Proceedings of The 6th IEEE International Symposium on Service Oriented System Engineering (SOSE 2011)

More Testable Service Compositions by Test Metadata

Marcelo Medeiros Eler
ICMC/USP
Sao Carlos/SP - Brazil
Email: mareler@icmc.usp.br

Antonia Bertolino
ISTI/CNR
Pisa - Italy
Email: antonia.bertolino@isti.cnr.it

Paulo Cesar Masiero
ICMC/USP
Sao Carlos/SP - Brazil
Email: masiero@icmc.usp.br

***Abstract*—In previous work we proposed testable services as a solution to provide third-party testers with structural coverage information after a test session, yet without revealing their internal details. However, service testers, e.g., integrators that use testable services into their compositions, 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 will help their testers to get a higher coverage. We show the approach on a case study of a real system that uses orchestrations and testable services.**

I. Introduction

Service Oriented Architecture (SOA) promises the rapid, low-cost development of loosely coupled and easily integrated applications even in heterogeneous environments [1]. Services can come in two flavors: single or composed. A composed service (or composition) is a service that delivers its functionality through the interaction of more services. Depending on the scheme of interaction, compositions form *orchestrations* or *choreographies* [2].

Everyone would agree that testing third party services before integrating them in an orchestration or choreography is very important. Unfortunately, it is not an easy task. Testing SOA applications is indeed challenging due to the complex nature of services and their characteristics of high dynamism and loose coupling. However, test cases should be chosen with accuracy [3], [4], as testing of services is expensive and costs may be associated with their execution. Low testability of compositions [5] makes things worse.

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" [6]. It is also an important quality indicator since its measurement leads to the prospect of facilitating and improving a service test process [7], [8].

Third-party services generally yield a low testability because they are provided as black boxes, only exposing a specified interface to their clients. This hampers the establishment and the tracing of testing criteria other than those based on the specification or based on the interface [7]. Such characteristic is a common issue of services and components when it comes to testing [9].

To overcome this issue, two similar approaches had been independently proposed in [10] and [11] to improve the testability of SOA applications by allowing for white-box testing of third-party services, but without revealing their internal details, 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 [10] and [11] 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 called by both approaches *testable services*.

Developers of orchestrations or choreographies (or, integrators) can thus test a third party testable service as a single service before integrating it into their application, or can test the composition by invoking the testable services in the context of 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 orchestration or the choreography that uses the testable service. In both cases the integrator can get a structural coverage analysis based on the test session carried out.

This coverage measure provides a feedback about the thoroughness of the executed tests, but, on the negative side, it provides no clue of how testing could be improved. 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 that, for example, 40% of the nodes of the service have not yet been covered, but 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.

Authors of the two testable services approaches join here their effort to find a common solution to this limitation shared by earlier approaches. The purpose of this paper is to enhance both approaches [10], [11] by proposing a solution to make testable services even *more testable*. We present a test metadata model for testable services, which is inspired by the concepts of built-in testing [9], [12] and metadata [13]. The approach is called MTxTM (for More Testable service by Test

978-1-4673-0412-2/11/$26.00 ©2011 IEEE

Metadata).

This paper is organized as follows. Section II presents a motivating scenario. Section III provides some background notions. Section IV depicts the built-in test metadata approach we propose in this paper. Section V shows a case study. Related work is surveyed in Section VI and concluding remarks are given in Section VII.

II. Motivating scenario

In this section we motivate our effort towards more testable service compositions. We consider a test scenario of a real application that is an open source E-Commerce framework, called Broad Leaf Commerce[1]. This framework is 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 test cases for almost all services.

For the purpose of our investigation, we selected a composed service called `RegisterService`, which is an orchestration using several services, among which a service called `CustomerService`. `RegisterService` has 2 public operations, while `CustomerService` has 7 public operations. Test sets for both `RegisterService` and `CustomerService` are available, called `ts-Register` and `ts-Customer`, respectively. Figure 1 shows an illustration of this scenario.

