Rehearsal: A framework for automated testing 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-2011-05

Abstract. *Web services are loosely-coupled software components designed to support interoperable machine-to-machine interaction over the Internet. Due these characteristics, simple web services can be combined in more complete ones by using: orchestration or choreographies. In an orchestration, a central node (the orchestrator) coordinates the flow of information from other participating services. Although straightforward and simple, its centralized nature leads to scalability. As a solution, choreographies have been proposed as decentralized and scalable solutions for composing web services. Nevertheless, inherent characteristics of SOA such as dynamicity, third-party and governance issues, and the decentralized flow of information, make the automated testing of choreographies difficult.*
The goal of this research is to present Rehearsal: a framework for automated testing of choreographies to support Test-Driven Development (TDD) of choreographies. This framework provides a set of features to assist the choreography developer during choreography testing. Through this framework: (i) choreography elements can be manipulated as Java Objects; (ii) web service operations can be invoked dynamically; and (iii) messages exchanged in the choreography can be intercepted and analyzed. Since real choreographies involve multiple internal and external services, the framework also provides mechanisms for mocking web services or invoking them in their real environments. With this set of features, we intend to support choreography testing in the entire choreography development process.

[*]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).

Contents

1. Introduction

Service-Oriented Computing (SOC) is now considered the new generation of distributed computing, being widely adopted. SOC aims to facilitate the development of distributed systems by providing mechanisms to overcome issues such as the lack of interoperability, high cost and complexity of integration. In such context, Service-Oriented Architecture (SOA) consists of an architectural model that guides the SOC paradigm, using web services as the building blocks of applications [Hew09]. Web services are loosely-coupled software components designed to support interoperable machine-to-machine interaction over a network. Due to these characteristics, simple web services can be combined to create more complete web services. From an external point-of-view, compound services are accessible as simple ones, and can also be composed in new services, recursively. The ability to effectively compose services is a critical requirement for achieving some of the most fundamental goals of Service-Oriented Computing [Erl07]. Thus, two main approaches have been proposed for composing web services: orchestrations and choreographies [Pel03].

In an orchestration, a central node (the orchestrator) coordinates the flow of information from other participating services. Although straightforward and simple, its centralized nature leads to scalability and fault-tolerance problems. As an alternative, choreographies have been proposed as decentralized and scalable solutions for composing web services. Each choreography participant plays a specific role, which determines its behavior in the choreography. Then, the interaction among the nodes is collaborative with no central point coordinating the information flow. As a result, choreographies are a good architecture for fully decentralized workflows [BBRW09].

Patient care in medical centers is an example of fully decentralized flow that can be modeled as a choreography [BPaRG09]. Normally, patient care involves the following stages: the patient registration, triage and examination, a medical appointment, exams, and medication. Each of these stages can be executed or supported by web services that can interact with each other to implement the entire workflow. In an orchestration, a central service must indicate which stage a patient must follow during the attendance. Thus, the orchestrator must deal with parallel requests since many patients are attended in parallel in a daily routine of a medical center. Besides, this central service must interact with services used by physicians, attendances and nurses that may need to retrieve information about the patient care (also in parallel).

Given the number of parallel interactions, the orchestrator can became a bottleneck of the system. In this scenario a choreography seems to be a more adequate approach for modeling the distributed flow. Each participant (e.g., patient, nurses, and physicians) can be modeled in a service that plays a specific role. As the patient progresses to a stage, the next role receive a message from the actual node to continue the attendance. Differently from orchestrations, the flow is decentralized, there is no central node coordinating and bypassing all the messages. For example, after the triage and examination have been done, the service playing the nurse role notifies the services playing the physician role. The later retrieve the triage results and let it available for the physician (human) to consult it.

In spite of all the benefits and advantages of web service choreographies, the automated testing of choreographies is challenging. Given the importance of choreographies, the functional behavior and scalability of choreographies must be assessed properly, otherwise, the choreography scalability may not be actually achieved. However, the inter-organizational integration of applications promoted by SOA can bring restrictions such as the impossibility to exercise third-party services in testing mode. As a consequence, services cannot be tested in isolation, before being integrated in the choreography. Moreover, some inherent characteristics of choreographies such as dynamicity and the decentralized flow of information, hamper the monitoring and interception of messages exchanged by the nodes of a choreography. Nevertheless, during choreography development, analyzing the messages exchanged by services can be an important mechanism for validating the integration of services. Due to these issues there is a lack of tools and techniques for automated testing of choreographies [BMS07, CDP09].

1.1. Motivation

In spite of some standards such as Web Service Choreography Interface (WSCI)[1], Web Services Choreography Description Language (WS-CDL)[2] and Business Process Model and Notation (BPMN2)[3] have been proposed for modeling choreographies, none of them have been defined as a standard solution. Consequently, the development process, including the testing activities, of choreographies is not disciplined. Many real choreographies are implemented using a combination of standards and *ad hoc* solutions.

Some tools, such as SoapUI[4] have been developed for testing atomic services. Since composed services are accessible as atomic services (from an external point of view), these tools can be used to validate the choreography as a service. Nevertheless, on this approach, the choreography is taken as a black-box, preventing the validation of services integrated inside the choreography, as well as the flow of information inside the choreography. Since standard choreographies are specified by global models before their deployment, some testing approaches [Pi410, WZZ$^+$10, ZPX$^+$10] have been proposed for validating these models. These approaches can detect design problems, using for instance, simulation. However, we believe that some failures and errors can be only detected by executing the services and the choreography.

In spite of their potential, the lack of stable standards, and the absence of more adequate testing tools, are obstacles for choreographies being widely adopted. Thus, efforts like the proposed testing framework to apply Test-Driven Development (TDD) of choreographies have a good potential to facilitate the choreography development and to leverage its adoption.

1.2. Goals

The main goal of this work is to develop a framework for automated testing of web service choreographies to support TDD (Test-Driven Development) [Bec03] of choreographies. During choreography development, the proposal framework will provide mechanisms for automating the testing of (1) the candidate web services (unit testing); (2) their integration in the choreography (integration testing), and (3), a part or the whole choreography (acceptance testing). With this framework, the software developer can apply automated tests on running choreographies, as well as on choreographies developed from scratch. To achieve our goal, the proposed framework will provide the following main features:

[1]WSCI: `http://www.w3.org/TR/wsci`
[2]WS-CDL: `http://www.w3.org/TR/ws-cdl-10`
[3]BPMN2: `http://www.omg.org/spec/BPMN/2.0`
[4]SoapUI: `http://www.soapui.org`

- *Abstraction of choreography:* The elements of a choreography such as roles, services, and messages exchanged must be represented by Java[5] objects. Through these objects, the developer will interact easily with the choreography elements for writing automated tests;
- *Dynamic generation of web service clients:* Given a web service URI, the framework must provide mechanisms for invoking the service operations dynamically.
- *Message Interceptor:* To apply integration tests, the framework must intercept, and then, collect the name and the content of messages exchanged among the services the developer wishes to the monitor;
- *Service mocking:* The framework must create a double service [Mes07] (e.g., a mock). Through this object, all web service operations can be mocked. This feature is particularly important when production web services cannot be tested (e.g., because of third-party rules) or when a specific scenario needs to be simulated.
- *Scalability testing support:* The framework must support scalability testing. The goal is to assess the choreography performance as long as the work load increases. To achieve that, we aim at providing features for: (i) applying multiple and parallel requests to services; (ii) measuring the response time of these services and (iii) creating more instances of existing services.

1.3. Document organization

In Chapter 2, we present the main concepts and terminologies used in this research. In Chapter 3, we discuss related works on automated testing of atomic web services and web services compositions. In Chapter 4, we explain the main features of the proposed framework. Finally, in Chapter 5 we present our conclusions and ongoing work.

[5]Java: `http://www.java.com/en`

2. Conceptualization and terminology

Software testing is a broad area of software development. Depending on the software community, the software development model adopted, and other context issues, terms and concepts related to software testing can be understood differently. For this reason, in this chapter, we present the main concepts and their respective meanings in the context of this work.

2.1. Verification and Validation (V&V)

Verification and Validation corresponds to a disciplined approach which aims to ensure quality during the whole software development [AM04]. On the one hand, Verification covers the activities to ensure that the software is built correctly regarding pre-defined standards and processes, or imposed design specifications. On the other hand, validation covers the activities that check if the software meets the user requirements. A wide set of activities belongs to V&V, such as formal technical reviews, quality and configuration audits, performance monitoring, simulation, documentation review, database review, algorithm analysis, and software testing.

2.2. Software testing

In this section, we introduce all software testing background related to this research. Despite this section refers to testing of traditional systems (client-server model), its understanding is needed for the testing approaches of web services choreographies presented.

2.2.1. Testing strategies

During a software development project, different testing strategies can be applied depending on the stage of development:

Unit tests focus on the small units 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].

Acceptance tests verify the behavior of the entire system or a complete functionality. They typically correspond to the execution of scenarios present in the use cases, features, or user stories specified by the customer. 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 objects are not used) [Mes07].

Integration tests lie between these two previous techniques and aim 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 the 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 is performed from the main module. Initially, all components that this module depends on are mocked or stubbed, then, these dependencies are replaced 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 entire cluster is integrated to other cluster.
- Big bang: all components are combined at once and tested as a whole.

2.2.2. Testing techniques

Depending on the goals of the test, availability of resources, and other factors, test strategies are combined with different testing techniques. In the functional or black-box technique, 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. During this validation, test criteria such as partitioning into equivalence classes and boundary value analysis [DMJ07] can be used.

Since it is impracticable to apply tests to cover all the possible inputs of program, normally, a subset containing the meaningful inputs is used in the functional test cases. This subset can be defined by using the partitioning into equivalence classes. First, the input domain is divided into equivalence classes. Then, at least one representative of each class is used as the test input. This representative is equivalent to other members of its class, so it is not necessary to repeat the test using the other members as inputs. Complementary to this test criterium, in boundary value analysis, the values above or below the equivalence classes boundaries are tested.

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".

2.2.3. Performance and scalability testing

The goal of performance testing is to apply enhancement strategies to maintain an acceptable system performance [Ngu01]. Although the terms performance, load, and stress tests are often used interchangeably, the following differences can be highlighted [Ngu01]:

- **Load testing** evaluates system performance to handle predefined load levels. Load testing measures how long the system under test takes to perform various program tasks and functions under normal, or predefined conditions.
- **Stress testing** evaluates the behavior when the system under test is pushed beyond its specified operational limits. Normally these tests aim at evaluating the responses to bursts of peak activity that exceed system limitations. In these situations, the goal is to determine whether the system crashes and/or recovers gracefully.

Scalability testing is similar to stress testing since both strategies aim at assessing the system performance in extreme situations to find out where the limits are [Liu09]. However, scalability testing also assesses how the system can be expanded to handle the required demand (e.g., number of users or volume of transactions). This kind of testing is important, since every business is expected to grow with time [Liu09]. Thus, with this test strategy, it is possible to measure the processing capability required to support the desired level of business operations.

2.2.4. Online vs Offline testing

Testing techniques can also be classified in online or offline testing, depending on when and where these techniques are applied. Online testing consists of determining whether a system complies with its intended behavior during its real life operation (i.e., in the production environment) [GGvD10]. Thus, all testing strategies and techniques, analysis (e.g., trace files), and simulations applied before the deployment of the system in its production environment, corresponds to offline testing. Tests invoking a system in the development or testing environment are also considered offline testing. In the literature, online is also called runtime monitoring or "passive testing" [Ber07]. However, since these activities may not limit themselves to passively observe the system under test, the term online testing can also be used for approaches that not only monitor, but also trigger proactive actions [BAP11].

