Supporting Test-Driven Development of Web Service Choreographies

Felipe Besson, Pedro M.B. Leal, Fabio Kon, Alfredo Goldman, Dejan Milojicic

Supporting Test-Driven Development of Web Service Choreographies

Felipe M. Besson, Pedro M.B. Leal,
Fabio Kon and Alfredo Goldman
Department of Computer Science
University of São Paulo
{besson, pedrombl, kon, gold}@ime.usp.br

Dejan Milojicic
Hewlett Packard Laboratories
Palo Alto, USA
dejan@hpl.hp.com

***Abstract*—Choreographies have been proposed as decentralized and scalable solutions for composing web services. Nevertheless, inherent characteristics of SOA such as dynamicity, scale, and governance issues make the automated testing of choreographies difficult. The goal of our research is to adapt the automated testing techniques used by the Agile Software Development community to the SOA context. To achieve that, we aim to develop software tools and a methodology to enable Test-Driven Development (TDD) of web service choreographies. In this paper, we present our first steps in that direction, a software prototype composed of *ad hoc* automated test case scripts for testing a choreography. Based on qualitative and quantitative assessments performed on our prototype, we derived the requirements and challenges for developing a generic automated testing framework for supporting TDD of web service choreographies.**

I. Introduction

Service-Oriented Computing has been considered the new generation of distributed computing, being widely adopted. Service-Oriented Architecture (SOA) aims at the implementation of Service-Oriented Computing by using web services as the building block of applications.

Composability of services is one of the SOA principles, however, few approaches for composing services have been proposed. Orchestration is a centralized approach for service composition. Although straightforward and simple, its centralized nature leads to scalability and fault-tolerance problems. To face this problem, choreographies of web services have been proposed as a decentralized scalable composition solution.

In spite of all the benefits and advantages of web service compositions, the automated testing of composed services has not yet received the needed attention. There are few techniques and tools directly applicable for testing these systems because of the dynamic and adaptive nature of SOA [3].

Some tools, such as SoapUI[1] and WebInject[2] have been developed for testing atomic services. Since composed services are accessible as atomic services (from the user perspective), these tools can be used in larger scopes. Nevertheless, on such approach, both orchestration and choreography are taken as black-boxes, preventing the use of testing strategies such as unit and integration tests. In the unit testing approach, each service participating in a composition is taken as a unit, while on the integration testing approach, the interaction among these services must be exercised and verified.

[1]www.soapui.org
[2]www.webinject.org

Because of some inherent characteristics of service compositions such as dynamicity, adaptiveness, cost of testing (in terms of time, human effort, and resources) and the impossibility to exercise third-party services in testing mode, there is a lack of tools to apply such testing strategy [1], [3].

The goal of our research is to adapt the automated testing techniques used by the Agile Software Development community to the SOA context. Therefore, we aim to develop a testing framework for supporting Test-Driven Development (TDD) of web service choreographies. In addition, our framework must provide mechanisms for enacting (starting) and stopping a choreography automatically on a cloud environment. This will be achieved by communicating with a middleware that manages the nodes of a choreography in virtual machines of a given cloud infrastructure, e.g., Amazon EC2 or Open Cirrus. This middleware is also being developed by our research group.

As our initial effort to achieve these goals, we developed a software prototype for automated testing of choreographies. This tool consists of: (i) bash scripts for enacting an *ad hoc* choreography automatically on a distributed environment and (ii) JUnit test scripts for applying automated unit, integration, and acceptance tests in the running choreography.

The paper is organized as follows: Section II presents and discusses related work in the field. In Section III, we describe our software prototype for automated testing of running choreographies. Then, in Sections IV and V we present qualitative and quantitative assessments. Finally, in Section VI, we draw our conclusions and describe the ongoing and future work.

II. Related Work

Since we are also interested in testing the components of choreographies, i.e., individual services, we started studying the existing software tools for automated testing of atomic services.

SoapUI is developed in Java and provides mechanisms for functional, regression, and performance tests. From a valid Web Service Description Language (WSDL) specification, the SoapUI tool provides features to build automatically a suite of unit tests for each operation and a mock service to simulate the web service under testing. It also provides mechanisms to measure test coverage.

