Int J Software Informatics, Volume 7, Issue 2 (2013), pp. 239–271 Tel: +86-10-62661040
International Journal of Software and Informatics, ISSN 1673-7288 http://www.ijsi.org

Email: ijsi@iscas.ac.cn, ijsi2007@gmail.com

Applying Structural Testing to Services Using Testing Interfaces and Metadata

Marcelo Medeiros Eler[1], Antonia Bertolino[2] and Paulo Cesar Masiero[1]

[1](ICMC/USP, Sao Carlos/SP, Brazil)

[2](ISTI/CNR, Pisa, Italy)

Abstract By their very nature, services are accessible only as black-boxes through their published interfaces. It is a well known issue that lack of implementation details may reduce service testability. In previous work, we proposed testable services as a solution to provide third-party services with structural coverage information after a test session, yet without revealing their internal details. However, integrators do not have enough information to improve their test set when they get a low coverage measure because they do not know which test requirements have not been covered. This paper proposes an approach in which testable services are provided along with test metadata that may help integrators to get a higher coverage. The approach is illustrated on a case study of a real system that uses orchestrations and testable services. A formal experiment designed to compare the proposed solution with a functional approach is also presented. The results show evidences that subjects using the testable service approach augmented with metadata can achieve better coverage than subjects using only a functional approach.
Key words: testable service; structural testing; coverage; metadata; evaluation; service-oriented architecture

Eler MM, Bertolino A, Masiero PC. Applying structural testing to services using testing interfaces and metadata. *Int J Software Informatics*, **Vol.7, No.2 (2013): 239–271. http://www.ijsi.org/1673-7288/7/i153.htm**

1 Introduction

Service Oriented Computing is an emerging paradigm that uses interoperable services as the building blocks to package bigger applications[20]. Particularly, Service Oriented Architecture (SOA) is an architectural style that promises rapid and low-cost development of loosely coupled and easily integrated applications even in heterogeneous environments[24]. According to this architecture, service providers refer to service brokers to publish their services, whereas service consumers refer to service brokers to find suitable services to compose in their applications.

Services can come in two flavors: atomic or composed. A composed service (or composition) is developed by integrators and may be described as an application (or part of it) that delivers its functionality through the interaction of more services.

This work is supported by the Brazilian funding agency CNPq and FAPESP (process 2008/03252-2) for the financial support and partially supported by the European Project FP7 IP 257178 CHOReOS.
Corresponding author: Marcelo Medeiros Eler, Email: marceloeler@gmail.com
Received 2012-09-28; Revised 2013-01-20; Accepted 2013-04-19.

Depending on the scheme of interaction, compositions form *orchestrations* or *choreographies*[16], and aggregate services that are possibly under the control of different ownership domains. The broad adoption of SOA depends on the confidence integrators have on services provided by third parties. One way to gain trust on services, and in general in any piece of software, is by testing[1, 6]. However, testing third party services is not an easy task due to their inherent low testability.

Testability has been defined as "the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met"[18]. It is also an important quality indicator since its measurement leads to the prospect of facilitating and improving a service test process[23, 27]. Third party services generally yield a low testability for two main reasons: the cost of the testing activity and the black box nature of services.

Testing a third party service may be expensive as costs may be associated with its execution[12, 25]. For example, there can be services to be paid per each invocation or others that, even if free of charge, only permit a limited number of invocations in a fixed period of time to avoid problems such as performance loss and denial of service. Therefore, test cases should be chosen with accuracy[12, 25].

Integrators have no access to the internal details of third party services since the latter are provided as black boxes. For them, a third party service is only an interface providing operations that can be invoked remotely. The encapsulation of third party services hampers establishing and tracing testing criteria based on implementation. Integrators can only use testing techniques based on the spectification or on the interface of the service[23]. Such characteristic is a common issue of services and components when it comes to testing[15]. There could however be situations in which interface-testing of services is not deemed sufficient and combining implementation and specification-based testing techniques is desirable. In fact, these two techniques are meant to find different types of failures[21] and their combined application may provide higher confidence.

Towards this purpose, two similar approaches have been independently proposed by Bartolini et al.[2, 3] and Eler et al.[14]. The two proposals aim at improving the testability of SOA applications by allowing for white-box testing of third-party services. In both approaches, the internal details of the services are not exposed thus preserving the encapsulation principle of SOA. The process and the infrastructure by which this coverage information is obtained varied in the two approaches (we refer to Refs. [1, 2] and Ref. [14] for the details). The basic idea is however the same: they suggested that the services are created with the capability to provide their clients with structural coverage information. Services offering such capability have been named *testable services* by both approaches.

Integrators can thus test a third party testable service and get a structural coverage analysis based on the test session carried out. The coverage analysis may refer to both control (all-nodes, all-edges, all-paths) and data flow (all-uses) criteria. Integrators can test the service in two different contexts: in isolation (unit testing) or from within a composition (integration testing). In the former situation, the integrator launches test cases to test the testable service through its interface, and in the latter situation, the integrator launches test cases to test the composition that uses the testable service. In both cases, the integrator can get a structural

coverage report.

The coverage measure provided by the testable service is a feedback about the thoroughness of the executed tests. Integrators can know how much (instructions, data, paths) of the testable service is being executed from the context of the composition. With such information, the integrators can decide to enhance their test set to exercise the testable service as much as possible from the context of the composition and prevent that untested code hides latent failures.

The testing facilities and the coverage measure provided by testable services, however, provide the integrators with no clue of how testing could be improved in the case the coverage score is low. The integrator does not get enough information to decide whether the coverage achieved is good or bad, nor to understand how the test set should be augmented to increase the coverage. Through the testable service approach, the integrator would know, for example, that 40% of service's control flow nodes have not been covered yet. However, the integrator would not know which are these nodes, because the source code or other models, such as the control flow, are not made available. This limitation was shared by both testable service approaches.

To find a solution, proposers of the two approaches joined their effort and introduce here an enhanced approach called More Testable Service by Test Metadata (MTxTM), which is based on a test metadata model inspired by the concepts of built-in testing[15, 28] and metadata[22]. MTxTM was designed to make testable services even more testable by providing integrators with information about the testing activity carried out by the testable service's developers. Integrators can use such information to evaluate the coverage reached on the testable services used and to create new test cases to exercise instructions, paths and data which have not been executed yet. In this paper, we expand on a previous version[13] by providing more insight into the motivations behind the approach and its potential applications. We also present a more extensive validation, through an exploratory case study and a novel experiment designed to compare MTxTM with a functional approach and to show evidences of the benefits of using a testable service approach.

This paper is organized as follows. Section 2 presents the basic concepts and an overview of the testable service approach. A motivating scenario in Section 3 shows why the coverage information by itself is not sufficient, and then the proposed test metadata model for testable services is presented in Section 4. Section 5 shows a case study used to perform the first evaluation of the proposed approach. An experiment designed to evaluate the MTxTM approach in comparison to a functional approach is presented in Section 6 and results are discussed in Section 7. Section 8 presents some related work and, finally, Section 9 presents the concluding remarks of this paper.

2 Testable Service Approaches to Allow for Structural Testing of Services

In previous work, we conceived two similar approaches to develop testable services to improve the testability of SOA applications by making services more transparent to external testers while maintaining the flexibility, the dynamism and the loose coupling of SOA. These services have been called *testable services*, because they are inspired by the concept of testable components[15]. The approach proposed by Bartolini et al.[2, 3] is called SOCT, while the approach proposed by Eler et al. is called BISTWS[14].

A testable service has been defined as a web service instrumented to collect

coverage information (instructions, data and paths exercised) and accessible through a WSDL interface that includes basic operations to enable the collection of testing data. The instrumentation of a service to transform it to a testable service is made by the developer of the service. This process can be manually done by the developers (SOCT) or fully automated by a tool or a testing service (BISTWS). During the instrumentation, probes are inserted into the service's code. Probes can be defined as additional instructions inserted at targeted locations to enable coverage data collection according to a specific coverage criterion. The coverage data collected by the probes are stored locally or sent to a service designed to collect these information and calculate the coverage measure when it is required.

The instrumentation of a testable service also adds operations to provide clients with structural testing facilities. These operations are used to define the boundaries of a test session and to retrieve coverage information regarding a specific test session, which has to be uniquely identified. A test session is thus a bounded set of interactions between the tester and the testable service during which coverage information is collected. The operations to set the boundaries of a test session and to get coverage information have been called `startTest`, `stopTest` and `coverageMeasure` in SOCT and `startTrace`, `stopTrace` and `getCoverage` in BISTWS.

Integrators can use a testable service to compose their orchestrations or choreographies and take advantage of its structural testing during testing activities. They can test either the provided operations of the testable service in isolation (similar to unit testing) or the testable service from the context of the composition. In both cases, the integrators can use the testable service operations to start a test session, launch a set of test cases, stop the test session and then can get a coverage measure report on structural testing criteria (control and data-flow). This report is a feedback about the thoroughness of the executed tests.

In Fig. 1, we present an abstract sequence diagram to illustrate an integrator using a testable service in the context of an orchestration. The integrator invokes the testable service that composes the orchestration to start a test session. Next, a test set designed to test the orchestration is executed. The orchestration interacts with the testable service and the testable service collects coverage data when executed. After executing the test set, the integrator invokes the testable service to stop the test session and then asks for a coverage analysis, which is produced by the testable service. In this diagram, we hide away details of how the coverage information is eventually achieved by the testable service, because it does not matter from the point of view of the integrator.

Despite the small differences between the approaches of Ref. [3] and Ref. [14], they are very similar and follow basically the same process and governance framework[10]. In both approaches the developer has to provide the testable service. The integrator has to start a testing session, execute a test set, stop the test session and ask for a coverage measure. Then, the coverage analysis is performed by a specific service (TCov or TestingWS).

From the point of view of the integrator, there is no difference between the two testable approaches mentioned before, in the sense that the integrator is ultimately interested in getting the coverage measure. They do not need to worry about code instrumentation and how the coverage information is stored or calculated, because

this is done by a tool that is accessed by them as a service. Therefore, from now on, in this paper we will not distinguish anymore between the two approaches. The case study and the experiment presented in this paper, however, will use the infrastructure provided by BISTWS[14].

