2011 IEEE International Conference on Services Computing

Model-Based Testing of Service-Oriented Applications via State Models

Andre Takeshi Endo and Adenilso Simao
Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo (USP), Brazil
{*aendo, adenilso*}*@icmc.usp.br*

***Abstract*—Service-oriented architectures and web services have been used to foster the development of loosely coupled, interoperable, and distributed applications. Mission-critical and business process systems can be implemented with them, requiring a high level of quality. Model-based testing allied with state models is a promising candidate due to its efficiency, effectiveness, and flexibility. In this paper, we propose a model-based testing process to verify service-oriented applications. Finite state machines are used to model and support the test case generation. We evaluated the applicability of our process with a case study using a prototype tool.**

***Keywords*-service oriented architecture; model based testing; finite state machines; web services**

I. Introduction

Service-Oriented Architecture (SOA) consists in an architectural style in which functionalities are decomposed into distinct units (services) [1]. These services, distributed over the Internet, can be reused and combined to create new and more complex enterprise applications. This process, named service composition, can be achieved by either cooperation among the services (choreography) or centralized coordination of services accessed in the composition (orchestration). Service-oriented applications have been developed using a set of XML standards (WSDL, SOAP, and UDDI), known as Web Services technology and implemented by various programming languages.

SOA and its more adopted implementation, Web Services, add features that need to be considered in the software development, such as distribution, lack of observability and control, and dynamic integration. In this context, the testing activity is essential for ensuring the quality of services. The testing approach should be dynamic, since SOA applications are constantly changing and are deployed in heterogeneous environments. This approach needs to be rigorous in order to guarantee the necessary level of reliability. Many consolidated testing approaches are applicable to the context of service-oriented applications, though they cannot be directly reused due to the dynamic and adaptive nature of services [2]. Thus, creating or adapting new testing approaches for service-oriented applications is necessary.

Among the existing testing techniques, Model-based Testing (MBT) is a promising candidate to provide these characteristics. MBT is an approach that derives test cases from models designed by the tester to describe the System Under Test (SUT) and support the testing activity. MBT works efficiently with software changes and, being based on well-defined models, the necessary level of formality is also provided. Using MBT, the test case generation is usually efficient since the tester can update the model and regenerate the test suite, avoiding error-prone manual changes [3]. Another characteristic which is interesting for SOA testing is the adoption of formal black-box models to support MBT. Black-box models are appropriate for service-oriented applications because internal details of services are usually not observable, the complexity of interactions and test harness can be abstracted, and the formality of the model contributes to more reliable tests. Moreover, MBT is most likely to be cost-effective when the execution of the generated tests can also be automated [4], such as service-oriented applications.

There are several studies that propose SOA testing using different types of model, such as Extended Finite State Machine (EFSM) [5], Graph Transformation Rules (GT Rules) [6], Stream X-Machine (SXM) [7], and Symbolic Transition System (STS) [8]. Although these studies provide approaches to test service-oriented applications, they are not concerned about establishing a process and describing the necessary steps in details. These studies usually describe complex interactions among the services by using state models. Among the mentioned modeling techniques, Finite State Machines (FSMs) have been widely studied by the Software Testing community and applied to various software domains [9], [10]. In particular, this state model has not been explored to test service-oriented applications.

Approaches that adopt rigorous models to support the tests have been proposed to obtain more formality and handle complex interactions [5], [6], [7], [8]. However, these approaches do not characterize and revisit the MBT process applied to service-oriented applications.

In this paper, we propose an MBT process for service-oriented applications using state models. We also present an evaluation of the process instantiation with a case study, showing that the process is applicable to test service-oriented applications.

Thus, the main contributions of this paper are twofold: (i) an MBT process for service-oriented applications that adds additional steps, tools, and artifacts, to tackle specific characteristics of this software class; and (ii) an exploratory evaluation of the process, considering its automation (tools) and practical usage (case study).

978-0-7695-4462-5/11 $26.00 © 2011 IEEE
DOI 10.1109/SCC.2011.77

The remainder of this paper is organized as follows. Section II provides concepts of SOAs, as well as a motivating example. Section III presents an MBT process for service-oriented applications via state models. Section IV shows results achieved during the evaluation of the process. Section V presents the related work. Finally, Section VI presents the conclusion and discusses future work.