Due to the distributed and dynamic nature of orchestrations and choreographies, there are yet few tools for testing and monitoring the services participating on such compositions. BPELUnit [5] provides mechanisms for specifying, organizing, and executing tests for a Business Process Execution Language (BPEL) process. Its goal is to exercise the internal behavior of such processes, validating its outputs by predefined inputs. In the context of choreographies, there are even fewer tools than for orchestrations. Pi4SOA[3] is a software tool for modeling choreographies in WS-CDL by producing the global model and, then, a BPEL specification for each participant, describing their role in the choreography. Once modeled, it is possible to validate the flow among the web services by simulation. This way, Pi4SOA provides design-time mechanisms to verify the global model specified in WS-CDL.

An initial effort in understanding the current scenario of testing techniques for orchestrations and choreographies was conducted by Bucchiarone [1]. Later, a more comprehensive survey covering SOA testing was conducted by Canfora and Di Penta [3]. Both works discuss and present alternatives for testing web service compositions based on testing strategies applied to traditional client/server systems.

Acceptance testing aims at verifying the behavior of the entire system or a complete functionality. It can be performed by taking the composition as an atomic service. In this situation, black-box tests and tools that can be applied are equivalent to atomic services [3]. In the unit testing approach, each participant is a unit to be tested. For choreographies, the expected behavior for each partner is defined by its role in the choreography. Thus, black-box techniques can be applied for validating this behavior against this specification role.

In the integration testing approach, the interaction among components (services) must be exercised and verified. Nevertheless, the lack of information about certain partners and the impossibility of exercising some third-party services prevent the integration tests. In the SOA context, through the dynamic binding property, the endpoints of a participating service are chosen dynamically. Such property can raise the integration test costs [2] since strong criteria might require testing all possible endpoints.

Model-Based Testing (MBT) can be an alternative to derive integration test cases. MBT refers to an approach to derive test cases from the exploitation of formal models. Some works in this direction try to derive test cases automatically from choreography specifications, applying algorithms defined for conformance checking [4]. Some tools have been developed to convert choreography models into UML diagrams, and then, derive test cases from these diagrams [6].

Zhou et al. have proposed a new approach for the validation of the choreography model [8], [7] by checking a global model written in WS-CDL to ensure the quality of its design. First, the choreography is parsed into a data-object graph. Then, through relational calculus, static validations are applied. Finally, the graph paths are simulated to validate the dynamic aspects of the choreography. A test framework generates automatically test inputs for these paths based on constraints defined by assertions.

Differently from these works, our approach focuses on supporting the developer in the task of writing a test suite in conformance to the agile methods practices. Based on customer requirements and knowledge of the internal architecture of the system, agile developers create a test suite composed of hundreds or thousands of automated tests that try to achieve a large code coverage serving as a safety net for refactoring and code evolution.

Our approach can also be seen as complementary to MBT testing in the sense that we are developing a framework and tool to facilitate the creation and execution of automated tests. Whether the test cases are defined by a human programmer or by an MBT algorithm is orthogonal to our work.

III. Software Prototype

Our prototype consists of *ad hoc* bash scripts for a choreography enactment, JUnit test cases for automated testing of the running choreography, and a user interaction prompt for executing the scripts and tests. In this section, we first present and explain the choreography developed and then, we present our automated test scripts and approaches for applying unit, integration, and acceptance tests on the running choreography.

A. The tested choreography

To validate our prototype, we designed and implemented a simple choreography for booking a trip on OK (Open-Knowledge)[4]. The choreography participants were essentially SOAP/WSDL services and RESTful web services.

A user plans to take a trip and informs the *traveler* service where and when to go. After ordering a trip through this choreography, the user can reserve an e-ticket, and finally, confirm (book) or cancel it. Initially, *traveler* invokes *travel agency*, which searches for the required flight on the *airline*. After selecting a flight, *traveler* requests a trip reservation to *travel agency*, which requests a flight reservation to the *airline*. After these two interactions, a user can request the *traveler* to cancel the reservation or to book it. Since this process of booking the trip is more complex than the previous one, we describe this process in Figure 1 using BPMN2[5].

