Journal of Systems and Software • October 2014 •
http://www.sciencedirect.com/science/article/pii/S016412121400209X

# Bringing Test-Driven Development to Web Service Choreographies

Felipe Besson, Paulo Moura, Fabio Kon

Department of Computer Science
University of São Paulo
{besson, pbmoura, kon}@ime.usp.br

Dejan Milojicic

Hewlett Packard Laboratories
Palo Alto, USA
dejan.milojicic@hp.com

**Abstract**

*Choreographies are a distributed approach for composing web services. Compared to orchestrations, which use a centralized scheme for distributed service management, the interaction among the choreographed services is collaborative with decentralized coordination. Despite the advantages, choreography development, including the testing activities, has not yet evolved sufficiently to support the complexity of the large distributed systems. This substantially impacts the robustness of the products and overall adoption of choreographies. The goal of the research described in this paper is to support the Test-Driven Development (TDD) of choreographies to facilitate the construction of reliable, decentralized distributed systems. To achieve that, we present Rehearsal, a framework supporting the automated testing of choreographies at development-time. In addition, we present a choreography development methodology that guides the developer on applying TDD using Rehearsal. To assess the framework and the methodology, we conducted an exploratory study with developers, whose result was that Rehearsal was considered very helpful for the application of TDD and that the methodology helped the development of robust choreographies.*

***Keywords:*** *Automated testing, Test-Driven Development, Web service choreographies*

## I. Introduction

Service-Oriented Architecture (SOA) is a set of principles and methods that uses services as the building blocks for the development of distributed applications. Web services can be composed to create more complete services and then to implement complex business workflows. Different from orchestrations, choreographies are a more scalable approach for composing services (Guimaraes et al., 2012). The interaction among the choreographed services is collaborative: coordination is distributed across all choreography participants. Locally, each choreography participant is only concerned with the actions it must take to play the desired role. For this reason, choreographies are a good candidate architecture for fully decentralized workflows (Barker et al., 2009).

Over the years, the way information is created, shared, used, and integrated in the Internet is changing at a fast pace. As a result, the current Internet is evolving towards the Future Internet, which is foreseen as a federation of services that will provide built-in mechanisms such as scalable service access, mobility of network and devices, in a secure, reliable, and robust way (Tselentis et al., 2010). With respect to service access, service choreographies are proposed as an adequate architecture to deal with the large scale nature of the Future Internet, which can be translated into a high number of interacting entities, parallel single service accesses, and massive service load (Issarny et al., 2011).

A few standards have been proposed for modeling choreographies, such as the Web Services Choreography Description Language (WS-CDL), a W3C standard candidate proposed in 2004,

and, more recently, the OMG Business Process Model and Notation version 2 (BPMN2). However, up to now, none of them have experienced wide adoption. This is also true for development methods such as the one proposed by the Savara project (Madurai, 2009). Due to the inherent characteristics of SOA – such as dynamism, inter-organization integration, reuse of service – the automated, and even the manual, testing of choreographies is not conducted properly. This scenario results in choreographies implemented using *ad hoc*, often chaotic, development processes. As a consequence, choreography development activities cannot be performed properly. Normally, neither the functional behavior nor scalability of choreographies is verified or assessed properly.

The goal of our research is to apply Test-Driven Development (TDD) to choreographies to facilitate their construction and improve their adoption. TDD is a design technique that guides the development of software through testing (Beck, 2003; Fowler, 2011). With TDD, test cases are written before the code which they test, in a technique called *test-first programming*, which forces the developer to think about what could possibly go wrong even before the implementation itself begins (Erdogmus et al., 2005). Experiments and empirical observations in the industry have shown that TDD increases code quality and reduces defect density. A case study (Bhat and Nagappan, 2006) conducted in Microsoft assessed the impact of TDD in two different teams. In the first one, although the initial development time of the project using TDD increased by 35%, the density of defects decreased by 62%. In the second case, the initial development time increased 15% while the density of defects decreased 76%. Based on this favorable retrospect, our research hypothesis was that TDD has a good potential to aggregate more quality to choreography development.