2.2.5. Test double

During software testing, often, the System Under Test (SUT) depends on other software components (classes, databases, systems, and so on) that can not be executed in "testing mode". Moreover, sometimes invoking a real Depended-On Component (DOC) makes the test too slow. In such situations, the real DOC can be replaced by a test double, an object that provides the same interface as the real DOC [Mes07]. According to Meszaros [Mes07], there are five kinds of test doubles:

- **Dummy:** objects that are passed as arguments to a method but are never actually used. Methods of the SUT may require objects as parameters, if neither the tests nor the SUT use these objects, they can be dummy objects;
- **Fake:** objects that provide the same functionalities of the real DOC, but using a simpler implementation. For instance, during the testing, a real database can be replaced by an in memory database fake object;
- **Stub:** objects that are pre-programmed to return pre-defined answers for calls received during the test. These answers can be defined as valid or invalid ones. The term "test stub" can be used to mean a temporary implementation that is used only until the real object becomes available. The goal behind the use of stub objects is to apply state validation. Thus, we determine whether the exercised method worked correctly by examining the state of the SUT and its collaborators [Fow07];
- **Spy:** stub objects that also provide mechanisms for capturing the stub answers for a later validation;
- **Mock:** objects that are pre-programmed to return pre-defined values when they receive expected calls. Differently from stubs, the goal of mocking is to apply behavior verification. Therefore, the exercised method worked correctly if the SUT made the correct calls on the mock objects that represented its collaborators [Fow07].

2.2.6. Test-Driven Development (TDD)

Test-Driven Development (TDD) consists of a design technique that guides the software development through testing [Bec03, Fow11]. TDD can be summed up in the following iterative steps:

- Write an automated test for the next functionality to be added into the system;
- Run all tests and see the new one fail;
- Write the simplest code possible to make the test pass;
- Run all tests and see them all succeed;
- Refactor the code to improve its quality.

In addition to these steps, according to Astels [Ast03], to apply TDD, developers should follow principles such as: (i) maintaining an exhaustive suite of programmer tests; (ii) only deploying code into the production environment if it has tests associated. Differently from unit tests, which are written to assess a method or class, programmer tests are tests written to define what must be developed. Programmer tests are similar to an executable specification since these tests help developers understand why a particular function is needed, to demonstrate how a function is called, or what are the expected results [Jef].

Having tests associated with the code, gives the developer confidence and courage to make changes and detect immediately (or in a short time) possible introduced problems. Thus, with the absence of tests, it is not possible to assure the correct behavior of the code when it is deployed or integrated into the production environment. Given this importance, in eXtreme Programming (XP) [Bec00], it is often said that a feature does not exist until there is a test suite associated to it.

As a design technique, TDD is not only about software testing but also a learning process. Applying different levels of tests, the development team can clarify the user and customer expectations, and then refine the system requirements [FP09].

2.3. Web service compositions

In the Service-Oriented Architecture (SOA) context, web services consist on the building blocks of applications [Hew09]. Due to its interoperability characteristics, atomic web services can be composed in more complex ones. In this section, we first present an overview of SOA describing its stakeholders, and how web services are used in the SOA life cycle. Then, we describe the main characteristics of web services, and two approaches for composing them.

2.3.1. SOA roles and operations

In the SOA life-cycle, different stakeholders interact with web services for developing or executing their operations. The typical SOA triangle, depicted in Figure 1, illustrates the involved stakeholders, its roles and its operations in SOA life-cycle. Once a provider develops a new web service, its description, containing for instance a WSDL document for SOAP web services, is published in a public or private registry of services. After the publication, clients send a query to the registry to find the desired service. Then, the registry returns a list of service interfaces which matches the client query. Finally, the client chooses an interface and then, can start interacting with the provider, these latter actions are denominated binding. Both querying and binding activities can be automated [PGFT11].

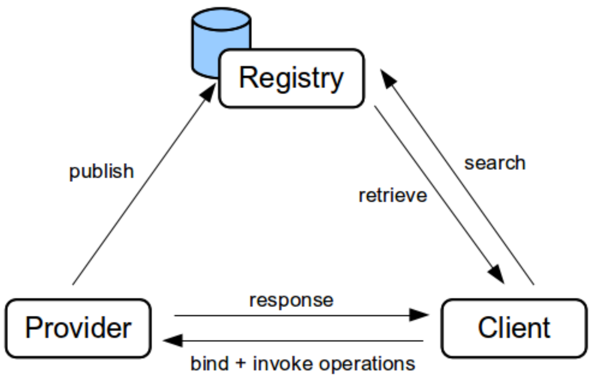


Figure 1. SOA triangle of roles and operations

2.3.2. Web services

According to the W3C [BHM$^+$04], a web service is a software component designed to favor the interoperable machine-to-machine communication using web protocols. Web services must be accessible through an endpoint. In addition, a web service exposes an interface in a machine-processable format for describing its operations. This interface may be specified in the Web Service Description Language (WSDL)[6]. A WSDL document describes the data types and messages exchanged by the web service operations and all needed information for binding with the web service. Based on the web service description, other software systems interact with it exchanging messages in a protocol such as the Simple Object Access Protocol (SOAP)[7], normally using HTTP. SOAP is a protocol, defined in XML, for exchanging structured information in a distributed and platform-independent way.

In spite of the benefits of using SOAP and WSDL, such as interoperability and the loose coupling (between a service provider and a consumer), XML manipulation raises several performance problems. For example, the translation of the XML to corresponding memory data structures has been problematic and is the main source of performance inefficiencies [PZL08]. Because of these issues, REpresentational State Transfer (REST) or Restful web services [RR07] can be an alternative to WSDL web services.

Initially, REST was proposed as an architectural style for building large-scale distributed hypermedia systems. However, because of the principles described below, REST started to be used as web services [PZL08]:

- **Resource identification:** a RESTful web service exposes its resources via the Internet. A resource can be for example a book from a book seller service. Each resource contains an identifier and it is accessible by a URI;

[6]WSDL: `http://www.w3.org/TR/wsdl`
[7]Soap: `http://www.w3.org/TR/soap`

- **Uniform interface:** The operations: PUT, GET, POST, and DELETE are used for creating, reading, updating, and deleting resources, respectively. Although REST web service can be used with other protocols, normally its used with HTTP;
- **Self-descriptive messages:** a given resource has only one identifier but it can have many representations (HTML, XML, JSON, PDF, JPEG, ...);
- **Stateful interactions through hyperlinks:** Every interaction with a resource is stateless, nonetheless, using the resource identifiers, stateful interactions can be performed. Several techniques exist to exchange state, e.g., URI rewriting, cookies, and hidden form fields.

2.3.3. Orchestration vs. Choreography

Composability of web services is one of the SOA principles [Erl07]. In such context, orchestration and choreography have been proposed as approaches for composing web services. Orchestration corresponds to a centralized approach where internal and external web services are composed into an executable business process [Pel03]. Some standards such as Business Process Execution Language (BPEL)[8] have been proposed for orchestrating services. In an orchestration, a central party (node) controls the interaction flow of the other parties.

A choreography is a collaborative interaction in which each involved node plays a well defined role. A role defines the behavior a node must follow as part of a larger and more complex interaction. When all roles have been set up, each node is aware of when and with whom to communicate, based on pre-established messages specified by a global model [BBRW09]. Therefore, when the choreography is started (enacted), there is no central entity driving the interaction of the whole choreography.

According to Ross-Talbot [Rt05], a choreography is a description of a peer-to-peer externally observable interactions that exist between the services. Differently from orchestrations, the interactions are described from a global or neutral point of view and not from any one services perspective. Thus, choreographies describe the common observable interactions (the messages flows) between the nodes but without defining how that role will be executed. In other words, a choreography defines the messages exchanged among the participating services; but it does not describe any internal action that occurs within the participating ones. These internal actions do not directly result in an externally visible effect such as an internal computation or data transformation [BDO].

Since orchestrations are executable processes, choreographies can be executed via distributed orchestrations [CHO11]. This way at a high level the global interactions of many participants are specified by the choreography model, using for instance, the languages and standards described in Section 1.1. Then, in a lower layer, to play a specific role, a service or a set of services can be orchestrated to deal with the message flow specified by the choreography. Figure 2 illustrates this approach.

2.4. Alternatives for web service composition testing

An initial effort for understanding the current scenario of testing techniques for orchestrations and choreographies was conducted by Bucchiarone [BMS07]. Later, a more comprehensive

[8]BPEL: `http://www.oasis-open.org/committees/wsbpel`

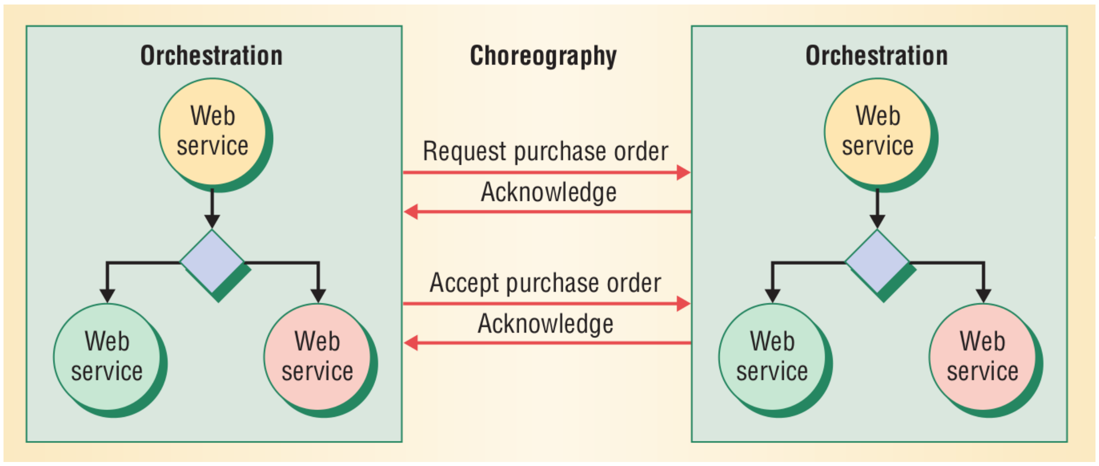


Figure 2. Orchestration of services playing roles in a choreography [Pel03]

survey to cover SOA testing was conducted by Canfora and Di Penta [CDP09]. These studies propose alternatives adapting the testing techniques and strategies for traditional software (see Section 2.2) to the context of web service compositions.

According to these studies, depending on the perspective and the availability of resources, different testing techniques and strategies can be applied. When the specification of an orchestration is available (e.g., in a BPEL file), this specification can also be taken as the code of the orchestration (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. Moreover, each service participating on the orchestration can be tested in isolation (as a unit).

However, from the user perspective, an orchestration can be taken as a unit since only its WSDL interface is exposed. In this situation, the process is available as a simple service. As a result, black-box testing techniques used for atomic services can also be applied for testing processes. The goal of these tests consists on exercising the services functionalities, verifying whether they match with the specification or the behavior expected by the customer, which are also the goals of acceptance tests. At the choreography side, acceptance tests can be applied taking the choreography as an atomic service. Nevertheless, if the internal representation of choreography is available, the unit tests can be performed 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 specific role in the choreography.

Since the tests at unit level are focused on just exercising the behavior of that unit, their integration must also be exercised. However, for orchestrations and choreographies, the lack of information about certain partners and the impossibility of exercising some third-party services make the integration testing complex. To minimize such issues, the dependent parties can be simulated. This can be done by creating stub or mock objects for the web services that the unit under testing communicates with.

3. Related work

In spite of all the benefits that choreographies can bring, the testing of this kind of composition raises new challenges. In this chapter, we present the main related works for testing web service choreographies. These works were grouped following the classification of web services composition testing presented in section 2.4. Thus, works related to atomic services represent unit and acceptance testing techniques, while those works focused on validating the internal choreography flow of information are the basis for integration testing.

3.1. Testing of atomic web services

Considering the SOA triangle (see Section 2.3.1), different stakeholders are involved in different phases of a SOA application life-cycle. The following works present approaches for testing atomic web services at different stakeholder perspectives.

3.1.1. Service client

There are some works for testing web service from the client perspective. We consider that the client role can be played by an end-user of the service, or by a developer, during the process of integrating services. The following approaches are applied after service binding.

SoapUI. This tool [Evi10] was developed in Java and provides mechanisms for functional and performance tests. From a valid WSDL, SoapUI provides features to build automatically a suite of unit test for each operation, and a mock service to simulate the web service under testing. Besides, this tool can be used for deploying the mocked service, and then, for making the WSDL interface available like the real service, but only the mocked functionalities are invoked. Finally, SoapUI has mechanisms to measure test coverage. Despite its name suggests a restriction to SOAP/WSDL web services, some mechanisms are provided to test other kinds of services, such as RESTful and JMS (Java Message Service). All these features are provided by the Open Source version (under the LGPL license[9]), but a commercial version of SoapUI also provides more resources, such as support for performance tests in large scale systems.

WS-TAXI. 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 shown in Figure 3, input and output parameters values must be provide manually (fields with "?" character).