The process of booking a trip starts with the user requesting this operation to the *traveler service*, which in Figure 1 is represented by a white envelope. Then, the *traveler* service makes a book trip request to the *travel agency*, which calls the *acquirer* to check whether the user can afford the flight and its services or not. The *acquirer service* notifies the purchase refusal to the *travel agency* and *airline services* if

[3]sourceforge.net/projects/pi4soa

[4]OpenKnowledge project: http://www.openk.org

[5]http://www.bpmn.org

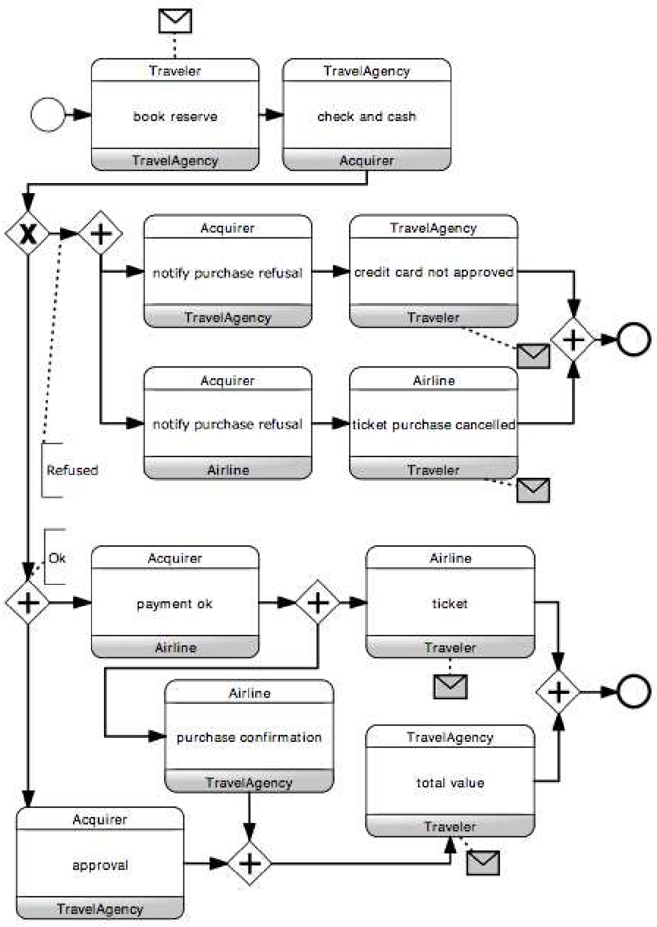


Fig. 1. Process of Book Trip operation

the user cannot afford the trip. In this case, these services send messages to the *traveler* reporting the refusal.

Otherwise, the *acquirer* service sends a payment confirmation to the *travel agency* and *airline* services. After that, the *airline* service confirms the flight price with the *travel agency* and sends the e-ticket to the *traveler*. After receiving the confirmation from the *acquirer* and the *airline*, the *travel agency* sends to the *traveler* a report (statement) with the total price paid. Finally, the *traveler* sends this response to the user which in Figure 1 is represented by a gray envelope.

B. Implemented Test cases

We developed automated test cases for applying the studied techniques and strategies on our choreography. All tests were developed using the JUnit framework and can be automatically compiled and executed by our software prototype.

1) Unit tests: In choreography context, services are considered the units for unit testing. Thus, our unit tests validate the service behavior by verifying each provided functionality. In our current prototype, to test SOAP web services, a Java SOAP client (developed using JAX-WS[6]) needs to be developed for each service endpoint (i.e., the client is specific for each endpoint). Once developed, the tests use this client to invoke the services. Thanks to the inherent flexibility of RESTful services, we developed a generic REST client (i.e., it is not restricted to a specific endpoint). Figure 2 shows two unit test cases for the *airline* web service.