This paper presents two major contributions:

1. Rehearsal, a novel framework for automated offline (development-time) testing of web service choreographies;

2. A choreography development methodology, which guides the developer to use the framework with TDD.

Both contributions were evaluated empirically via an exploratory study following sound Empirical Software Engineering principles.

Our research involved a three-year study on the requirements and architecture for provisioning a powerful tool and useful methodology for the development of robust choreographies. During this period, we carried out a comprehensive study of the academic literature on the subject, a detailed analysis of related software tools, and discussed the subject with tens of programmers, researchers, and practitioners from the software industry. Our research findings indicate that the tool and the methodology, as described in this paper, aid developers significantly in the complex task of building large-scale decentralized systems composed of collections of web services.

Differently from the related works, Rehearsal and the development methodology provide features and guidelines for testing and developing choreographies following agile software development concepts. Therefore, with this framework, the developer can write tests before the service implementation. Following the agile culture, the tests are created by developers based on the choreography specification. This process is performed incrementally, and the tests guide the design of the choreographed services. During the service integration, the exchanged messages can be intercepted and validated through service proxies, which helps the developer to monitor and debug the choreography behavior at the development environment. Moreover, Rehearsal provides a feature to emulate (mock) third-party services that cannot be used during offline tests. This feature can be invoked through a fluent interface that is similar to the interface of Java mocking

tools (e.g., Mockito[1] and EasyMock[2]).

This paper is structured as follows. Section II provides a brief background, a practical choreography example, and requirements for a comprehensive testing infrastructure to deal with real problems arising from testing choreographies. Section III discusses related work and its relationship with our approach. In Sections V and VI, we present Rehearsal and our methodology proposal and how these artifacts address the problems presented in our choreography example. To assess both artifacts, an exploratory study has been conducted with advanced Computer Science students. The study design and obtained results are presented and discussed in Section VII. Then, an industrial validation carried out in the context of CHOReOS project is summarized in Section VIII. Finally, we draw our conclusions and discuss ongoing and future work in Section IX.

## II. Fundamental concepts

In this section, we present a brief introduction to web service choreographies and Test-Driven Development (TDD), two fundamental concepts related to this work. Then, we present the FutureMarket, an example of a choreography for distributed shopping. This example guides the explanation of the Rehearsal features in the next sections. Finally, we present the existing challenges of choreography testing that this work aims to cover.

### I. Web Service Choreographies

The ability of composing web services effectively is one of the critical requirements for service oriented computing (Erl, 2007). In this context, orchestrations and choreographies have been proposed as approaches for composing web services. Orchestrations correspond to a centralized approach where internal and external web services are composed into an executable business process (Peltz, 2003). Some standards, such as the Business Process Execution Language (BPEL)[3], have been proposed for orchestrating services. In an orchestration, a central party (node) controls the interaction flow of the other parties, unlike in choreographies, where the control is decentralized.

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 (Barker et al., 2009). Therefore, when the choreography is started (enacted), there is no central entity driving the interaction of the whole choreography. In a web service choreography, each role is formally specified through a Web Service Description Language (WSDL) document.

Since orchestrations are executable processes, choreographies can be executed via distributed orchestrations (Autili and Ruscio, 2011). In this approach, the developer associates an orchestration to each choreography role. During the choreography information flow, many coordinators (orchestrations) are then responsible for different parts of the flow. In this manner, there is no entity keeping a global interaction state or, in other words, coordinating the entire business flow.

In an abstract (higher) level, the messages exchanged among the choreography roles are specified by using modeling languages (e.g., BPMN 2). However, in an executable (lower) level, a role implementation consists of an orchestration of a service or a set of services. In this case, the WSDL interface exposed by the orchestration must be compatible with the interface required by the implemented choreography role.

