5.65 test cases out of 31 in the original test suite) and still reach about 82% of block coverage (results for branch coverage are similar).

A reduced test suite is valid only for a specific service implementation. The SOCT Tester can assume that no changes were made to the implementation as long as the reduced test suite produces the same coverage results (this might not be completely true, but there is no possibility of a better knowledge with a black-box application such as a web service). However, after a change has been detected through a different coverage report, the full test suite should be executed from scratch, applying TRH again to find a new reduced test suite.

Of course, the relevance of the reduced test suite is limited to exercising the highest possible coverage in the web service. The usefulness of the reduced test suite is represented by the fact that additional tests do not cover any uncrossed probe, but does not mean that the SOCT Tester has no interest in running them
Table 3: Reduced test suites.

<table>
<thead>
<tr>
<th>Permutation</th>
<th>Number of Tests</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>83.87%</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>80.65%</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>83.87%</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>80.65%</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>83.87%</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>77.42%</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>83.87%</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>83.87%</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>80.65%</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
<td>80.65%</td>
</tr>
<tr>
<td>11</td>
<td>6</td>
<td>80.65%</td>
</tr>
<tr>
<td>12</td>
<td>6</td>
<td>80.65%</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>90.32%</td>
</tr>
<tr>
<td>14</td>
<td>7</td>
<td>77.42%</td>
</tr>
<tr>
<td>15</td>
<td>7</td>
<td>77.42%</td>
</tr>
<tr>
<td>16</td>
<td>5</td>
<td>83.87%</td>
</tr>
<tr>
<td>17</td>
<td>7</td>
<td>77.42%</td>
</tr>
<tr>
<td>18</td>
<td>6</td>
<td>80.65%</td>
</tr>
<tr>
<td>19</td>
<td>3</td>
<td>90.32%</td>
</tr>
<tr>
<td>20</td>
<td>7</td>
<td>77.42%</td>
</tr>
</tbody>
</table>

Mean 5.65 81.77%

to verify the functional correctness. The typical scenario where the reduced test suite is useful is one where the full test suite has already been executed and the SOCT Tester knows that the service behaves correctly under all test cases. Once a change is detected, the SOCT Tester will be interested to run the whole test suite (and not only the reduced one) again.

5.2.2. Testable Service regression detection

Regression testing is fundamental in SOA testing, because of its high dynamicity and continuous evolution. SOCT can help regression testing through the coverage reports that can detect whether an external service implementation has been modified. Note that this is slightly different from traditional regression testing. Normally, regression testing is performed when a change is made to a system, to ascertain that tests that previously passed are still successful, so usually a change in the system is the stimulus for regression testing. In this case, we suggest that tests are repeated to detect whether a change has occurred.

Our suggested use of SOCT for SOA regression detection is that a test suite reduced according to the TRH heuristic is periodically run, and the achieved coverage measures are compared with the previous ones. The underlying hypothesis is that the same reduced Testing Session must achieve the same coverage percentage every time it is executed on the same Testable Service, unless the service has been changed. We note that the Testable Service, however, could still be modified without resulting in a change in the percentage of coverage reported by TCov. The intention here is simply to give a warning that something has been changed in the Testable Services based on the additional coverage information available through TCov. In such a case, the tester may decide to fully execute the complete test suite, to identify possible new problems, and to register the new coverage measure and identify a new minimum set of tests.
We empirically investigated this approach by mutating the WorldTravel service. The mutants were generated from a non-instrumented version of the code, with SOCT instrumentation added separately to each mutated version. As the instrumentation and setup of the experiment were largely done manually, and repeating them for each single mutant required significant time and effort, we limited the number of mutants to 20, making a random selection as follows. We first randomly picked 20 lines from the code, and then on each of these lines we again randomly picked one mutation operator for Java [20] applicable to the picked line. Where no mutation was applicable (e.g., the picked line only included a parenthesis or a comment), we considered the first valid subsequent line.

Once the mutants were generated, we executed on each of them the TRH-reduced test suite, measuring the coverage obtained. The coverage results thus obtained were compared with those obtained by executing the same reduced test suite on the original WorldTravel service, so as to verify whether the TCov percentage would help in detecting changes in the code.

We ran all the 20 reduced test suites from the experiment described in the previous section on the set of 20 mutants and on our golden version, the original WorldTravel service (summing up to a total of $20 \times 21 = 410$ reduced Testing Sessions), and we collected coverage information. The results obtained are shown in Table 4. Each row of the table shows a separate mutant, while the columns represent the reduced test suites from the previous experiment. The table shows the behavior of the test suites against the mutants, with the following meanings:

- mutants which produce a different output from the golden version but have the same coverage pattern are marked with a slash (/);
- those which differ both in output and in the coverage pattern are marked with an asterisk (*);
- mutants which produce the same output as the golden version and also have identical coverage results are unmarked;
- finally, mutants with identical outputs but differing in the coverage pattern are marked with a dagger (†).