WS-TAXI [BBMP09] was proposed to improve this feature. 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 presented on Figure 3, WS-TAXI would generate test cases (permuting the data input) to cover all possibilities of options (lines 09 – 22). For instance, for the skeleton of Figure 3, WS-TAXI could generate a test cases permuting which option of *¡ChoiceElem¿* elements could be filled, or other test cases could permute the occurrence of the *¡occurElem¿* element. Also, WS-TAXI could generate test cases where the element *¡secondInput¿* could be

[9]http://www.gnu.org/licenses/lgpl-2.1.html

```xml
<soapenv:Envelope
        xmlns:soapen="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:test="http://www.example.org/test/">
        <soapenv:Header>
           <headerPart>?</headerPart>
        </soapenv:Header>
        <soapenv:Body>
           <test:operation>
              <test:ChoiceElem>
                 <!--You have a CHOICE of the next 4 items at this level -->
                 <field1>?</field1>
                 <field2>?</field2>
                 <field3>?</field3>
                 <!-- 0 to 3 repetitions:-->
                 <occurElem>?</occurElem>
                 <AllElem>
                    <!--You may enter the following 2 items in any order-->
                    <AllElem1>?</AllElem1>
                    <AllElem2>?</AllElem2>
                <\/AllElem>
              </test:ChoiceElem>
              <secondInput>?</secondInput>
           </test:operation>
        </soapenv:Body>
</soapenv:Envelope>
```

Figure 3. Skeleton generated by soapUI [BBMP09]

filled or not.

TTR. This tool is specific for REST testing. TTR (Test-The-REST) [CK09] provides mechanisms for functional testing (using the black box strategy) and non-functional testing (e.g., performance tests). Similarly to SoapUI, this tool provides support for testing the CRUD operations over REST service resources (see Section 2.3.2). TTR provides a script-like language for the test specification and the execution. A TTR test case is composed of an ID, the URL of the resource, the HTTP operation to be executed, input data, and expected output data. A test case can also be composed of other test cases. Similarly to SoapUI, test cases have to be written in XML.

SOCT. In all tools aforementioned, the client can only apply black-box testing to validate the web service operations. The goal of the Service Oriented Coverage Testing (SOCT) tool [BBEM09] is to provide web services with associated testing coverage information. Thus, the client receives services already tested, instead of applying himself/herself black-box testing to validate these services. During the development of an orchestration or a choreography, the service integrator can choose the best services by looking at their testing coverage information. To achieve their goals, the authors of SOCT propose an adaption of the entities which compose the SOA triangle (see Section 2.3.1). In the proposed approach, first the provided services must be instrumented, they are called *Testable Services*. Second, an entity called *TCov provider* is introduced between the testable services provider and the client. Finally, the new SOA workflow is defined by the following steps:

1. the SOCT Tester, which can represent the client, sends a “start a testing session” message and the test to the desired *Testable Service* (TS);
2. the test is executed in the *TS provider*, which sends a log with the coverage data to the

Coverage Collecting Service (CCS), an internal service of *TCov provider*;

3. after the testing session has been finished, the SOCT tester invokes the *Coverage Reporting Service* (CRS), another internal service of *TCov* to retrieve the coverage data.

Other software tools: There are many software tools for testing web services at the client perspective. In Table 1, we present some studied tools with their vendors, URL, and types of license.

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
REST-assured	Jayway	http://code.google.com/p/rest-assured	Apache 2.0
SoapSonar	CrossCheck Networks	http://www.crosschecknet.com	Proprietary
SOATest	Parasoft	http://www.parasoft.com	Proprietary
TestMaker	PushToTest	http://www.pushtotest.com	GPL
WebInject	Corey Goldberg	http://www.webinject.org	GPL

Table 1. Web Services Testing tools

3.1.2. Services registry

The following works describe approaches for applying automated online testing within the service registry.

Audition framework: This work was initially proposed in the PLASTIC (Pervasive and Trustworthy Network)[10] project, and later on, in the TAS^3 (Trusted Architecture for Securely Shared Services) project. The goal of TAS^3 is to provide a trustworthy infrastructure for web services and users to exchange personal data information. The Audition Framework [BDAP09] consists of a framework for validating the TAS^3 web services to guarantee the trustworthiness of the participating services. To assure secure personal data information exchange, all services must provide the functionalities they are supposed to, and also implement all authentication aspects required by TAS^3. Through the Audition framework, before being allocated on an *Audition Service Registry* (ASR), the services are audited. First, ASR creates on-the-fly testers for each service being registered. Then, the ASR creates and assigns assertions for the test cases that will be performed. The assertions certify that a given actor is playing a given role into the testing choreography. Finally, the tester invokes the service functionalities (online testing) and the ASR collects the service replies. If the replies match the expected results, the service is accepted, otherwise, it is denied. Once the auditing succeed, the ASR indexes the service into the directory service and a testing certificate will be associated with it.

(role)CAST: This work consists of the implementation of the online testing architecture proposed by the Audition Framework described above. Within a service federation, a network of services from different organizations, (role)CAST (ROLE CompliAnce Service on-line Testing)[11] [BAP11] proposes mechanisms for validating authentication, authorization, and identi-

[10]PLASTIC: `http://www.ist-plastic.org`
[11]roleCAST: `http://labsewiki.isti.cnr.it/labse/tools/rolecast`

fication in the federation. It is achieved by the entities *Test Driver* and *Test Robot*. The goal of the *Test Driver* is to configure and to execute *Tester Robot* instances. An instance of a *Tester Robot* is in charge of:

1. loading test cases for a *Test Cases Repository*;
2. validating if a web service is allowed to join the federation by checking its identification information with an id provider (through a secure SOAP channel);
3. making a real request to the service under testing;
4. verifying the test case results with an Oracle;
5. saving these results in a *Test Results Repository*.

To generate test cases automatically, (role)CAST provides mechanisms for creating *Test Drivers* by deriving test cases from UML diagrams.

3.1.3. Service provider

Before publishing and providing a service, a service developer can apply unit and acceptance tests using a development or even the production environment. In the approach proposed by Jia Zhang [Zha11], the tests are triggered by an external entity but they are executed within the service provider. In her work, Zhang proposes an approach for automated testing of atomic web services using a mobile agent-based infrastructure. A mobile agent refers to a composition of software and data that is able to travel from a computer to another over a network and autonomously and automatically continue their execution on the destination computer. Thus, the main idea is to use agents for decreasing the costs of communication that is inherent for web service on-line testing, mainly in the case of performance testing.

Instead of launching a suite of test cases (invoking the web services using Soap) and receiving each test result from the client side, which is a common practice used, for example in SoapUI, the idea is to send the test cases to a mobile agent placed at the web service server. Then, this agent executes all the tests and sends the results back to the client. This framework is built with the tools HP LoadRunner [HP11] and IBM Aglet [IBM11]. The former tool is an environment for automating the testing of systems in general, not only web services. The latter is an open source library for the development of mobile agents in Java.

3.2. Testing of web services choreographies

The related works presented previously are focused on testing atomic web services, the following related works propose specific approaches for automated testing of choreographies. We only present those testing approaches in accordance with the scope of our main goals (see Section 1.2). Systematic reviews such as [PGFT11, RPIT11] presented more works related to testing of web service compositions.

3.2.1. Before enactment

In this section, we present related works that provide mechanisms for testing a choreography before its enactment. Thus, the approaches proposed by these works are not applicable to a running choreographies.

PI4SOA: With this software tool [Pi410] is possible to test the 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 its role in the modeled choreography. Based on some components such as partners, roles, services, and interactions (message exchanges), the choreography is modeled by dragging and dropping these components in an Eclipse-based IDE. After modeled, it is possible to validate the flow among the web services by simulation, which is not performed using real services. This way, at design time, this tool provides mechanisms to verify the global model specified in WS-CDL.

CDLChecker: The main goal of this tool consists on validating and simulating choreography models. CDLChecker [WZZ+10] is an Eclipse plugin for checking a choreography global model written in WS-CDL to ensure the quality of its design. The tool is supported by a simulation engine that parses this model into a data-object structure that stores the data, interactions, and the flow of information contained in the model. Using these stored elements, the activities described on the model are exercised by simulation to validate the dynamic behavior of the choreography. Besides, CDLChecker provides a module to apply static validations on the choreography model. This module applies relational calculus to validate the WS-CDL specification. All constraints between the choreographies nodes are checked in a static way. The goal of this module is to facilitate the specification design, detecting faults as soon as possible.

Later, based on the simulation engine, the authors developed a framework for automatically testing input generation for WS-CDL documents [ZPX+10]. A symbolic execution technique is applied to generate test inputs while assertions, introduced in the WS-CDL documents, are treated as the test oracles. According to Sen [SMA05], in symbolic execution, the program is executed using symbolic variables in place of the concrete values for inputs. Each conditional expression in the program represents a constraint that determines an execution path. The goal is to generate concrete values for inputs to exercise different paths. In this work, choreography models are simulated using the generated test inputs and they are validated in conformance with the pre-defined assertions.

3.2.2. After enactment

The following works propose testing approaches applicable for a running choreography, not depending on whether it is running on a development, testing, or production environment.

BPELUnit: Mayer and Lübke proposes an architecture for creating BPEL automated testing frameworks. This architecture is divided into four layers [ML06]. The *Test specification* layer describes how the tester will express the tests. Two extreme techniques are proposed: (i) *data-centered approach*, which consists of the validation of data structures, for example, a predefined SOAP message is compared to the one sent by the process under test (PUT); and (2) *logic-centered approach*, in which a programming language is used to describe all test logic.

The *Test organization* layer characterizes the tests arrangement. An approach can consist on grouping tests in test suites to assist the developer usage. *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 assisted 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. The last layer (*Test Results*) is responsible for gathering the results and statistics obtained and presenting them to the end-user.

Based on the architecture described above, Mayer and Lübke developed the BPELUnit framework. The developer specifies the test in XML. To allow fast testing, SOAP details received and sent by the PUT are hidden from 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 the PUT itself. As previously mentioned, all the choices on the execution layer depend on the used BPEL engine. BPELUnit supports this diversity with the development of adapters for each BPEL engine. It uses the real-life approach that deploys the application PUT, executes the tests, and undeploys it.

An Online testing application: Greiler proposes a software application for detecting faults during a dynamic reconfiguration of a service composition [GGvD10] . The goal is to apply online testing to detect five types of faults:

- Publishing: incorrect service description, or deployment;
- Discovery: discovering no service, or the wrong service;
- Composition: missing or incompatible components (integration faults);
- Binding: binding denied, or referenced to a wrong service;
- Execution: service crashed, or incorrect results.

During an online reconfiguration, the service to be integrated (new service) is deployed, in the production environment, in parallel with the old service. In this approach, the service implementation is not checked against its own specification (interface), but against the expectation of another requesting service. Each service, including the new service, contains a test suite to assess the functionalities of its required services. Before being published and integrated, the test suite of the new service is executed in three steps. First, *discovery tests* are applied to check whether all required services can be discovered in the registry. Second, *binding tests* are applied to check whether the required services can be bound, and to validate their interfaces. Finally, *composition tests* execute the required services to validate the message exchange. If all tests pass, the new service is published and can replace the old one.

According to Greiler, specially during online testing, the services must be aware that they are being tested and test activities must be isolated. Thus, in this approach, the services are deployed in two forms: operational and testable. The operational service responds to the real requests in the composition. The testable service extends the operational service by inheritance or delegation and provides mechanisms for other services to test the service. To validate this approach, a software prototype[12] was developed based on the Open Services Gateway Initiative (OSGI)[13] SOA framework for publishing and deploying the services.

3.3. Comparison with related works

Regarding these related works , none of them is specific for or favors TDD of choreographies. However, there are some points of intersection between some of these works and our framework. The most significant similarities and differences are presented below.

3.3.1. Testing of atomic web services

Taking into account the related works for testing of atomic web services, we can highlight the following aspects.

SoapUI. We can classify SoapUI as an internal dependency of our work. Our feature for generating clients dynamically uses SoapUI to build the Soap envelopes. Originally, SoapUI consists on a graphical IDE for applying functional and performance testing on web services, but it is also available as an API. Since the simpler version of SoapUI is Open Source (under the LGPL License), we developed our dynamic client based on this API.

WS-TAXI. This tool provides mechanism for generating test cases automatically. This is one of our future goals, but differently from WS-TAXI, we want to generate test cases in Java instead of XML.