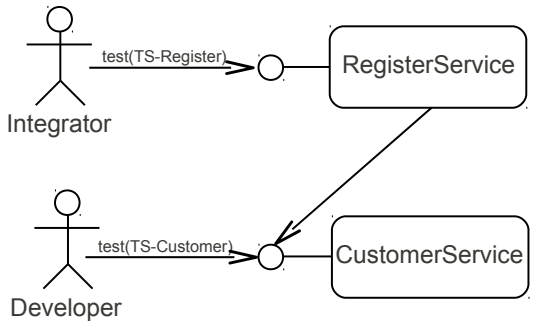

Fig. 1. Motivating scenario

According to the testable service approaches, we transformed the `CustomerService` into a testable service (testable services are introduced in the next section; for the purpose of this example, take it to mean that we instrumented it, pretending to be the `CustomerService` developers) and got the coverage reached in two situations. In the first situation we execute `TS-Customer`, which tests `CustomerService` in isolation. Table I shows the structural coverage achieved in this situation for the whole service and 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 were created by the developers that have access to the source code of the service.

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

TABLE I
Structural coverage reached when testing `CustomerService` as a single service

	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%

TABLE II
Structural coverage reached when testing `CustomerService` in the context of the orchestration `RegisterService`

	Orchestration context		
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%

In the second situation, we executed the test set called `TS-Register` to test the orchestration `RegisterService`, which also invokes the `CustomerService` operations. Table II shows the structural coverage of `CustomerService` when invoked from within the context of `RegisterService`. Notice that the coverage is now relatively low. This probably happens because `TS-Register` was meant to test `RegisterService` without taking into account how much of the `CustomerService` was being executed from that context.

By using the testable version of `CustomerService` the integrator can see that the coverage of `CustomerService` is low when tested from the perspective of `RegisterService`. The limitation of the testable service approach is that the integrator does not have enough information neither to evaluate if the measure reached on `CustomerService` is good nor to create more test cases to raise the coverage.

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

[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.

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.

2) **Relative coverage:** Different clients may use different operations of a service or even use the same operations but in different contexts. A car shop, an estate agent and a university, for example, may use the same bank service to make loans to their clients. The car shop may use short time duration loans, while the 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 which are not used from 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 usage profile for each client of the testable service.

III. Background

A. Testable service approach

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. *Testable services* [10], [11] are services instrumented so to provide their clients with coverage information regarding a test session execution. There are three main stakeholders in the testable services approaches: the testable service, the integrator and a coverage provider. The testable service is a service instrumented to collect information about the code structure (e.g., instructions, branches and data) executed during a test session. The integrator uses the testable service in an orchestration or choreography and uses its interface to get a coverage analysis after a test session execution. The coverage provider is a service responsible for using the information about the execution of the testable service and performing the structural coverage analysis.

In the SOCT approach [10] the instrumentation of the testable service should be done by the developers themselves, while the information about the execution of the testable service is collected by a coverage provider. This coverage provider is called TCov and is also in charge of generating the coverage analysis information. In the BISTWS approach [11], instead, the instrumentation is automatically done by the coverage provider called TestingWS and the data generated on its execution are locally stored. TestingWS is also used to calculate the structural coverage of the testable service.

Beyond these differences, the SOCT and the BISTWS approaches are very similar and follow basically the same process and governance framework. In both approaches the integrator initiates the process by opening a Testing Session, which is uniquely identified. The integrator then starts launching test cases on the testable service and can afterwards collect the coverage information using the session identifier. In both approaches the testable service has operations to define the boundaries of a test session and to retrieve coverage information: `startTest`, `stopTest` and `coverageMeasure`, respectively, in SOCT; and `startTracing`, `stopTracing` and `getCoverage` in BISTWS.

From the point of view of the integrator, it does not matter who instruments the testable service or how the coverage information will be calculated. The integrator only wishes to perform a test session and discover how much of the testable service is executed when the orchestration is tested. In this case, either of SOCT or BISTWS approaches could be used to produce the testable service. The only difference for the integrator would be the names of the operations to define the boundaries of the test session or to get the coverage analysis, but the results from their perspective would be the same. Therefore, from now on, in this paper we will not distinguish anymore between SOCT of BISTWS.

We refer to the abstract sequence diagram in Figure 2, showing an integrator using a testable service in the context of an orchestration, hiding away details of how the coverage information is eventually achieved.