[1] http://code.google.com/p/mockito
[2] www.easymock.org
[3] BPEL: http://www.oasis-open.org/committees/wsbpel

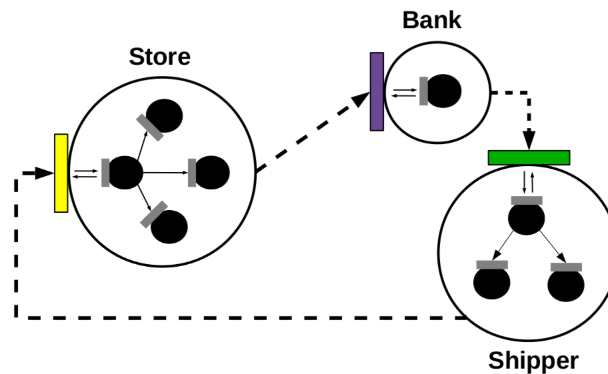


**Figure 1:** *Orchestration of services playing roles in a choreography (Peltz, 2003)*

Figure 1 depicts a role implementation through orchestrations. The orchestration in the left side implements the *Store* role. During the choreography execution, this orchestration sends a payment message to the *Bank* role that forwards part of the payment to the *Shipper* role. Finally, this orchestration sends a confirmation message back to the *Store* role. During the message exchange, the information flows across the choreography roles in a decentralized way.

## II. Test-Driven Development (TDD)

Test-Driven Development (TDD) consists of a design technique that guides software development through testing (Beck, 2003; Fowler, 2011). TDD can be summed up in the following iterative steps:

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

In addition to these steps, according to Astels (2003), to apply TDD, developers should follow principles such as: maintaining an exhaustive suite of programmer tests and 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 (Jeffries, 2011).

Having tests associated with the code, gives the developer confidence and courage to make changes and detect immediately (or in a short time) potential problems introduced into the code. 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) (Beck, 2004), 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 (Freeman and Pryce, 2009).

## III. FutureMarket Choreography

The FutureMarket consists of a web service choreography, which provides a service for distributed shopping following this workflow:

1. A customer provides a shopping list to the choreography;
2. The price of each list item is queried from multiple supermarket services to find which one has the lowest price;
3. The choreography returns to the customer the total, least expensive price of its list and provides features for purchasing and delivering the items.

To provide this workflow, the choreography is composed of services playing the *Customer*, *Supermarket*, and *Shipper* roles, which we implemented as three BPEL orchestrations. Figure 2 presents a BPMN2 diagram that specifies the interaction of these roles during the *Purchase* operation, which is executed after the minimum prices for each item are found.