TTR. Differently from TTR (Test-The-REST), our approach for testing RESTful services is not based on XML. Our client for REST (presented in Section **??**) is built upon the tool TestAssured[14]. As a result, the tests can be written using Java.

SOCT. For the time being, we do not intend to measure the testing coverage of web services. Besides, such approach requires an adaptation of the traditional SOA triangle. We aim at providing a feature that does not require this adaptation.

Audition framework and (role)CAST. Differently from these two works, the goal of our framework is to assist in the development of the choreography, or in other words, we aim to apply test at development-time. The Audition Framework and (role)CAST focus on runtime testing, also called online testing. Since the authors of these works are also members of the CHOReOS project, as a future work, our framework can be integrated to these works to provide also runtime testing.

Mobile agent-based framework. Our framework acts at the client side of interaction instead of acting at the provider (server) side. A major problem of applying performance testing of web

[12]Online tool: `http://swerl.tudelft.nl/bin/view/MichaelaGreiler/OnlineTesting`
[13]OSGI: `http://www.osgi.org`
[14]http://code.google.com/p/rest-assured

services lies on the latency of the network. The mobile agent-based framework overcomes this problem by executing the tests at the server side. Thus, it is possible to measure the response time of the service discounting the time spent with communication. Since our long term goals are also to apply performance and QoS testing, this related work can be our starting point to support these kinds of testing.

3.3.2. Testing of Choreographies

Regarding the works related to testing of web services choreographies, we can highlight the following aspects:

PI4SOA and CDLChecker. Although these tools are related to choreography testing, they only provide mechanisms for validating the message exchange using simulation. We are interested in validating this exchange invoking the choreography operations under testing. CDLChecker could be used during the choreography specification (design time) to validate the choreography models. However, both the PI4SOA and CDLChecker tools are coupled to the WS-CDL language while we are interested in testing choreographies specified in other languages, for instance, BPMN2.

BPELUnit. Since a choreography can be composed of distributed orchestrations (see Section 2.3.3), our work is related to the fundamentals of BPELUnit. More specifically, we want to explore more and then to adapt the architecture in layers for BPEL testing proposed. The tests for BPELUnit must be written in XML, and this tool is available as an Eclipse plugin. We aim to develop a framework that provides features for writing tests in Java. Moreover, we do not have the intention to be restricted to BPEL, we aims to explore choreographies and orchestration defined in other languages.

A online testing application [GGvD10]. Differently from the goal of this online application, we aim at providing mechanisms for testing at development-time, defined as offline testing by the authors of this related work. Besides, another major difference lies on the technologies used. The online tests proposed can only be applied on web services that are developed and executed under the OSGI SOA framework. We aim at supporting the testing of traditional Soap and REST services. In spite of these differences, we are also interested on keeping the controllability and isolation of testing. To achieve that, we aim at providing the service mocking feature for minimizing or avoiding side effects on the internal state of the services under testing.

4. Rehearsal: Our testing framework

This proposed framework is part of the CHOReOS[15] and Baile[16] projects. The goals of these projects are to study web service choreographies and Clouds in large-scale environments which involve thousands of users and hundreds of web services. More specifically, for both projects, this framework belongs to the research lines related to Verification and Validation (V&V) of choreographies. In this context, the main contribution of this work lies on providing features for the automated testing of choreographies at development-time.

Enactment corresponds to the term used for designating a choreography that is already running. The tests we want to provide aims at supporting the development (construction) of the choreography. For this reason, during the testing activities, we can say that the choreography is being rehearsed. When all tests have passed, the final choreography can be enacted. In the following sections, we present in detail Rehearsal: our framework for automated testing of web service choreographies.

4.1. Supported testing strategies

During the choreography development, Rehearsal will provide features for testing the entire process of composing services. This process starts at the selection of the services that will be integrated and ends when the entire choreography is tested. To achieve that, based on the works presented in Section 2.4, we propose multiple levels of testing. As depicted in Figure 4, during the development of a choreography (represented by the external circles), unit, integration and acceptance strategies are applied. We describe in detail each one of these techniques in the next sections.

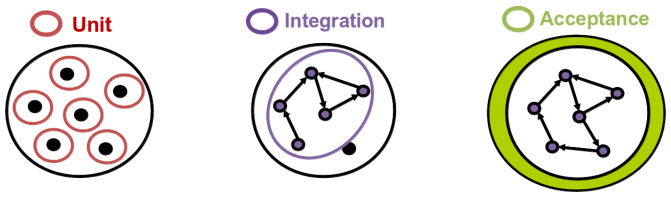


Figure 4. Levels of testing

4.1.1. Unit testing

Unit tests focus on verifying the behavior of small units of software, which can be a single class or a method. In the choreography context, we consider the small unit of software as a web service, thus, our unit tests validate the service behavior by verifying each provided functionality. To achieve that, each functionality is verified using black-box testing. As described in Section 2.3.3, to play a role, a node can also be a composition, for instance an orchestration of atomic services [Pel03]. By definition, an orchestration is accessible as an atomic service from a user perspective and unit tests can be applied to validate the role behavior. In this case, a role is a unit, and we are applying conformance tests to validate whether the role is being played correctly or not.

[15]CHOReOS project: `http://www.choreos.eu`
[16]Baile project: `http://ccsl.ime.usp.br/baile`

4.1.2. Integration testing

Integration tests aim to solve the problems produced when unit tested components are integrated. We propose an approach for applying integration testing based on message exchange validation. Thus, we propose two levels of integration:

- *Role*: during the composition of web service(s) into a role, the messages exchanged inside the local orchestration (that defines the role) are validated;
- *Choreography*: during the composition of roles into a choreography, the messages exchanged by the roles are validated.

We propose an approach for supporting both levels of integration. After a web service is integrated, we verify whether the 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 to the expected values.

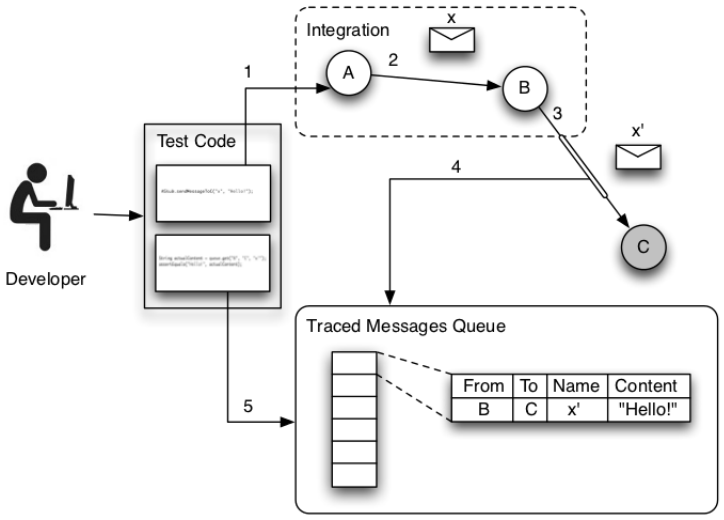


Figure 5. Integration test flow example

Figure 5 shows an example of this approach. As depicted in this figure, the developer is integrating the A and B services. At development-time, the developer specifies in the test code what services must be invoked as well as what messages must be intercepted and validated. In the Figure 6 we present a draft version of a test code for validating the integration of the services A and B. In this example, we want to validate the message sent from service B to service C.

```
serviceA.invoke("x", "Hello!");

String actualContent = queue.get("B", "C", "x'");
assertEquals("Hello!", actualContent);
```

Figure 6. Test code

During the execution of this test code, in the first step of Figure 5, after the services deployment, the framework invokes the service A. Then, service A sends a message to service

B (step 2). Our framework collects the output message from B and stores them in a queue, this is performed in the third and fourth steps. When the execution is over, the collected data is validated against the expected results (step 5).

4.1.3. Acceptance testing

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

4.2. Choreography development process

As explained before, one of the goals of CHOReOS and Baile projects is to support the development of choreographies involving thousands of users and hundreds of web services. In such context, a developer can develop the choreography from scratch, or from an existing choreography. Since choreographies are fully distributed, the latter scenario is more reasonable. Figure 7 illustrates this scenario.

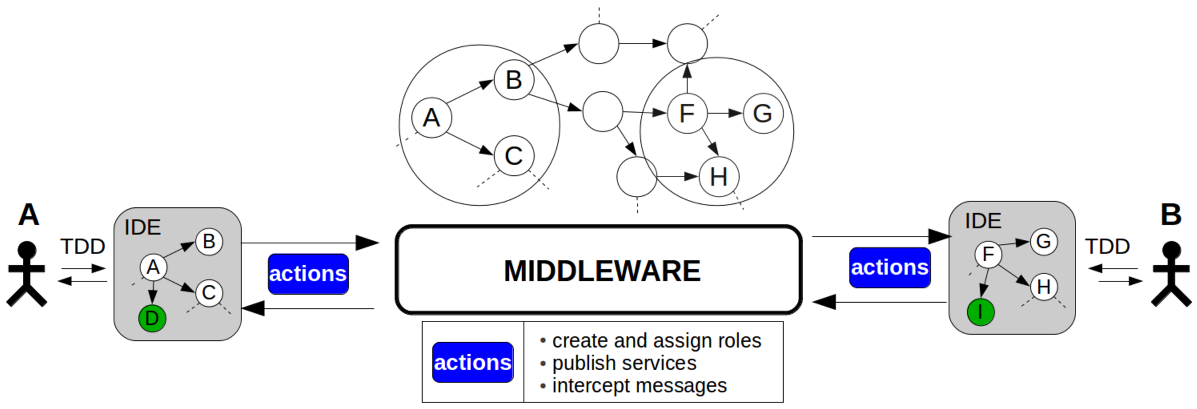


Figure 7. Choreography development process

As shown in Figure 7, a choreography is already enacted and the developers A and B are adding new roles on it. Both developers can interact with the choreography by using the CHOReOS Middleware [VIG+10]. The Middleware aims at providing all the required software infrastructure to enact and to interact with an enacted web service choreography. Thus, the developer A, for instance, can develop and integrate the new role D by applying the actions depicted in this Figure. Even if the choreography has been developed from scratch, the developer can also interact with the middleware by performing these actions to build and enact the choreography. We consider as choreography development the process composed at least of the following activities:

- Adaptation or creation of web services or roles;
- Integration of web service into roles;
- Integration of existing or new roles to create the choreography;

- Validation of a part or the entire choreography.

The CHOReOS project proposes a development process model to cover the above activities and other ones related to the choreography specification, governance, and runtime adaptation. Figure 8 presents the activities, artifacts, and their relationships within the CHOReOS development process model. Our framework can be used in combination with other development process models but, within the CHOReOS project, Rehearsal provides features for applying TDD in the activities highlighted (not colored in gray) in Figure 8.

As described briefly in Section 1.2, we aim to support Test-Driven Development (TDD) of choreographies. Since our integration tests are based on messages exchange, our framework provides features for applying TDD in the activity: "Monitoring and V&V Planning". The V&V planning artifacts consist of the integration tests written by the developer before the choreography implementation. Then, during the test execution, our framework monitors the messages by intercepting and validating their contents. All test cases writing using our framework, and needed artifacts (e.g., files for the configuration of the development environment and servers) are part of the artifact "Monitoring and V&V Configuration".