[6]http://jax-ws.java.net

```
public class AirlineWSTest {

    private AirlineWSService service;
    private AirlineWS stub;

    final String TA_NAME = "Agile Travels";
    final String RESERVATION = "R3153-1|2000";
    final String USER = "John Locke";

    @BeforeClass
    public static void publishAirlineService() {
        Bash.deployService("airline");
    }
    ...

   @Test
   public void shouldFindFlight() {
       flight = stub.getFlight(destination, date);

       assertEquals("3153", flight.getId());
       assertEquals("Milan", flight.getDestination());
       assertEquals("12-21-2010", flight.getDate());
       assertEquals("09:15", flight.getTime());
   }

   @Test
   public void shouldBeAnAuthorizedTravelAgency() {
       assertTrue(stub.isTravelAgencyAuthorized(TA_NAME));
   }
```

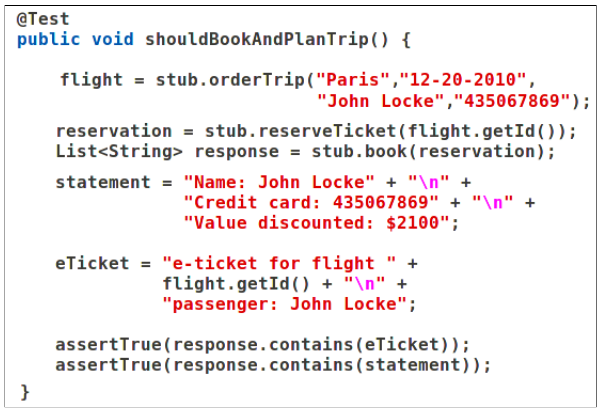
Fig. 2. Unit test case for Airline service

Airline service is a stateless service for searching flights. Any other services or end-users can search for flights. However, this service just allows authorized travel agencies to reserve and book flights. Our unit test cases focus on testing these functionalities. As seen in Figure 2, we first search for a specific flight and, then, all information retrieved is validated against the expected ones.

2) Acceptance tests: Differently from other testing strategies, acceptance tests verify the behavior of the entire system or complete functionality. From the point of view of an end-user, the choreography is available as an atomic service. Thus, the acceptance test validates the choreography as a unit service, testing a complete functionality. In this context, this type of test is similar to the approaches of unit testing using the black-box model, and there is no need to know how the service is implemented.

On our approach, a developer specifies the tests by calling a service that activates the choreography. Before the execution, the developer needs to execute a script that enacts the choreography and deploys the services. Then, the tests are executed and the actual results are compared with the expected output values.

In the choreography example explained above, the *traveler* peer is the service that triggers it. Therefore, to test the Order Trip Operation, the developer calls the method on the *traveler* web service and compares the returned object properties with the expected ones. The entire flow performed by a user that completes his/her plan can be seen in Figure 3.


```
@Test
public void shouldBookAndPlanTrip() {

    flight = stub.orderTrip("Paris","12-20-2010",
                            "John Locke","435067869");

   reservation = stub.reserveTicket(flight.getId());
   List<String> response = stub.book(reservation);

   statement = "Name: John Locke" + "\n" +
               "Credit card: 435067869" + "\n" +
               "Value discounted: $2100";

   eTicket = "e-ticket for flight " +
             flight.getId() + "\n" +
             "passenger: John Locke";

   assertTrue(response.contains(eTicket));
   assertTrue(response.contains(statement));
}
```

Fig. 3. Acceptance Test example

3) Integration tests: Integration tests intend to solve the problems found when unit tested components are integrated. Their goal is to verify the unit interfaces and interactions among system components. Based on the discussion presented by Bucchiarone [1] for integration testing of choreographies, we defined an approach for applying integration tests. After all services have been tested at the unit level, the approach focuses on integrating each service at a time in the choreography. Once a service is integrated, the choreography is enacted by the developer. Then, using runtime monitoring of the choreography, the framework verifies whether the service just integrated behaves as expected. This step is achieved by checking the messages sent by that component. For each message, its name, destination, and content are compared with the expected values.

