Towards Verification and Validation of Choreographies*

Felipe M. Besson, Pedro M. B. Leal, Fabio Kon

Department of Computer Science
Institute of Mathematics and Statistics
University of São Paulo (USP)

{besson,pedrombl,fabio.kon}@ime.usp.br

Technical Report No: RT-MAC-2011-01

Abstract. *This technical report presents a study about software tools and approaches for testing web service compositions. Our goals consist on understanding the current scenario of Verification and Validation (V&V) activities (focusing on automated tests) in the SOA context. We also present a prototype we developed to illustrate different types of automated test case scripts for testing web service choreographies.*

Contents

1 Introduction **3**

2 Conceptualization and terminology **3**

2.1 Testing Strategies 3

2.2 Testing Techniques 4

2.3 Testing Tools 4

3 Testing tools for SOA **5**

3.1 Atomic Web services testing 5

3.2 Web Services Composition Testing 7

3.2.1 Testing Orchestrations 7

3.2.2 Testing Choreographies 8

4 Methodologies for testing web service compositions **8**

4.1 Testing strategies and techniques 8

4.1.1 Orchestration 9

4.1.2 Choreography 9

4.2 Practical methodologies 10

4.2.1 The Savara project 11

*The research leading to these results has received funding from 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).

5 Our Prototype **12**
5.1 The tested choreography 12
5.2 Proposed test cases 14
5.2.1 Acceptance tests scenarios 16
5.2.2 Airline unit tests examples 18
5.3 Implemented test cases 18
5.3.1 Unit tests 18
5.3.2 Acceptance tests 21
5.3.3 Integration tests 21

6 Conclusions and Future Works **24**

7 Acknowledgements **24**

1. Introduction

Service-Oriented Computing has been considered the new generation of distributed computing, being widely adopted. In such context, SOA (Service-Oriented Architecture) consists on an architecture aiming at the implementation of Service-Oriented Computing fundamentals by using web services as the building block of applications [Hew09].

Since the composability of services is one of the SOA principles [Erl07], some approaches for composing services have been proposed. Orchestration is a centralized approach of service composition. Some standards such as WS-BPEL (Web Services Business Process Language) [OAS02] implement this kind of composition. Although straightforward and simple, its centralized nature has scalability and fault-tolerance problems. To face this problem, choreographies of web services have been proposed as decentralized composition solutions. Some languages, such as WSCI (Web Service Choreography Interface) [W3C02], WS-CDL (Web Services Choreography Description Language) [W3C04], and BPMN2 (Business Process Model and Notation) [OMG10] have been also proposed as choreography standards.

In spite of the importance and advantages of web service compositions, just lately the verification and validation of atomic and composed services have received the needed attention [ABS07]. Nevertheless, the dynamic and adaptive nature of SOA makes most of the existing techniques not directly applicable for testing services and service-centric systems [CDP09].

This technical report is a study to understand the current scenario of Verification and Validation (V&V) activities in the context of web services compositions (more specifically choreographies) with focus on automated testing techniques and strategies. V&V is a wide discipline that evolves also monitoring and governance issues, but these latter topics are out of the scope of this document. Then, the goal of this work is to document and study software tools, methodologies and techniques for writing and executing automated test of web services and web services composition.

This document is organized as follows: Section 2 presents a brief conceptualization of the existing test strategies and techniques for traditional systems (following the client-server model). Section 3 describes the main software tools for testing services and service composition, while methodologies for creating automated tests for web services compositions are addressed on Section 4. In Section 5, we present a prototype composed of *ad hoc* scripts for automated choreography tests. Finally, in Section 6, we draw our conclusions and describe ongoing and future work.

2. Conceptualization and terminology

In this section, we introduce some concepts and terminologies related to software testing. Despite this brief conceptualization refers to testing of traditional systems (client-server model), its understanding makes needed for the testing approaches of web services composition presented in this technical report.

2.1. Testing Strategies

During a software development project, different testing strategies can be applied depending on the stage of development. Unit tests focus on the smallest unit of software present in the source code. These tests verify the behavior of a single class or method, and are not directly related

to the requirements of the project, except when a key chunk of business logic is encapsulated within a specific class or method [Mes07].

At the other extreme, there are acceptance tests to verify the behavior of the entire system or complete functionality. They typically correspond to the execution of scenarios present in the use cases, features, or user stories. They do not depend on the implementation. Normally, they are slower than the other test strategies because they exercise all layers of the system, accessing the real components (mock components are not used) [Mes07]. These tests are also called functional, customer, or end-users tests.

Between these two approaches of testing, there are the integration tests, which attend to solve the problems produced when unit tested components are integrated. Their goal is to verify the unit interfaces and interactions. A set of integration tests must be built to explicit goal of exercising these interactions and not the unit functionalities [Del97]. In this context, integration tests are also called component tests, which aim to verify components consisting of groups that collectively provide some service [Mes07]. There are some strategies to perform this integration [Pre01]:

- Top-down: the integration was performed from the main module. Initially, all components that this module depends on are mocked or stubbed, then, these dependencies are substituted incrementally by its real implementation until the system is totally integrated;
- Bottom-up: the integration starts in the atomic modules (i.e., components at the lowest levels in the program structure). These components are grouped in clusters. Once all components of a cluster are integrated, the cluster is integrated. Since second cluster integration contain real components, no stubs are needed;
- Big bang: all components are combined at once and tested as a whole.

2.2. Testing Techniques

Depending on the goals of the test, availability of resources, and other factors, test strategies are combined with different test techniques. When functional or black-box testing are used, the entity (method, class or system) under testing is considered as a complete entity and the internal structure is ignored [Mye04]. Then, when some valid and invalid data are provided to the entity under testing, it just verifies whether the actual results are compatible with the expected ones. On the other hand, in the white-box testing technique, internal structure of program aspects as data flow and its internal logic are verified and validated [Mye04]. For this reason, this technique is also called "structural test".

2.3. Testing Tools

Currently, there are many automated testing frameworks. Open source xUnit[1] tools, such as JUnit, SUnit, and CPPUnit, are widely used among developers. The usage of these tools is particular to certain programming languages. Long-established languages have their own xUnit tool, such as Java and C.

There are also mocking tools, such as Mockito [2] and EasyMock [3], which support these automated test frameworks to generate double components easily. Some tools, such as Cu-

[1] xUnit tools: http://xprogramming.com/software.htm
[2] Mockito: http://mockito.org/
[3] EasyMock: http://easymock.org/

cumber [4], Jbee [5], and Fitness [6] favor the automated acceptance test writing. They focus on the readability and the ease understanding for non-technical or business customers. Then, these tools encourage customers to write automated acceptance tests by themselves.

3. Testing tools for SOA

Since web services are the building blocks of SOA [Hew09], we started our study investigating those tools for testing web services as a unit or, in other words, for testing the functionalities of a web service individually. Some tools used for testing web service compositions were also studied and analyzed.

3.1. Atomic Web services testing

The are many software tools for different test techniques at web service level. In Table 1, we present some studied tools with its types of license. Nevertheless, we focus our efforts on tools for functional testing.