II. Preliminaries

A. SOA and Web Services

In a SOA, all functionalities are provided as services. According to Papazoglou and Heuvel [11], a service is a well-defined software module that does not depend on other services, i.e. it is self-contained. A service is a black box, since implementation details are hidden and only its interface is available. A SOA is based on three entities: *provider*, *consumer*, and *registry* [12]. An intermediate layer among the services, named Enterprise Service Bus (ESB), can be included in a SOA. The ESB is responsible for controlling, routing, and translating messages exchanged by the services [11]. Using the Web Services technology, a SOA is realized through three main XML standards: SOAP, WSDL, and UDDI. SOAP is a W3C protocol used to define the structure of messages exchanged among the services. WSDL is a W3C standard used to describe the service interface, including details like operations, data types, and adopted protocols. UDDI is an OASIS standard that defines a set of functionalities to support description and discovery of services.

A group of services can be assembled to create a new value-added service via composition. In a **service composition**, many services can be combined in a workflow to model and execute complex business processes. Service composition can be developed either as orchestration or as choreography. In **service orchestration**, there is a main entity that is responsible for coordinating the partner services. Currently, the most widespread language to implement a service orchestration is the Web Service Business Process Execution Language (BPEL) [13]. In **service choreography**, there is no control entity and all partner services work cooperatively. There are several languages used to describe a service choreography, e.g. Web Services Choreography Description Language (WS-CDL) [14].

B. Motivating Example

In this section, we introduce an example to illustrate the issues we address in this paper. ThirdPartyCall-SOA is an application based on the third party call service proposed by the Parlay-X services [15]. Some extra functionalities and modules were included to consider more SOA concepts. Figure 1 presents a generic architecture of the ThirdPartyCall-SOA application. ThirdPartyCall-SOA is composed of three services that can be distributed in different hosts:

- **Third Party Call Service (TPCS):** this service contains the main functionalities for a third-party application to manage call sessions among different participants. Seven operations are available in the WSDL interface of this service. TPCS provides an extra operation to receive notifications about the participants connected in a call.
- **Call Monitor Service (CMS):** this service represents an instance of a monitor that observes the participant's status in a call. CMS provides two operations to respectively subscribe and unsubscribe in a monitoring process of a participant's status. This status can be connected, hangUp, notReached, and busy.
- **Service Registry (SR):** this service is a registry specific to inform the available call monitor services. SR is an instance of the Apache jUDDI[1], an open-source Java implementation for the standard UDDI.

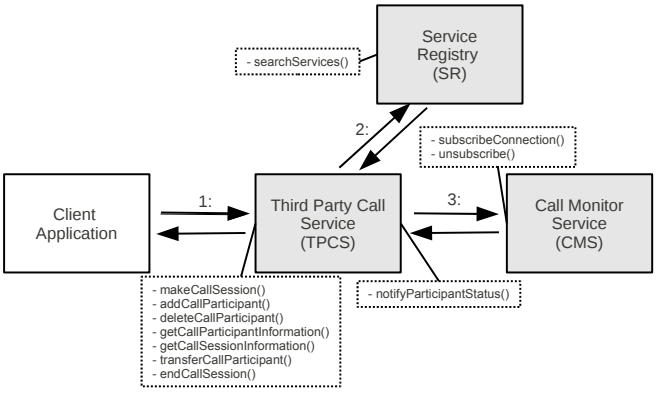

Figure 1. Architecture of the ThirdPartyCall-SOA application.

A typical usage of ThirdPartyCall-SOA is as follows. Initially, some application starts a call session with one or two participants. For instance, a stock quote monitoring application starts a session between the stockbroker and his/her client because some stocks reach a threshold value [15]. TPCS invokes an SR operation that searches for call monitor services. TPCS subscribes for each participant using CMS operations. At this point, CMS creates dynamically a stub to access the notification operation of TPCS. Thus, if some participant's status changes, CMS uses this stub to notify TPCS. CMS simulates the possible status changes using random choices after some time. After the establishment of a call, different actions can be made via TPCS operations, such as add/remove/transfer participants, query the call/participant status, and end the call session.

The application of MBT in service-oriented applications, such as ThirdPartyCall-SOA, is motivated by the potential for automation. In an adaptive and dynamic context of SOAs, it is important that tests are automated and can be

[1]http://ws.apache.org/juddi/index.html

easily changed. MBT usually reacts promptly to changing requirements and can be an interesting solution in this context. However, the distribution of services over the Internet and the usage of XML standards hinder the information gathering during the tests and the establishment of a test harness. For instance, in a SOA environment it is difficult to monitor other integrated services, besides the service under test.