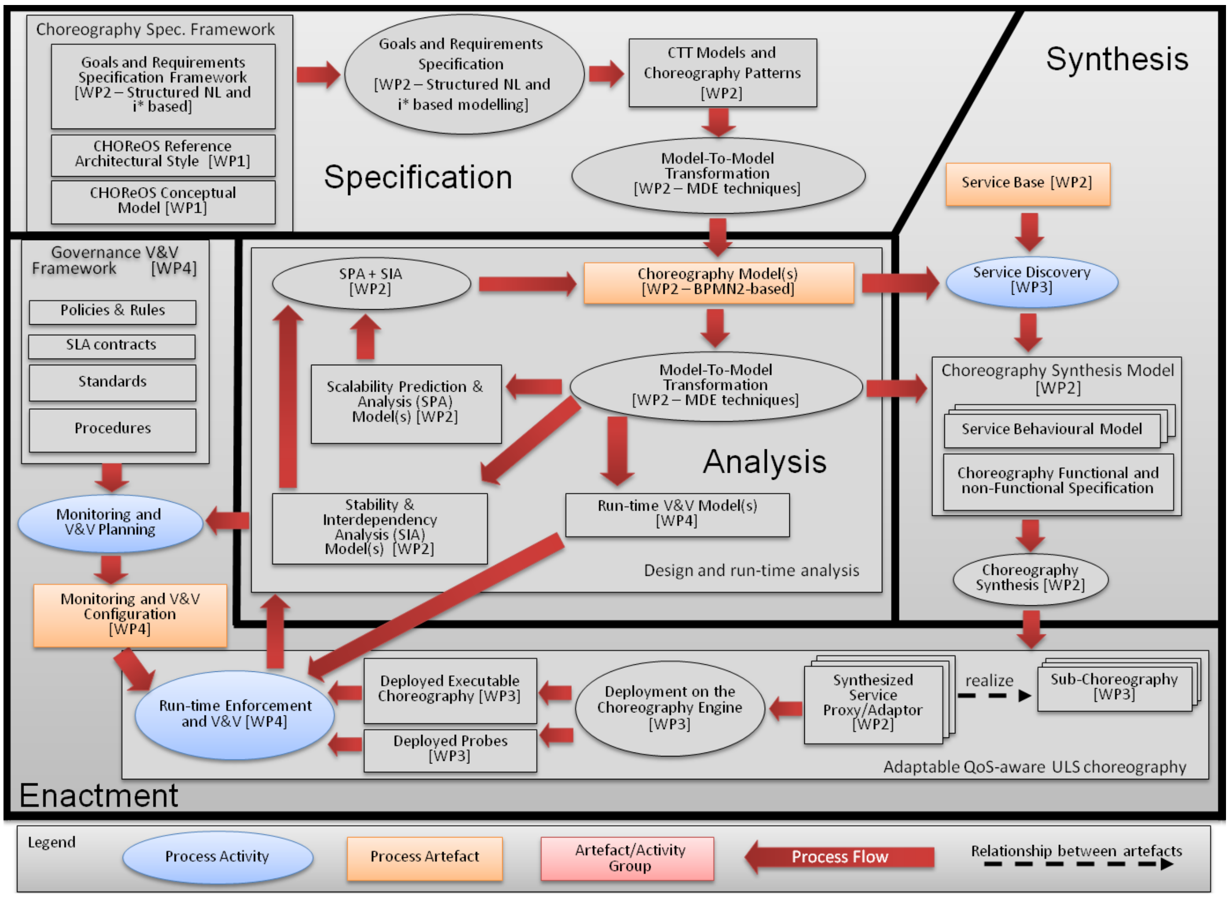

Choreography Spec. Framework
Goals and Requirements Specification Framework [WP2 – Structured NL and i* based]
CHOReOS Reference Architectural Style [WP1]
CHOReOS Conceptual Model [WP1]
Goals and Requirements Specification [WP2 – Structured NL and i* based modelling]
CTT Models and Choreography Patterns [WP2]
Model-To-Model Transformation [WP2 – MDE techniques]
Specification
Synthesis
Service Base [WP2]
Service Discovery [WP3]
Choreography Synthesis Model [WP2]
Service Behavioural Model
Choreography Functional and non-Functional Specification
Choreography Synthesis [WP2]
Governance V&V Framework [WP4]
Policies & Rules
SLA contracts
Standards
Procedures
Monitoring and V&V Planning
Monitoring and V&V Configuration [WP4]
SPA + SIA [WP2]
Choreography Model(s) [WP2 – BPMN2-based]
Scalability Prediction & Analysis (SPA) Model(s) [WP2]
Model-To-Model Transformation [WP2 – MDE techniques]
Analysis
Stability & Interdependency Analysis (SIA) Model(s) [WP2]
Run-time V&V Model(s) [WP4]
Design and run-time analysis
Run-time Enforcement and V&V [WP4]
Deployed Executable Choreography [WP3]
Deployed Probes [WP3]
Deployment on the Choreography Engine [WP3]
Synthesized Service Proxy/Adaptor [WP2]
realize
Sub-Choreography [WP3]
Adaptable QoS-aware ULS choreography
Enactment
Legend
Process Activity
Process Artefact
Artefact/Activity Group
Process Flow
Relationship between artefacts

Figure 8. CHOReOS development process model

During their execution, unit, integration, and acceptance tests are validated against models that specify the choreography. Thus, these choreography models are an input for validation applied by Rehearsal. In addition, our framework interacts with service bases and service discovery entities to bind and invoke the services under testing. These entities are also depicted in Figure 8. Since our tests are always applied in a running choreography, Rehearsal is related to the activity: "Runtime enforcement and V&V". However, in our case the runtime enforcement activities are applied in a development or testing environment, and not in the final production environment.

To support these activities and other ones related to other development process models, Rehearsal will provide the features explained in the following sections.

4.3. Framework features

To understand how to apply the testing strategies deeply to cover the activities belonging to the choreography development process, we developed a software prototype [BLK+11b, BLK+11a]. Our prototype consisted of: (a) *ad hoc* bash scripts for choreography enactment; (b) JUnit[17] test cases for applying automated unit, integration, and acceptance tests in a running choreography, and (c) a user interaction prompt for executing the scripts and tests.

To validate our prototype, we implemented an example choreography for booking a trip using the OpenKnowledge(OK)[18] framework. In this example, a user plans to go on 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. To provide such features, the *Traveler* service interacts with other services such as: *Airline*, *Acquire*, and *Travel Agency*.

The choreography participants were essentially SOAP/WSDL services and RESTful web services. Based on the experience and results acquired from this software prototype, we derived the main features this framework must provide to achieve our goals. These features are explained in the next sections.

4.3.1. Abstraction of choreography

To facilitate the testing writing, we proposed an approach for abstracting a choreography into Java objects. With this feature, all the choreography elements (partners, services, messages exchanged, etc.) can be represented by Java objects. Given a choreography specification, the developer can write unit, integration, and acceptance tests for the internal elements of the choreography.

As can be seen in Figure 9, the *Choreography* object is created from a choreography diagram (e.g., BPMN2) and corresponds to a composite in which the developer can interact with all choreography services. Each *Service* object is associated to one or more choreography roles. Through this object, the developer can retrieve the WSDL URI to invoke the desired services. Web service compositions are recursive, a *Service* object can be a composition of other services. Thus, from a *Service* object, a list of of its internal services can be extracted.

Using these objects, the developer can write tests before implementing the choreography. During test writing (design-time) the elements specified in the test cases are not checked. Then, in the first phase of execution, all of these Java objects (e.g., a role name, endpoints,

[17]JUnit: `http://www.junit.org`
[18]OpenKnowledge project: `http://www.openk.org`

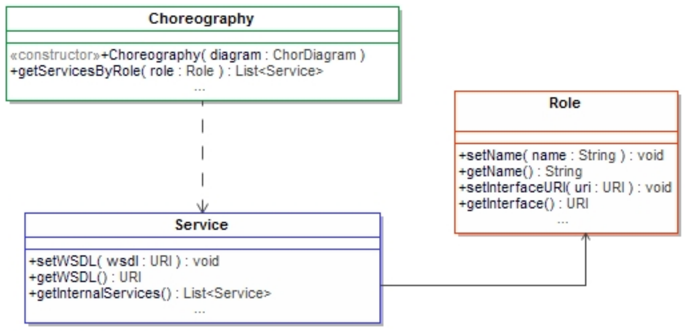

Choreography
«constructor»+Choreography(diagram : ChorDiagram)
+getServicesByRole(role : Role) : List<Service>
...
Role
+setName(name : String) : void
+getName() : String
+setInterfaceURI(uri : URI) : void
+getInterface() : URI
...
Service
+setWSDL(wsdl : URI) : void
+getWSDL() : URI
+getInternalServices() : List<Service>
...

Figure 9. Public interface of Abstraction Choreography

and message names) are validated against the BPMN2 diagram. As explained previously, this framework depends on other systems belonging to the CHOReOS and Baile projects. One of the framework dependencies is the CHOReOS Middleware. Thus, on the second phase of execution, the proposed framework will interact directly with the middleware to invoke the operations specified by the test cases.

In addition, this feature can be used with other Rehearsal features as explained in the next sections. For example, the endpoints extracted from a *Role* object can be dynamically invoked using WSClient, or mocked using the *Service Mocking* feature.

4.3.2. Dynamic generation of web service clients

For applying unit and acceptance testing, we aim at providing automated mechanisms where the developer can request service operations, and then, validate the response. During prototype development, we identified some tools for applying that, such as: Apache Axis[19] and JAX-WS[20]. With these tools, it is possible to create stub objects (also called clients) from a valid WSDL specification and to make requests to SOAP services. This creation process is not totally automated, human intervention is needed for creating and using the stub objects. Besides, if the WSDL specification of the requested service changes existing clients need to be generated again. To overcome these problems, we included in our framework a feature for the dynamic generation of web service clients. With this feature, the developer can interact with a service without creating stub objects. Given a web service interface (in WSDL), its operations can be requested dynamically. Our solution for Soap/WSDL services uses SoapUI (see Section 3.1.1) to build and retrieve Soap envelopes automatically. Figure 10 presents the execution process of our dynamic client for interacting with Soap/WSDL web services.

In the first step of Figure 10, the developer, using the object *WSClient*, specifies what operation must be invoked. This object represents our dynamic client and its public interface is described in Figure 11.

To request the desired operation, the developer invokes the method *request* providing the operation name and its arguments. This method supports as parameters, primitive types (e.g., int, String, and so on), or complex ones, which are represented by the object *Item*.

[19]Apache Axis: `http://axis.apache.org/axis`
[20]Jax-ws: `http://jax-ws.java.net`

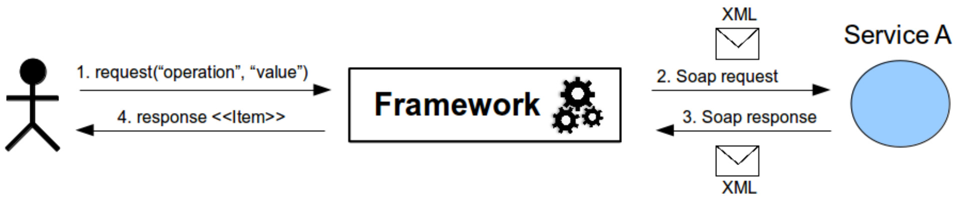


Figure 10. Execution process for interacting with Soap/WSDL services.

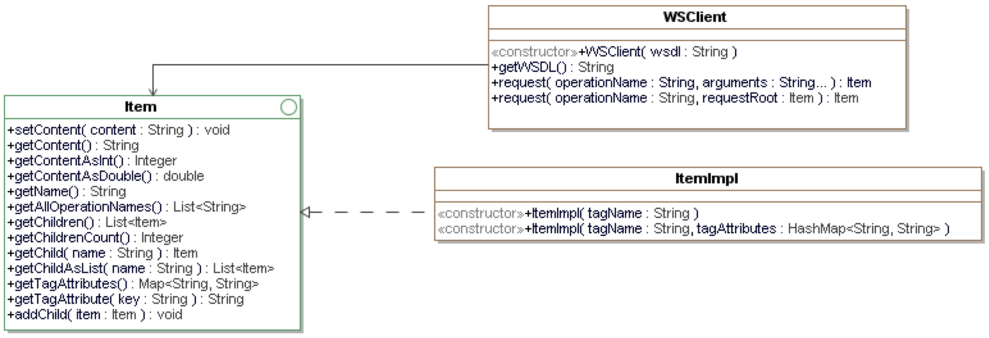

WSClient
«constructor»+WSClient(wsdl : String)
+getWSDL() : String
+request(operationName : String, arguments : String...) : Item
+request(operationName : String, requestRoot : Item) : Item
Item
+setContent(content : String) : void
+getContent() : String
+getContentAsInt() : Integer
+getContentAsDouble() : double
+getName() : String
+getAllOperationNames() : List<String>
+getChildren() : List<Item>
+getChildrenCount() : Integer
+getChild(name : String) : Item
+getChildAsList(name : String) : List<Item>
+getTagAttributes() : Map<String, String>
+getTagAttribute(key : String) : String
+addChild(item : Item) : void
ItemImpl
«constructor»+ItemImpl(tagName : String)
«constructor»+ItemImpl(tagName : String, tagAttributes : HashMap<String, String>)

Figure 11. Public interface of WSClient and related entities.