Test Tool	Vendor	URL	License
GH Tester	Green Hat	http://www.greenhat.com	Proprietary
HP Service Test	HP	http://www.hp.com	Proprietary
Lisa	iTKO	http://www.itko.com	Proprietary
Matador Tech Tester	Matador Tech Corp	http://www.matadortech.com	Proprietary
SilkTest	Borland	http://www.borland.com/us/products/silk/silktest	Proprietary
RESTClient	–	http://code.google.com/p/rest-client	Apache 2.0
SoapSonar	CrossCheck Networks	http://www.crosschecknet.com	Proprietary
SOATest	Parasoft	http://www.parasoft.com	Proprietary
SoapUI	Eviware	http://www.soapui.org	LGPL
TestMaker	PushToTest	http://www.pushtotest.com	GPL
WebInject	Corey Goldberg	http://www.webinject.org	GPL

Table 1. Web Services Testing tools

SoapUI [Evi10] is developed in Java and provides mechanisms for functional, regression, and performance tests. From a valid WSDL (Web Service Description Language), this tool can provide (i) features to build automatically a suite of unit test for each operation, and (ii) a mock serviceto simulate the web service under testing. Besides, the mocked service can be deployed by the tool, making the WSDL interface available like the real service, but only the mocked functionalities are invoked. Finally, SoapUI has mechanisms to measure the test coverage.

In spite of its name suggests restriction to SOAP/WSDL web services, some mechanisms are provided to test other kind of services like REST (Representational State Transfer) and JMS (Java Message Service). All these features are provided by the Open Source version, but a commercial version of SoapUI also provides more resources such as features to apply performance tests in large scale systems.

For testing SOAP/WSDL web services, as mentioned before, SoapUI provides a mechanism that automatically generates a skeleton of test for the operations presented in the WSDL. Although it automates the test creation, the produced test cases are incomplete, as seen in Figure 1 input and output parameters values must be provide manually (fields with “?” character).

[4] Cucumber: http://cukes.info/
[5] http://sites.google.com/site/jbeetest/
[6] http://fitnesse.org/

```
1 <soapenv:Envelope
2         xmlns:soapen="http://schemas.xmlsoap.org/soap/envelope/"
3         xmlns:test="http://www.example.org/test/">
4         <soapenv:Header>
5            <headerPart>?</headerPart>
6         </soapenv:Header>
7         <soapenv:Body>
8            <test:operation>
9               <test:ChoiceElem>
10                 <!--You have a CHOICE of the next 4 items at this level -->
11                 <field1>?</field1>
12                 <field2>?</field2>
13                 <field3>?</field3>
14                 <!-- 0 to 3 repetitions:-->
15                 <occurElem>?</occurElem>
16                 <AllElem>
17                    <!--You may enter the following 2 items in any order-->
18                    <AllElem1>?</AllElem1>
19                    <AllElem2>?</AllElem2>
20                </AllElem>
21              </test:ChoiceElem>
22              <secondInput>?</secondInput>
23           </test:operation>
24        </soapenv:Body>
25 </soapenv:Envelope>
```

Figure 1. Skeleton generated by soapUI [BBMP09]

To improve this feature of SoapUI, WS-TAXI [BBMP09] was proposed. Its goal is to automatically fill these empty fields by deriving XML instances from an XML schema. With this tool, test cases are generated from all possibilities of data combination for skeletons produced by SoapUI. In the example of Figure 1, WS-TAXI would generate test cases (varying the data input) to cover all possibilities of options (lines 09 – 21).

Similarly to the tools described above, WebInject [Gol10] automates the sending of messages (usually XML payloads) and verifies and validates the response messages returned by the the service under testing. It can be used to test SOAP/WSDL and REST web services, as well as any component that has HTTP interfaces (e.g., JSP, ASP, CGI, PHP, AJAX, Servlets, HTML Forms). The tests for WebInject must be written in XML, which can be considered a drawback in some situations. Because of the verbosity of XML, tests may not be clear as desired.

Some approaches are focused on REST testing. TTR (Test-The-REST) [CK09] provides mechanisms for applying Functional testings (using a black box strategy) and non-functional testing (applying performance tests). Similar to SoapUI, this tool provides features for testing the CRUD operations over REST services resources. CRUD operations in REST context normally correspond to the HTTP methods: POST (for Create), GET (for Read), PUT (for Update) and DELETE (for Delete). TTR is a script-like tool while SoapUI provides a GUI (Graphical User Interface) for tests specification and execution.

A TTR test case is composed of an identification, the URL of Resource, the HTTP Method to be executed, input data, and output data expected. A test case can be composed of other test cases, and it can assist integration test among features of some web service. But similarly to WebInject, test cases have to be written in XML.

Considering the importance of SOA, as observed in Table 1, several automated testing tools for atomic services have been developed. While some tools are dedicated to SOAP/WSDL services, others can also be used for REST and JMS services. As a drawback, in some cases, test cases must be specified in XML, sometimes making its specification more verbose and

error-prone than desired.

3.2. Web Services Composition Testing

Because of the dynamic nature of orchestrations and choreographies, there are few softwares tools for testing and monitoring the services participanting on such compositions. A systematic review about unit testing approaches for BPEL [ZAGS09] carried out in 2009 analyzed 27 papers, in which only four were related to practical frameworks. BPEL4WS Unit Testing [LSJZ05] and BPELUnit [ML06] present an architecture and a model for specifying unit tests on BPEL process. Since the BPELUnit [ML06] is the most active software, we focused on the description of this tool in the next section.

3.2.1. Testing Orchestrations

BPEL process is based on its interaction with web services. Since they are loosely coupled, it is not hard to test its implementation by creating a harness around the process under testing. With this intention, Mayer and Lübke propose an architecture for creating BPEL automated testing frameworks. This framework is divided in four layers: test specification, test organization, test execution and test results [ML06].

Test specification layer describes how the tester will express his/her tests. Two extreme techniques are proposed: (i) *data-centered approach*, which consists on the validation of data structures, for example, a predefined SOAP message is compared with that one sent by the process under test (PUT), and (2) *logic-centered approach*, in which a programming language must be used to describe all test logic. The first one is simple, but it is less expressive while the latter is more expressive, but also complex.

Test organization layer characterizes the tests arrangement. An approach can consist on grouping tests in test suites to assist the developer usage. This organization has an advantage: the usage of the same setup and tear down methods for all related tests.

Tests specifications are executed by a framework by creating a wrapper around the PUT to validate the PUT outputs and send pre-defined inputs. It can be achieved by using two different approaches. On *Simulated testing*, the BPEL process is simulated in a controlled environment assisting by an engine to perform debugging, control outputs and simulate inputs. The *Real-life testing approach* actually deploys the process and replaces the real web service partners with mocks. Both techniques depend on the specific engine implementation since they are not standardized.

The last layer is responsible for gathering the results and statistics obtained and presenting them to the end-user. A testing framework could have different metrics to assess this information. Test coverage is one of the metrics that can be used. Using the simulated testing approach, the testing framework, operating in debugging mode, can call methods from the BPEL engine during the simulation. While executing the real-life testing approach, the mocks used could log their activities for subsequent validations.

Based on the architecture described above, Mayer and Lübke report their choices for each layer to develop the BPELUnit. The developer specifies the test in XML language. To allow fast testing, SOAP details received and sent by the PUT are abstracted to the developer.

The framework supports specification of three interaction types with the PUT: one-way, two-way synchronous, and two-way asynchronous.

On the organization layer besides grouping the tests cases, it is mandatory to have the setup and teardown methods to set up and later shut down the web service partners and PUT itself.

As previously mentioned, all the choices on the execution layer depend on the used BPEL engine. BPELUnit supports these diversity with the development of adapters to each one. It uses the real-life approach which deploys the application PUT, executes the tests, and undeploys it.

3.2.2. Testing Choreographies

In the context of choreographies, there are even less tools than for orchestration. Some efforts for testing this kind of composition are found in the literature but they are mostly focused on model checking of choreographies, or data flow analysis from some structure derived from static artifacts. Therefore, none of them provide mechanisms for testing a choreography execution, which is one of our goals. Although some of those tools were also analyzed.