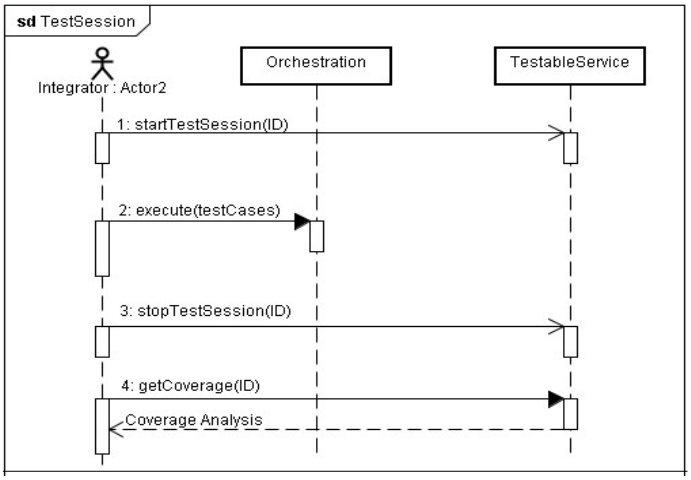

Fig. 2. Sequence Diagram of a generic test session of a testable service

B. Built-in testing and metadata

Software components and web services have many similarities. Both of them are self-contained composition units and can only be accessed through their explicit published interfaces [9]. According to [14], the lack of more information about third party components brings many problems to the validation,

to the maintenance and to the evolution of component-based applications. In the context of testable services, this is also the reason why integrators cannot improve their test set and get a better coverage of the testable service in many situations. Two approaches in the literature to handle the issue of lack of information in component testing are: Built-in Testing and Metadata.

Built-in testing is an approach created to improve the testability of software components based on the self-testing and defect-detection concepts of electronic components. The general idea is to introduce functionalities into the component to provide its users with better control and observation of its internal state [9], [15]. 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 [16].

In the regular mode, the built-in test capabilities are transparent to the component user and the component does not differ from other components. In the testing mode, however, the component user can test the component with the help of its built-in testing features. The component user can invoke the respective methods of the component, which execute the test, evaluate autonomously its results, and output a test summary [16].

A testable service is a type of built-in testing entity because its testability was improved by operations that enable its observation regarding structural coverage information. Moreover, a testable service can also be set to a regular or to a test session mode. A testable service, however, has neither self testing capabilities nor built-in test cases made available to its clients.

In component-based development the problem of lack of information is also usually handled by providing some metadata within the software component. Metadata provide extra information about the component other than the interface specifications. Metadata range from finite-state-machine models and QoS-related information to plain documentation. It may consist of coverage information, test cases, abstract representations of source code or assertions about security properties [13], for example.

Some authors make distinction between *a priori* and *on-demand* metadata. A priori metadata provide information that was previously created and is attached to the component when it is released. On-demand metadata provide information calculated/generated during runtime.

In the sense that it provides testing information, built-in testing can itself be considered a metadata approach. Bundell and coauthors [17] pointed that metadata should be used to provide information to help component users on analysis and testing activities. They proposed that test specifications and a set of test cases to test each interface should be provided as metadata along with software components. In the next section we present our solution to handle the problem of lack of information in the testable service architectures using both concepts of built-in testing and metadata.

IV. The MTxTM Approach

Lack of information and relative coverage is the main cause of the issues we mentioned before in our motivating example. We devised an approach in which testable services are published along with both a priori and on demand built-in test metadata to provide integrators with more information to help them to improve the test set when the coverage achieved is low or not satisfactory. We also propose incremental usage profile to minimize the relative coverage issue.

The a priori metadata should be created by the developer of the testable service and they consist of the test set used to test the service during development time (similarly to the built-in test concept). The on demand metadata is automatically generated by the testable service using the a priori metadata. The usage profile is created by the integrator who uses the testable service within a composition. In the next subsections we explain in details these three main components of the MTxTM approach, which is an extension to our previous work regarding testable services.

A. A priori metadata

Developers usually test their services before their publication using available testing techniques, strategies and plans. An instance of a typical scenario of creating test cases involves the following activities:

1) The developers creates 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 developer executes 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 developer looks at the source code or use flow models to discover which parts of the service were not executed yet and then creates test cases to cover the uncovered test requirements (instructions, branches, data, ...).

External clients of testable services can perfectly perform 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 motivating example presented at the beginning of this paper, Table III presents the test cases of the testable service `CustomerService` provided by its

Listing 1. XML structure of the a priori metadata

```
<testcase id="tc-04" description="..."
  operation="createFromId" return="boolean">
<input name="ID">1004</input>
<input name="user"> Cust4 </input>
<input name="passwd">Cust4Psswd</input>
<input name="chAnsw1">Answ1</input>
<input name="chAnsw2">Answ1</input>
<input name="email">cust4@broad.leaf</input>
 ...
<expected>true</expected>
</testcase>
```

TABLE IV
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							

developer. Note that there are test cases for each operation of `CustomerService` and for each test case there is a description related to the reasoning to create it.