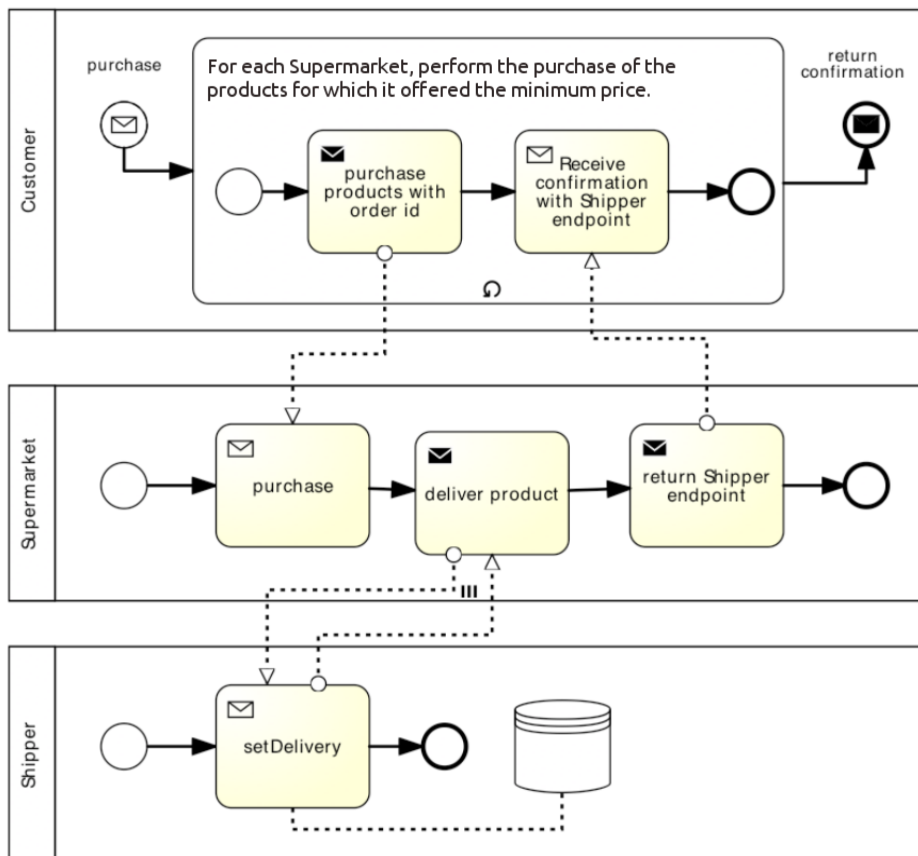


**Figure 2:** *Purchase workflow*

Figure 2 presents three lanes; each of them represents a process or, in other words, a choreography role. Besides, it is possible to represent the messages exchanged inside and across the roles. As can be noticed, the coordination is decentralized. While new requests are delivered to the *Customer* role, existing requests are processed by the other choreography roles in parallel.

## IV. Challenges in choreography testing

Automated testing of choreographies is challenging due to their characteristics. In Figure 2, for example, a reasonable testing framework for the presented choreography operation should cover, at least, the following aspects:

- **Atomic services correctness.** All individual services, including third-party ones, must work as expected. Thus, they must be tested in isolation.
- **Message exchange correctness.** Hundreds of messages may be exchanged in parallel when the choreography is enacted in a production environment. Thus, tests must simulate such scenario and, then, verify whether all messages are delivered in the expected order, within the expected response time, and with the expected content.
- **Message exchange debugging.** During the development, problems such as errors in the message exchange may be investigated through debugging mechanisms.

As explained in Section III, there are tools for assessing the **correctness of atomic services**. However, in all of them, the WSDL interface of the service under test must be provided to write the test cases, preventing the use of TDD, which requires writing the test before implementing the service. Besides, there is a lack of tools supporting the testing of third-party services, which may not be available at choreography development and offline testing time. To overcome these problems, Rehearsal provides a feature for generating web service clients dynamically. This way, tests can be written even if the WSDL interface does not yet exist. To deal with third-party services, Rehearsal provides a feature for mocking (emulating) web services.

Regarding the **message exchange aspects**, Enterprise Service Bus (ESB) systems can be used to intercept and collect messages. However, these systems are heavy (require considerable resources to run), maintain hundred of services, and normally are only available in acceptance test or production environments. These characteristics do not favor the ESB usage during choreography development. During development, the computer resources needed by an ESB may not be available for running the tests. Besides, in an ESB, messages may be collected only via log files, which makes their integration with test cases written by the programmer difficult.

In general, debugging is conducted passively in ESBs; in other words, the developer cannot pause the choreography execution to inspect the messages received by a web service. Full-layer debugging strategies are out of the scope of this paper. With Rehearsal, we provide features for debugging only those services that are under the developer's control. For these cases, messages are intercepted by service proxies that are introduced in the choreography. With the proxy, a developer can retrieve all messages received by a certain service. Besides, if the proxies are running in a developer IDE that supports debugging of Java objects, the developer can pause the choreography and inspect the message exchange at runtime. Nevertheless, the debugging of distributed systems consists of a topic already covered by our research group (Mega and Kon, 2004). As a future work, Rehearsal can be integrated with our previous work on the *Global Online Debugger*(Mega and Kon, 2004), an open source debugger that aims at being an extensible and portable tool for developers of distributed object applications.

All Rehearsal features and the methodology proposal are presented in detail in Sections V and VI.