Pi4SOA [Pi410] is a software tool for testing interactions among participants of a choreography from a global model specified in WS-CDL (Web Service Choreography Description Language). Essentially, the purpose of Pi4SOA is modeling choreographies in WS-CDL, by producing the global model and then, a BPEL specification for each participant describing their role in the choreography modeled.

Based on some components such as partners, behaviors and interactions (message exchanges), the choreography is modeled by dragging and dropping these components, since the tool is available as an Eclipse plugin. Once modeled, it is possible to validate the flow among the web services by simulation, which is not performed using real service. Then, at design time, this tool provides mechanisms to verify the global model specified in WS-CDL.

4. Methodologies for testing web service compositions

An initial effort for understanding the current scenario of testing techniques for orchestrations and choreographies was conducted by Bucchiarone [ABS07]. Later, a more comprehensive survey to cover SOA testing was conducted by Canfora and Di Penta [CDP09]. In this section, we discuss test techniques and methodologies presented in these studies and other complementary ones. Then, we present some practical methodologies for web services composition development, focusing on its V&V activities.

4.1. Testing strategies and techniques

Some issues such as dynamicity, adaptiveness, lack of control, and cost of testing are intrinsic properties that limit the testability of service-centric systems [CDP06]. Nevertheless, based on traditional test strategies and techniques (see Section 2), some alternatives for testing service composition can be classified as presented in Figure 2.

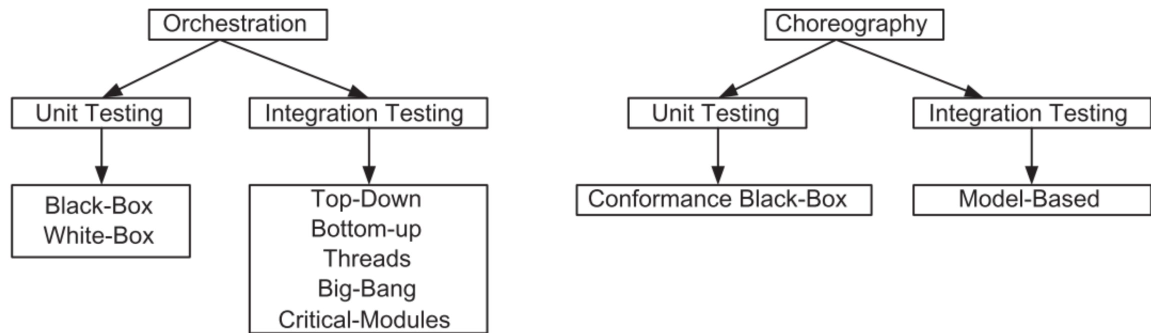

Figure 2. Alternatives for testing web service compositions [ABS07]

4.1.1. Orchestration

An orchestrator is a program, or code (e.g., a BPEL specification) that defines the behavior of a particular partner. In the unit testing context, an orchestrator can be taken as a unit. The goal of tests consists on exercising its behavior and verifying whether they match the specification or the behavior expected by the customer. Depending on the nature of the orchestrator, black-box or white-box techniques can be applied. When a WS-BPEL process is visible from an external point of view, only its WSDL interface is exposed. In such situation, the process is available as a common service, then the black-box tests that can be applied for such process are equivalent to atomic services [CDP09].

When an orchestrator is specified in BPEL and executed by a BPEL engine, this specification can be also taken as the code of the orchestrator (if the engine is correct). In this case, white-box, or implementation-based techniques can be applied. For this kind of technique, mechanisms of structural testing such as control and data flow graphs can be used, and test cases can be derived from these structures. Yuan [YLS06] proposes a graph search approach to BPEL test case generation which deals with BPEL special characteristics such as concurrency and synchronization.

Since these tests at unit level are focused on just exercising the behavior of that unit, the behavior of dependent parts must be simulated. This can be done by creating stubs or mock objects for the web services which the unit under testing communicates with. On the integration testing approach, the interaction among components must be exercised and verified, thus, some strategies for integration presented in Section 2 can be adopted. The real web services should be invoked, but exercising third-party services cannot be possible in testing mode. This is one of the most significant problem that integration testing of web services must overcome.

4.1.2. Choreography

At the choreography side, the unit tests can be applied following the same approach defined for orchestration, i.e., each participant is a unit to be tested. Differently from orchestration, 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.

Similarly to the case of orchestrations, integration tests face the same problems for choreographies. The lack of information about certain partners and the impossibility of exercising some third-party services make this kind of test more difficult. Moreover, the dynamic binding property, in which the endpoint of a participating service is chosen dynamically, can raise serious integration issues [CDP06]. Due to this property, strong criteria might require testing all possible endpoints that can be assigned to a choreography. In the example illustrated in Figure 3, each abstract interface (hotelSearch, checkFlight, getShuttleTicketPrize and getCabPrice) has some candidates to take on these roles. Because of the dynamic binding, all possible combinations of bindings (24 in total) must be integrated and exercised one-by-one.

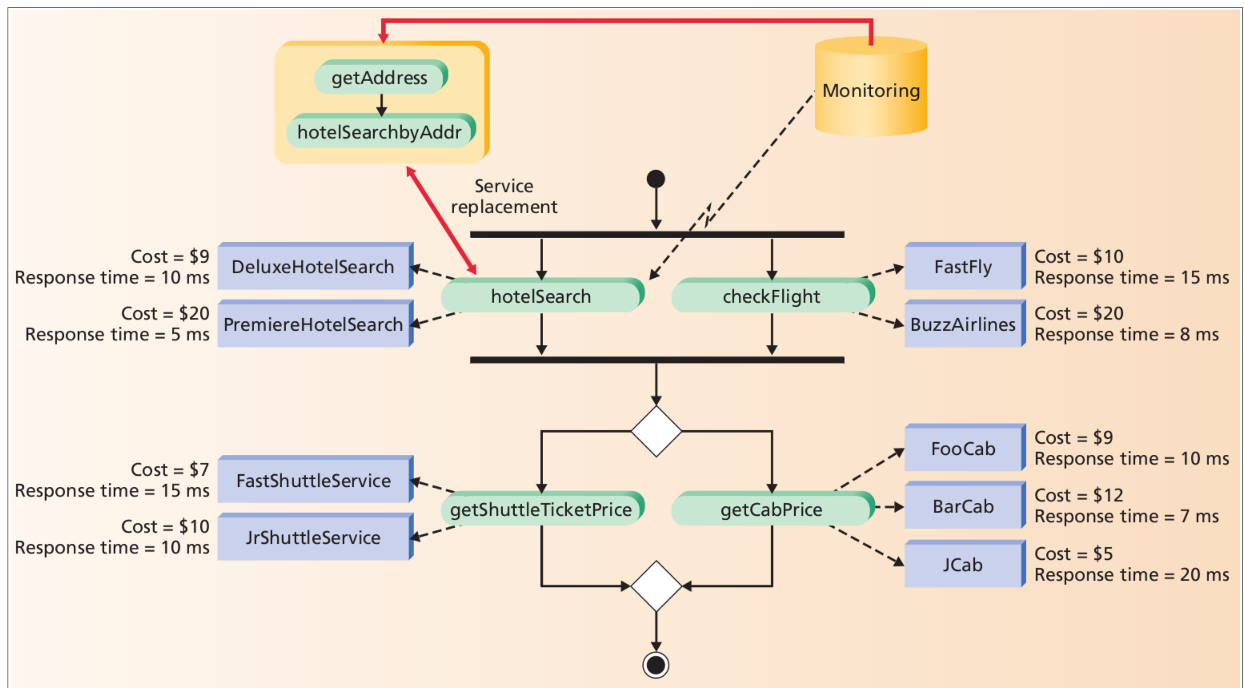

Figure 3. Dynamic binding on the integration test [CDP06]

According to Canfora and Di Penta [CDP06], some heuristics can be reduced to decrease the number of possible endpoints for testing. For instance, the test integration strategy can ignore those endpoints that violated some QoS or business rule. In the situation of Figure 3, if the service providers establish a business rule, defined in a global agreement, then the cost of the services must be within $15, and just six possibilities would be exercised.