Within a Soap envelope, complex types are represented in XML. To avoid XML manipulation, we developed the *Item* object. This object consists of a recursive data structure for representing complex types that can be used as an operation parameter or for representing an operation response. Thus, during the testing of a service composition, *Item* objects representing a response received from a service can be directly used as input for the testing of other operation.

Figures 12 and 13 illustrate a Soap envelope containing complex types and its equivalent Item object. As can be seen in these figures, the developer does not have to specify the operation and parameters namespace information. In the example, we do not have to specify the namespace "ns" associated to the operation "getProductByName".

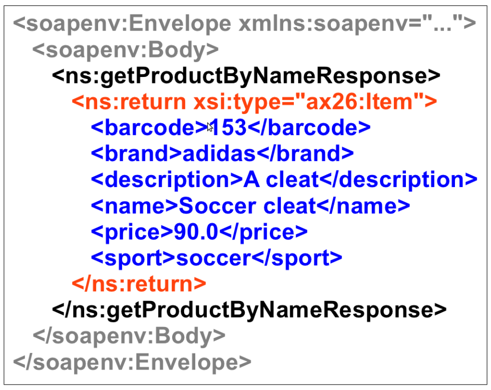

```
<soapenv:Envelope xmlns:soapenv="...">
  <soapenv:Body>
    <ns:getProductByNameResponse>
      <ns:return xsi:type="ax26:Item">
        <barcode>153</barcode>
        <brand>adidas</brand>
        <description>A cleat</description>
        <name>Soccer cleat</name>
        <price>90.0</price>
        <sport>soccer</sport>
      </ns:return>
    </ns:getProductByNameResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Figure 12. Soap response envelope

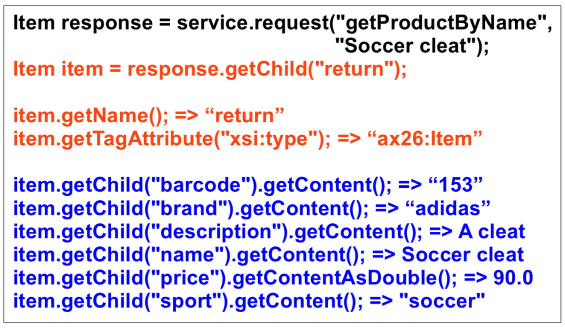
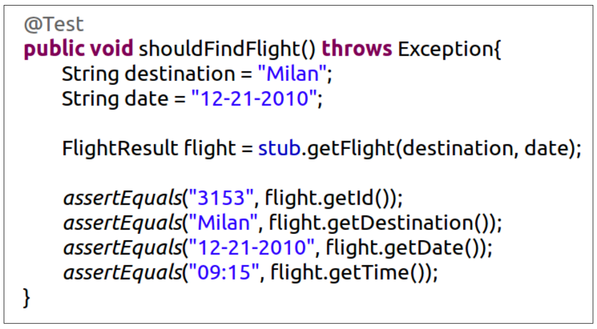

```
Item response = service.request("getProductByName",
                                "Soccer cleat");
Item item = response.getChild("return");

item.getName(); => "return"
item.getTagAttribute("xsi:type"); => "ax26:Item"

item.getChild("barcode").getContent(); => "153"
item.getChild("brand").getContent(); => "adidas"
item.getChild("description").getContent(); => A cleat
item.getChild("name").getContent(); => Soccer cleat
item.getChild("price").getContentAsDouble(); => 90.0
item.getChild("sport").getContent(); => "soccer"
```

Figure 13. Equivalent Item

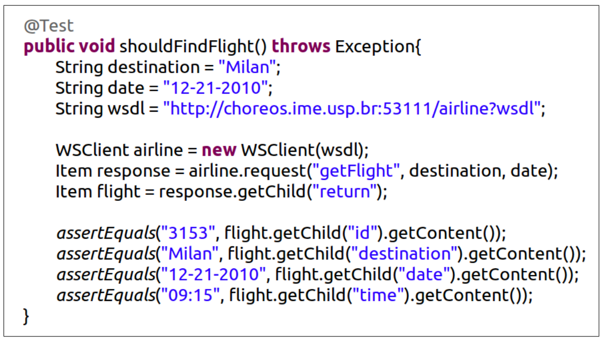
In the *WSClient* execution process (Figure 10), once the operation request has been defined, our framework creates and submits the corresponding Soap-request envelope to the service under testing (step 2). Then, in step 3, the framework collects the Soap-response envelope. Finally, in the step 4, the framework composes and returns the *Item* object representing the response.

To assess the *WSClient* object, we compared an unit test case written using it with the same test case written in our prototype, using JAX-WS. Figures 14 and 15 illustrate this comparison.



```
@Test
public void shouldFindFlight() throws Exception{
    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());
}
```

Figure 14. Software Prototype (test case A)



```
@Test
public void shouldFindFlight() throws Exception{
    String destination = "Milan";
    String date = "12-21-2010";
    String wsdl = "http://choreos.ime.usp.br:53111/airline?wsdl";

    WSClient airline = new WSClient(wsdl);
    Item response = airline.request("getFlight", destination, date);
    Item flight = response.getChild("return");

    assertEquals("3153", flight.getChild("id").getContent());
    assertEquals("Milan", flight.getChild("destination").getContent());
    assertEquals("12-21-2010", flight.getChild("date").getContent());
    assertEquals("09:15", flight.getChild("time").getContent());
}
```

Figure 15. WSClient (test case B)

As can be seen in these figures, for the code snippet depicted, the test case written for the prototype is four lines smaller than the test case written using *WSClient*. However, the test case A uses the object *FlightResult* which is a stub object generated by the JAX-WS. Since the service under testing provides other operations, other stub objects have been created and must be known for testing those operations. Thus, in this case, the code needed for testing is longer than the code presented. The test case B is independent of stub generations. The code snippet presented is only what the developer needs for testing this operation. Besides, if the service WSDL has changed, we just have to adjust the operation and parameters names, if needed.

We also included in the framework a feature for supporting the interaction with REST services. This feature facilitates the invocation of CRUD operations. After providing the URI of the REST service, the framework assists the developer to apply the POST, GET, PUT, or DELETE operation on the resources of the service under testing. Our solution for testing RESTful services is based on TestAssured (see Section 3.1.1). Figure 16 presents the execution process of our dynamic client for interacting with REST services.

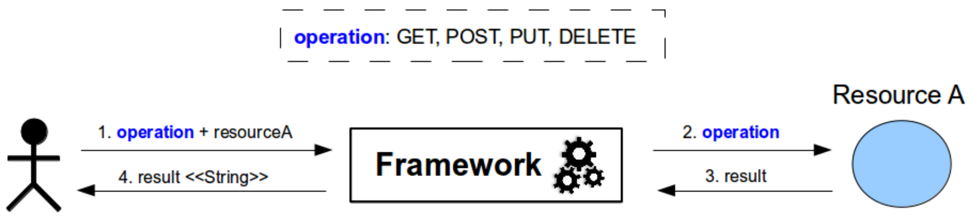
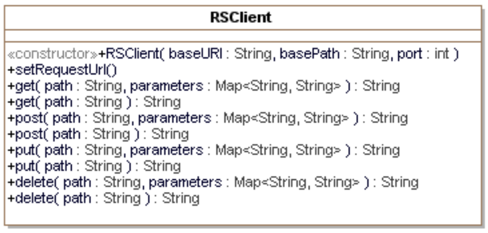


Figure 16. Workflow for interacting with REST services.

In step 1, to interact with the desired resource, the developer provides the service location and the desired operation (GET, POST, PUT or DELETE) that must be applied over the resource. This process is performed using RSClient, our dynamic client implementation for RESTful services. As depicted in Figure 16, the framework invokes the resource desired by the developer (step 2), collects the response, and then, presents the results to the developer, on steps 3 and 4, respectively.

The response of a REST service is not defined by a standard such as WSDL for Soap web services, the framework just returns the response content received from the service under testing. As discussed in Section 2.3.2, the response can be represented in different data formats such as HTML, text, XML, JSON, PDF, and JPG. Due to this fact, in the current version of this feature, the validation of the response content is up to the developer.

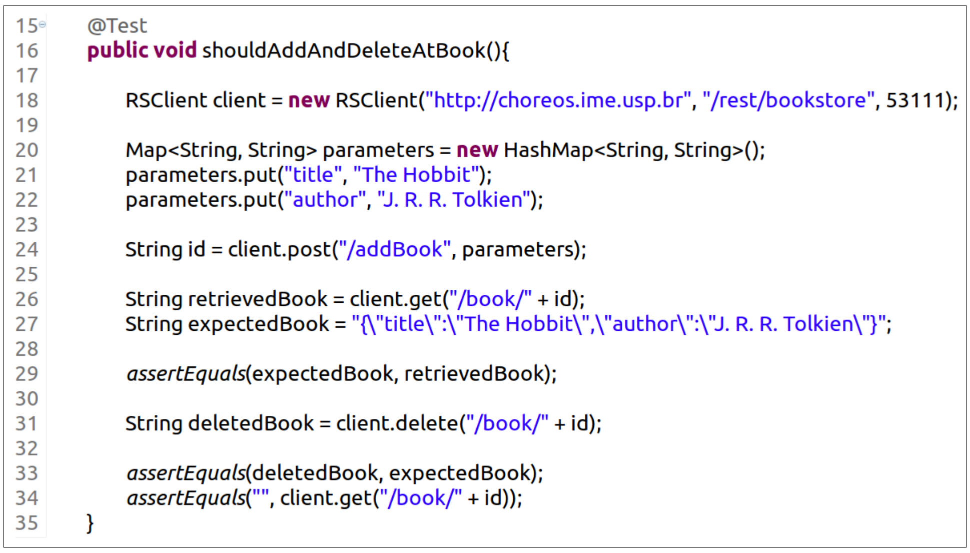
The most important advantages provided by RSClient are: (i) to abstract the communication issues for the developer and (ii) to provide a simple interface for applying the CRUD operation over the resources. Figure 17 presents the public interface of RSClient.



```
RSClient

«constructor»+RSClient( baseURI : String, basePath : String, port : int )
+setRequestUrl()
+get( path : String, parameters : Map<String, String> ) : String
+get( path : String ) : String
+post( path : String, parameters : Map<String, String> ) : String
+post( path : String ) : String
+put( path : String, parameters : Map<String, String> ) : String
+put( path : String ) : String
+delete( path : String, parameters : Map<String, String> ) : String
+delete( path : String ) : String
```

Figure 17. Public interface of RSClient

Figure 18 presents an example of application of GET, POST, and DELETE operations using RSClient. In this example, first, the developer specifies the service information such as its URI, base path, and also the port where the service is available. This information is specified inside the RSClient constructor method. Then, the POST operation is applied to create a new book (lines 20-24). To validate the resource creation, the GET operation is applied over the resource just created (“book”).