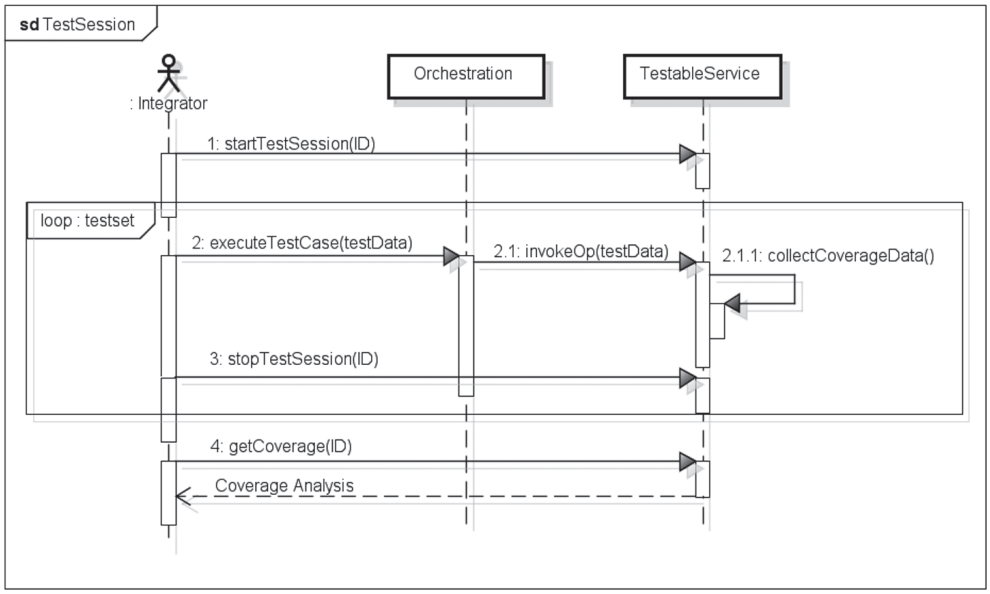

Figure 1. Sequence Diagram of a generic test session of a testable service

The integrators may be worried instead about how reliable or trustworthy is the coverage measure obtained. In fact, they have no way of knowing if the instrumentation introduced by the developer can correctly measure the coverage score or even that it was not manipulated to developer's advantage, since it was introduced by the developers themselves. Addressing this issue is a matter of governance, as discussed in Ref. [10], and several solutions could be taken for a real-world implementation. One could be a service agreement contract between the integrator and the developer (the confidence on the coverage would be the same attributed to the service) or the instrumentation and coverage tool could be certified by an independent certifier. The coverage tool could also be developed and operated by a third party trusted company. We do not expand further on governance aspects beyond our approach: the problem is important but is out of scope for this paper.

3 The BroadLeaf Commerce Test Scenario

We use the BroadLeaf Commerce framework as an illustrative scenario to present our proposed approach. It is an open source E-Commerce framework[1]* composed by several services, each providing a specific feature such as customer and catalog management, shopping cart, order, shipping and payment processes. The developers provide a complete instantiation of the framework and its source code, including the

[1] `http://www.broadleafcommerce.org`

test cases for almost all services. We use this software to illustrate and motivate our approach. For this purpose, we selected a service called `RegisterService`, which uses several services, among which a single service called `CustomerService`. Figure 2 shows an illustration of this scenario.

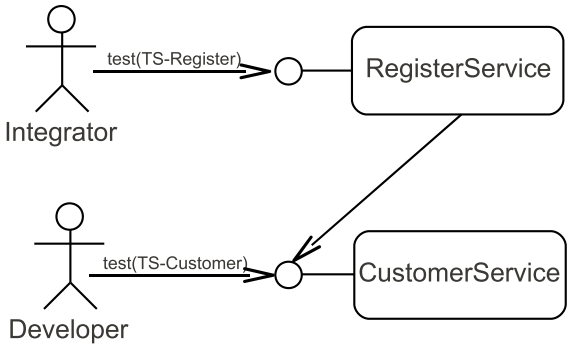

Figure 2. Test scenario of RegisterService and CustomerService

The developer of `CustomerService` designed a test set to test the seven operations of its interface. We call this test set `TS-Customer`. We transformed `CustomerService` into a testable service by instrumentation and executed `TS-Customer` in the context of a test session. The coverage analysis obtained with this execution is presented in Table 1. The coverage measure is given for the whole service and for each operation. The coverage for the whole service is given by the formula $TRcov/TR$, where TR is the sum of all test requirements of the service (for all operations) and $TRcov$ is the sum of the test requirements that were covered. We can notice that the coverage is quite high, since the test cases have been created by the developers, who have access to the source code of the service.

Table 1 Structural coverage reached when testing `CustomerService` as a single service

Service	**all-nodes**	**all-edges**	**all-uses**
`CustomerService`	97%	93%	87%
By operation	**all-nodes**	**all-edges**	**all-uses**
createFromId	90%	92%	72%
registerCustomer	100%	100%	100%
saveCustomer	90%	86%	84%
readById	100%	100%	100%
readByUsername	100%	100%	100%
readByEmail	100%	100%	100%
changePassword	100%	100%	80%

The integrator who created `RegisterService` designed a test set to test the two operations of its interface (`TS-Register`). We executed `TS-Register` to check how much of much of `CustomerService` would be exercised from within `RegisterService`. The coverage analysis generated after this test session is presented in Table 2. Notice that the coverage is now relatively low. This probably happened because `TS-Register`

was meant to test `RegisterService` without taking into account how much of the `CustomerService` was being executed in that context.

Table 2 Structural coverage reached when testing `CustomerService` in the context of the orchestration `RegisterService`

Service	**all-nodes**	**all-edges**	**all-uses**
`CustomerService`	50%	34%	34%
By operation	**all-nodes**	**all-edges**	**all-uses**
createFromId	54%	35%	25%
registerCustomer	100%	66%	65%
saveCustomer	54%	33%	33%
readById	0%	0%	0%
readByUsername	100%	100%	100%
readByEmail	0%	0%	0%
changePassword	0%	0%	0%

By using the testable version of `CustomerService`, the integrator can see now that its coverage is low when tested from the perspective of `RegisterService`. This coverage report by itself is a valuable information to the integrator, however, it does not help to analyze the reasons why the coverage is low or, even better, how it could be improved. This issue can be mitigated by the metadata model we designed for testable services, which provides integrators with information to evaluate the quality of the test set of the orchestration and to create more test cases to increase the coverage.

4 The MTxTM Approach: Testable Services Even More Testable

Testable services provide the integrator with detailed feedback about how much a service is exercised during its validation or the validation of the composition in which it is integrated. This advance on service testing is already a great advantage brought by testable services, but there is still room for improvement. As said, the coverage measure provided by the testable service is a feedback about the thoroughness of the executed tests, but integrators do not know whether the coverage achieved is satisfactory or how to improve their test set to reach a better coverage. When the coverage achieved is high, the test set of the service or the orchestration is probably good, but when the coverage is low, it does not necessarily mean that the test set is bad. Indeed, coverage of a service could be low for two different reasons[2†]:

1. **Insufficient test cases:** The most obvious reason for low coverage, especially at the beginning of testing, is that the test set is weak: in devising the test cases, the tester has probably neglected to consider some interesting behaviors. In traditional white box testing, by inspecting the source code and looking at the parts that have not been covered, the tester can usually identify new significant test cases. In service testing, however, it is not possible, even

[2] The coverage could of course be lower than 100% also because of infeasible requirements (unreachable paths). This issue, however, is not peculiar to services, and should be handled by the developers of the service.

considering testable services, because the source code is not available. It requires extra information to help integrators discovering which instructions, paths and/or data were not exercised yet to create new meaningful test cases. It is difficult to create new test cases based only on interfaces or specifications.

2. **Relative coverage:** Different customers may use different operations of a service or even use the same operations in different contexts. A car shop, a real estate agent and a university, for example, may use the same bank service to take loans to their customers. The car shop may use short time duration loans, while the real estate agent uses long time duration loans and student loans can start to be paid after many years. If we consider a test session of the car shop, for example, the coverage measured over the whole bank service code would be low and not realistic, because the other two types of loan would never be used from that context. We call *irrelevant* the test requirements covered by functionalities not used within a specific context.

The first issue can be addressed by adding more test cases. However, the integrators cannot know which are the test requirements of the testable service that were not executed, and why. The second issue arises because the coverage ratio is computed over a set of test requirements that is unrealistic; this issue should be addressed by considering a personalized coverage analysis for each context in which a service is used.

Considering this scenario, we investigated how these issues are addressed in the field of software components, since they share many similarities with services. Both of them are self-contained composition units and can only be accessed through explicit published interfaces[15]. Built-in testing is an approach created to improve the testability of software components. The general idea is to introduce functionalities into the component to provide its users with better control and observation of its internal state[15, 17].

A component developed under the built-in testing concept can also contain test cases or the capability to generate test cases. These test cases can be used by the testable component itself for self-testing, and by external users as well[5].

Another way of improving the testability of service components is using metadata. Metadata provide extra information about the component other than the interface specifications and range from finite-state-machine models and QoS-related information to plain documentation. Bundell and coauthors[7] pointed that metadata should be used to provide information to help component users on analysis and testing activities. Thus, metadata may consist of coverage information, test cases, abstract representations of source code or assertions about security properties[22], for example.

To address the problem of the insufficient test cases, inspired by the concepts of built-in testing and metadata approaches from researches on software components, we propose publishing test metadata along with testable services to help integrators to improve their test set[13]. To address the problem of relative coverage, we propose using an orchestration profile that is used by the testable service to generate coverage analysis tailored for each context in which it is used. The overall solution we propose is the MTxTM approach.

In particular, the test metadata designed for testable services can come in two flavors: *a priori* and on demand. *A priori* metadata provide information that was previously created and is attached to the testable service when it is released. On-demand metadata provide information calculated/generated during runtime, based on the *a priori* metadata. In the next subsections, we show details of both *a priori* and on demand metadata and following of the usage profile.