Other approaches like MBT (Model-Based Testing) 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 from choreography specifications, applying algorithms defined for conformance checking [FHKW09]. Some tools have been developed to convert choreography models into UML diagrams, and then, derive test cases from these diagrams [SWK09].

4.2. Practical methodologies

After this study on tests for choreographies, we investigated some practical approaches for V&V. In particular, Savara methodology (described bellow) is not just a testing methodology,

but rather a methodology related to all stages of a choreography development. Nonetheless, at each stage of development, the produced artifacts are verified and validated. Furthermore the choreography monitoring (at design and runtime) consists on a significant stage of Savara. For these reasons, we decided to include a discussion about SAVARA in this technical report. In the future, we intend to propose and develop a new practical methodology that uses TDD (Test-Driven Development) [Bec02] to guide choreographies development. Since one of our goals is also supply the industry demands, existing technologies and standards like Savara must be studied and analyzed.

4.2.1. The Savara project

The Savara Project [Red10] is a set of software tools that comprises the Enterprise Architecture to support a methodology called "Testable Architecture". The methodology aims to ensure that any artifacts defined during the development life cycle of a choreography can be validated with other artifacts in previous and subsequent phases.

This methodology formally has its foundations in pi-calculus. Pi4SOA features (see Section 3.1) are used for verifying and validating some models developed in WS-CDL [Mad09]. According to Bhavish Kumar Madurai, Co-chair of the SAVARA Project, the proposed methodology is aligned to the TOGAF Iterative methodology principles [Mad09] for Enterprise Architectures. TOGAF (The Open Group Architecture Framework) [Ope10] is a collective effort of some entities such as consulting and system integrator organizations, as well as end-users. TOGAF aims at defining detailed processes, methodology, and artifacts for efficient and effective delivery of enterprise architectures for any organization regardless of the size.

During the project development, a file property records information about the artifacts defined in each phase of the software development lifecycle, and the relationships between them. This file is used by Savara tools to validate and ensure that each artifact with its dependencies is valid in respect to each other [BY10]. The development process of Savara is divided into the following phases:

1. Modeling the scenarios: Scenarios representing the business requirements are modeled based on UML use cases diagrams and documents. These scenarios correspond to a logical sequence of messages among the partners of the choreography to be developed.
2. Global Model definition: Based on the present scenarios, a global model that represents the business flow from an external perspective is developed. Thus, this model specifies the message exchange (in SOAP) among partners but not inside a specific component. This artifact can be written in WS-CDL or BPMN2.
3. Local Model: Defines the interface ("the contract") for each web service. These artifacts are used to validate the global Model in order to guarantee its correct implementation, using the Pi4SOA simulation feature. At this point, the web services can be associated within a Registry/Repository. The Local Model is a kind of orchestration process defined for each service.
4. Service Design: The contracts defined in the Local Model are designed internally. If a service represents a database, it could be defined as the database schema. At this point, for example, a BPEL specification and WSDL interfaces are generated for the services.

5. Service Implementation and Testing: Based on the artifacts generated by the previous phase, the services are implemented and BPEL specifications are deployed in BPEL Engines. Since these artifacts (e.g., BPEL specification) were generated based on models, its logic is correct, assuming that the BPEL Engine is correct. At this point, unit tests are applied in the web services.
6. Runtime: the Savara project proposes a runtime monitoring tool to ensure that business processes are executing correctly in accordance with the business requirements defined on the scenarios and global models.

5. Our Prototype

Based on the study and analysis previously presented, we developed a software prototype for automated testing of composed services. This tool consists of *ad hoc* automated test scripts developed using the JUnit [Bec10] framework. During the development, we tried to identify and apply, whenever feasible, studied testing techniques. We also developed a service choreography to validate our prototype. This work represents a first step towards building a generic testing framework and methodology for Choreographies, which is our long-term goal.

5.1. The tested choreography

We designed and implemented a simple choreography on OK (OpenKnowledge)[7], a system which facilitates the interaction among peers without a pre-run-time knowledge of who to interact with or how interactions will proceed. From the OK kernel, fully distributed peer-to-peer applications can be developed [BPB+09].

Applying these principles, we developed a choreography for planning and booking a trip based on the example provided in the WSCI specification[8]. A user plans to take a trip and informs the Traveler service where and when to go. At this point, the choreography participants consist of: Traveler, Travel Agency, Airline, and Acquirer.

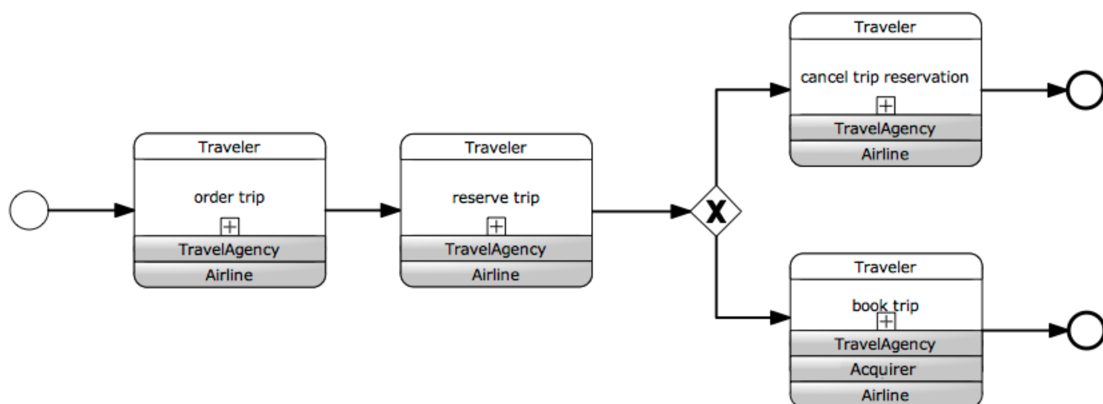

Figure 4. Choreography global view

[7]OpenKnowledge project: http://www.openk.org/
[8]WSCI official page: http://www.w3.org/TR/wsci

Through this choreography, after ordering a trip, the user can reserve an e-ticket, and finally, confirm (book) or cancel it. Figure 4 presents this flow in OMG's BPMN2 format. As we can observe, each operation is a process which, in BPMN2, is represented by a box that contains the symbol "+".

In our example, the Traveler service initiates the Order Trip process, and then invokes the Travel Agency which searches for the required Flight on the Airline. In this type of diagram, since the Traveler started the process, it is represented differently from the entities called, which in this case, are the Travel Agency and the Airline (both are painted in gray). This interaction is illustrated on Figure 5. Notice that the input message (request) is represented by a white envelope while the output message (response) is represented by a gray one. Even whether the message output of some process consists on the input of another one, this is the representative syntax for those situations.