## III. Related work

An initial effort for understanding the current scenario of testing techniques for web service compositions was conducted by Bucchiarone et al. (2007). Later, a more comprehensive survey to cover SOA testing was conducted by Canfora and Penta (2009). These studies propose alternative approaches, adapting the traditional software engineering testing techniques and strategies to the context of web service compositions.

According to these studies, in the web service composition context, the smallest unit of software is the web service. Thus, testing the individual web services corresponds to *unit testing*. Internally to a service composition, *integration testing* aims at assessing the interactions among the units forming the composition. Finally, *acceptance testing* is conducted when an orchestration or choreography is tested from the end-user perspective. In the next sections, we present the works related to our goals based on this classification.

### I. Atomic web service testing

Regarding the tools used for testing atomic (individual) web services, SoapUI (Eviware, 2010) provides mechanisms for functional testing. From a valid WSDL (Web Service Description Language) specification, SoapUI provides features to automatically build a set of XML-Soap request envelopes to test service operations. The tool also provides a feature for mocking web services. Rehearsal uses SoapUI internally to build Soap envelopes at runtime, thus we classify SoapUI as an internal dependency of our work.

SoapUI provides a mechanism to generate automatically a test skeleton for the operations presented in the WSDL. Although it automates the test creation, the produced test cases are incomplete and must be filled in by the programmer. WS-TAXI (Bartolini et al., 2009b) was proposed to improve this feature. Its goal is to automatically fill in these empty fields by deriving XML instances from an XML schema. With this tool, test cases are generated from all possibilities of data combinations for skeletons produced by SoapUI.

TTR (Test-The-REST) (Chakrabarti and Kumar, 2009) 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.

SOCT (Service Oriented Coverage Testing) (Bartolini et al., 2011) and BISTWS (Built-in Structural Testing of Web Services) (Eler et al., 2010) are approaches for applying structural (white-box) testing in web services. The goal of these works is to calculate the test coverage of testing suites. As a drawback, the services under test must be instrumented, which might be impossible for third-party services. To enable integration testing, as explained in the next section, Rehearsal supports functional (black-box) testing.

Besides these works, wsrbench consists of an approach to assess the robustness of web services (Laranjeiro et al., 2012). Robustness testing aims at analyzing the service behavior when tested under invalid and exceptional inputs. The goal of this work is to help with service selection by providing web services with robustness metrics to clients (end users) and developers.

(role)CAST (ROLE CompliAnce Testing) (Bertolino et al., 2011) focuses on applying compliance testing, aiming at testing services published in a registry. Its goal is to automatically apply predefined tests on new services. In comparison, Rehearsal's goal is development-time testing. However, the same compliance tests created using Rehearsal can be reused by tools similar to (role)Cast.

Arikan and colleagues introduced a Generic Testing Framework for the Internet of Services (Arikan et al., 2012) in which test cases are specified in XML and the framework takes care of

generating and executing corresponding JUnit tests. In addition to behavioral tests, this framework also supports stress, scalability, and parallel tests.

Most of these tools focused on testing the web services at the client side. However, Zhang (2011) proposes a framework to test web services at server side by using mobile agents. In this work, an agent starts the test execution in a machine and ends it in a different one. The goal of this work is to reduce the communication costs that are inherent to web service performance testing.

## II. Web service compositions testing

Pi4SOA (Pi4 Technologies Foundation, 2010) and CDLChecker (Wang et al., 2010) are tools designed to test choreographies specified in the WS-CDL format, but they only provide mechanisms for validating message exchange using simulation. We are interested in validating message exchange by invoking the real choreography, working in real systems.

ValiBPEL-Web (Endo et al., 2008) is a tool to apply structural testing on BPEL processes. This tool provides a web interface for instrumenting the process under testing, applying the test cases, and analyzing the results. To cover these features, the BPEL process is mapped into a Parallel Control Flow Graph (PCFG). BPEL activities such as *Receive*, *Reply*, *Invoke*, and *Pick* correspond to the graph nodes. Message exchanges, both internal and external ones, correspond to the graph edges. Then, after executing the tests, the tool measures test coverage, based on the defined test requirements, and presents the results.