4.1 A priori metadata

The *a priori* metadata are created by the developer of the testable service and consist of the test set used to test the service during development time (similarly to the built-in test concept). Developers usually test their services before their publication using available testing techniques, strategies and plans. An instance of a typical scenario to create test cases involves the following activities:

1. The developers create test cases for each operation of the service considering its specification. They could use, for instance, functional testing techniques, such as category partition or boundary analysis.

2. The developers execute the test cases and get structural coverage information to evaluate how much their test set has exercised the code of the service under test.

3. The developers inspect the source code or use data flow models to discover which parts of the service were not executed yet and then create test cases to cover the uncovered test requirements (instructions, branches, data, ...).

Integrators of testable services can perfectly perform the steps 1 and 2 above, but they cannot perform step 3 because they do not have access to the source code as the developers do. For this reason, we believe that the developers of testable services should pack the test cases created during development time and export them as the *a priori* metadata of the testable service. The *a priori* metadata should also include the reasoning performed by the developer to design each test case.

In MTxTM we assume that the *a priori* metadata represents the best effort of the developer and that the coverage obtained is the highest possible. In many cases, the structural coverage reached will not be 100% because of infeasible requirements (like unreachable paths, for example).

Considering the example of a testing scenario of a subset of the BroadLeaf Commerce framework presented before, the *a priori* metadata of `CustomerService` is the test set `ts-Customer`. Table 3 presents a subset of `ts-Customer`. These test cases have been designed to test an operation of `CustomerService` called `createFromID`. Notice that for each test case there are the input data and a description that represents the reason why that test case was created.

Table 3 A subset of the *a priori* metadata of `CustomerService`

boolean createFromID (id,user,passwd,email,chAnsw1,chAnsw2,register)								
TC-ID	**ID**	**user**	**pwd**	**email**	**chAnsw1**	**chAnsw2**	**register**	**Description**
tc-01	1001	cst1	c1pwd	c1@blf.com			true	Create and register
tc-02	1002	cst2	c2pwd	c2@blf.com			false	Create and don't register
tc-03	1003	cst3	c3pwd	c3@blf.com	A1-c3		true	Create (answer 1)
tc-04	1004	cst4	c4pwd	c4@blf.com	A1-c4	A2-c4	true	Create (answer 1 and 2)
tc-05	null	cst5	c5pwd	c5@blf.com			true	Create (auto generated ID)
tc-06	1001							Create (existent ID)

Metadata published along with software components or services must have a predefined format. In our approach, we defined an XML structure in which each test case of the *a priori* metadata must be expressed. Each test case must contain the following information: a unique identification (tc-ID); the name of the operation it refers to; the name and the value of each input parameter; dependencies (some test cases should be executed after the execution of other test cases); the expected result given by an oracle; and a quick description in free text explaining why the test case was created. The following listing shows tc-04 in a simplified XML format.

```
<testcase id="tc-04" description="..."
          operation="createFromId" return="boolean">
   <input name="ID">1004</input>
   <input name="user"> cst4 </input>
   <input name="passwd">c4pwd</input>
   <input name="chAnsw1">A1-c4</input>
   <input name="chAnsw2">A2-c4</input>
   <input name="email">c4@blf.com</input>
   <input name="register"> true </input>
   <input name="description"> Create (answer 1 and 2) </input>
   <expected>true</expected>
 </testcase>
```

4.2 On demand metadata

The on demand metadata is automatically generated by the testable service during runtime, based on the *a priori* metadata. They consist of suggestions provided to help integrators to improve the coverage on the testable service reached so far.

Suppose that an integrator tested a service from the context of a composition and achieved low coverage. He/she can request the on demand metadata of the testable service and receive suggestions of test cases that should be executed to cover the test requirements not executed yet. Considering that the testable service is being executed from within a composition, the integrator must analyze the input data of the suggested test cases and identify an integration test for the composition that invokes the testable service with those or similar inputs. This should not be too difficult a task, because integrators own the source code of the composition and thus should be able to create suitable test cases for the composition to repeat the same

test configuration implied by the on demand metadata. There can be situations in which the suggested test cases cannot be repeated and this can be handled by usage profiles (see next subsection).

Developers who write the *a priori* metadata should include enough information in the description of each test case to help integrators understand the reason why that particular test case was created. A good description of the test case may in fact help integrators to create test cases to exercise the integration between composition and testable services. One way to provide a good reason to create a test case is using black box criteria, such as category partition and boundary values, that always refer to the specification of the service. When the test case is created to exercise a particular branch or use of data, the description could explicitly show which kind of data and values should be used as input values. The developer does not have to disclose details of branch conditions or secret business decisions, but needs to specify which public business rules decision the test case refers to, without showing how it was implemented.

The *a priori* and the on demand metadata are not meant to allow integrators to understand the inner structure of each operation of the service. They are meant to help integrators to reach better coverage by using test case suggestions to cover test requirements that possibly were not covered yet. Integrators must understand the business rules of the composition and of the testable service to be able to create integration tests using the test cases suggested as on demand metadata.

In our approach, the order of the test cases suggested as the on demand metadata is not chosen by chance. By applying a common greedy heuristics[26], those test cases that cover more uncovered test requirements come first. In this way, the integrator can try, for example, to use only a few among the first test cases to improve the coverage of the test set, instead of using all the suggested test cases. We use such heuristics to delimit the number of test cases.

A testable service can identify which test cases of the *a priori* metadata should be provided as on demand metadata because for each implemented criterion it keeps track of the test requirements covered by each test case of the *a priori* metadata. As soon as the *a priori* metadata is packed within the testable service, the latter is executed on each test case and a list of which test requirements is covered by which test case is made. Table 4 shows an example of this list considering the test cases for the operation `createFromID` of the *a priori* metadata of `CustomerService`. Note that this list refers to the test cases that cover the test requirements generated by the criterion all-nodes. A similar table is generated for each criterion implemented by the testable service and for which there is a probe to collect coverage data.

Table 4 Nodes of the operation `createFromID` covered by each test case of the *a priori* metadata

Test case	1	2	3	4	5	6	7	8	9	10
tc-01	X			X	X	X			X	X
tc-02	X			X	X				X	X
tc-03	X			X	X	X	X		X	X
tc-04	X			X	X	X	X	X	X	X
tc-05	X	X		X	X	X			X	X
tc-06	X		X							

The generation of the on demand metadata works as follow. The integrator requests the on demand metadata using a test session identifier. The testable service uses the test session identifier to make a list of the test requirements which were not covered during that specific test session. Next, the testable service uses the *a priori* metadata to make a list of test cases that cover the test requirements which were not covered during the test session informed. The testable service then returns this list to the integrator as the on demand metadata.

Suppose, for instance, that a integrator requested the on demand metadata generated by `CustomerService` after a test session in which nodes 2 and 3 of `createFromID` have not been covered. The testable service would use the *a priori* metadata and identify that tc-05 and tc-06 cover nodes 2 and 3 and they would be provided as suggestions to the integrator.

4.3 Incremental usage profile

The developer of a service does not know in advance which orchestrations or choreographies it will be used in, thus when the service is tested in isolation at development time, it is tested without considering any context. Consequently, the *a priori* metadata are generic. The on demand metadata are also generic because the testable service does not know which functionalities of the service would be or would not be used by the integrator that requests the coverage information (see the notion of relative coverage introduced at the beginning of this section).

To overcome this issue, we propose that the integrator who uses the testable service within a composition should create a usage profile that identifies which operations of the testable services are actually used. This profile also expresses which test requirements should be excluded from the coverage measure calculation. This information can thus be used by the testable service to calculate a customized measure for each context and to generate specific on demand metadata based on the operations and the test requirements that should be considered.

The usage profile of a composition must provide the following information to the testable service used:

- An identifier for future profile updates.
- An identification of the orchestration or choreography that is using the testable service.
- A list of the operations of the testable service that will be actually used by this integrator. The testable service will calculate the coverage based on this list instead of considering all operations of the service. The on demand metadata will also be generated based only on the operations actually used and informed in this section of the usage profile.
- A list of irrelevant or contextually infeasible test cases. More precisely, these are the test cases in the *a priori* metadata that would never be executed in some contexts. This can happen if the test case refers to a not used operation or to a situation in which the combination of parameter values cannot be produced in the given composition.

We consider that a testable service may be tested in iterations. The information in the usage profile can be used by the testable service at each iteration to refine the calculation of coverage and revise the on demand metadata, so to suggest additional test cases based only on operations that are really used. Integrators can provide the testable service with updates to the usage profile information at any time.

A practical use of a usage profile is presented in the next section.

5 Exploratory Case Study

In this section, we present an exploratory case study performed as a first assessment of the MTxTM approach. For this purpose, we reused the test scenario of the Broad Leaf Commerce framework, in which there is an orchestration called `RegisterService` that uses another service called `CustomerService` (as illustrated in Fig. 2). `RegisterService` provides two public operations, while `CustomerService` provides seven ones. Test cases are available to test both `RegisterService` and `CustomerService`. This example is small, but genuine, since the services and test sets were reused from a real application.

In our exploratory case study, one of the authors played the role of the integrator that wants to test and evaluate the integration between the composition and `CustomerService`. The single service was instrumented and transformed into a testable service, thus the integrator can get a coverage report about how much of `CustomerService`'s code was exercised after testing `RegisterService`. The execution of the test cases of `CustomerService` achieved a high coverage (see Table 1), while the execution of the test cases of the orchestration (`RegisterService`) obtained a lower coverage regarding the `CustomerService` structure (see Table 2).