In the example, TPCS provides a set of operations that can change the service state. Moreover, there are interactions with other services, such as SR and CMS. These interactions can be modeled by using state models. Generating tests that cover state models and check their states is an accurate way to verify these interactions. In ThirdPartyCall-SOA, there are different aspects that should be tested, such as individual services and their interaction. Considering all these aspects in only one state model would increase its complexity and size. In MBT, it is particularly necessary to manage the complexity and size of the models to avoid the state space explosion, while meaningful tests can still be generated. As noted in the example, we can build complex models even for simple service-oriented applications. For instance, ThirdPartyCall-SOA has different contexts that should be tested, e.g. the participant's status, number of participants in a call session, and the communication between TPCS and CMS.

There are also issues on using MBT and state models to test a service-oriented application like ThirdPartyCall-SOA. The test harness is complex due to the distributed nature of SOAs and the adoption of XML standards. The order of invocations and the state of the services are important to assure the reliability of service-oriented applications. However, complex and large state models can be built and, as a consequence, limiting the tests and model maintainability.

III. State-based Testing Process for Service-oriented Applications

Motivated by the issues discussed in the previous section, we propose a model-based process to test service-oriented applications with state models. We add to the process elements that were necessary in a SOA context. Elements present in generic MBT processes [4], [16] are revisited, highlighting the differences whenever necessary. Figure 2 represents the elements of the process as artifacts (Section III-A), tools (Section III-B), and the services under test. We also discuss the steps of MBT in service-oriented applications in Section III-C.

A. Artifacts

There are three types of artifacts in the proposed testing process: legacy artifacts, manual artifacts, and generated artifacts. Legacy artifacts are resources produced during the development of service-oriented applications. In Figure 2, we call them as service artifacts and can include interface descriptions (e.g. WSDL), composition specifications (e.g. BPEL, WS-CDL), and semantic information (e.g. OWL-S).

Manual artifacts are produced during the process by the testers. In Figure 2, the manual artifacts are the state model, test selection criteria, the concrete adaptor, and the linking information. We define *linking information* as the artifact that has references to the state model and service artifacts making connections between them. This artifact is important during the testing process, since it includes mechanisms to increase the level of automation, providing information to automatically generate code used in the test harness.

Generated artifacts are automatically obtained by using the supporting tools. In Figure 2, the generated artifacts include the abstract test suite, the abstract adaptor, and the tests report. The idea of the abstract adaptor artifact is to provide the necessary infrastructure for the tester to implement the concrete adaptor without concerns not related to tests.

B. Supporting Tools

In the proposed process, artifacts are processed by tools that produce new artifacts. Figure 2 presents the software tools needed to automate some tasks of the process. We describe each of them as follows.

State Model-based Test Generator: this module receives as input the state model and test selection criteria and generates abstract test cases. It is desirable that test selection criteria are already implemented in this module in order to automate the test case generation. The artifact *linking information* can also be considered during the test case generation. In this case, the algorithms and test criteria should be adapted to use the information included in this artifact.

State Model-to-SOA Adaptor Generator: this tool aims at decreasing the effort to develop the adaptor. It also reduces the complexity of the test harness, generating as much test code as possible. Thus, the tester can focus on the core of the adaptor implementation, leaving the repeated and tedious work to the tool. The state model, linking information, and service artifacts are received as input to generate an abstract adaptor that can be realized in a flexible way. However, the abstract adaptor contains pre-defined and fixed structures used to enable the test execution by the State Model-based Test Runner.

State Model-based Test Runner: this tool is responsible for mapping the abstract test cases to executable ones using the concrete adaptor. It is able to invoke operations of single services and compositions under test. As output, it produces a tests report detailing the execution of each test case. In complex test scenarios involving integration with other services, it communicates with the ESB-based Test Mediator to obtain more information and analyze the results.