Hwang et al. (2011) proposed a method to apply compliance testing for choreography services. The goal of this work is to verify whether the service contract is compatible with the contract specified in the choreography global model. Each service is represented by a Finite State Machine (FSM). In this model, the transition function represents the service requests and responses. For example, a service (FSM), in the $r$ state, moves to the $r'$ state, when receiving a request for a specific operation. At design-time, services are mapped to FSMs and, then, the method identifies which services are in compliance with the choreography roles.

With BPELUnit, Mayer and Lübke (2006) propose an architecture for testing web service processes. This architecture consists of four layers. In the first layer, test cases are specified. Then, in the second layer, these tests are organized in the framework. At this point, wrappers are created around the processes to execute the test cases. In the third layer, the tests are executed by simulation or by invoking the real services. Finally, the test results are presented in the forth layer. BPELUnit consists of a framework that implements these four layers for validating BPEL processes.

Greiler et al. (2010) propose a software application for detecting faults during a dynamic reconfiguration of a service composition. 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 functionality 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.

| Identifier | Requeriment description |
|---|---|
| R1 | Create clients to invoke operations of Soap web services |
| R2 | Create clients to interact with REST service resources |
| R3 | Allow test case writing for services that do not have contracts (interfaces) defined |
| R4 | Intercept (at the development environment) messages exchanged among the services of a composition |
| R5 | Emulate (mock) web services |
| R6 | Configure a timeout when invoking a web service operation |
| R7 | Provide objects that represent the choreography elements |
| R8 | Validate the contract (roles) of choreography services |
| R9 | Provide features that are not coupled to specific choreography and orchestration technologies, languages, or tools |

**Table 1:** *Comprehensive choreography testing framework requirements*

## III. Existing choreography development methodologies

With regard to efforts related to choreography development, we can highlight the Savara project (Madurai, 2009). This project provides a set of tools to develop a choreography by following the principles of a methodology called Testable Architecture. The goal of this methodology is to assure that any artifact produced during a specific development phase can be validated based on artifacts produced in the previous development phases. In the first three phases of this methodology, the choreography requirements are collected and modeled. As a result, the global and the local choreography models are produced. Then, based on these models, the services are implemented and the choreography is enacted and monitored to verify its correct behavior. The goal of our TDD methodology is to provide mechanisms to specify and develop the choreography following a test-driven approach.

In the next section, we present the requirements for a choreography testing framework and show how the works described in this section meet those requirements.

## IV. Requirements for Testing Tools

We started addressing the challenges presented in Section IV by developing a first Rehearsal prototype. This prototype consisted of: (a) *ad hoc* bash scripts to implement a choreography; (b) JUnit[4] test cases; and (c) a console that allows the user to execute the scripts and the test cases.

To evaluate the prototype, we developed an example choreography for planning and booking trips by using the OpenKnowledge (OK)[5], a framework for the peer-to-peer communication of distributed software components. In this choreography, the user indicates to the *Traveler* service where and when he/she intends to travel. Then, the user can reserve and buy the flight tickets. During this flow, the *Traveler* service interacts with other services such as *Airline*, *Acquire*, and *Travel agency*.

Based on the experience and results achieved from this prototype, we derive the requirements for building a comprehensive web service choreography testing framework. In the Table 1, we present these requirements.

In the Table 2, we present which of the presented requirements are covered by the related works presented in Section III.