We defined that the developers of testable service using the MTxTM aproach must express the a priori metadata using a XML structure in which each test case must contain the following information: a unique identification (tc-ID); the name of the operation it refers; 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. Listing 1 shows the tc-04 in a simplified XML format.

As soon as the test set created by the developer is packed and provided to the testable service as the a priori metadata, the testable service executes each test case and creates a list of which test requirements are covered by each test case. Table IV shows the nodes of the operation `createFromID` covered by each test case of the a priori metadata of `CustomerService`. The same list is created for each operation and for each criterion supported by the testable service.

B. On demand metadata

The on demand metadata of the MTxTM consist of suggestions provided to help clients to improve the coverage of the testable service reached so far. These metadata are generated on the fly upon request by some client after a test session and they are based on the a priori metadata.

The generation of the on demand metadata works as follow. The client 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 client as the on demand metadata.

Suppose, for instance, that a client requested the on demand metadata of `CustomerService` considering a test session which not covered the nodes 2 and 3 of the operation `createFromID`. The testable service would use the a priori metadata and identify that tc-05 and tc-06 cover the nodes 2 and 3 and they would be provided as suggestions to the client.

If the client is testing the testable service in isolation, it is straightforward for him/her to use the test cases in the on demand metadata to improve the coverage. If the client is instead an integrator and is testing an orchestration or a choreography, then the provided test cases could not be invoked immediately, as the testable service is not directly invoked. The client will have to analyze the input data of the suggested test cases and identify an integration test for the orchestration or choreography that invokes the testable service with those or similar inputs. It should be an easy task because integrators own the source code of the composition and they should be able to create suitable test cases for the composition to repeat the same test configuration suggested by the on demand metadata. The situations in which the suggested test cases cannot be repeated can be handled by usage profiles (see next subsection).

In the MTxTM approach, we also propose that the order of the test cases suggested as the on demand metadata is not chosen by chance. By applying a common greedy heuristics [18], those test cases that cover more uncovered test requirements come first. In this way, the client of the testable service 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. The information of the on demand metadata is the same as that of the a priori metadata.

C. Incremental usage profile

The developer of a service does not know in advance in which orchestrations or choreographies it will be used, 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 request the coverage information (see the notion of relative coverage introduced in Section II).

To overcome this issue and customize the test metadata for the context in which the testable service is used, we also propose in the MTxTM approach an incremental usage profile that characterizes the testable service's client. It provides the following information:

- An identification of the orchestration or choreography that is using the testable service.
- An identifier for future profile updates.

TABLE III
THE A PRIORI METADATA OF CUSTOMERSERVICE (TS-CUSTOMER)

	Operation: createFromID							
TC-ID	**ID**	**user**	**passwd**	**email**	**chAnsw1**	**chAnsw2**	**register**	**Description**
tc-01	1001	cust1	c1passwd	c1@bleaf.com			true	Create and register (true)
tc-02	1002	cust2	c2passwd	c2@bleaf.com			false	Create and don't register (false)
tc-03	1003	cust3	c3passwd	c3@bleaf.com	Answ1-ct3		true	Create with chAnswer1
tc-04	1004	cust4	c4passwd	c4@bleaf.com	Answ1-c4	Answ2-c4	true	Create with chAnswer1 and 2
tc-05	null	cust5	c5passwd	c5@bleaf.com			true	Create using an auto generated ID
tc-06	1001							Create using an existent ID
	Operation: registerCustomer							
	ID	**user**	**passwd**	**email**	**Description**			
tc-07	null	cust7	c7passwd	c7@bleaf.com	Register and create a customer at the same time (id=null)			
tc-08	1002	cust2	c2passwd		Register an unregistered customer using a valid ID			
	Operation: readByID							
	ID	**Desc.**						
tc-09	1001	Invoke the operation using an existent ID						
tc-10	7777	Invoke the operation using an invalid ID						
	Operation: readByUsername							
	user	**Desc.**						
tc-11	cust4	Invoke the operation using an existent username						
tc-12	custN	Invoke the operation using an invalid username						
	Operation: readByEmail							
	email	**Desc.**						
tc-13	c3@...	Invoke the operation using an existent email						
tc-14	cN@...	Invoke the operation using an invalid email						
	Operation: changePassword							
	user	**New**	**Confirm**	**Desc.**				
tc-15	cust23	c23New	c23New	Try change the passwd using an invalid customer				
tc-16	cust4	c4New	c4New	Try to change the passwd using a valid customer				

- A list of the operations of the testable service that will be actually used by this client. 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 the context of this client. 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. Clients 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 Section V.