ESB-based Test Mediator: this tool is essential when the service-oriented application is integrated with other services,

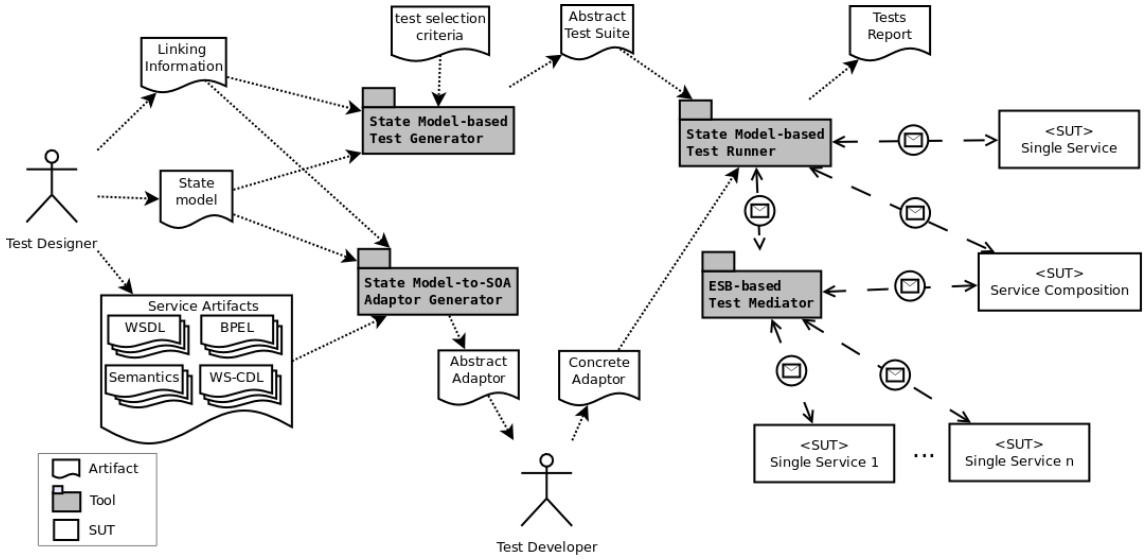

Figure 2. State-based testing process for service-oriented applications.

i.e. it involves a service composition. The Mediator can be used in two modes: monitoring and simulating. In the monitoring mode, it can be used to observe and record (SOAP) messages and to provide the necessary data for the State Model-based Test Runner evaluates the tests. In the simulating mode, services are not available or developed and the mediator works as stubs for them. The Mediator is integrated with an ESB, using its capabilities to monitor and simulate the involved services in a service-oriented application. If the service-oriented application is already based on an ESB, the mediator can be included in the bus as a component. Otherwise, more configuration for deploying the application in the ESB is needed, though this task can also be automated. Scripts can be used to deploy the services under test in the ESB.

C. Steps of the Process

After describing artifacts and tools, we discuss each step of the MBT process for service-oriented applications using state models.

1. Selection of a test scenario: It is usually infeasible to create and maintain only one model for the entire application. A specific part or feature of the SUT can be selected to be tested, i.e. to define a *test scenario*. Thus, the tester should identify test scenarios that are complex enough to create a model and simple enough to avoid too many states and transitions. As consequences of these scenarios, meaningful tests can be generated and the state space explosion problem is controlled.

Identifying and specifying test scenarios is helpful for SOAs since the tester can divide the tests into less complex contexts. For instance, a test scenario can be initially defined for each service. If a single service is too complex, two or more scenarios can be selected for this service. In a later step, more scenarios can be selected for the service composition. Moreover, the tester can use the available artifacts, such as interface specifications (WSDL) and composition descriptions (BPEL, WS-CDL), as information source to define test scenarios.

2. Building of the test model: The tester builds a state model to represent the behavior observed in the test scenario. In this context, we assume that the model is created exclusively to support the tests. This assumption will guarantee a redundancy necessary to the testing: the intended and the actual behavior [16]. Moreover, reusing analysis and design models (specifications) may not be possible since service-oriented applications are defined and assembled dynamically. This step should be performed by a tester with more skills in modeling (test designer in Figure 2).

Considering the application as a black box, a state model can be modeled by controlling inputs and observing outputs. Thus, the model abstracts the complexity and the tester can focus on states and transitions that compose the test scenario. In other words, the tester can exclusively concentrate on the modeling activity. Although there are a couple of differences between single services and compositions, state models are applicable in both contexts [5], [17]. State models also help in testing dynamical services since even though it is not possible to know which service will be integrated, it is possible to model its expected behavior.

At this step, the *linking information* artifact is produced connecting test models and service artifacts. The service artifacts are used as supporting resources. In a service composition context, the presence of the composition description can help in the modeling process. First, a model can

be partially or completely derived from these descriptions. Another possibility is to use these descriptions to verify the models built by the tester.

3. Definition of test selection criteria: Usually, infinitely many tests we can generate from a model. Thus, it is necessary to restrict the size of the test suite, defining test selection criteria. They are usually related to model coverage and are implemented by algorithms that traverse the model. This step can be completely automated if standard test selection criteria are adopted and implemented in State Model-based Test Generator.