```
15      @Test
16      public void shouldAddAndDeleteAtBook(){
17
18          RSClient client = new RSClient("http://choreos.ime.usp.br", "/rest/bookstore", 53111);
19
20          Map<String, String> parameters = new HashMap<String, String>();
21          parameters.put("title", "The Hobbit");
22          parameters.put("author", "J. R. R. Tolkien");
23
24          String id = client.post("/addBook", parameters);
25
26          String retrievedBook = client.get("/book/" + id);
27          String expectedBook = "{\"title\":\"The Hobbit\",\"author\":\"J. R. R. Tolkien\"}";
28
29          assertEquals(expectedBook, retrievedBook);
30
31          String deletedBook = client.delete("/book/" + id);
32
33          assertEquals(deletedBook, expectedBook);
34          assertEquals("", client.get("/book/" + id));
35      }
```

Figure 18. RSClient: Example of usage

As a result, a JSON object is retrieved, and then validated. This is illustrated in lines 26–29. Finally, we want to delete the resource created. This is achieved by providing the book id in the DELETE operation. To validate the result, we apply a GET operation by providing the id of the book deleted and this operation must return an empty String. The lines 31–34 illustrate the DELETE operation. Notice that the developer just have to specify the test cases, all the HTTP communication issues are performed automatically.

4.3.3. Service mocking

The inter-organizational integration of services is one of the inherent characteristics of SOA. In spite of its advantages, this integration also brings difficulties for testing such as the absence of a testing environment for invoking services. Without a testing environment, some service operations, for instance the non-idempotent ones, cannot be tested completely. For those services, we cannot provide mechanisms for applying unit testing yet. Nevertheless, we aim at providing mechanisms for applying integration tests even when these services are involved. To achieve that, we proposed the usage of mock objects (see Section 2.2.5). This feature provides mechanisms for mocking all operations of a SOAP/WSDL that cannot be tested online.

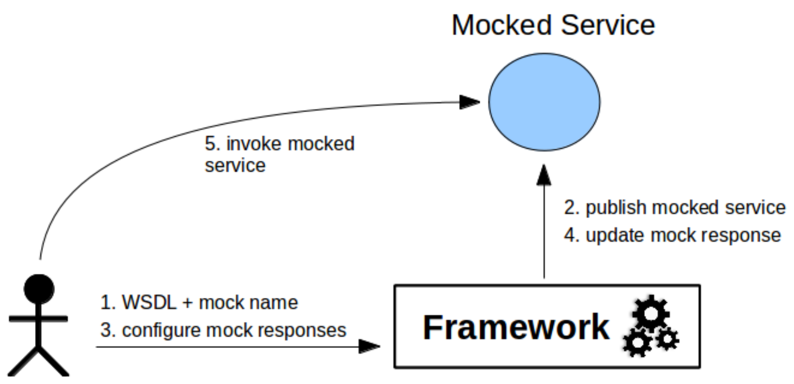


Figure 19. Workflow for mocking services

In the Figure 19, we present the basic workflow for using this Rehearsal feature. In the first step, the developer provides the real service WSDL URI and also the mock name. Then, the framework, using the SoapUI Mock API, builds and publishes the mock object letting it available in an URI composed by the mock name provided. It is also possible to define the host name and the port in which the mocked service will be published. Even if the mocked service is already running, the developer can change its response messages. As presented in the Figure 20, the WSMock object represents the mocked service. Through this object, the developer can define different responses. Moreover, this feature can assist in fault scenarios simulation. Using the features *doNotRespond* and *crash* is possible to configure the mock to do not respond or simulate a crash behavior, respectively.

In the third step of Figure 19, the actual mock behavior can be changed by the developer, and in the fourth step, the framework applies the changes in the running mock. Finally, the developer can invoke the mocked operation in the test cases (step 05). In the Figure 21, we have a concrete example of how to use the service mocking feature. In this example, we are mocking a real registry service which is published in the URI specified in line 12. In this

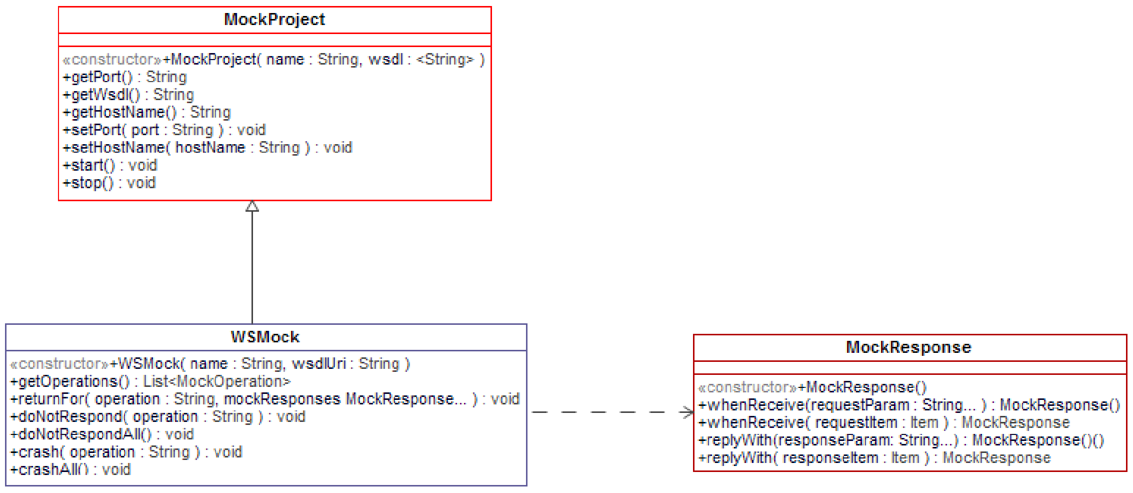
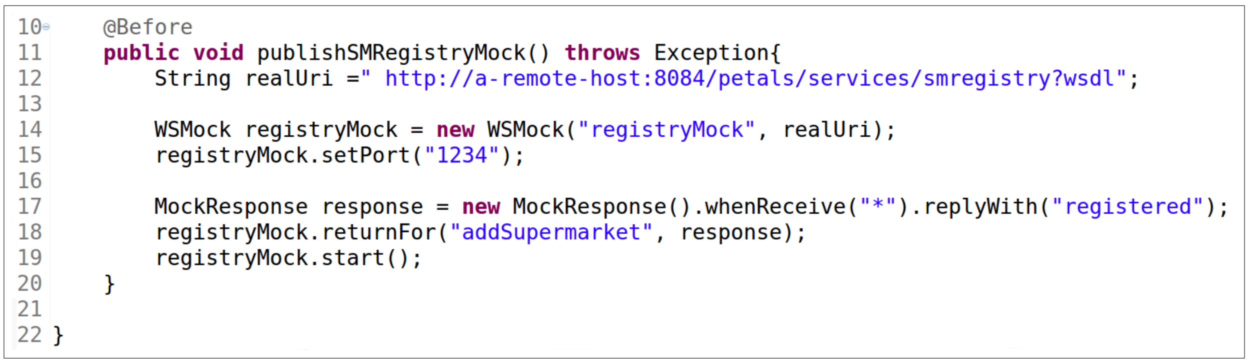

MockProject
«constructor»+MockProject(name : String, wsdl : <String>)
+getPort() : String
+getWsdl() : String
+getHostName() : String
+setPort(port : String) : void
+setHostName(hostName : String) : void
+start() : void
+stop() : void
WSMock
«constructor»+WSMock(name : String, wsdlUri : String)
+getOperations() : List<MockOperation>
+returnFor(operation : String, mockResponses MockResponse...) : void
+doNotRespond(operation : String) : void
+doNotRespondAll() : void
+crash(operation : String) : void
+crashAll() : void
MockResponse
«constructor»+MockResponse()
+whenReceive(requestParam : String...) : MockResponse()
+whenReceive(requestItem : Item) : MockResponse
+replyWith(responseParam: String...) : MockResponse()()
+replyWith(responseItem : Item) : MockResponse

Figure 20. Public interface of Service Mocking

example, the registry service is only available in the production environment, then, we can simulate the service to apply developer tests at development-time. Thus, in lines 14–15, we instantiate a *WSMock* object, in lines 17–18, we define the response this object must provide for any parameter received. Finally, in line 19, we started the mock and its WSDL interface is published on the URI:*http://localhost:1234/registryMock?wsdl*.



```
10    @Before
11    public void publishSMRegistryMock() throws Exception{
12        String realUri =" http://a-remote-host:8084/petals/services/smregistry?wsdl";
13
14        WSMock registryMock = new WSMock("registryMock", realUri);
15        registryMock.setPort("1234");
16
17        MockResponse response = new MockResponse().whenReceive("*").replyWith("registered");
18        registryMock.returnFor("addSupermarket", response);
19        registryMock.start();
20    }
21
22 }
```

Figure 21. Example of web service mocked with WSMock

4.3.4. Message interceptor

To apply our approach for integration tests, the messages exchanged among the services must be intercepted and validated, as described in details in Section 4.1.2. Our prototype supports this approach by instrumenting the choreography under testing. Then, the messages are stored and retrieved from a database server. In previous work [BLK+11a], we evaluated our prototype, in spite of being efficient, we detected some deficiencies. The instrumentation applied let our solution coupled to the Open Knowledge framework. Moreover, the prototype has only supported the interception of simple and short messages, basically Strings. However, in real choreographies, more complex messages (e.g., specified in XML) are exchanged.

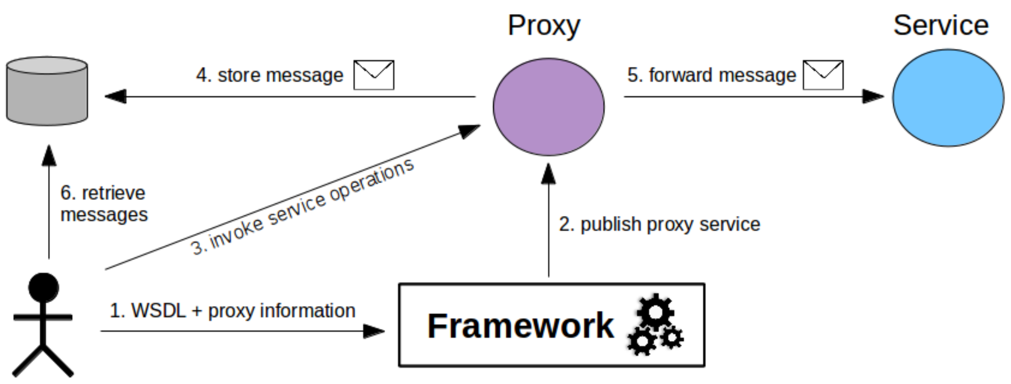


Figure 22. Workflow for intercepting messages

Our framework aims at implementing this approach. To overcome the deficiencies of our prototype, we provide features for: (i) intercepting and collecting complex messages; and (ii) retrieving the messages collected. In the Figure 22, we present the basic workflow for using the message interceptor feature. In the first step, the developer provides the WSDL URI of the service that will have its messages intercepted; and the proxy information, such as the port and the host name where the proxy will be deployed. Then, the framework will publish the proxy and the developer can interact with the service under testing through it (steps 02 and 03 respectively).

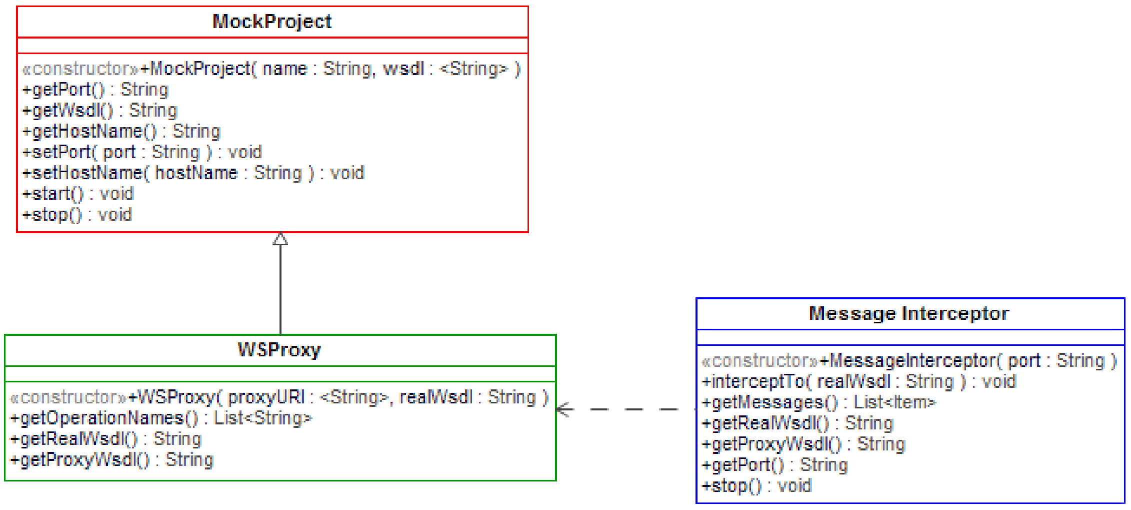
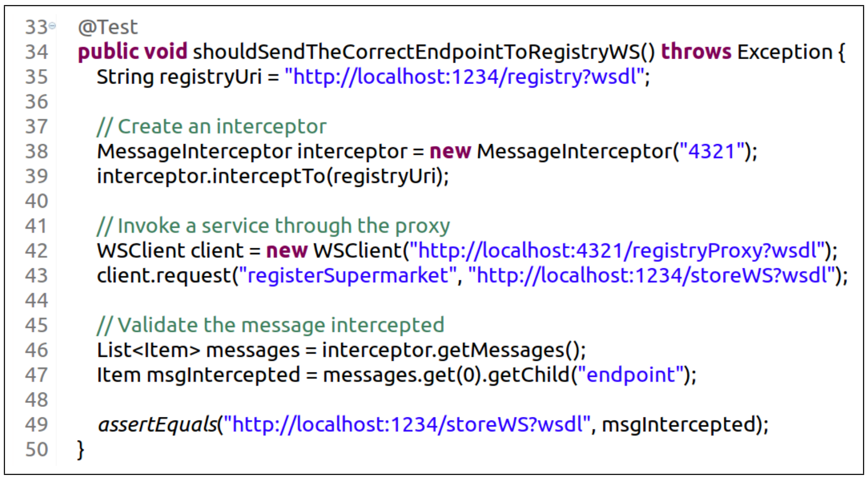

MockProject
«constructor»+MockProject(name : String, wsdl : <String>)
+getPort() : String
+getWsdl() : String
+getHostName() : String
+setPort(port : String) : void
+setHostName(hostName : String) : void
+start() : void
+stop() : void
WSProxy
«constructor»+WSProxy(proxyURI : <String>, realWsdl : String)
+getOperationNames() : List<String>
+getRealWsdl() : String
+getProxyWsdl() : String
Message Interceptor
«constructor»+MessageInterceptor(port : String)
+interceptTo(realWsdl : String) : void
+getMessages() : List<Item>
+getRealWsdl() : String
+getProxyWsdl() : String
+getPort() : String
+stop() : void

Figure 23. Public Interface of Message Interceptor

Similarly to WSMock, the proxy was developed using the SoapUI Mock API. In the step 04, all messages sent to real service will be intercepted by the proxy, stored, and then forward to the real service, in the step 05. The messages are stored in an instance of Message Interceptor. As can be seen in the Figure 23, through the Message Interceptor object the developer can define which service must be intercepted, interact with the proxy created, and also retrieve the intercepted messages, which is performed in the step 06. In Figure 24, we present a test case which uses the Message Interceptor to validate the messages sent to the registry service (line 35).