D. Business Model

We believe that the test metadata model proposed for testable services could change not only the way developers and integrators technically interact in testing of compositions, but also change the business model behind SOA orchestrations and choreographies. The structural testing capability of testable service can already bring competition advantage on the market; we believe that the enhanced testability of "more testable" services can aggregate even more value to market. Consideration in depth of a business model 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:

- Regular service. Provides only its interface to its clients.
- Testable service. Provides its interface and structural testing capability.
- Testable service with a priori test metadata.
- Testable service with a priori and on demand test metadata.

V. CASE STUDY

In this section we report a first assessment of the proposed approach on the case study presented in Section II. The example used is small, but genuine, since we used existing services and test sets reused from a real application.

As we illustrated (see Figure 1), the e-commerce instantiation of the BroadLeafCommerce framework has an orchestration called `RegisterService` that uses another service called `CustomerService`. `RegisterService` has 2 public operations, while `CustomerService` has 7 public operations. The developers of this e-commerce application provide a test set to test `RegisterService` and also a test set for testing `CustomerService`.

In this case study, one of the authors played the role of the integrator and `CustomerService` represents the "more testable" service. The integrator wants to improve the coverage of `CustomerService` reached when testing

RegisterService, with respect to that shown in Table I. The study focuses on the following research questions.

- **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 which improve the coverage of the testable service;

RQ1-b whether the integrator can identify the irrelevant test cases from the test metadata; and

RQ1-c whether the usage profile is useful to generate a more meaningful 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.

A. Improving coverage using MTxTM

The execution of the test cases of CustomerService achieved a high coverage (see Table I), while the execution of the test cases of RegisterService orchestration obtained a lower coverage regarding the CustomerService structure (see Table II). This study starts from this context and the in integrator wants to increase the coverage of CustomerService when tested from within the RegisterService orchestration. This 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 Figure 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 some 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.

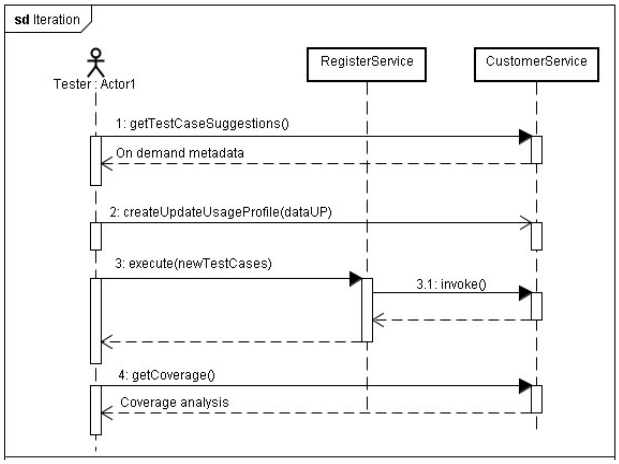

Fig. 3. Activities performed by the integrator at each iteration

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 internals), 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.

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

1) First iteration:: The integrator executed the original test set of RegisterService that contains 4 test cases and obtained the coverage presented in Table II. The on demand test metadata progressively ordered suggested by CustomerService after the execution of the test cases are 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 III (Col. 5, 6 and 7).

2) Second iteration:: The integrator created new test cases to test RegisterService based on the test cases tc-04, tc-03 and tc-01 (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 that are actually 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 V shows the usage profile of RegisterService created in this iteration.

TABLE V
REGISTERSERVICE USAGE PROFILE

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 7 test cases) after submitting the usage profile to CustomerService. Table VI (Col. 2,3 and 4) 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 the 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 VI
STRUCTURAL COVERAGE ANALYSIS OF CUSTOMERSERVICE IN THE SECOND ITERATION

	Second 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%

After the test session CustomerService generated the on demand metadata and suggested the following test cases: 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. It is not, however, a mistake of the CustomerService on providing the on demand metadata. This happens because RegisterService handles the input data before invoking CustomerService and it is not always possible to recreate the same conditions in which the CustomerService was tested as a single service. This fact evidences that despite using the same service, orchestrations handle data in different ways and require different functionalities from a single operation.