In such a context, the integrator can exploit the *a priori* and/or the on demand metadata provided by the testable service to improve the test set of the composition in order to increase the coverage achieved on the single service. The integrator can also create a usage profile for the composition to get a customized coverage analysis. The improvement of the composition test set is done in iterations. At each successive iteration the integrator can augment the test set of `RegisterService` (TS-Register) by using the on demand test metadata provided by `CustomerService`. Precisely, at each iteration the integrator must perform the following activities (as shown in Fig.3):

1. Get the list of suggested test cases provided by the testable service as on demand metadata.

2. Use the test cases suggested by the testable service to create new test cases (if possible) to augment the test set of `RegisterService` (orchestration) and to get a better coverage on `CustomerService`. The integrator cannot directly use the suggested test cases, but must adapt the input information to create new test cases of the orchestration, which has different operations with different input parameters. This activity cannot be easily automated but it should be easy to create test cases manually since the integrator has full control over the orchestration and knows how each input parameter is handled by the orchestration to invoke `CustomerService`.

3. Create or update the usage profile of the orchestration by defining the list of

operations of the testable service that are actually used by the orchestration; and by identifying possible irrelevant test requirements, which are indirectly identified (as the integrator does not have direct visibility of testable service implementation), by those test cases among the suggested ones that cannot be executed in the context of the orchestration.

4. Execute the new test cases created to augment the test set TS-Register.
5. Get the new structural coverage analysis of `CustomerService`.

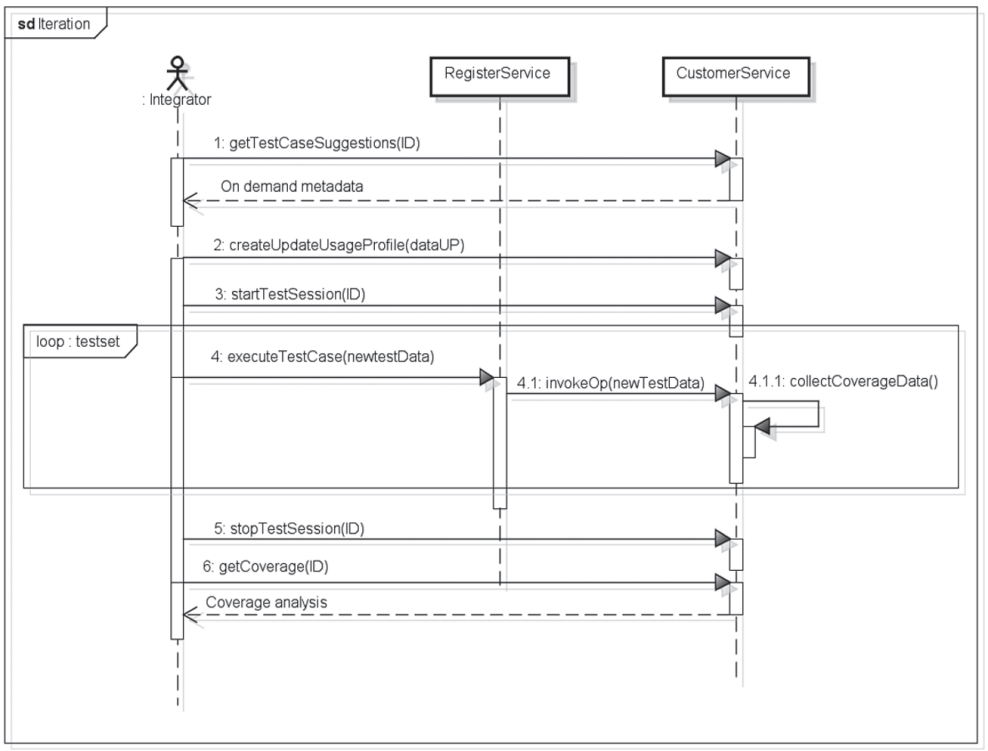

Figure 3. Activities performed by the integrator at each iteration

This process is repeated until the integrator is satisfied with the coverage reached or no more useful test cases are suggested.

We proposed the following research questions to guide our exploratory study:

- **RQ1:** Are the test metadata useful for helping integrators improve the coverage percentage of the test set of a composition? In particular, we will assess:
  - *RQ1-a* whether by using the on demand metadata of the testable service the integrator can create more orchestration test cases to improve the coverage of the testable service;
  - *RQ1-b* whether the integrator can identify irrelevant test cases from the test metadata; and
  - *RQ1-c* whether the usage profile is useful to generate a more realistic (customized) coverage analysis report.

– **RQ2** Is the MTxTM approach more effective than a random test generation approach to improve the coverage reached? In particular, we will assess:
  – *RQ2-a* whether by using the same number of additional test cases, the coverage reached using MTxTM is greater than the coverage achieved using random test cases; and
  – *RQ2-b* whether, when the coverage reached by MTxTM is the same or higher than the coverage reached by random test cases, the number of MTxTM test cases is lower than the number of random test cases.

5.1 First iteration

After executing the test set of `RegisterService` and getting the coverage presented in Table 2, the integrator invoked the testable service and asked for test case suggestions (on demand metadata). The list of test cases suggested by the testable service is the following: tc-04, tc-03, tc-01, tc-02, tc-06, tc-08, tc-16, tc-05, tc-15, tc-09, tc-11, tc-12, tc-13. The detail of each test case is presented in Table 3 (Col. 5, 6 and 7).

The integrator analyzed the input data of tc-04, tc-03 and tc-01 and created new test cases for `RegisterService` (these are the first ones in the list, we recall that the test cases are ordered according to how many more uncovered test requirements they cover). In this case study, we arbitrarily set to three the number of added test cases at each iteration. Next, the integrator identified the operations of `CustomerService` used by `RegisterService` and created the *usage profile* of the orchestration. The integrator also identified the test cases that would never be executed in the context of `RegisterService` and set the section "irrelevant". Table 5 shows the usage profile of `RegisterService` created in this iteration.

Table 5 Differences among three dimensions models

Name	`RegisterService`
ID	OPF-RS-001
Operations	createCustomerFromId
	registerCustomer
	saveCustomer
	readById
	readByUsername
Irrelevant test cases	tc-11,tc-13, tc-15,tc-16

The integrator executed the augmented test set of `RegisterService` (now with seven test cases –the four original test cases plus the three new test cases) after submitting the usage profile to `CustomerService`. Table 6 shows the coverage reached after the execution of this test set. Notice that the coverage values have increased and the operations that are not used by `RegisterService` does not appear anymore in the coverage analysis. The test requirements exclusively covered by the test cases identified in the "irrelevant" section of the usage profile are not considered in computing the coverage ratio. In italic we show the coverage measures of `CustomerService` which would be obtained without considering the usage profile.

Table 6 Structural coverage analysis of `CustomerService` in the first iteration

Service	all-nodes	all-edges	all-uses
`CustomerService`	84%	69%	67%
Without usage profile	*67%*	*58%*	*56%*
By operation	**all-nodes**	**all-edges**	**all-uses**
createFromId	81%	71%	52%
registerCustomer	100%	100%	100%
saveCustomer	72%	53%	51%
readById	66%	50%	44%
readByUsername	100%	100%	100%

5.2 Second iteration

The integrator asked again for test cases suggestions and the testable service generated this list of test cases as on demand metadata: tc-03, tc-04, tc-06, tc-01, tc-05, tc-09, tc-02. The curious thing about this list is that the test cases already used in the second iteration (namely, tc-01, tc-03 and tc-04) appear again. This happens because `RegisterService` does not invoke `CustomerService` using all the input data used in the test case because the data is processed before invoking `CustomerService`. This fact shows that despite using the same service, orchestrations handle data in different ways and require different functionalities from a single operation. It is not always possible to recreate the same conditions in which the `CustomerService` was tested as a single service.

The integrator decided not to consider the already used test cases (tc-04, tc-03 and tc-01) suggested by `CustomerService`. New test cases for `RegisterService` were created based on the test cases tc-06, tc-05 and tc-09. There was no need to update the usage profile of `RegisterService` in this iteration.

The test set of `RegisterService` now contains ten test cases and obtained the coverage measures presented in Table 7 when it was executed. Note that the coverage achieved is further increased. The coverage obtained for the operation `saveCustomer` is the same as the coverage obtained in the first iteration. This happened because the testable service is not invoked with the same inputs as suggested by the test cases of the on demand metadata, even when new test cases are created for the orchestration using the suggestions of the testable service. The reason of this situation is that the orchestration transforms the input data received from the test cases execution and somehow invokes the testable service with a different combination of input data.

Table 7 Structural coverage analysis of `CustomerService` in the second iteration

Service	all-nodes	all-edges	all-uses
`CustomerService`	90%	79%	78%
Without usage profile	*72%*	*67%*	*65%*
By operation	**all-nodes**	**all-edges**	**all-uses**
createFromId	90%	92%	72%
registerCustomer	100%	100%	100%
saveCustomer	72%	53%	51%
readById	100%	100%	100%
readByUsername	100%	100%	100%

5.3 Third iteration

The on demand metadata obtained by the integrator this time were the following: tc-03, tc-04, tc-01, tc-02. Of these, tc-04, tc-03 and tc-01 have been already used and the only difference between the test cases tc-01 and tc-02 is the value of the input parameter "register", which is not taken as input parameter by any operation of `RegisterService`. Also, the coverage reached by the test cases of the composition was already high. Hence the integrator decides to stop test set augmentation and not to proceed with further iterations.

5.4 Improving coverage using random test cases

The case study presented above showed a complex process to improve the coverage of `CustomerService` when invoked in the context of `RegisterService`. A natural question arises whether the required effort is worth, or instead the integrator could anyway increase the coverage easily by just continuing to test. Hence, we performed a comparison with additional random test cases as a baseline. We used an application provided by the web site GENERATE DATA[3‡] to generate the random test cases. We set the number and the type of the parameters of the operations of `RegisterService` and generated 120 test cases using random data. The number and the combination of the input parameters of each random test case was also selected randomly.

In the first iteration of this study, we created 20 test sets using the test set `TS-Register` augmented with six new random test cases. We decided to use six new test cases in this study because it is the number of the new test cases created using the MTxTM approach. We executed these 20 test sets and calculated the average of the coverage reached for each operation and for the whole service. In the second iteration we executed the original test cases of `RegisterService` plus 50 new random test cases, and in the third iteration we executed the original test set of `RegisterService` plus 100 new random test cases. Rows 4 to 6 of Table 8 show the coverage analysis obtained by each iteration of this case study using random test cases.