All test suites behaved in the same manner on a given mutant (either all killed a mutant, or none did). In particular, as the table shows, there were 7 live mutants (not detected by output difference), or 35% of the total number of mutants. The most interesting issue is that three of these mutants (15% of the total number of mutants) that have the same output in the mutated and the original version of Worldtravel can be detected through differences in the coverage measures reported by TCov, and a reduced test suite is sufficient to detect them.

We note that it is common for the implementation of a web service to change without notice and still generate the same outputs as new features have been added, new parameters included into the operations invocation, or faults been
fixed. The SOCT environment offers an additional way for the orchestrator to be aware of such changes (whose impact on coverage would likely be easier to detect than a single line mutation as in our case study).

5.3. SOCT Viability

The SOCT approach requires some changes on the Testable Service, changes which may affect its performance. To evaluate the runtime impact of such changes, we analyzed the services both with and without the TCoV instrumentation under two settings:

- a raw analysis of temporal overhead with a single SOCT Tester, without any concurrent invocations;
- an analysis of the extent of the capabilities of our implementation, by stressing it with concurrent clients.

Our first experiment was to run an incremental number of invocations in three different execution modes:

1. invoking the Testable Service without instrumenting it. Probes are not even present in the code, and the \texttt{startTest} and \texttt{stopTest} operations are not available;
2. invoking an instrumented Testable Service, but without enabling the TCoV logging. Probes are present, but since \texttt{startTest} and \texttt{stopTest} are not invoked in this mode, no logging is ever sent to TCoV. This execution mode allows us to estimate the raw impact of the computation time introduced by the instrumentation;
3. invoking the instrumented Testable Service with TCoV logging enabled, with TCoV hosted on a remote server.

In all three execution modes, one test consisted of sequentially invoking the service an incremental number of times, from 1 to 100; this means that the first test is made up of a single invocation, the second of two sequential invocations, and so on. Every invocation consists of a single valid request sent by ST1 to the ITRS; we are not much interested in this experiment in the content of the request or in its response, so all invocations are identical. In the third invocation mode, each test is also a separate Testing Session, to also account for the delays introduced by the \texttt{startTest} and \texttt{stopTest} operations. The results, outlined in Figure 6, reveal that the three execution modes show no significant difference in response times. Apart from a few occasional spikes (probably due to accidental server load or network congestion), the three graphs mostly overlap, with response times substantially identical at both sides of the graph.

We then moved on to the second experiment, focused on concurrency. We forked a client an increasing number of times (again, 1 to 100), and each child process made a concurrent invocation to the service. We did this using an instrumented Testable Service, TCoV-enabled Testable Service, and with a non-instrumented Testable Service (we left out the experiment with the disabled instrumentation because we consider the overhead it introduces trivial).
Figure 6: Response times of successive invocations of the SOCT Tester.

Figure 7(a) displays the response times of the invocation batches. Generally, the response times for the non-TCoV mode is somewhat smaller, until around 70 tasks. However, probably due to limitations of the test machine running the Geronimo application server, some of the forked threads fail, as shown in Figure 7(b), and, though at a certain point (73 concurrent threads) the system saturates, failing completely to respond to any request at all, failures appear earlier in the TCoV-enabled mode than the other (47 concurrent tasks vs. 63, apart from a couple of early spikes).

5.4. Putting the Findings in Context

The results of the case study provide preliminary evidence that TCoV can be successfully instantiated to operate on a SOA orchestration, can generate useful information for improving the testing of a set of orchestrated services, and can operate without introducing significant overhead on the Testable Services while operating with hundreds of requests.

These findings, however, must be considered within the context of the choices we made for our study, which might threaten their external validity.

First, the Testable Service we selected, WorldTravel [39], is a benchmark that may not reflect the population of web services available in terms of its complexity, feature availability, and deployment scale. WorldTravel, however, is among the first larger benchmarks available for the research community trying to devise techniques to validate web services and web services orchestrations, and we are among the first researchers using it to assess the approach we have developed.
Figure 7: Experiments with multiple tasks.
Second, there are other applications that could exploit TCov data that we did not assess, multiple governance scenarios that we did not explore, and many other measures and settings to assess the value of the applications, scenarios, and implementations that were not part of our setup. Our choices were meant to show the value of SOCT in at least one scenario that we have seen in practical settings, but clearly more studies are necessary to better understand the strengths and limitations of SOCT.