3) Third iteration:: 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 10 test cases and obtained the coverage measures presented in Table VII. Note that the coverage achieved is further increased. The following four test cases are the on demand metadata generated by CustomerService this time: 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. Hence the integrator decides to stop the test set augmentation.

TABLE VII
STRUCTURAL COVERAGE ANALYSIS OF CUSTOMERSERVICE IN THE THIRD ITERATION

	Third 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%

B. Improving coverage using random test cases

The case study presented above showed a somewhat complex process to improve the coverage of CustomerService when invoked in the context of RegisterService. A natural question arises whether this 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 a first iteration of this study, we executed 20 times the test set provided by the developers (ts-Register) plus six new random test cases. We decided to use 6 new test cases in this study because it is the number of new test cases created using the MTxTM approach. In a 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 VIII shows the coverage analysis obtained by each iteration of this case study using random test cases.

C. Discussion

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 create new test cases to the test set ts-Register. In fact six new test cases have been

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

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 VIII shows, for each approach, the number of test cases created for `RegisterService` and the coverage reached on `CustomerService` at 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).

TABLE VIII
SUMMARY OF THE RESULTS OF THE CASE STUDY

Approach	Iter.	#TC	all-nodes	all-edges	all-uses
Random	1st	10	74%	56%	56%
Random	2nd	54	84%	68%	64%
Random	3rd	104	84%	75%	71%
MTxTM	1st	4	50%	34%	34%
MTxTM	2nd	7	84%	69%	67%
MTxTM	3rd	10	90%	79%	78%

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 makes adaptations 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. If we look at Table VIII, 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" [3]. 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" [3] and cause a denial-of-service phenomena [4]. 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 [3]. It is also important to keep the number of test cases low if the services under test are charged on a per-use basis [4].

D. Threats to validity

In this subsection we discuss the threats to the validity of this case study, regarding the construct, internal and external validity. To ensure construct 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 represents as closely as possible the 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. It is to minimize this confounding factor that we only used the test data provided on the BroadLeafCommerce web site. We cannot however exclude its impact, and we plan to perform further case studies using independent subjects.

The external validity of this study case cannot be assured. The objects 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 in order to generalize the results we obtained with this case study.

VI. RELATED WORK

Service testing is actively researched, as recently surveyed by Canfora et al. [4]. 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 investigated by Bucchiarone et al. [19].

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" [4]. The need to enhance black-box testing with coverage information of tested services has also been recognized by Li et al. [20]. 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. [21]. Similarly to [10] and [11], 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, which we already introduced in Section II. Briand et. al. [22] 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.

VII. Concluding Remarks

This paper presented a test metadata model for testable services using concepts of built-in testing and metadata. The proposed approach enhances the way integrators can interact with testable services. Integrators can test the testable service in isolation or in the context of an orchestration or choreography and get more than a structural coverage analysis report, as in previous approaches [10], [11]. They can also get the a priori (static) and/or the on demand (dynamic) metadata of the testable service.

MTXTM helps address the two issues we identified in the motivating scenario. The `insufficient test cases` issue is mitigated by both a priori and on demand metadata provided by the testable service. The case study we performed shows 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 `relative coverage` issue is mitigated by the usage profile that is used to calculate a suitable coverage considering the context from which the testable service is called.

A related question is whether it is realistic to expect service developers to create these specific metadata since this requires additional effort. 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 [13] for which they could charge a fee, thus enhancing the value of their services.

As future work we intend to perform further evaluation of the approach via more formal experiments. We also intend to study how the MTxTM approach could be used in the context of choreographies, whereas here it was applied for orchestrations. For this, the approach will be used in the context of the CHOReOS project [4], which aims at implementing a framework for scalable choreography development. The MTxTM approach will be used in the Governance, Verification and Validation support of the framework.

VIII. Acknowledgements

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

[4] www.choreos.eu

References

[1] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-oriented computing: a research roadmap," *Int. J. Coop. Inf. Syst.*, vol. 17, no. 2, pp. 223–255, 2008.