Table 8 Summary of the results of the case study

Approach	Iter.	#NEW-TC	all-nodes	all-edges	all-uses
Initial Context	-	-	50%	34%	34%
Random	1st	6	74%	56%	56%
Random	2nd	50	84%	68%	64%
Random	3rd	100	84%	75%	71%
MTxTM	1st	3	84%	69%	67%
MTxTM	2nd	6	90%	79%	78%

5.5 Threats to validity

To ensure construction validity, we took the case study from a real environment and the orchestration within which the testable service has been tested is an application that is used in practice. The way this case study was performed also

[3] `http://www.generatedata.com/`

represents as closely as possible a typical scenario of a testing activity in practice. Therefore, we do not see major threats to construct validity, i.e., the case study represents the intended concept of the MTxTM approach.

A major confounding factor of the internal validity of the case study is that its subjects were the same proposers of the approach under evaluation. To minimize this confounding factor we only used the test data provided on the BroadLeafCommerce web site, we cannot however exclude its impact. The study presented in Section 6 is a small experiment using independent subjects.

The external validity of this study case cannot be assured. The services of the case study are real, but just one case study may not be representative and we cannot generalize the results of this specific study for all situations or for all applications of the domain. Further evaluation is required to generalize the results obtained with this case study.

5.6 Answering the research questions

Considering the case study performed, we attempt to provide preliminary answers to both RQ1 and RQ2. The integrator was able to analyze the on demand metadata provided by `CustomerService` and augment the test set `TS-Register` with new test cases. In fact, six new test cases have been created and this raised the coverage of `CustomerService` (RQ1-a). The integrator was also able to identify the irrelevant requirements of the on demand metadata and create an usage profile for the orchestration (RQ1-b). The testable service used the usage profile of `RegisterService` and produced a coverage analysis specific for that profile, disregarding the operations not used and the "irrelevant" test cases (RQ1-c).

Concerning RQ2, Table shows, for each approach, the number of new test cases created for `RegisterService` and the coverage reached on `CustomerService` for each iteration. Note that the coverage reached by the test cases created by MTxTM is higher than the coverage achieved by the random approach (RQ2-a). MTxTM leads to the creation of new six test cases to reach a coverage measure that is higher than the coverage achieved by the random approach after creating new 100 test cases (RQ2-b).

RQ2-b is related to the effort to raise the coverage of `TS-Register`. Using MTxTM the integrator cannot use the on demand metadata as they are. The integrator needs to study the input values and adapt them to create test cases suitable to the operations of the orchestration. This requires human interaction and takes more time than using a random approach, for example, which can generate and execute 100 test cases in a very short time. If we look at Table , the coverage reached by the random approach after creating and executing 100 new test cases is not so far from the coverage reached by MTxTM after creating 6 new test cases. The time to create 6 new test cases was longer than the time to create 100 new random test cases. In SOA testing, however, the lesser the number of test cases the better.

Indeed, when testing a third-party service on line, "superfluous requests to Web Services may bring heavy burden to the network, software, and hardware of service providers, and even disturb service users' normal requests"[25]. Besides, "if the service provider allows massive vicious requests to a Web Service within a short time, the requests may congest the network or even crash the service's server"[25] and cause

a denial-of-service phenomena[12]. In fact, there are services that define the upper limit of the number of requests that can be performed by a client. If the number of invocation exceeds the limit, the extra requests are ignored[25]. It is also important to keep the number of test cases low if the services under test are charged on a per-use basis[12].

6 An Experiment to Evaluate MTxTM

After the first assessment of MTxTM, we also performed a formal evaluation of the MTxTM approach. We report here an experiment designed and executed to measure how effective MTxTM is on supporting integrators to reach high coverage on a single service when testing its integration with a composition. For the purpose of this investigation, we compared MTxTM with a Functional approach.

The experiment was performed from the perspective of integrators that had to create test cases for a composition that uses a testable service. The main purpose of the integrator was to test the integration between the composition and the testable service to exercise the structure of the testable service as much as possible. The second purpose was to find failures on the testable service.

During the test activity, subjects using the Functional approach could use only the results of the test cases and functional testing criteria to decide whether the test cases created were enough or should be improved. Subjects of the MTxTM approach, on the other hand, could make this decision using the functional results of the test cases, functional testing criteria, structural coverage analysis and on demand metadata provided by the testable service.

The designed experiment has four independent and two dependent variables. The independent variables are the following: MTxTM; the Functional approach; the experimental objects and the subjects experience on software testing. The dependent variables are the following: the structural coverage obtained for the testable service under test and the number of failures found.

6.1 Hypotheses

Two null and two alternative hypotheses were defined for the experiment, considering the two dependent variables: the coverage achieved and the number of failures. The null (H0) and the alternative (H1) hypotheses are the following:

- **H0$_1$:** the structural coverage obtained by the MTxTM approach is less or equal to the coverage obtained by the functional approach, i.e, $\text{COV}_{MTxTM} <= \text{COV}_{FT}$.
- **H1$_1$:** the structural coverage obtained by the MTxTM approach is greater than the coverage obtained by the functional approach, i.e, $\text{COV}_{MTxTM} > \text{COV}_{FT}$.
- **H0$_2$:** the number of failures found by the MTxTM approach is less or equal to the number of failures found by the functional approach, i.e, $\text{FAIL}_{MTxTM} <= \text{FAIL}_{FT}$.
- **H1$_2$:** the number of failures found by the MTxTM approach is greater than the number of failures found by the functional approach, i.e, $\text{FAIL}_{MTxTM} > \text{FAIL}_{FT}$.

6.2 Subjects and experimental objects

The subjects of the experiment were 12 graduate students of the Software Engineering Laboratory of the University of Sao Paulo. The subjects were selected by convenience, since most of them were attending an Experimental Software Engineering course and had already taken a software testing course.

Two experimental objects were used in this experiment: `WSGolfReservation` and `GolfChampionship`. The first experimental object is a real world service reused from an open source web-based application[4§] designed for golf clubs to manage reservations. This real service has 10 public operations and 571 LOC. The latter experimental object is an application developed by one of the co-authors to manage players and matches of golf championships. It uses `WSGolfReservation` to make reservations of golf courses to allocate matches of the championship. This application has 9 operations and 230 LOC.

`WSGolfReservation` was transformed in a testable service using the BISTWS approach[14]. A test set was created to test each operation of `WSGolfReservation`. This test set reached 100% of coverage for the all-nodes, all-edges and all-uses criteria for all operations and it was used as the *a priori* metadata of `WSGolfReservation`.

Subjects of MTxTM had access to the interface of the testable service to get the coverage analysis and the on demand metadata, while subjects of the Functional approach could only access the interface of the composition `GolfChampionship`. In this way, the coverage reached by the subjects using the Functional approach was calculated but was not available to them.

6.3 Preparation

The level of experience on software testing of each subject was measured using an ordinal scale: 1-high, 2-medium and 3-low. Low level means that the subject knows software testing from six months to one year and have never worked with software testing. Medium level means that the subject knows software testing from one to three years and have already applied software testing in at least one project. High level means that the subject knows software testing from one year or more and works with software testing in the industry or as research theme.

Coincidentally, the number of subjects for each level was even and we split them into 6 pairs of equal experience. For each pair of subjects, we randomly selected which one would use the MTxTM approach while the other, automatically, would use the Functional approach. The subjects were then split in two groups with balanced experience. Subjects using MTxTM were identified by even numbers while subjects using the Functional approach were identified by odd numbers. Table 9 shows the distribution of subjects and approaches. Since all subjects tested the same experimental objects (`WSGolfReservation` and `GolfChampionship`), Table 9 is also the design of the experiment.

[4] `http://sourceforge.net/projects/golf-reserve/`

Table 9 Design of the experiment

MTxTM		Functional	
Subjects	**Experience**	**Subjects**	**Experience**
Subject 2	High	Subject 3	High
Subject 12	High	Subject 9	High
Subject 4	Medium	Subject 1	Medium
Subject 6	Medium	Subject 5	Medium
Subject 10	Medium	Subject 7	Medium
Subject 8	Low	Subject 11	Low

6.4 Operation

Before executing the activities of the experiment, the subjects received one hour of training according to their experience and the approach used. Each subject has also received a detailed description of the two experimental objects and a guideline to execute each task required by the experiment.

Figure 4 shows a sequence diagram representing the testing activities performed by subjects (integrators) using the Functional approach. The integrator executes test cases and the only information available about the test session is the result of each test case. The integrator then has to decide to create more test cases to the composition based on functional testing criteria. Note that the subject has only access to the interface of the composition. The subject may also invoke the testable service's operations, but may not use the testing interface to get coverage information and/or test metadata.

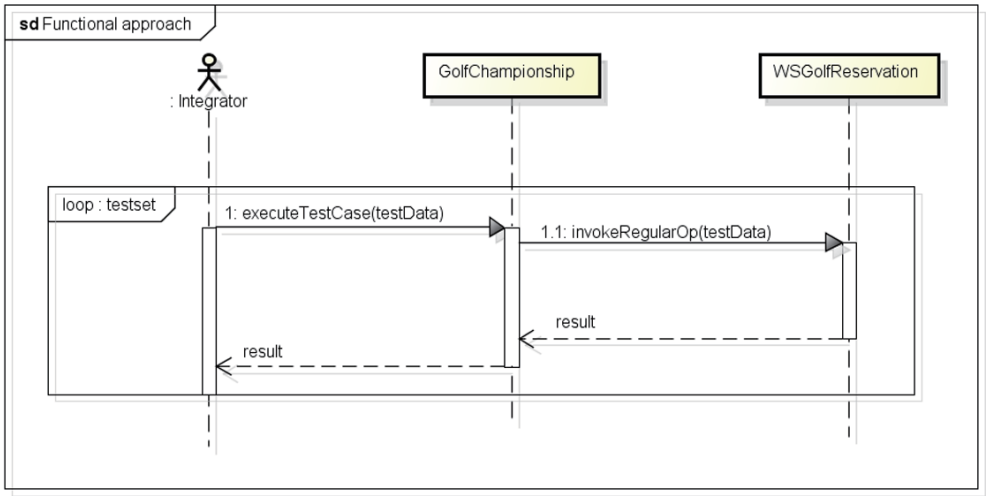

Figure 4. Sequence Diagram of the testing activities performed by subjects using the Functional approach

Figure 5 shows a sequence diagram that represents the testing activities performed by subjects (integrators) using MTxTM. The integrator invokes the testable service to start a test session, executes a set of test cases and then invokes

the testable service again to communicate that the test session is over. The integrator then creates or updates the usage profile of the application `GolfChampionship` to inform `WSGolfReservation` on which operations are actually used and which among the test cases suggested as the on demand metadata (if it is the case) are irrelevant. Next, the integrator gets a coverage analysis to evaluate how much of the testable service was executed regarding instructions, paths and data. The integrator can also get the on demand metadata (test case suggestions) to create more meaningful test cases to the composition and then achieve higher coverage.