As can be observed in the Table 2, none of the related works cover all of the defined requirements. In particular, we can notice that SoapUI, an internal Rehearsal dependency, is the tool

[4]JUnit: http://www.junit.org
[5]OpenKnowledge: http://www.openk.org

| | R1 | R2 | R3 | R4 | R5 | R6 | R7 | R8 | R9 |
|---|---|---|---|---|---|---|---|---|---|
| **Testing of atomic web services** | | | | | | | | | |
| SoapUI (Eviware, 2010) | • | • | | | • | • | | • | • |
| TTR (Chakrabarti and Kumar, 2009) | | • | • | | | | | | • |
| WS-TAXI (Bartolini et al., 2009b) | • | | | | | | | • | • |
| wsrbench (Laranjeiro et al., 2012) | • | | | | | | | • | • |
| SOCT (Bartolini et al., 2009a) | • | | | | | | | • | • |
| BISTWS (Eler et al., 2010) | • | | | | | | | • | • |
| (role)CAST (Bertolino et al., 2011) | • | | | | | | | • | • |
| Mobile-agent framework (Zhang, 2011) | • | | | | | | | • | • |
| **Testing of choreographies** | | | | | | | | | |
| PI4SOA (Pi4 Technologies Foundation, 2010) | | | | • | | | • | • | |
| CDLChecker (Wang et al., 2010) | | | | • | | | • | • | |
| Compliance testing approach (Hwang et al., 2011) | | | | • | | | | • | • |
| ValiBPEL-Web (Endo et al., 2008) | | | | • | | | | • | |
| BPELUnit (Mayer and Lübke, 2006) | • | | | • | | | | • | |
| Online testing framework (Greiler et al., 2010) | | | | • | | | | • | |

**Table 2:** *Requirements covered by the related works*

that covers more requirements than the other tools. Apart from the compliance testing approach (Hwang et al., 2011), all tools focused on choreography testing are coupled to a specific technology, tool or choreography language. Even thought all tools cover the requirement R4 (message intercepting), only the ValiBPEL-Web and BPELUnit intercept real messages of the choreography. In the other tools, the intercepting and validation occur in a message exchange simulation.

Based on this scenario, we developed the Rehearsal features and the TDD methodology to cover all the requirements of a comprehensive web service choreography testing framework. These features, as well as the Rehearsal architecture, are present in the next section of this paper.

## V. The Rehearsal framework

In the context of web service choreographies, when all services are deployed correctly, the choreography is ready to be enacted. In our work, we consider that, during all activities happening before this point at development time, specially testing activities, the choreography is being "rehearsed".

Rehearsal[6] is available as open source software under the GNU Lesser General Public License (LGPL) and was developed within the CHOReOS[7] and Baile[8] projects. The goals of these projects are to study, develop, and use web service choreographies in large-scale environments, in particular, those related to Cloud Computing and the Future Internet. In both projects, this framework belongs to research lines related to Verification and Validation (V&V) of choreographies.

### I. Rehearsal Features

In this section, we cover in detail the main features of Rehearsal, namely dynamic generation of web service clients, item exploration, message interception, service mocking, abstraction of

[6]Rehearsal page: http://ccsl.ime.usp.br/baile/VandV
[7]CHOReOS: http://www.choreos.eu
[8]Baile: http://ccsl.ime.usp.br/baile

choreographies, and scalability exploration.

### I.1 Dynamic generation of web service clients

In the SOA choreography system model, web services are considered the smallest system units. In the Rehearsal approach, these units can be validated by using the functional (black-box) testing technique. As explained previously, orchestrations and choreographies are accessible as atomic services from the end user perspective. Thus, tests focused on validating the compositions from this perspective are called *acceptance tests*. Since the system under test as a whole looks like an atomic service, tools used in unit testing can also be applied to acceptance testing of choreographies.

To cover these testing strategies, Rehearsal provides a feature in which the developer can invoke and validate the web service operation automatically. During the development of the first Rehearsal prototype, we identified some tools for this purpose, such as Apache Axis[9] and JAX-WS[10]. With these tools, the developer can create stub objects (also called clients) from the WSDL specification to communicate with web services. This process is not totally automated and human intervention is needed to create and use such objects. Besides, if the WSDL specification changes, existing clients must be generated again.