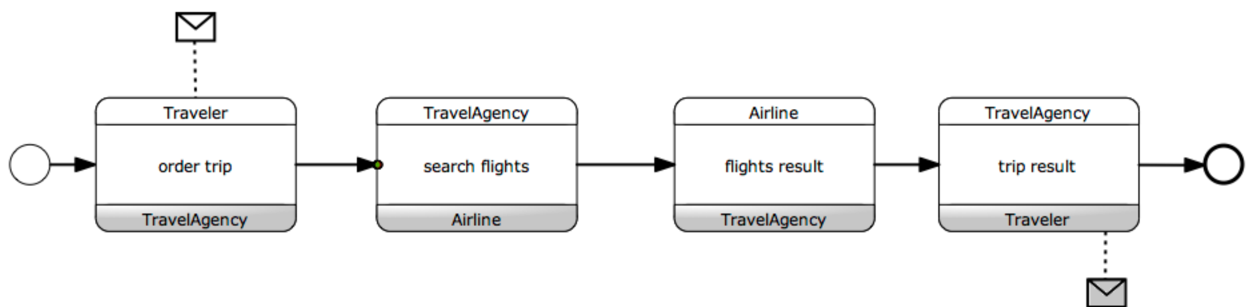

Figure 5. Process of the Order Trip operation

The Reserve Trip process can be seen on Figure 6. Traveler requests a trip reservation to the Travel Agency, which requests a flight reservation to the Airline. Then, the response travels all the flow back until the Traveler service. After these two interactions, a user can request the Traveler to cancel the reservation or to book it.

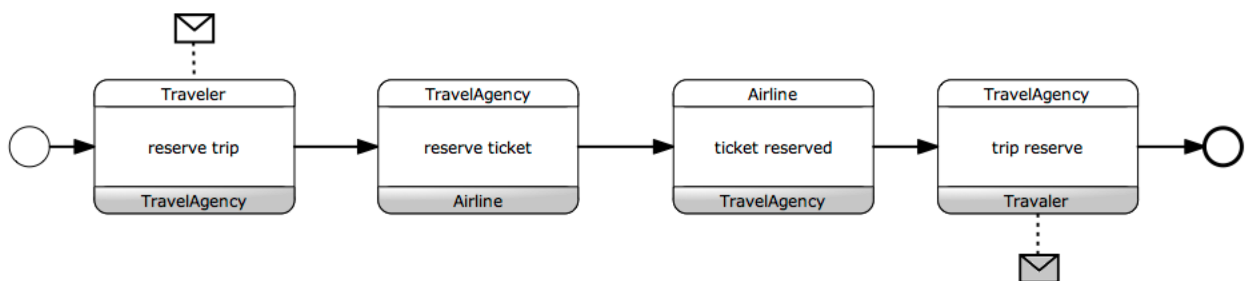

Figure 6. Process of Reserve Trip operation

As can be seen, the Cancel Reservation process (Figure 7) is simple. The return value from the Airline and the Travel Agency services are just confirmation messages.

The process described in Figure 8 is more complex than the others. After the Traveler service requests the Book Trip operation, the Travel Agency calls the Acquirer to check whether the user can afford to buy or not for the flight and its services.

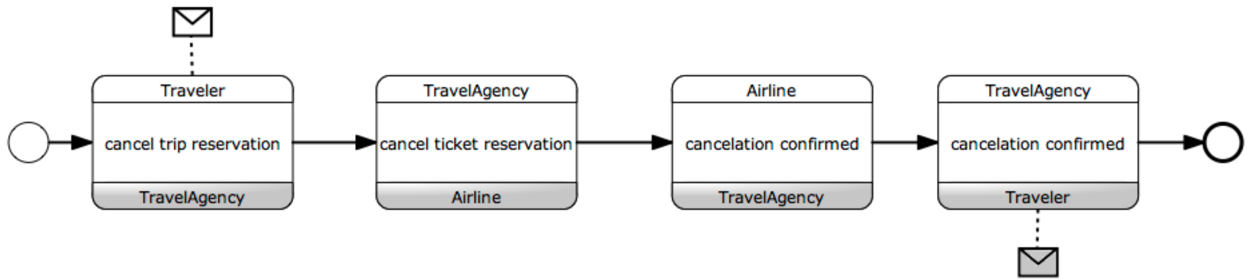

Figure 7. Process of Cancel Trip operation

The Acquirer service sends failure notifications to the Travel Agency and Airline services if the user cannot afford to buy the trip. In this case, these services send messages to the Traveler reporting the failure.

Otherwise, the Acquirer service sends a payment confirmation to the Travel Agency and Airline services. After that, the Airline service confirms the flight price to 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.

Each peer of our choreography is an atomic web service. These services can be accessed by other applications as well as reused by other compositions. We developed our services using different technologies. Traveler and Airline are SOAP/WSDL services while Travel Agency and Acquirer are RESTful web services.

Even though some choreography standard languages (WSCI, WS-CDL, and BPMN2) have been defined, to the best of our knowledge, none of them has been completely implemented. Therefore, we had some difficulties to design and implement our choreography.Since we needed a running choreography for testing, the OpenKnowledge system was selected as the best option for our preliminary work. In spite of OK not being originally designed for SOA applications, it is simple and easy to define and distribute interaction models with it.

As explained before, standardized web service compositions are accessible as atomic services from the exterior. For this reason, we adapted the OK choreography to become accessible as an atomic service. As an advantage, all developed test scripts and the tools are compatible with those standard compositions. Since the choreography was developed incrementally, we tried to apply, whenever possible, TFD (Test-First Development) [Mes07].

5.2. Proposed test cases

Before starting to implement automated test cases for the presented choreography, we first wrote in pseudo-code some tests we intended to implement. Through this "draft" version of tests, we derived unit and integration tests by analyzing the web services isolately and the choreography flow, respectively.

Some web service participants of a choreography are atomic services, and can be accessed outside the choreography. Other ones, such as the Traveler Service, were developed exclusively for our composition. We can describe the internal behavior of each services as