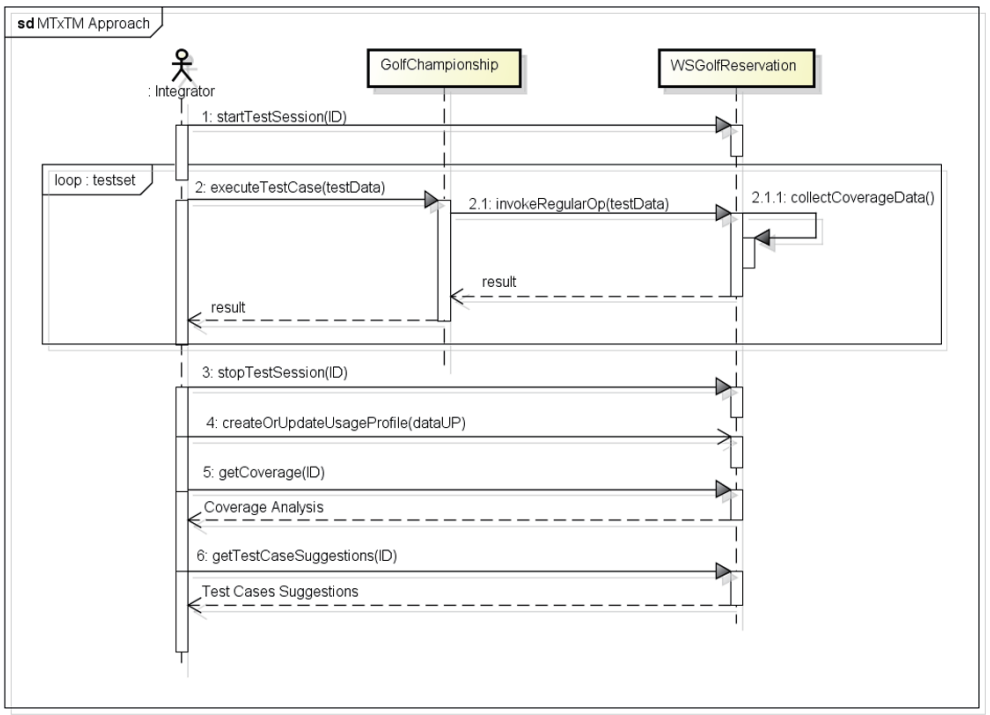

Figure 5. Sequence Diagram of the testing activities performed by subjects using MTxTM

The subjects performed the activities of the experiment in pairs during two weeks in a controlled environment. They had from 1 to 4 hours to execute their tasks and all of them followed the schedule, the recommendations and the rules of the experiment. Subjects using MTxTM were recommended to stop testing when no more test cases are suggested by the testable service or when the test cases suggested were not meaningful anymore, i.e., by using them the composition could not invoke the testable service.

After the experiment, the subjects filled in a form reporting the (i) test cases created; (ii) the number of failures found; (iii) the structural coverage analysis (only subjects using MTxTM); (iv) the difficulty to use the test metadata (only subjects using MTxTM); and (v) any relevant observations. The coverage reached by subjects using the Functional approach was not available to them, but it was anyhow collected and calculated by us to compare it with the coverage reached by subjects of MTxTM.

6.5 Results

The forms filled in by the subjects were validated and summarized. The results are shown in Table 10. The coverage presented is the average of the coverage percentage of these three criteria: all-nodes, all-edges and all-uses. The number of failures is the number of different failures detected by the test cases.

Table 10 Results of the experiment

		MTxTM			Functional	
Experience	**Subject**	**Coverage**	**Failures**	**Subject**	**Coverage**	**Failures**
High	2	85%	3	3	37%	3
High	12	83%	7	9	72%	2
Medium	4	82%	3	1	74%	5
Medium	6	85%	5	5	75%	2
Medium	10	85%	6	7	81%	5
Low	8	85%	3	11	56%	2

6.6 Threats to validity

Three threats to the internal validity of our experiment were found: the experience of the students, the experimental objects and the productivity under evaluation. The experience of the students may impact the results, but we tried to mitigate this threat by providing a specific training for each level of experience and also by splitting the subjects into two balanced groups. As the results of the experiment show, the experience was not a determinant factor in this experiment.

There were two experimental objects used in this experiment. One of them was developed by a co-author of the MTxTM approach. The developer could have, even unconsciously, created an application to give advantage to the metadata approach. However, after further investigation of the results of the experiment, we realized that the application prevents many configurations of data to be sent to the testable approach. This indeed hampers the MTxTM approach to be much more efficient than the Functional approach on reaching a higher coverage. If it were not the case, probably almost all subjects would have obtained up to 100% coverage for all criteria.

Students are more productive when they are under evaluation and this could affect the results of the experiment. In this case, although most of them were taking the Experimental Software Engineering course, the experiment was not executed in that context. The students were not forced to join the experiment, but most of them did because they wanted to have an experience of participating in an experiment and also to collaborate with an ongoing research of the Software Engineering group.

There are two threats to the external validity of the experiment. The population of subjects is not statistically representative since it is a homogeneous group of graduate students, and not professionals. Also, the experiment was not conducted in a real environment. Therefore, the conclusions of this experiment cannot be generalized for a professional environment. However, even in a limited environment, the results obtained with the experiment is an evidence that using testable services and test metadata helps integrators improving their test set to reach high coverage.

6.7 Data analysis

A descriptive analysis of the results of the experiment considering the coverage percentage and the number of failures was performed, which are related to the first and second hypothesis, respectively. Figure 6 shows box plots of the experiment results.

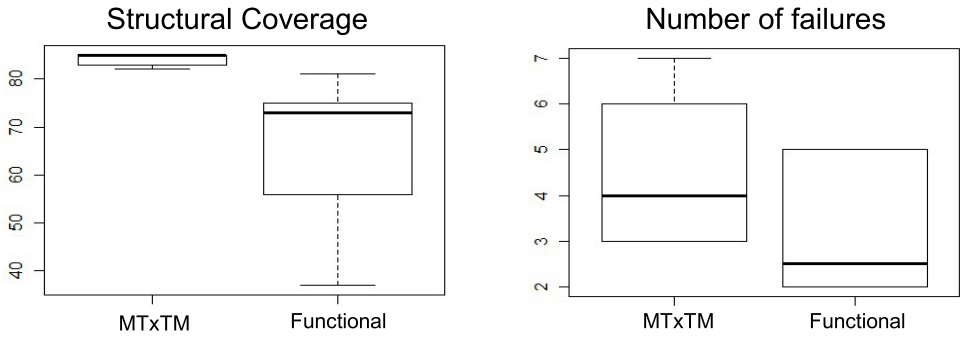

Figure 6. Box plots of the experiment results

Regarding the coverage percentage, MTxTM seems to reach a much better coverage than the Functional approach. Note that the highest coverage reached by a subject using the Functional approach is less than the lowest coverage obtained by a subject using MTxTM. Regarding the number of failures found, MTxTM seems to be slightly better than the Functional approach.

An analysis of the results was performed considering the experience of the subjects disregarding the approach used. This analysis is important to check whether the experience of the subjects have influenced the result. Figure 7 shows the box plots of the results for each level of experience considering the structural coverage. It seems that the experience has not influenced the results since there are not great differences among the results for each level of experience. If the experience

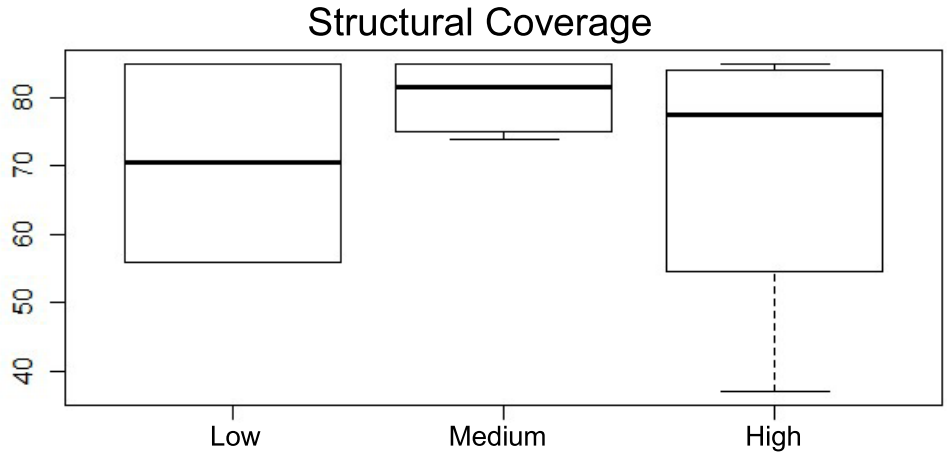

Figure 7. Box plots of the structural coverage considering the level of experience

were a determinant factor to affect the results, the subjects with high experience would have obtained the highest coverage while the subjects with low experience would have reached the lowest coverage, but this was not the case. On the contrary, one of the subjects with high experience reached the lowest coverage: 36%. Most of the subjects of the three level of experience obtained more than 56% coverage.