Third, our study is executed in a simplified contextual setting. Our assessment does not account, for example, for the complex costs and benefits that a service provider must consider such as those associated with the long term deployment and maintenance of a testing service. Similarly, our assessment barely considers the contractual agreements that may govern the provision of such service.

Finally, there are various infrastructure elements whose implementation may have impacted the internal validity of the results obtained. For example, different implementation choices for TCov, for the clients using the Testable Service, or for the test generation process may lead to different results. The regression detection study of course depends on the mutants generated, and the validity of the results may be threatened by the low number of mutations executed. To address this concern, we provided detailed accounts of the choices we made. We also make our code available upon request to other researchers who wish to reproduce the results or compare them against other alternatives. Concerning the viability study, some implementation-dependent issues are directly discussed in the results report (end of Section 5.3).

6. Conclusions and future work

We have presented SOCT, an approach that whitens SOA testing, overcoming what is apparently a contradiction in terms: it empowers a SOA orchestrator to obtain coverage information on an invoked external web service, yet without breaking the implementation neutrality of the latter, which is one of SOA founding principles. Such an attainment is possible if providers are willing to instrument their services so that users invoking 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 by a service provider that we called TCov.

We have motivated the need for whitening SOA testing, defined the approach, and implemented an instance of it. We have performed a case study on a publicly available benchmark. The study results confirmed that TCov can be successfully instantiated to operate on a SOA orchestration and can generate useful information for improving the testing of a set of orchestrated services, for example for test suite reduction and for regression detection. We also provided some preliminary evidence that introduced overheads are not significant from the point of view of the runtime performance. Concerning the added technical effort to set the SOCT environment up, both from the perspective of TCov
(which is a lean and simple SOAP service), and from that of the Testable Service (with the aid of a provided client library for TCov and tools for automatic instrumentation), we have amply discussed the respective requirements that can easily be addressed.

Beyond these, we are aware that putting SOCT into practice requires an agreement and commitment from all involved stakeholders, in terms of rules, policies, and enforcement procedures, which altogether form the SOA governance [7]. The approach requires, for example, that developers of Testable Services instrument their code, reveal program information, and incorporate additional services for external testing, all of which imply additional resources. It also requires that contractual agreements are established between the TCov providers and the Testable Service providers, or the SOA orchestrators invoking them, which may imply further costs. 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 tools for making services more transparent and, in the end, more trustworthy.

In the future, we plan to continue investigating SOA white-box testing, extending this paper in several ways. First, we want to refine several aspects of the SOCT instantiation, such as those associated with the estimation of the coverage domain and the efficient collection of coverage information. In particular, we would like to provide guidelines for the placement of the probes into the Testable Service. We also would like to leverage a large body of existing work on minimizing the overhead caused by instrumentation probes, and the collection of additional information such as a failure states to also help the orchestrator to refine the tests and the test oracles.

Second, we would like to investigate novel strategies that the orchestrator could apply to improve the test cases when structural testing coverage is not sufficient. A possibility in this case could be that the service provider might suggest supplementary information or, more bluntly, test cases, to achieve better coverage. However, the main problem in doing this is that the Testable Service would be available for a number of orchestrators, each with his or her own purpose, who might use the service differently from all the others, calling different operations and so on. The Testable Service might provide a set of test cases for general use of the service, but the tests which might be required by one orchestrator might be completely useless to another. For this reason, it is preferable that the Testable Service only introduces the facilities to ease the testing on the orchestrator’s part, without attempting to respond to their specific requirements. Alternatively, some fuzz testing approach could be attempted, but this would raise issues of costs and side effects.

Third, we plan of course to perform other larger-scale studies to obtain a better understanding of the cost, performance and overall impact on the quality of the service, and potential benefits of SOCT. We note that the implementation and the results obtained in this paper set the stage for such costly studies.

Finally, we also intend to explore and formalize the governance framework behind SOCT, so to make explicit all assumptions and agreements required on service developers and orchestrators who are willing to adopt the approach.
Acknowledgements

This work was partially supported by the following projects: the TAS³ Project (EU FP7 IP No. 216287), the Italian MIUR Project D-ASAP (Prin 2007), and the National Science Foundation Award #0915526. We also wish to present our gratitude to the reviewers for their important suggestions which we believe improved this work a lot.


Table 4: Behavior of the reduced test suites against the mutants.

<table>
<thead>
<tr>
<th>Mutant</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>2</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>3</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>4</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>5</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>6</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>7</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>10</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>11</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>12</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>13</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>14</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>15</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td>†</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>17</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>18</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>19</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>20</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

/ different output * different output and coverage † different coverage