4. Generation of abstract tests: This step is automated using the tools State Model-based Test Generator and State Model-to-SOA Adaptor Generator. These tools produce a set of abstract test cases and an abstract adaptor. Abstract test cases are derived from state models as input/output sequences. Possible changes in the application are handled by regenerating tests for a changed test model. This step and the following ones are performed by other testers (test developer in Figure 2).

5. Concretization of tests: In this step, the abstract test cases are transformed into executable ones. An adaptor that mediates the level of abstraction between the model and the SUT is developed. The tester uses the abstract adaptor as the initial point to concretize the tests. As interface descriptions (e.g. WSDL) are common in web services, much effort is saved since the source-code used to interact with the service can be automatically generated. If the tester adds extra information during the modeling connecting models and service artifacts (*linking information*), more effort can be saved.

6. Execution of tests: This step uses the abstract test suite and the concrete adaptor to execute the tests on the SUT. The tools State Model-based Test Runner and ESB-based Test Mediator perform this step automatically.

7. Analysis of results: The tool State Model-based Test Runner also automates this step. If some fault is detected, the tester can use a set of resources/artifacts to identify and correct it, such as abstract and concrete test cases, recorded SOAP messages, and the state model.

8. Verification and validation of produced artifacts: This orthogonal step verifies and validates artifacts produced along the process. State models and concrete adaptors are the main artifacts that need verification and validation in this step.

The state model needs to be validated w.r.t. the requirements of the service-oriented application. As an advantage of MBT, the model can be built in initial stages of the development process and faults can be revealed earlier. If there exist semantic or composition descriptions, these artifacts are an additional information to verify the test model. The concrete adaptor should be also verified since, as a program, it can also include faults. The execution of the tests (Step 6) can be used as a test for the adaptor.

IV. Exploratory Study

We conducted an exploratory study to analyze the proposed testing process for service-oriented applications. Our goal is to provide an initial evaluation of the testing process concerning the level of automation and its practical usage in service-oriented applications. We divided the study into two parts: first, we developed a prototype tool, named JStateModelTest, to instantiate the supporting tools proposed in the process; then, a case study was performed using two service-oriented applications.

A. JStateModelTest Tool

As the tools proposed in the testing process (Section III-B), the JStateModelTest tool is composed of four modules, named as *test-generator*, *adaptor-generator*, *runner*, and *mediator*. Currently, the tool with its four modules has about 3500 Lines of Code (LOC). First, we developed the tool using the programming language Java and its available frameworks for web services. We also suppose that Java is the programming language used to write the tests. Second, the modeling technique initially supported by the tool is the traditional Mealy's FSM [9]. An FSM is a machine composed by states and transitions. For each transition, an input symbol is consumed and an output symbol is produced. An FSM can be represented by a state diagram, which is a directed graph so that nodes are states and edges are transitions. The edges are annotated with inputs and outputs associated with the transition. Examples of FSMs can be seen in Figures 3 and 4. Each module is described in details as follows.

`test-generator`: This module implements a version of the P-method [18] for test case generation from FSMs. The method is able to generate a full fault coverage test suite and also includes optimizations to generate shorter test sequences. It is also able to increment user-defined test suites, enhancing their fault detection capability. Although we chose the fault coverage, it is not difficult to implement other test selection criteria or using external tools. The only restriction is to generate abstract test suites in a format readable by the runner.

`adaptor-generator`: The basic idea to implement adaptors was to associate the adaptor with a class and input/output symbols with methods. These associations are done with Java annotations. The annotation information is kept at runtime by the JVM, because the `runner` needs to access this information to run the tests. As a consequence, we have a simple and flexible way of creating concrete adaptors. Moreover, the `adaptor-generator` is able to create an abstract adaptor (template) from the test model. The necessary structure is generated and the tester just needs to implement input and output symbols. Another possibility is to use the adaptor class as a facade for developing complex interactions with the SUT.

`runner`: This module uses Java reflection mechanisms to call the corresponding method for each input or output being tested. This information is retrieved by the usage of Java annotations. Each input and output of a test case is called in sequence and if, in the end, all respective methods returned true, the test case passed. Otherwise, `runner` stops the test case execution and shows a fail message with the problematic input or output. Another functionality is a timeout threshold that can be assigned, limiting the execution time of each input and output.