Figure 8 shows the box plots of the results for each level of experience considering the number of failures found. The experience of the subjects seems to have some influence on the results in this case. The number of failures found by subjects with low experience is less than the number of failures found by subjects with medium or high experience. We cannot assure that the experience is the only determinant factor because there were only two subjects with low experience. Note that there are subjects with medium and high experience that have found only two failures, which was the lowest number of failures found in this experiment. But, in general, the subjects with medium and high experience seem to have found more failures than the subjects with low experience. The subjects with medium experience seem to have found the same number of failures as the subjects with high experience.

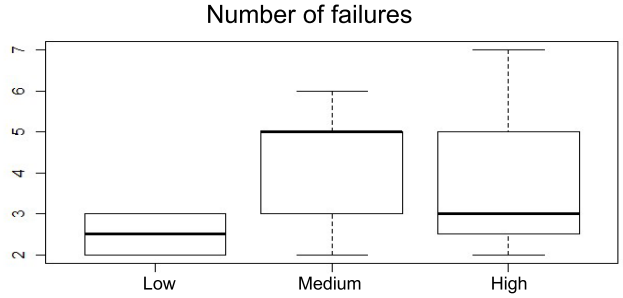

Figure 8. Box plots of the number of failures found considering each level of experience

6.8 Hypothesis testing

The unpaired T test was used for testing the hypotheses of this experiment according to the recommendation of Wohlin et al.[29] given the design of the experiment. There are six samples generated by the MTxTM approach (m) and six samples produced by the Functional approach (n). According to the t-value distribution, $\mathtt{t}_{10}$ (m+n-2) is 1.812 considering a 95% confidence interval and the one sided test. The value of $\mathtt{t}_{10}$ is also used to validate our hypotheses.

The first hypothesis is related to the coverage reached by each approach. The T test was performed to compare the samples of the MTxTM and the Functional approach and the following results were obtained. `P-value` is 0,02043 and `t` is 2.7273. These results allow us to refute the null hypothesis $H0_1$ and accept the alternative hypothesis $H1_1$ since $\mathtt{p\text{-}value} < 0.05$ and $\mathtt{t} > \mathtt{t}_{10}$. Actually, the $H0_1$ can also be refuted using a 97.5% confidence interval.

The second hypothesis is related to the number of failures obtained by each approach. The T test was performed to compare the samples of the MTxTM and the Functional approach and the following results were obtained. `P-value` is 0,09303 and `t` is 1,4231. These results do not allow us to refute the null hypothesis $H0_2$ and

accept the alternative hypothesis $H1_2$ since $\texttt{p-value} > 0.05$ and $\texttt{t} < \texttt{t}_{10}$. $H0_2$ can be refuted and $H1_2$ accepted using a 90% confidence interval because in this case the $\texttt{t}_{10}$ is 1.372.

6.9 Data analysis based on the experience of the subjects

The results were also analyzed considering the experience and the approach separately. Figures 9 and 10 show box plots created for the coverage reached by the MTxTM and the Functional approach, respectively, according to the level of experience of the subjects. Note that the experience has not influenced the results. It seems that the MTxTM helped the subjects with low experience to reach the same coverage reached by the subjects with high experience.

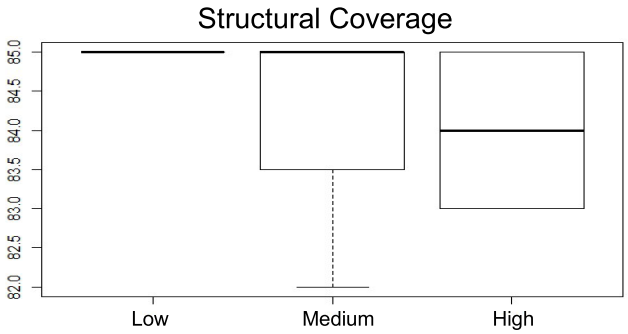

Figure 9. Box plots of the coverage reached by MTxTM considering each level of experience

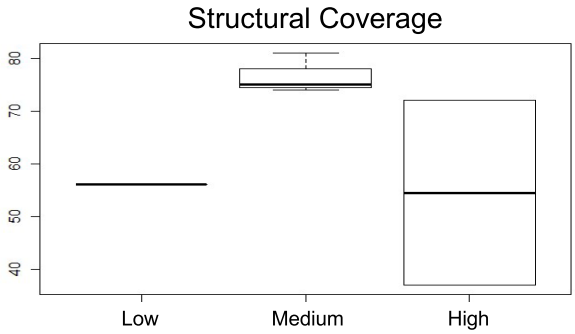

Figure 10. Box plots of the coverage reached by the Functional approach considering each level of experience

Figures 11 and 12 show the box plots created for the number of failures found by MTxTM and the Functional approach, respectively, according to the level of experience of the subjects. In this case, the experience of the subjects seems to have influenced the results. The number of failures found by the subjects with high experience seems to be greater than the number of failures found by the subjects with medium and low experience. Equally, the number of failures found by the subjects with medium experience seems to be greater than the number of failures found by the subjects with low experience.

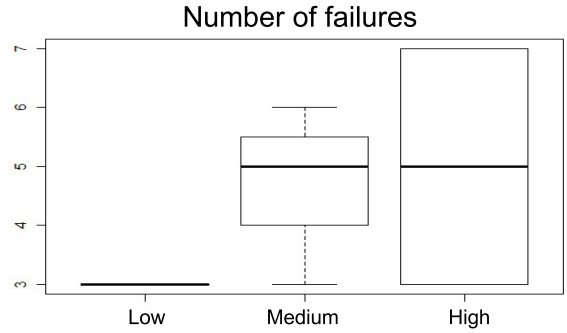

Figure 11. Box plots of the number of failures found by MTxTM considering each level of experience

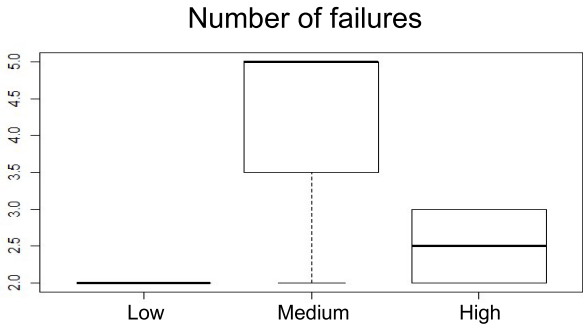

Figure 12. Box plots of the number of failures found by the Functional approach considering each level of experience

In the functional approach results, on the other hand, the experience of each subject does not seem to have influenced the results. The number of failures found by the subjects with medium experience seems to be greater than the number of failures found by the subjects with high experience. Further investigation was not performed to discover why this happened.

6.10 Failures analysis

An analysis of the failures found by integrators using both approaches was performed. Integrators using the functional approach found 7 distinct failures while integrators using the MTxTM approach found 11 distinct failures, which included all the 7 failures found by the functional approach. The list of the 7 common failures found by both approaches are the following:

1. The application allows creating two players with the same identifier.
2. The application allows creating matches using invalid dates (month 15)
3. The application allows scheduling matches for passed dates.
4. The application allows a manager of a championship to cancel a match of other championships.

5. The application allows finishing a match that has not been started.
6. The application allows starting a match that has been already started.
7. The application allows winning a player that has not played the match.

Integrators using MTxTM approach found the 7 failures above and these 4 other failures:

8. The application allows creating a match with only one player.
9. The application allows finishing a match already finished.
10. The application allows starting a match that have been cancelled.
11. The application allows starting, cancelling or finishing a match without checking whether the input data *user* is a registered user.

Most of the failures were found by test cases created based on specifications. Some of the failures, however, were found only by those integrators that used the MTxTM approach. Specifically, failures numbered from 8 to 11 were found after the execution of test cases that were based on the on demand metadata provided by the testable version of WSGolfReservation. The suggestions used to create those test cases were the test cases designed by the developer of WSGolfReservation to cover some specific branches and nodes of the code. As the integrators using only the functional results had no access to the structural coverage analysis, they did not realize that there were parts of the code that had not been executed. Integrators using MTxTM, on the other hand, had access to the coverage analysis and used the test cases suggestions (on demand metadata) to raise the coverage and ended up finding four more distinct failures.

7 Discussion

The MTxTM approach was firstly evaluated in an exploratory case study and then in a formal experiment. The exploratory case study has shown that the integrator was able to use the metadata information to create more test cases to test the orchestration and consequently improve the coverage reached on the testable service. The experiment described in this paper was intended to measure how effective the MTxTM approach is on supporting integrators to reach better coverage and finding failures of testable services.

The main purpose of software testing is to reveal failures of the software under test. The MTxTM approach is not a technique specifically designed to maximize the number of failures found, it is conceived to help integrators to create test cases to cover the parts of the code that were not exercised yet. As for any coverage-based approach, our assumption is that, consequently, the approach can help reveal latent failures hidden behind untested code. In the experiment, we found that MTxTM was not very useful to increase the number of failures found in comparison with the Functional approach, at least in this particular context. However, although the number of failures found by both approaches are not so different, the number of distinct failures found by integrators using MTxTM was greater than the number of

distinct failures found by integrators using only functional testing. This happened because integrators using MTxTM created test cases aiming at improve the coverage achieved so far driven by the test metadata. This is also a confirmation that functional and structural testing are complementary and should be used together even in service oriented contexts where the source code is not available.

Two related questions arise. The first one is whether it is realistic to expect service developers to create these specific metadata since this requires additional effort. The second one is whether integrators will really want to create test cases to meet testing criteria other than those based on specification and interface, as category partition and boundary values, for example.

Regarding the first question, we believe that for developers committed to create services with good quality the test scaffolding activities we propose in this paper would be straightforward. Further, if it were empirically demonstrated that provision of testable services with metadata could enhance the quality of orchestrations or choreographies, developers might be motivated to provide such metadata as an optional value-added feature[22] for which they could charge a fee, thus enhancing the value of their services.