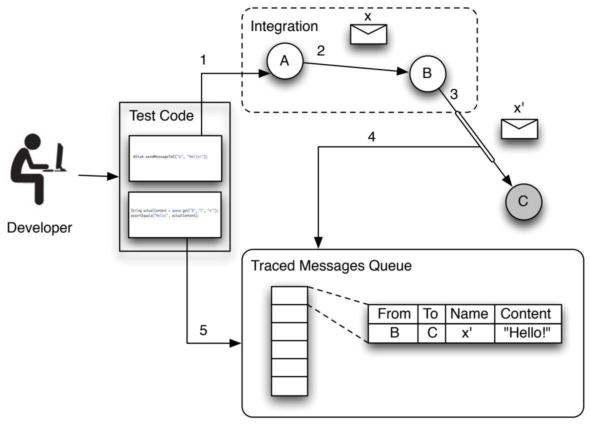


Fig. 4. Integration test flow example

We developed and included in our prototype an *ad hoc* framework that implements the steps of this approach. Figure 4 shows an example of how the framework works. As depicted, the developer is integrating the A and B services. In the first step, the developer activates the choreography, invoking service A. During the choreography enactment, messages are exchanged between services A and B (step 2). Our framework monitors the choreography, collecting the output messages from B and storing them in a queue. This is performed in the third and fourth steps. When the choreography is over, the tool compares what the developer has specified with the collected data (step 5).

The framework is still coupled to OK choreographies, but one of our goals is to make this tool generic. The complete source code of our prototype including 16 unit tests, 17 acceptance tests, and 8 integration tests can be downloaded as an open source software from http://ccsl.ime.usp.br/baile/VandV.

IV. Qualitative Assessment

SoapUI provides functionalities for the automatic generation and execution of test cases from a valid URI; but the tester must still fill in the XML-Soap envelope, which can be cumbersome. As depicted in Figure 2, on our unit tests, the tester interacts with the web services under testing in an object-oriented way, by invoking methods instead of manipulating XML-Soap envelopes. However, our web service client generation is not fully automatic. In our ongoing work we are seeking to combine the automatic client generation of SoapUI with the high-level, object-oriented testing scheme of our approach (free of the burdens of XML hacking).

Since SoapUI does not provide support for integration tests, we developed a new approach (described in Section III-B3). To illustrate the benefits of our approach, we implemented a use case based on our choreography example. Considering that the *travel agency* can search for flights in more than one airline, suppose that another airline service is integrated into the choreography. This new service is from a Brazilian provider, and consequently, it charges all tickets in BRL (Brazilian Real), but our choreography only works with USD (United States Dollar). Initially, all unit tests for this new component pass and the incompatible currency is not noticed at this stage. Then, the integration test detects that the *acquirer* service charged the ticket price incorrectly since its service does not apply currency conversions. In this example, our approach reveals the error and points to where, in the choreography, it could be fixed. To correct the error, one must add, for instance, a currency converter service between the travel agency and acquirer services.

In the absence of our approach, one is limited to acceptance testing strategies to validate a service integration. In this case, the choreography is taken as a black-box, preventing the tester to discover where exactly the error occurred. With our approach, after identifying a problem, we can start collecting and analyzing the messages exchanged to isolate the problem. In our current prototype, for collecting a specific message, the whole choreography functionality flow must be performed. Thus, our integration tests take, at least, the same time that an acceptance test for that choreography flow takes. To improve our approach, we intend to develop a mechanism that can stop the choreography after collecting the desired message.

V. Quantitative Assessment

As described previously, our integration testing approach must collect the messages exchanged among the services.

Such procedure might cause an overhead in the choreography execution. We have conducted a quantitative assessment to evaluate possible overheads. In this assessment, we first deploy our choreography example (see Section III-A) on a cluster. Each service choreography was allocated on a dedicated node with a Pentium 4, 3.00GHz processor, with 1GB of RAM, running GNU/Linux and connected to a 100Mb/s LAN.

The goal of this assessment was to compare the execution time of a choreography functionality using and not using our approach for monitoring the choreography at runtime. We have chosen the *order trip* functionality for the experiment. In this execution, 4 messages are exchanged among the services. We measured the execution time of 1, 2, 4, 8, and 16 sequential *order trip* executions, collecting and not collecting all the 4 messages exchanged. First, each sequence was executed 30 times, e.g., in the case of 8-sequential *order trip* executions, we had 30 samples, each one with the 8-sequential executions. Then, we extracted the average and standard deviation of these samples. Our results are presented on the Table I.