`mediator`: This module was based on Mule ESB[2], which is a lightweight Java-based ESB that includes many functionalities to integrate existing systems. To use the module `mediator`, it is necessary that all services whose messages will be monitored be routed by mule ESB, i.e. all the communication with these services will firstly pass by the ESB. We developed this module as a web service integrated with the Mule ESB, providing operations to retrieve messages (*monitoring*). It is also possible to intercept and produce different messages to simulate certain situations (*simulation*), such as timeout occurrences and emulating unavailable services.

A Java interface can be added to an adaptor class through dependency injection (the `runner` module is responsible for adding the dependency at runtime). Thus, specific messages can be accessed and checked inside the adaptor written by the tester. Another characteristic is that other ESBs can be used without changing the developed adaptors. A new mediator may be developed for a different ESB by implementing the standard mediator interface and configuring the `runner` module.

B. Case Study

In this section, we present a case study aiming at evaluating the applicability of our testing process. We used the JStateModelTest tool to support the case study conduction. In this study, two service-oriented applications were tested: *ThirdPartyCall-SOA*, a small application presented in Section II; and *QualiPSo-Factory*, a more complex and real-world application.

ThirdPartyCall-SOA: We selected two scenarios *number of participants* and *participant's status*. The former verifies the number of participants in a call session of the TPCS service. This scenario involves different ways to increase/decrease the number of participants, besides restrictions like a maximum number of participants and session termination after removing all participants. The latter models the possible participant's status during a call session. Figure 3 depicts the FSM test model for the *number of participants* scenario.

QualiPSo-Factory: QualiPSo-Factory is a collaborative environment to support the development of open source software[3]. It has been developed as a service-oriented application by different partners from industry and academy in the context of the *Quality Platform for Open Source Software* (QualiPSo) project[4].

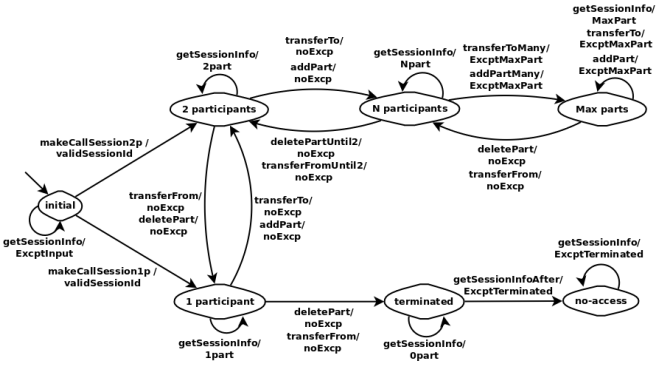

Figure 3. FSM for number of participants test scenario (*SC-1-1*).

QualiPSo-Factory is composed by a set of services, including functionalities concerning project management, issue tracker, version control, coverage testing, calendar, VOIP, and so on. The services are integrated with the core module that provides essential functionalities, such as security, notification, and semantics. The core implements two types of services, internal and external. The internal services are invoked only by other trusted services deployed inside the factory. The external services are visible for the Factory users and can be accessed by them as web services. We applied the proposed process to test external services provided by the core module (QualiPSo-Factory 0.6). In this context, four test scenarios were selected. Figure 4 depicts an FSM model for a test scenario involving access control. To conduct this study, we needed to integrate JStateModelTest tool with JUnit [19] and Maven [20].

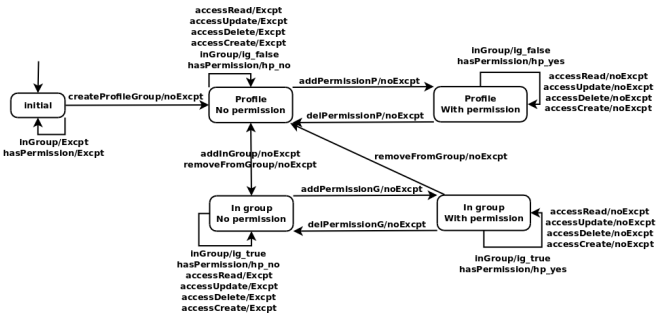

Figure 4. FSM for profile and group access control (*SC-2-3*).

Analysis of Results: Table I summarizes the data collected for the two applications. It was selected two test scenarios for ThirdPartyCall-SOA (*SC-1-1, SC-1-2*) and

[2]http://www.mulesoft.org/

[3]http://qualipso.gforge.inria.fr/

[4]http://www.qualipso.org/

Table I
DATA ABOUT TEST SCENARIOS IN *ThirdPartyCall-SOA* AND *QualiPSo-Factory* APPLICATIONS.