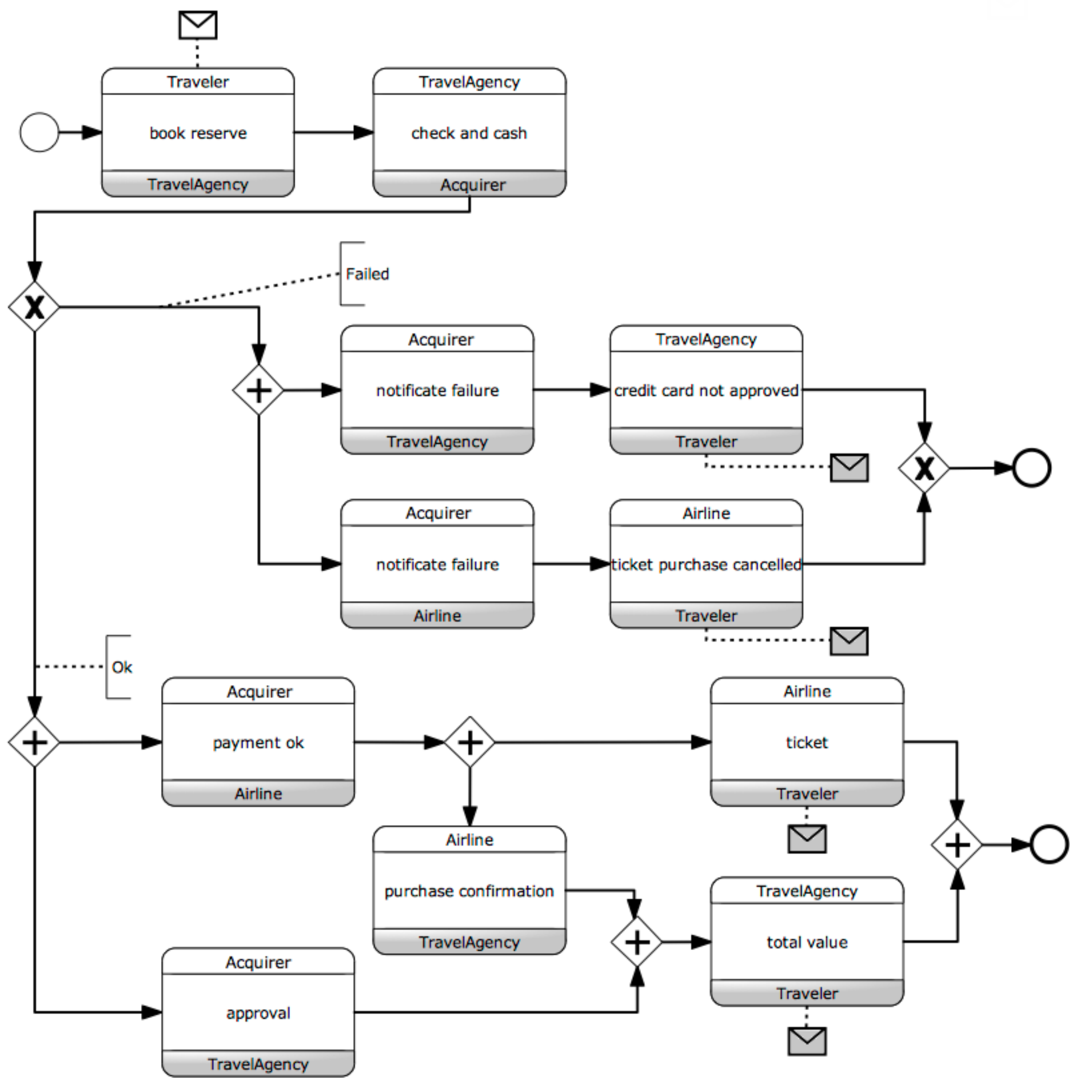

Figure 8. Process of Book Trip operation

follows:

- **Traveler:** this service corresponds to the “trigger” of the choreography. In this first version, it just forwards the received messages to the web service Travel Agency, and interacts directly with users. In more realistic scenarios, this service can store user information and invoke other services to verify/collect data from user (e.g., payment information, historical requests);
- **Travel Agency:** stores information about travelers (end-users), for instance their registers. This functionality is not specified in this choreography model (see Figure 4), but every time a user makes an order trip request, it is invoked.
- **Airline:** it 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. Then, users can olny reserve and book tickets with the assistance of an authorized Travel Agency service.
- **Acquirer:** this service manages the financial relationships among the services. Then, it provides internal functionalities such as checking user credit, creating accountants and effecting payments.

The goals of unit tests proposed below consist on apply black-box techniques to exercise these services internal behaviors. In this section, we present a high-level acceptance test specification for verifying and validating some scenarios of choreography. Then, some unit test cases for testing the Airline service are presented.

A concrete implementation of these tests is described in details in the next section. The Figure below presents some acronyms and resource names that will be used in the tests description.

Web Services

TE: Traveler
TA: Travel Agency
AL: Airline
AQ: Acquirer

Resources

user: name, identification, credit card number
trip: destination, data, time
flightX: airline name, price
reserveX: (reserve confirmation)
credit card statement: (payment confirmation)
eTicketX: a valid ticket for flight X

Figure 9. Acronyms and resource names

5.2.1. Acceptance tests scenarios

Using black-box, we present below some scenarios and the respectives test cases written to validated it.

1. Successful Book Trip

// First Step
// User request Traveler WS which activates the choreography

request: TE.orderTrip(trip, user);
expected result: "flightA"

// Second Step
// User invokes traveler WS to make a reservation

request: TE.reserveTicket(flightA);
expected result: "reserveA"

// Third Step
// User invokers Traveler WS to book

request: TE.bookTicket(reserveA);
expected results:

1. "Ok"
2. "credicard estatement" from TA.sendEstatement();
3. "eTicketA" from AL.sendTickets();

2. Book Trip Canceled

// First Step
// // User invokes Traveler WS which starts the choreography

request: TE.orderTrip(trip, user);
expected result: "flightA"

// Second Step
// User invokes Traveler WS to make a reservation

request: TE.reserveTicket(flightA);
expected result: "reserveA"

// Third Step
// User invokes Traveler WS to cancel the reservation

request: TE.cancel(reserveA);
expected result: "canceled"

3. Unsuccessful Book Trip (no flights available)

// First Step
// User request Traveler WS which starts the choreography

request: TE.orderTrip(trip, user);
expected result: "no flights"

5.2.2. Airline unit tests examples

In the following test specifications, some functionalities provided by the airline service are tested at unit level. To apply these tests, some clients must be generated to invoke the service under testing. In the next section, these mechanisms are approached in details.

1. **Search for a flight (flights available)**

request: AL.searchFlight(trip);
expected result: "flightA"

2. **Search for a flight (flights not available)**

request: AL.searchFlight(trip);
expected result: "no tickets available"

3. **Reserve a flight**

request: AL.reserveFlight(flightA, travelAgencyID);
expected result: "reserve"

5.3. Implemented test cases

In this section, we describe our approaches for implementing automated tests for the choreography previously presented. Therefore, all approaches to implement unit, integration, and acceptance tests are described. Since we consider integration tests the most important strategy for web service composition, the section about it is more detailed than the others.

5.3.1. Unit tests

Unit tests focus on verifying the behavior of the smallest unit of software which can be a single class or a method. As discussed in Section 4.1, on the choreography context, services are considered the units for unit testing. Therefore, our unit tests validate the service behavior by verifying each provided functionality. In real compositions, often, only the service interface is available, so we decided to focus our efforts on testing techniques related to this scenario. Then, each functionality of the service under testing is validated using the black box testing model.

In our current prototype, to test SOAP web services, a Java SOAP client (developed using JAX-WS[9]) needs to be developed for each service endpoint (i.e., the client is specific for

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

each endpoint). Once developed, the test specification uses this client to invoke the services. For guaranteeing the independence of these unit tests, at the beginning of the test suite, the prototype publishes the service and at the end, turns it off. Figure 10 shows some unit test cases for the Airline web service.

```
public class AirlineWSTest {

        private AirlineWSService service;
        private AirlineWS stub;

        final String TA_NAME = "United Airlines";
        final String FLIGHT_ID = "3153";
        final String RESERVE = "R3153-1|2000";
        final String USER = "John Locke";

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

        @AfterClass
        public static void unpublishAirlineService() {
                Bash.undeployService("airline");
        }

        @Before
        public void setUp(){
                service = new AirlineWSService();
                stub = service.getAirlineWSPort();
        }

        @Test
        public void shouldFindFlight() throws RemoteException{
                String destination = "Milan";
                String date = "12-21-2010";

                FlightResult 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() throws RemoteException{
                assertTrue(stub.isTravelAgencyAuthorized(TA_NAME));
        }

        ...
```