TABLE I
INTERCEPTION MESSAGES OVERHEAD

# of executions	average w/o monitoring	average w/ monitoring	overhead
1	4.51 (0.59)	4.6 (0.6)	0.08
2	9.35 (0.7)	9.6 (0.89)	0.25
4	19.28 (0.77)	19.14 (0.69)	-0.14
8	38.51 (0.86)	38.38 (0.54)	-0.13
16	77.32 (1.21)	77.93 (1.52)	0.62

The second and third columns represent the average times and their standard deviation (in parentheses) for exchanging the 4 messages without monitoring and monitoring the choreography, respectively. As can be observed in the last column, the overhead is considerably smaller than the standard deviation. Based on these measurements, the monitoring overhead can be negligible. Since we do not need to store the whole XML-Soap message but only its content, the message monitoring process is relatively fast. Nevertheless, new assessments must be applied to analyze the overhead when multiple requests are performed in parallel. In the future, we intend to evaluate the overhead behavior in a more realistic choreography (with hundreds of nodes and thousands of messages exchanged).

VI. CONCLUSIONS AND FUTURE WORKS

In our ongoing work, one of our goals is to develop a TDD methodology that will help developers and project leaders to deal with the key-issues involved in testing large-scale, distributed, Internet systems and will guide them in the production of effective and efficient test suites for web service choreographies. To achieve this goal, we first intend to develop an open source testing environment to support the methodology proposed. Based on the results of the current work and on the lessons learned from the prototype development, we can derive some requirements and challenges that must be faced to achieve our future goals.

A. Requirements and Challenges

The requirements for the proposed environment include (1) the definition or adaptation of a simple language for specifying the deployment of choreographies across the network in a reproducible way, (2) the construction of a tool for parsing specifications written in this language and enacting the choreography, and (3) the development of a framework for writing and executing automated tests for service choreographies.

To scale up existing efforts for testing choreographies and to meet these requirements, we must overcome some obstacles. First, we have to deal with the lack of observability: since some services export only their interfaces, this prevents white-box testing in some cases. Some inherent characteristics of service compositions such as dynamicity, adaptiveness, third-party rules, and governance issues must also be solved to automate the integration tests. Besides, frameworks for automated testing must be generic, i.e., not coupled to a specific choreography but applicable in any choreography.

Finally, some environment issues such as the decentralized flow of information, multiple party communication, and parallelism must be adequately treated by our future framework for writing and executing automated tests for service choreographies.

ACKNOWLEDGMENT

This research was funded by HP Brasil under the Baile Project and from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement number 257178 (project CHOReOS - Large Scale Choreographies for the Future Internet).

REFERENCES

[1] H. M. A. Bucchiarone and F. Severoni. Testing service composition. In *8th Argentine Symposium on Software Engineering (ASSE'07)*, Argentina, 2007.

[2] G. Canfora and M. Di Penta. Testing services and service-centric systems: challenges and opportunities. *IT Professional*, 8(2):10 –17, march-april 2006.

[3] G. Canfora and M. Di Penta. Service-oriented architectures testing: A survey. In A. De Lucia and F. Ferrucci, editors, *Software Engineering*, volume 5413 of *LNCS*, pages 78–105. Springer, 2009.

[4] L. Frantzen, M. N. Huerta, Z. G. Kiss, and T. Wallet. On-The-Fly Model-Based Testing of Web Services with Jambition. In *5th International Workshop on Web Services and Formal Methods – WS-FM 2008*, 2009.

[5] P. Mayer and D. Lübke. Towards a BPEL unit testing framework. In *Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications*, TAV-WEB '06, pages 33–42, New York, NY, USA, 2006. ACM.

[6] A. Stefanescu, S. Wieczorek, and A. Kirshin. MBT4Chor: A model-based testing approach for service choreographies. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, ECMDA-FA '09, Berlin, Heidelberg, 2009.

[7] Z. Wang, L. Zhou, Y. Zhao, J. Ping, H. Xiao, G. Pu, and H. Zhu. Web services choreography validation. *Service Oriented Computing Applications*, 4, December 2010.

[8] L. Zhou, J. Ping, H. Xiao, Z. Wang, G. Pu, and Z. Ding. Automatically testing web services choreography with assertions. In *Proceedings of the 12th international conference on Formal engineering methods and software engineering*, ICFEM'10, pages 138–154. Springer-Verlag, 2010.