[2] H. Hass and A. Brown, "Web Services Glossary,W3C Working Group Note," 2004. [Online]. Available: http://www.w3.org/TR/ws-gloss/

[3] S. shan Hou, L. Zhang, T. Xie, and J. su Sun, "Quota-constrained test-case prioritization for regression testing of service-centric systems," in *Proc. IEEE ICSM*, 2008, pp. 257–266.

[4] G. Canfora and M. Di Penta, *Service Oriented Architecture Testing : A Survey*, ser. LNCS. Springer, 2009, no. 5413, pp. 78–105.

[5] M. H. Mustafa Bozkurt and Y. Hassoun, "Testing web services: A survey," Department of Computer Science, King's College London, Tech. Rep. TR-10-01, January 2010.

[6] IEEE, "IEEE Standard Glossary of Software Engineering Terminology," Tech. Rep., 1990. [Online]. Available: http://dx.doi.org/10.1109/IEEESTD.1990.101064

[7] L. O'Brien, P. Merson, and L. Bass, "Quality attributes for service-oriented architectures," in *Proc. of the Int. Workshop on Systems Development in SOA Environments*, 2007, p. 3.

[8] W. T. Tsai, J. Gao, X. Wei, and Y. Chen, "Testability of software in service-oriented architecture," in *Proc. of the 30th Annual Int. Computer Software and Applications Conf.*, 2006, pp. 163–170.

[9] H.-G. Gross, *Component-Based Software Testing with UML*. Springer, 2005.

[10] C. Bartolini, A. Bertolino, S. Elbaum, and E. Marchetti, "Bringing white-box testing to service oriented architectures through a service oriented approach," *J. Syst. Softw.*, vol. 84, pp. 655–668, April 2011. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2010.10.024

[11] M. M. Eler, M. E. Delamaro, J. C. Maldonado, and P. C. Masiero, "Built-in structural testing of web services," *Brazilian Symp. on Soft. Engineering*, vol. 0, pp. 70–79, 2010.

[12] Y. Wang and G. King, "A european cots architecture with built-in tests," in *OOIS '02: Proc. 8th Int. Conf. Object-Oriented Information Systems*. London, UK: Springer-Verlag, 2002, pp. 336–347.

[13] A. Orso, M. J. Harrold, and D. Rosenblum, "Component metadata for software engineering tasks," in *2nd Int. Workshop on Engineering Distributed Objects (EDO 2000)*. Springer, 2000, pp. 129–144.

[14] M. J. Harrold, A. Orso, D. Rosenblum, G. Rothermel, and M. L. Soffa, "Using component metadata to support the regression testing of component-based software," Tech. Rep. GIT-CC-00-38, 2000.

[15] J. Hornstein and H. Edler, "Test reuse in CBSE using built-in tests," in *Workshop on Component-based Software Engineering*, 2002.

[16] S. Beydeda and V. Gruhn, "State of the art in testing components," in *Int. Conf. on Quality Software (QSIC). IEEE Computer*. Society Press, 2003, pp. 146–153.

[17] G. Bundell, G. Lee, J. Morris, K. Parker, and P. Lam, "A software component verification tool," in *Int. Conf. on Software Methods and Tools (SMT)*. IEEE, 2000, pp. 137– 146. [Online]. Available: citeseer.ist.psu.edu/bundell00software.html

[18] S. Tallam and N. Gupta, "A concept analysis inspired greedy algorithm for test suite minimization," *SIGSOFT Softw. Eng. Notes*, vol. 31, pp. 35–42, Sept. 2005. [Online]. Available: http://doi.acm.org/10.1145/1108768.1108802

[19] A. Bucchiarone, H. Melgratti, and F. Severoni, "Testing service composition," in *Proceedings of the 8th Argentine Symposium on Software Engineering*, 2007.

[20] Z. J. Li, H. F. Tan, H. H. Liu, J. Zhu, and N. M. Mitsumori, "Business-process-driven gray-box SOA testing," *IBM Syst. J.*, vol. 47, no. 3, pp. 457–472, 2008.

[21] A. Benharref, R. Dssouli, M. A. Serhani, and R. Glitho, "Efficient traces' collection mechanisms for passive testing of web services," *Information Software Technology Journal*, vol. 51, no. 2, pp. 362–374, 2009.

[22] L. C. Briand, Y. Labiche, and M. M. Sówka, "Automated, contract-based user testing of commercial-off-the-shelf components," in *ICSE '06: Proc. 28th Int. Conf. on Software Engineering*. New York, NY, USA: ACM, 2006, pp. 92–101.