Figure 10. Unit test case for Airline service

Because of the nature of REST services, we have a generic client (i.e., it is not restricted to a specific endpoint). This tool was adapted from an open source GUI rest client [RC10] and integrated into our prototype. Similarly to SOAP web services, before running the unit test case, the prototype deploys the service, and at the end, turns it off. In Figure 11, some unit test cases for the Travel Agency service are presented.

```java
public class TravelAgencyWSTest {

	final static String BASE_URL = "http://localhost:9881/travelagency";
	private static RESTClient client;


	@BeforeClass
	public static void publishTravelAgencyService() {
		Bash.deployService("travelagency");
		client = new RESTClient();
		client.setBaseURL(BASE_URL);
		Bash.cleanTravelAgencyDatabase();
	}

	@AfterClass
	public static void unpublishTravelAgencyService() {
		Bash.undeployService("travelagency");
	}

	@After
	public void tearDown(){
		Bash.cleanTravelAgencyDatabase();
	}

	@Test
	public void shouldCreateUser() {
		String body = "John Locke|421543-2";
		String resourceLocation = client.POST("/users", body);

		assertEquals("http://localhost:9881/travelagency/users/1", resourceLocation);
	}

	@Test
	public void shouldRetrieveUserCreated() {
		String body = "John Locke|421543-2";
		String resourceLocation = client.POST("/users", body);
		String resourcePart = resourceLocation.substring(BASE_URL.length(), resourceLocation.length());
		String response = client.GET(resourcePart);

		assertEquals("John Locke|421543-2", response);
	}

	...
```

Figure 11. Unit test case for Travel Agency service

During the development of atomic services, we faced some problems related to non-idempotent operations (for instance POST). Test cases that usually need certain resources previously created can become dependent of other test cases. This behavior is not acceptable for automated unit tests. To solve this, a strategy could be cleaning all persistence structures affected by the test after its execution.

5.3.2. Acceptance tests

Differently from others 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 such context, this type of test is similar to the approaches of unit tests using the black-box model, and there is no need to know how the service is implemented.

In our approach, a tester specifies the tests by calling a service that activates the choreography. Before the execution, he/she needs to execute a script that enacts the choreography and deploys a service. Then, these 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, shown in Figure 5, the tester calls the method on the Traveler Web Service and compares the returned value with the expected one. The whole flow performed by a user that completes his/her plan can be seen in Figure 12.

```
@Test
public void shouldBookAndPlanTrip() throws RemoteException{

    Flight flight = stub.orderTrip("Paris", "12-20-2010", "John Locke", "435067869");
    String reserve = stub.reserveTicket(flight.getId());
    List<String> response = stub.bookReserve(reserve);

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

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

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

}
```

Figure 12. Acceptance Test example

Each operation of the scenario tested in Figure 12 can be tested separately as presented in Figure 13.

5.3.3. Integration tests

Integration tests attend to solve the problems produced when unit tested components are integrated. Their goal is to verify the unit interfaces and interactions. Our approach for applying integration tests is based on the techniques presented by Bucchiarone [ABS07] as in the incremental integration strategies (see Section 2). After all services have been tested at unit

```java
@Test
public void shouldOrderTrip() throws RemoteException{
    Flight flight = stub.orderTrip("Paris", "12-20-2010", "John Locke", "435067869");

    assertEquals("available",flight.getStatus());
    assertEquals("2142", flight.getId());
    assertEquals("2000", flight.getPrice());
    assertEquals("United Airlines", flight.getAirline());
}

@Test
public void shouldReserve() throws RemoteException{
    Flight flight = stub.orderTrip("Paris", "12-20-2010", "John Locke", "435067869");
    String reserve = stub.reserveTicket(flight.getId());
    assertEquals("R2142-1", reserve);
}

@Test
public void shouldBookAndReceiveAnStatement() throws RemoteException{
    Flight flight = stub.orderTrip("Paris", "12-20-2010", "John Locke", "435067869");
    String reserve = stub.reserveTicket(flight.getId());
    List<String> bookResponse = stub.bookReserve(reserve);
    String statement = "Name: John Locke...";

    assertTrue(bookResponse.contains(statement));
```

Figure 13. Acceptance test for operations testing

level, the approach focus on integrating each service at a time in the choreography. Once a service is integrated, the choreography is actived by the tester. Then, it is verified whether the component (service) newly integrated acts as expected. This step is achieved by checking the messages sent by that component. For each message sent, its name, destination, and content are compared with the expected values.

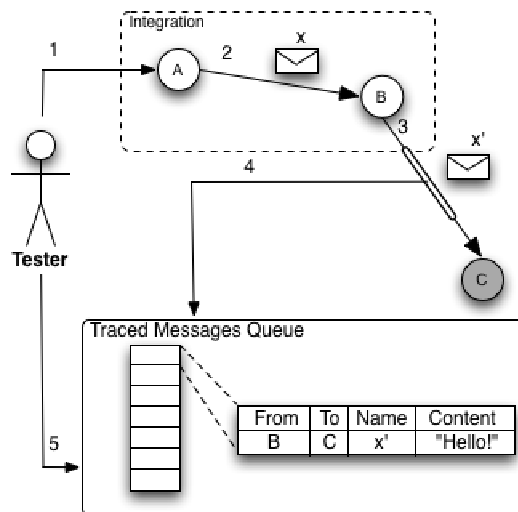

Figure 14. Integration test flow example.

We developed and include in our prototype an *ad hoc* framework that supports implements the steps in this approach. Figure 14 shows an example of that approach usage. As depicted, the tester is integrating the A and B components. In the first step, the tester activates the choreography, invoking the service A. During the choreography enactment, messages are exchanged between services A and B (step 2). Our framework collects the output messages fom B and stores them in a queue. This is performed in the third and fourth steps. When the choreography is over, the tool compares what the tester has specified with the collected data

(step 5).

Some integration testing techniques presented in Section 2.1 can be combined with our approach. In the top-down procedure, the integration begins with the main module, so its dependencies need to be mocked, which in the example would be the C service. In the bottom-up approach, the integration starts with the most independent modules. In this case, there are no dependencies, so the C component could be a service already integrated in the choreography.

To demonstrate the benefits of this kind of test, we propose an example based on the choreography presented in the previous section. Considering that the Travel Agency can search for flights from more that one airline, suppose that another airline service is integrated to the choreography. This new service is from a Brazilian provider, and consequently, it charges all tickets in BRL (Brazilian Real), but our choreography just 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. This situation and the tests that caught this incorrect behavior are presented in Figure 15 and Figure 16

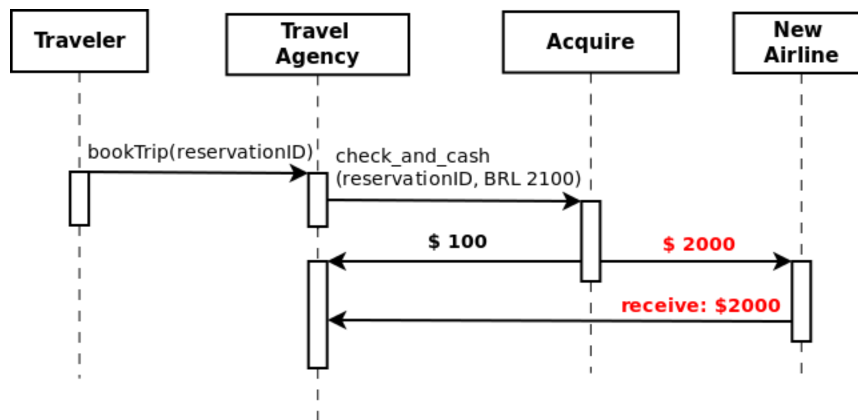

Figure 15. Message flow after new Airline service integration

```
travelerStub.bookReserve(reserve);
String expected = queue.get("airline", "travelagency", "value_paid");

assertEquals("BRL 2000", expected);
```

Figure 16. Test case to verify new Airline service integration error

Thus, these tests revealed an error in the choreography that should be fixed, for example, by adding a currency converter component. We have used both integration and acceptance testing strategies to validate our choreography flow. With integration testing, some problems of interface matching, like the currency conversion, are identified when a new service is integrated. As a result, it is discovered that an error or failure emerged with the current integration. Even though integration tests are the most efficient ones, they also are more difficult to be applied. On the other hand, acceptance tests are easier to be applied, since the composed service is taken as an atomic one, but they are also slower.