	APP-1: ThirdPartyCall-SOA		APP-2: QualiPSo-Factory (Core)			
Measure	**SC-1-1**	**SC-1-2**	**SC-2-1**	**SC-2-2**	**SC-2-3**	**SC-2-4**
FSM Test Model						
# states	7	5	4	5	5	5
# transitions	26	20	13	28	33	22
# input symbols	12	10	8	10	13	9
# output symbols	10	7	3	6	6	3
Concrete Adaptor						
# Lines of Code (LOC)	487	334	206	216	368	199
# methods	31	22	13	18	23	14
average # LOC for input symbol	11.9	9.5	13.8	8.7	9.6	10.8
average # LOC for output symbol	14.4	15.3	1.6	2.3	7.3	1.6
average McCabe Cyclomatic Complexity (CC)	2.4	2.2	2.2	1.8	2.3	2.0
Test Suite						
# test cases	41	31	12	28	33	30
average test length	4.4	3.6	4.6	4.0	4.6	5.3

four for QualiPSo-Factory (*SC-2-1...SC-2-4*). For each test scenario, it presents data about the FSM test model, the concrete adaptor, and the test suite.

The FSM models have from four to seven states, loops and an infinite number of possible paths. FSMs were partially specified and, as seen in Figures 3 and 4, presented a manageable size to be manipulated by graphical tools. We measured the effort necessary to implement the concrete adaptors using LOC. The total LOC for each adaptor depends on the test model size, more specifically the number of inputs and outputs. Some developed code was automatically generated and reused. Moreover, the annotations-based structure of the abstract adaptor (produced by `adaptor-generator` module) fosters the production of less complex code ($\approx$ 2.2 McCabe CC). It produced an average of $\approx$10.7 LOC to implement each input symbol and $\approx$7.1 LOC for the output symbols. Data about the concrete adaptors was collected using Eclipse Metrics[5]. The `test-generator` and `runner` were essential to automate the generation and execution of test suites. The `mediator` was used to observe and verify messages exchanged during the tests. The test suites have an average length of 4.4 input symbols for test case. The number of test cases varies according to the model complexity (number of states, transitions, inputs, and outputs).

Although the two applications have different sizes and domains, the data considering isolated test scenarios are similar. The key difference is the number of test scenarios that will increase according to the application complexity and size. We note four main results of the case study. First, the selection of test scenarios restricts the test model size, keeping the model manageable and meaningful. Moreover, it shows that FSMs can be applied to test service-oriented applications. Second, a reasonable effort for developing adaptors can be reached by using adequate structures and source-code generation. Third, test cases are automatically generated and more tests can be included by adopting alternative test methods. Fourth, the JStateModelTest tool was helpful to support the testing process usage.

[5]http://metrics.sourceforge.net/

V. RELATED WORK

MBT for service-oriented applications has been studied in recent years, giving more formality to the testing process. Heckel and Mariani [6] propose a high quality service registry that incorporates automated web service testing before the registration. GT Rules are used to specify the behavior of the web service at the conceptual level. The test case generation uses a domain-based strategy, named partition testing. The input domain is divided into subsets based on WSDL types and constraints of the GT Rules.

Keum et al. [5] propose to use EFSMs for modeling and testing web services. It is defined a procedure to derive an EFSM by means of a WSDL specification. The procedure is based on filling templates to support the EFSM modeling. Test cases are generated to cover control and data flow of the model.

Dranidis et al. [7] introduce a new approach to verify the conformance between the service implementation and a formal specification. The SXM is used to model the service behavior and its testing method is used to generate test cases. XMDL-O language is used to describe the processing functions, modeling the request, response, pre-conditions, and effects. Applying the SXM testing method, a complete set of input sequences can be generated.

Frantzen et al. [8] present a tool, called Jambition, that generates test cases for web services. The model STS is used to specify the functional aspects of the service, including variables and guard conditions. The input data is generated based on guard transitions using the constraint solver of the GNU Prolog. This data is executed and the tool randomly selects a new operation on-the-fly.

Mei et al. [17] propose a new model to describe a service choreography that manipulates data flow by means of XPath queries. The authors motivate the testing challenge so that XPath queries can raise different expectations which lead to faults in the choreography. XPath queries are represented using XPath Rewriting Graphs (XRGs). Based on Labeled Transition System (LTS), an LTS-based Choreography model is proposed with XRGs attached in transitions that represent service invocations. New test adequacy criteria are also proposed.