```
33  @Test
34  public void shouldSendTheCorrectEndpointToRegistryWS() throws Exception {
35    String registryUri = "http://localhost:1234/registry?wsdl";
36
37    // Create an interceptor
38    MessageInterceptor interceptor = new MessageInterceptor("4321");
39    interceptor.interceptTo(registryUri);
40
41    // Invoke a service through the proxy
42    WSClient client = new WSClient("http://localhost:4321/registryProxy?wsdl");
43    client.request("registerSupermarket", "http://localhost:1234/storeWS?wsdl");
44
45    // Validate the message intercepted
46    List<Item> messages = interceptor.getMessages();
47    Item msgIntercepted = messages.get(0).getChild("endpoint");
48
49    assertEquals("http://localhost:1234/storeWS?wsdl", msgIntercepted);
50  }
```

Figure 24. Message Interceptor Example

As depicted in Figure 24, in lines 38–39, a Message Interceptor is instantiated to intercept the messages sent to the WSDL URI specified in line 35. During the instantiation, a proxy providing the same WSDL interface of the real service will be published on the port *4321*. Then, through the proxy, the operation *registerSupermarket* belonging to the real service is invoked (lines 42–43). At this point, this message is intercepted, stored, and then. forwarded to the service published in the URI specified in the line 35. Finally, in lines 46 to 49, the intercepted messages are retrieved and their content validated.

4.3.5. Scalability testing support

The goal of the CHOReOS and Baile projects is to support the development of large scale choreographies which are composed of thousands of users and hundreds of services. In such context, the choreography should be developed to meet the required demand and scalability. For this reason, in addition to supporting functional testing, our framework will also support scalability testing. To assess the choreography or atomic service performance as long as the work load increases, we aim at providing features for applying multiple and parallel requests to services and for measuring the response time of these services. The goal is to verify whether the functional tests keep working properly when the choreography is exposed to different levels of workloads.

To assess how the choreography can scale when it is exposed to large scale workloads, we aim at providing a feature for creating more instances of existing services. With this feature, the developer can dynamically create and deploy new instances of the services under testing, and then, assess the new choreography performance. To implement that, our framework will interact with the *Node Pool Manager* which is a CHOReOS middleware component for resource allocation. Through this component, the framework can create, destroy, and deploy on a testing environment new instances of the services under testing.

5. Conclusions and Future Works

Rehearsal has been developed incrementally. All releases have been published monthly in the Baile Project website [21]. By the end of this year, apart from the "Abstraction Choreography Feature", all features presented in this technical report are almost completly developed. Currently, we are working on the code refactoring, to improve its quality, and on little feature improvements. In the case of the "Abstraction Choreography Feature", since this feature depends to the CHOReOS middleware system, which is still under development, we will develop *ad hoc* scripts that map choreography diagrams into Abstract Choreography objects, such as roles and services. With these scripts we can start the Rehearsal assessment.

We intend to assess the Rehearsal performance and effectiveness. In the case of performance assessment, execution time will be our metric. We intend to analyze the time overhead demanded from the Rehearsal features when attending complex test cases. To assess the Rehearsal effectiveness when applying Test-Driven Development (TDD), we intend to apply an experiment involving Computer Science Students. The students will be divided in groups to develop a choreography role using a TDD methodology. One group will use Rehearsal while the another group, existing testing tools. The goal is to study the benefits and difficulties faced by the each group during the development. The TDD methodology and the experiment protocol are under development. All results obtained with the performance and effectiveness assessment will be analyzed and reported in future publications.

[21]Rehearsal Page: `http://ccsl.ime.usp.br/baile/VandV`

References

[AM04] Alain Abran and James W. Moore. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Press, 2005 edition, 2004.

[Ast03] D. Astels. *Test-Driven Development: A Practical Guide*. Prentice Hall PTR, July 2003.

[BAP11] Antonia Bertolino, Guglielmo De Angelis, and Andre Polini. (role)CAST: A Framework for On-line Service Testing. In *7th Internation Conference on Web Information Systems and Technologies*, WEBIST, Noordwijkerhout, Netherlands, 2011.

[BBEM09] Cesare Bartolini, Antonia Bertolino, Sebastian Elbaum, and Eda Marchetti. Whitening SOA testing. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 161–170, New York, NY, USA, 2009. ACM.

[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.

[BBRW09] Adam Barker, Paolo Besana, David Robertson, and Jon B. Weissman. The benefits of service choreography for data-intensive computing. In *Proceedings of the 7th international workshop on Challenges of large applications in distributed environments*, CLADE '09, pages 1–10, New York, NY, USA, 2009. ACM.

[BDAP09] Antonia Bertolino, Guglielmo De Angelis, and Andrea Polini. On-line validation of service oriented systems in the European Project TAS3. In *Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, PESOS '09, Washington, DC, USA, 2009. IEEE Computer Society.

[BDO] Alistair Barros, Marlon Dumas, and Phillipa Oaks. A Critical Overview of the Web Services Choreography Description Language (WS-CDL). *BPTrends Newsletter*.

[Bec00] Kent Beck. *Extreme programming explained: embrace change*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[Bec03] Kent Beck. *Test-driven development: by example*. Addison-Wesley, Boston, 2003.

[Ber07] Antonia Bertolino. Software Testing Research: Achievements, Challenges, Dreams. In *2007 Future of Software Engineering*, FOSE '07, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[BHM$^+$04] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. W3C Note NOTE-ws-arch-20040211. Available on: ¡www.w3.org/TR/ws-arch/wsa.pdf¿, 2004.

[BLK$^+$11a] Felipe M. Besson, Pedro M.B. Leal, Fabio Kon, Alfredo Goldman, and Dejan Milojicic. Supporting Test-Driven Development of Web Service Choreographies. In *The 5th International Open Cirrus Summit*, Moscow, Russia, 2011.

[BLK$^+$11b] Felipe M. Besson, Pedro M.B. Leal, Fabio Kon, Alfredo Goldman, and Dejan Milojicic. Towards automated testing of web service choreographies. In *Proceeding of the 6th* *international workshop on Automation of software test*, AST '11, pages 109–110, Waikiki, Honolulu, HI, USA, 2011. ACM.

[BMS07] Antonio Bucchiarone, Hernán Melgratti, and Francesco Severoni. Testing Service Composition. In *8th Argentine Symposium on Software Engineering (ASSE'07)*, Mar del Plata, Argentina, 2007.

[BPaRG09] Paolo Besana, Vivek Patkar, D avid Robertson, and David Glasspool. Sharing Choreographies in OpenKnowledge: A Novel Approach to Interoperability. *Journal of Software*, 4:833–842, 2009.

[CDP09] Gerardo Canfora and Massimiliano Di Penta. Service-Oriented Architectures Testing: A Survey. In *Software Engineering*, volume 5413 of *Lecture Notes in Computer Science*, pages 78–105. Springer Berlin / Heidelberg, 2009.

[CHO11] CHOReOS. CHOReOS Perspective on the FI and initial conceptual model (D 1.2). http://choreos.eu/bin/Download/Deliverables, 2011.

[CK09] Sujit K. Chakrabarti and Prashant 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 Eduardo 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.

[DMJ07] Márcio Eduardo Delamaro, José Carlos Maldonado, and Mario Jino. *Introdução ao teste de software*. Elsevier Editora Ltda, 2007.

[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.

[Fow07] Martin Fowler. Mocks Aren't Stubs. Available on: http://martinfowler.com/articles/mocksArentStubs.html, 2007.

[Fow11] Martin Fowler. Test-Driven Development. Available on: http://www.martinfowler.com/bliki/TestDrivenDevelopment.html, 2011.

[FP09] Steve Freeman and Nat Pryce. *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley Professional, 1 edition, 2009.

[GGvD10] Michaela Greiler, Hans-Gerhard Gross, and Arie van Deursen. Evaluation of online testing for services: a case study. In *Proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems*, PESOS '10, pages 36–42. ACM, 2010.

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

[HP11] HP. Loadrunner. Available on: ¡http://www8.hp.com/us/en/software/software-product.html?compURI=tcm:245-935779¿, 2011.

[IBM11] IBM. Aglets. Available on: ¡http://www.trl.ibm.com/aglets¿, 2011.

[Jef] Ron Jeffries. What is extreme programming?

[Liu09] Henry H. Liu. *Software Performance and Scalability: A Quantitative Approach (Quantitative Software Engineering Series)*. Wiley, 2009.

[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.

[Ngu01] Hung Q. Nguyen. *Test Applications on the Web*. Wiley Computer Publishing, 2001.

[Pel03] Chris Peltz. Web Services Orchestration and Choreography. *Computer*, 36:46–52, October 2003.

[PGFT11] Marcos Palacios, José García-Fanjul, and Javier Tuya. Testing in Service Oriented Architectures with dynamic binding: A mapping study. *Information and Software Technology*, pages 171–189, March 2011.

[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.

[PZL08] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. "big" web services: making the right architectural decision. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, pages 805–814, New York, NY, USA, 2008. ACM.

[RPIT11] Hazlifah Mohd Rusli, Mazidah Puteh, Suhaimi Ibrahim, and Sayed Gholam Hassan Tabatabaei. A comparative evaluation of state-of-the-art web service composition testing approaches. In *Proceeding of the 6th international workshop on Automation of software test*, AST '11, pages 29–35, New York, NY, USA, 2011. ACM.

[RR07] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly, 2007.

[Rt05] Stephen Ross-talbot. Orchestration and Choreography: Standards, Tools and Technologies for Distributed Workflows. In *NETTAB Workshop - Workflows management: new abilities for the biological information overflow*, 2005.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[VIG$^+$10] Hugues Vincent, Valérie Issarny, Nikolaos Georgantas, Emilio Francesquini, Alfredo Goldman, and Fabio Kon. CHOReOS: scaling choreographies for the internet of the future. In *Middleware '10 Posters and Demos Track*, Middleware Posters '10, pages 8:1–8:3, New York, NY, USA, 2010. ACM.

[WZZ$^+$10] Zheng Wang, Lei Zhou, Yongxin Zhao, Jing Ping, Hao Xiao, Geguang Pu, and Huibiao Zhu. Web Services Choreography Validation. *Service Oriented Computing Applications*, 4, December 2010.

[Zha11] Jia Zhang. A Mobile Agent-Based Tool Supporting Web Services Testing. *Wireless Personal Communications*, 56:147–172, 2011.

[ZPX$^+$10] Lei Zhou, Jing Ping, Hao Xiao, Zheng Wang, Geguang Pu, and Zuohua 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.