6. Conclusions and Future Works

This technical report presented our initial research about the current scenario of V&V activities in web services compositions. We focused the research on existing tools and techniques for testing atomic and composed services. Based on the results, we also developed a prototype for automated choreography tests.

When taken as an atomic service, both orchestrations and choreographies testing approaches are equivalent to the techniques for atomic services testing (see Section 4.1). Nevertheless, when the implementation of the orchestrator is available, unit tests can exercise the flow orchestrated using white-box testing techniques.

Although there are some approaches for testing choreographies in the literature, we noticed that these proposed works do not execute the tests on the choreography (running software). In such approaches, the services that compose a choreography are not really invoked by the tests. These testing techniques focus on validating the models that describe the choreography.

The nature of SOA makes the execution of integration tests for composed services more difficult. Because of the lack of service information, how it is implemented, third-part rules, and governance issues, the adoption of this testing strategy may not be feasible in some cases. To experience these issues, we developed a software prototype for testing choreography automated. One of its feature provides mechanisms for applying integration tests by capturing and analyzing messages exchanged among the choreography participants.

For future works, we plan to develop a framework and a set of tools for testing choreographies. With this set of tools, we intend to provide mechanisms for: (i) specifying the needed environment to test and monitor a choreography (running software); (ii) writing and executing tests automated.

All tools proposed will support a TDD methodology for writing high quality automated tests for choreographies. The lessons learned from the prototype development will be used as a valid input for the methodology proposal we intend to develop. All testing techniques and strategies as well as the practical methodologies studied consist on the basis to achieve our goals.

7. Acknowledgements

The research leading to these results has received funding from 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

[ABS07] H. Melgratti A. Bucchiarone and F. Severoni. Testing service composition. In *8th Argentine Symposium on Software Engineering (ASSE'07)*, Mar del Plata, Argentina, 2007.

[BBMP09] Cesare Bartolini, Antonia Bertolino, Eda Marchetti, and Andrea Polini. WS-TAXI: A WSDL-based testing tool for web services. *Software Testing, Verification, and Validation, 2008 International Conference on*, 0:326–335, 2009.

[Bec02] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, November 2002.

[Bec10] Kent Beck. Automated testing framework. Available on: <http://www.junit.org/>, 2010.

[BPB+09] Paolo Besana, Vivek Patkar, Adam Barker, David Robertson, and David Glasspool. Sharing choreographies in openknowledge: A novel approach to interoperability. *JSW*, pages 833–842, 2009.

[BY10] Gary Brown and Jeff Yu. Savara 1.1 – Getting started guide, 2010. Available on: http://docs.jboss.org/savara/releases/1.1.x/gettingstartedguide.

[CDP06] G. Canfora and M. Di Penta. Testing services and service-centric systems: challenges and opportunities. *IT Professional*, 8(2):10 –17, march-april 2006.

[CDP09] Gerardo Canfora and Massimiliano Di Penta. Service-oriented architectures testing: A survey. In Andrea De Lucia and Filomena Ferrucci, editors, *Software Engineering*, volume 5413 of *Lecture Notes in Computer Science*, pages 78–105. Springer Berlin / Heidelberg, 2009.

[CK09] S.K. Chakrabarti and P. Kumar. Test-the-rest: An approach to testing restful web-services. In *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. COMPUTATIONWORLD '09. Computation World*, pages 302 –308, nov. 2009.

[Del97] Marcio E. Delamaro. *Mutação de interface: Um critério de adequação interprocedimental para o teste de integração.* PhD thesis, University of São Paulo – Physics Institute, SP, Brazil, 1997.

[Erl07] Thomas Erl. *SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007.

[Evi10] Eviware. SoapUI, web services functional testing tool. Available on: <http://http://www.soapui.org/>, 2010.

[FHKW09] L. Frantzen, M. N. Huerta, Z. G. Kiss, and T. Wallet. On-The-Fly Model-Based Testing of Web Services with Jambition. In R. Bruni and K. Wolf, editors, *5th International Workshop on Web Services and Formal Methods – WS-FM 2008*, number 5387 in Lecture Notes in Computer Science, pages 143–157. Springer, 2009.

[Gol10] Corey Goldberg. Free tool for automated testing of web applications and web services. Available on: <http://www.webinject.org/>, 2010.

[Hew09] Eben Hewitt. *Java Soa Cookbook*. O'Reilly Media, 1 edition, March 2009.

[LSJZ05] Zhongjie Li, Wei Sun, Zhong Bo Jiang, and Xin Zhang. Bpel4ws unit testing: Framework and implementation. In *Proceedings of the IEEE International Conference on Web Services*, ICWS '05, pages 103–110, Washington, DC, USA, 2005. IEEE Computer Society.

[Mad09] Bhavish Kumar Madurai. Getting serious about enterprise architecture (whitepaper on alignment between testable architectures and TOGAF), 2009. Available on: <http://docs.jboss.org/savara/whitepapers/>.

[Mes07] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, May 2007.

[ML06] Philip Mayer and Daniel 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.

[Mye04] Glenford J. Myers. *The Art of Software Testing, Second Edition*. Wiley, 2 edition, June 2004.

[OAS02] OASIS. Web Services Business Process Language. Available on: <http://www.oasis-open.org/committees/wsbpel/>, 2002.

[OMG10] OMG. Business Process Model and Notation. Available on: <http://www.bpmn.org/>, 2010.

[Ope10] Open Group. TOGAF (The Open Group Architecture Framework). Available on: <http://www.opengroup.org/togaf/>, 2010.

[Pi410] Pi4 Technologies Foundation. Pi calculus for SOA. Available on: <http://sourceforge.net/projects/pi4soa/>, 2010.

[Pre01] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 5th edition, 2001.

[RC10] Rest Client. Java application to test RESTful webservices. Available on: <http://code.google.com/p/rest-client/>, 2010.

[Red10] Red Hat and Cognizant Technology Solutions. Savara: Testable architecture, 2010. Available on: <http://www.jboss.org/savara>.

[SWK09] Alin Stefanescu, Sebastian Wieczorek, and Andrei 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, pages 313–324, Berlin, Heidelberg, 2009. Springer-Verlag.

[W3C02] W3C. Web Service Choreography Interface version 1.0. Available on: <http://www.w3.org/TR/wsci/>, 2002.

[W3C04] W3C. Web Service Choreography Description Language version 1.0. Available on: <http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/>, 2004.

[YLS06] Yuan Yuan, Zhongjie Li, and Wei Sun. A graph-search based approach to bpel4ws test generation. In *Proceedings of the International Conference on Software Engineering Advances*, pages 14–, Washington, DC, USA, 2006. IEEE Computer Society.

[ZAGS09] Zulfa Zakaria, Rodziah Atan, Abdul Azim Abdul Ghani, and Nor Fazlida Mohd. Sani. Unit testing approaches for BPEL: A systematic review. *Asia-Pacific Software Engineering Conference*, 0:316–322, 2009.