We note that there is no concern about establishing a process or tailoring the approaches in an MBT process. Another issue is that a test model is used for the entire service-oriented application and no discussion is provided on its complexity. Although the studies present evidences of practical applicability, the concretization of abstract test cases is not discussed.

VI. Conclusion and Future Work

In this paper, we presented an MBT process for service-oriented applications that uses state models. The process was described discussing improvements to overcome issues about testing of service-oriented applications. Finally, we present an exploratory study of the process, evaluating the applicabibility of the process and aspects of implementation.

Our proposal advances toward providing a formal, flexible and automated process to test service-oriented applications. We do not differ between single services and service composition during the process description, being applied for the both contexts. Thus, the tester does not need to change the strategy when testing different levels of integration. We achieved some practical results in the case study, presenting reasonable development effort and complexity. Regarding the tools, the development of prototypes provides evidences that a high level of automation can be reached.

As future work, we intend to define formally the linking information artifact and propose new test generation algorithms for SOAs. Other test selection criteria and the fault detection capability have also been studied. Moreover, we will investigate the process in the context of other SOA technologies, such as BPEL, WS-CDL, and REST.

Acknowledgments

This work was partially financially supported by FAPESP/Brazil (Grant 2009/01486-9) and CNPq/Brazil (Grant 474152/2010-3).

References

[1] MacKenzie et al., "OASIS reference model for service oriented architecture 1.0," 2006. [Online]. Available: http://docs.oasis-open.org/soa-rm/v1.0/

[2] G. Canfora and M. Di Penta, "SOA: Testing and self-checking," in *International Workshop on Web Services - Modeling and Testing*, 2006, pp. 3–12.

[3] Dalal et al., "Model-based testing in practice," in *International conference on Software engineering*, 1999, pp. 285–294.

[4] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., 2006.

[5] Keum et al., "Generating test cases for web services using extended finite state machine," in *IFIP International Conference on Testing of Communicating Systems*, 2006, pp. 103–117.

[6] R. Heckel and L. Mariani, "Automatic conformance testing of web services," in *International Conference on Fundamental Approaches to Software Engineering*, 2005, pp. 34–48.

[7] Dranidis et al., "Formal verification of web service behavioural conformance through testing," *Annals of Mathematics, Computing & Teleinformatics*, vol. 1, no. 5, pp. 36–43, 2007.

[8] Frantzen et al., "On-the-fly model-based testing of web services with jambition," in *International Workshop on Web Services and Formal Methods*, 2008, pp. 143–157.

[9] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines - a survey," *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090–1123, 1996.

[10] Hierons et al., "Using formal specifications to support testing," *ACM Computing Surveys*, vol. 41, no. 2, pp. 1–76, 2009.

[11] M. P. Papazoglou and W.-J. Heuvel, "Service oriented architectures: approaches, technologies and research issues," *The International Journal on Very Large Databases*, vol. 16, no. 3, pp. 389–415, 2007.

[12] M. N. Huhns and M. P. Singh, "Service-oriented computing: Key concepts and principles," *IEEE Internet Computing*, vol. 9, no. 1, pp. 75–81, 2005.

[13] Jordan et al., "OASIS web services business process execution language (WSBPEL) v2.0," 2007. [Online]. Available: http://docs.oasis-open.org/wsbpel/2.0/

[14] Kavantzas et al., "Web services choreography description language version 1.0," 2005. [Online]. Available: http://www.w3.org/TR/ws-cdl-10/

[15] OSA, "Parlay x web services," 2009. [Online]. Available: http://www.3gpp.org/ftp/Specs/html-info/29199-01.htm

[16] A. Pretschner and J. Philipps, "Methodological issues in model-based testing," in *Model-Based Testing of Reactive Systems*, 2004, pp. 281–291.

[17] L. Mei, W. K. Chan, and T. H. Tse, "Data flow testing of service choreography," in *Symposium on the Foundations of Software Engineering*, 2009, pp. 151–160.

[18] A. Simao and A. Petrenko, "Fault coverage-driven incremental test generation," *Computer Journal*, vol. 53, pp. 1508–1522, 2010.

[19] JUnit, "Junit.org resources for test driven development," 2011. [Online]. Available: http://www.junit.org

[20] ASF, "Apache maven project," 2011. [Online]. Available: http://maven.apache.org