Concerning the second question, we believe that there may be integrators who want to use only black box testing and others that want to use both black box and white box testing. A regular service is suitable for the first type of integrators, but not for the former. A testable service, on the other hand, is suitable for both types of integrators, since their structural testing facilities are only activated when operations to start and to stop a test session are invoked.

The probes inserted by instrumentation can bring overhead to the testable service, although the probes are only activated during a test session. We have done a performance analysis to measure the overhead inserted by instrumentation in a SOA architecture using testable services.

We executed a test set several times against a regular service and measured the average response time. Next, we executed the testable version of this service against the same test set. Regarding the testable version, we measured the average response time of the test set execution inside and outside a test session.

As we show in Table 11, the overhead of average response time of the testable service outside a test session is 2.65% and inside a test session is 5.26%. In general, the response time of a testable service was greater than the response time of a regular service. However, there were specific executions in which the response time of the testable version execution was faster than the response time of the regular version. This happened because in some executions the overhead due to the network was greater than the overhead brought by the testing code.

Table 11 Analysis of the overhead brought by the instrumentation of a testable service

Version	Average response time	Overhead
Non-testable	2070	0%
Testable	2125	2.65%
Testable (during a test session)	2179	5.26%

More in general, we believe that the test metadata model proposed for testable

services could change not only the way developers and integrators technically interact when testing compositions, but also change the business model behind SOA orchestrations and choreographies. The structural testing capability of testable service can already bring competition advantage to the market; we believe that the enhanced testability of "more testable" services can aggregate even more value to market. Presenting a business model in depth is outside the scope of this paper. In brief, we foresee a business scenario in which many versions of a service would be made available with varying costs depending on the testability features it provides:

- A regular service providing only its interface to its clients.
- A testable service providing its interface and structural testing capability.
- A testable service with *a priori* test metadata.
- A testable service with *a priori* and on demand test metadata.

8 Related Work

Service testing is actively researched, as can be seen in some recent surveys[6, 11, 12]. We focus here on testing of compositions of services that might have been developed by independent organizations. The issues encountered in testing a composition of services are surveyed by Bucchiarone et al.[9].

Most 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"[12]. The need to enhance black-box testing with coverage information of tested services has also been recognized by Li et al.[19]. A "grey-box testing" approach is introduced, in which, after test execution, the produced test traces are collected and analyzed by the so-called BPELTester tool. However, the assumption of BPELTester that the orchestrator can access and analyze service execution traces breaks the loose coupling between service provider and service user.

The idea of leveraging service execution traces is also pursued by Benharref et al.[4]. Similarly to Ref. [3] and Ref. [14], 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 structural coverage, in ESOA services are monitored (passive testing) against a state model.

This paper is aimed at exploiting the coverage measures obtained for testable services for improving the test set. This is inspired by metadata and built-in testing. Briand et al.[8] presented an approach in which the developer must provide metadata with constraints called CSPE (Constraints on Succeeding and Preceding Events). The metadata is then used by the tester to generate test cases to cover each constraint of the CSPE. In this approach the tester uses the constraints to generate the test set and the coverage is given according to the constraints defined by the developer. In our approach the metadata is provided to help testers improve the coverage of the testable service even when it is tested in the context of an orchestration. The test

cases suggested are presented with real input values while constraints generally refer to generic situations.

9 Concluding Remarks

Testing services and service compositions has been described as a challenging task, specially when integrators want to use testing techniques beyond interface and/or specification based[6, 11, 12]. Services are black box by nature and their inherent encapsulation hampers applying implementation based testing techniques. Testable services have been proposed as a solution to address this issue.

Testable services provide their customers with testing interfaces and metadata. The testing interface is used to invoke operations to start and to stop a test session and then get structural coverage analysis regarding control and data flow criteria.

This paper presented a metadata model proposed to make testable services even more testable by providing their customers with both *a priori* and on demand metadata. The *a priori* metadata is the test set used by the developer to test the testable services and the on demand metadata are test cases suggested to improve the coverage measure reached so far. *A priori* and on demand metadata can be used to evaluate the coverage achieved and also to create new meaningful test cases to improve the coverage obtained. Customers of testable services can also exploit usage profiles to get coverage analysis specialized to the context in which the testable service is used. The proposed solution addresses the main reasons of low testability, namely lack of control and observability; as well as the reasons why a low coverage is obtained when testing a service: insufficient test cases and relative coverage.

This paper also presented an exploratory case study performed as a first assessment of the MTxTM approach and a formal experiment designed to compare MTxTM with a Functional approach. The results of the exploratory case study have shown that MTxTM could help integrators to create new meaningful test cases to the orchestration using a testable service and then to increase the coverage measure that was low at the beginning. The results of the formal experiment have shown that subjects using the MTxTM approach were able to obtain higher coverage than subjects using only a Functional approach.

As future work, we intend to perform further experimentation using real subjects and experimental objects. In this future experiment, we intend to seed failures in the experimental objects to better observe the impact of MTxTM on the number of failures found by the integrators. We believe that applying MTxTM in real environments will produce more realistic evidences of the benefits of using testable services in service oriented computing.

Acknowledgement

The authors would like to thank the Brazilian funding agency CNPq and FAPESP (process 2008/03252-2) for the financial support. This work has been partially supported by the European Project FP7 IP 257178 CHOReOS.

References

[1] Bertolino A, Angelis GD, Kellomaki S, Polini A. Enhancing service federation trustworthiness

through online testing. IEEE Computer, 2012, 45(1): 66–72.
[2] Bartolini C, Bertolino A, Elbaum S, Marchetti E. Whitening SOA testing. Proc. of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. ACM. 2009. 161–170.
[3] Bartolini C, Bertolino A, Elbaum S, Marchetti E. Bringing white-box testing to service oriented architectures through a service oriented approach. Journal of Systems and Software, April 2011, 84: 655–668.
[4] Benharref A, Dssouli R, Serhani MA, Glitho R. Efficient traces' collection mechanisms for passive testing of web services. Information Software Technology Journal, 2009, 51(2): 362–374.
[5] Beydeda S, Gruhn V. State of the art in testing components. International Conference on Quality Software. IEEE Computer. Society Press. 2003. 146–153.
[6] Bozkurt M, Harman M, Hassoun Y. Testing and verification in service-oriented architecture: a survey. Software Testing, Verification and Reliability. 2012.
[7] Bundel GA, Lee G, Morris J, Parker K, Lam P. A software component verification tool. International Conference on Software Methods and Tools. IEEE. 2000. 137–146.
[8] Briand LC, Labiche Y, Sówka MM. Automated, contract-based user testing of commercial-off-the-shelf components. Proc. of the 28th International Conference on Software Engineering, New York, NY, USA. ACM. 2006. 92–101.
[9] Bucchiarone A, Melgratti H, Severoni F. Testing service composition. Proc. of the 8th Argentine Symposium on Software Engineering. 2007.
[10] Bertolino A, Polini A. SOA test governance: Enabling service integration testing across organization and technology borders. Proc. of the IEEE International Conference on Software Testing, Verification, and Validation Workshops. ICSTW '09. Washington, DC, USA, IEEE Computer Society. 2009. 277–286.
[11] Canfora G, Di Penta M. Testing services and service-centric systems: challenges and opportunities. IEEE IT Professionnal, 2006, 8(2): 10–17.
[12] Canfora G, Di Penta M. Service Oriented Architecture Testing : A Survey. Number 5413 in LNCS. Springer. 2009. 78–105.
[13] Eler MM, Bertolino A, Masiero PC. More testable service compositions by test metadata. 6th IEEE International Symposium on Service Oriented System Engineering. IEEE Computer Society. Washington, DC, USA. 2011. 204 –213.
[14] Eler MM, Delamaro ME, Maldonado JC, Masiero PC. Built-in structural testing of web services. Brazilian Symposium on Software Engineering, 2010, 1: 70–79.
[15] Gross HG. Component-Based Software Testing with UML. Springer. 2005.
[16] Hass H, Brown A. Web Services Glossary,W3C Working Group Note. 2004.
[17] Hornstein J, Edler H. Test reuse in CBSE using built-in tests. Workshop on Component-based Software Engineering. 2002.
[18] IEEE. IEEE Standard Glossary of Software Engineering Terminology. [Technical report]. 1990.
[19] Li ZJ, Tan HF, Liu HH, Zhu J, Mitsumori NM. Business-process-driven gray-box SOA testing. IBM Systems Journal, 2008, 47(3): 457–472.
[20] Mackenzie MC, Laskey K, Mccabe F, Brown PF, Metz R. OASIS Reference Model for Service Oriented Architecture v1.0, OASIS Standard. 2006.
[21] Myers GJ. The Art of Software Testing. Wiley, New York. 1979.
[22] Orso A, Harrold MJ, Rosenblum D. Component metadata for software engineering tasks. 2nd International Workshop on Engineering Distributed Objects. Springer. 2000. 129–144.
[23] O'Brien L, Merson P, Bass L. Quality attributes for service-oriented architectures. Proc. of the International Workshop on Systems Development in SOA Environments. 2007. 3.
[24] Papazoglou MP, Traverso P, Dustdar S, Leymann F. Service-oriented computing: a research roadmap. International Journal of Cooperative Information Systems, 2008, 17(2): 223–255.
[25] Hou SS, Zhang Lu, Xie T, Sun JS. Quota-constrained test-case prioritization for regression testing of service-centric systems. Proc. IEEE ICSM. 2008. 257–266.
[26] Tallam S, Gupta N. A concept analysis inspired greedy algorithm for test suite minimization. SIGSOFT Software Engeneering Notes, Sept. 2005, 31: 35–42.
[27] Tsai WT, Gao J, Wei X, Chen Y. Testability of software in service-oriented architecture. Proc.

of the 30th Annual International Computer Software and Applications Conference. 2006. 163–170.

[28] Wang Y, King G. A European COTS architecture with built-in tests. Proc. of the 8th International Conference on Object-Oriented Information Systems. London, UK. Springer-Verlag. 2002. 336–347.

[29] Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A. Experimentation in software engineering: an introduction. Kluwer Academic Publishers. Norwell, MA, USA. 2000.