As an alternative to stubs manipulation, Rehearsal facilitates the dynamic generation of web service clients. With this feature, the developer can interact with web service clients without requiring *a priori* creation of stubs. This way, test cases can be written before the creation of WSDL interfaces. The service operations can be designed during the process of writing the tests. To make the tests pass, the developer starts coding the service implementation, generating as a result its contract (WSDL file). This feature uses SoapUI (see Section III) to build, send, and receive Soap envelopes. Figure 3 depicts the workflow for using this feature.

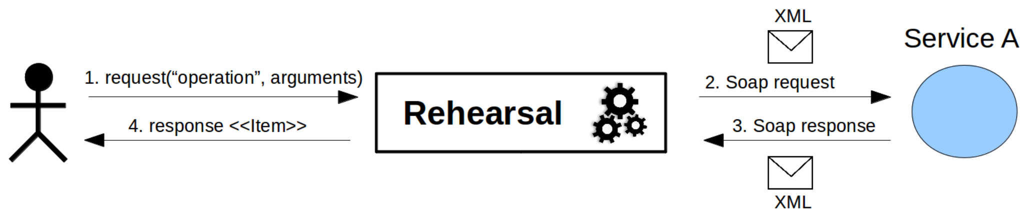


**Figure 3:** *Workflow to invoke web services*

In the first step of Figure 3, the developer, using the `WSClient` object, specifies which operation needs to be invoked. This object represents the dynamic client. To invoke the desired operation, the developer uses the `request` method by providing the operation name and parameters. This method supports primitive types (e.g., `int` and `String`). To avoid XML manipulation in the complex types, Rehearsal provides the `Item` object, which is a recursive data structure to represent XML documents. This object can be used to represent request and response of web services.

Figures 4 and 5 present a Soap envelope describing a complex type object and an `Item` object that represents it. In the workflow presented in Figure 3, when an operation is invoked, Rehearsal generates and sends a Soap envelope to the tested service (Step 2). Then, in Step 3, the framework collects the Soap response. Finally, in Step 4, the framework maps the response to an `Item` object that is returned to the developer.

[9]Apache Axis: `http://axis.apache.org/axis`
[10]JAX-WS: `http://jax-ws.java.net`

```
<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 4:** *Request Soap Envelope*

```
Item response =
      service.request("getProductByName",
                      "Soccer cleat");
Item item = response.getChild("return");
item.getName(); => "return"
item.getTagAttribute("xsi:type"); => "ax26:Item"
item.getContent("barcode"); => "153"
item.getContent("brand"); => "adidas"
item.getContent("description"); => "A cleat"
item.getContent("name"); => Soccer cleat
item.getContentAsDouble("price"); => 90.0
item.getContent("sport"); => "soccer"
```

**Figure 5:** *Equivalent `Item` object*

To evaluate test writing using the `WSClient`, a test case has been written using this Rehearsal feature and also using the JAX-WS tool. Figures 6 and 7 present both test case implementations. As can be seen, in the code snippet, the test case A is two lines shorter than the one written by using the `WSClient`. However, test case A uses the `Flight` object, a stub object that must be generated a priori by JAX-WS by using command-line or graphical tools. Since the tested service contains other operations, other stub objects must also be created and need to be integrated by the developer in the test cases. Thus, the test case written with the `WSClient` is a little longer but requires less work from the developer that does not need to deal with stub creation.

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

    Flight 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 6:** *Example of stubs (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.getContent("id"));
    assertEquals("Milan", flight.getContent("destination"));
    assertEquals("12-21-2010", flight.getContent("date"));
    assertEquals("09:15", flight.getContent("time"));
}
```

**Figure 7:** *WSClient (test case B)*

On the other hand, test case B is free of stub generation. The presented code snippet is all that the developer needs to test the `getFlight` operation. Besides, if the WSDL interface changes, only the operation name and parameters need to be changed in the test cases.

### I.2 Item Explorer

To help in the `Item` object creation, we developed the *Item Explorer* tool that automates the building of such objects. From a valid WSDL, this tool automatically generates Java code skeletons for retrieving web service requests and responses that can be integrated into the developer test cases. Figure 8 presents the Item Explorer graphical user interface depicting its features.

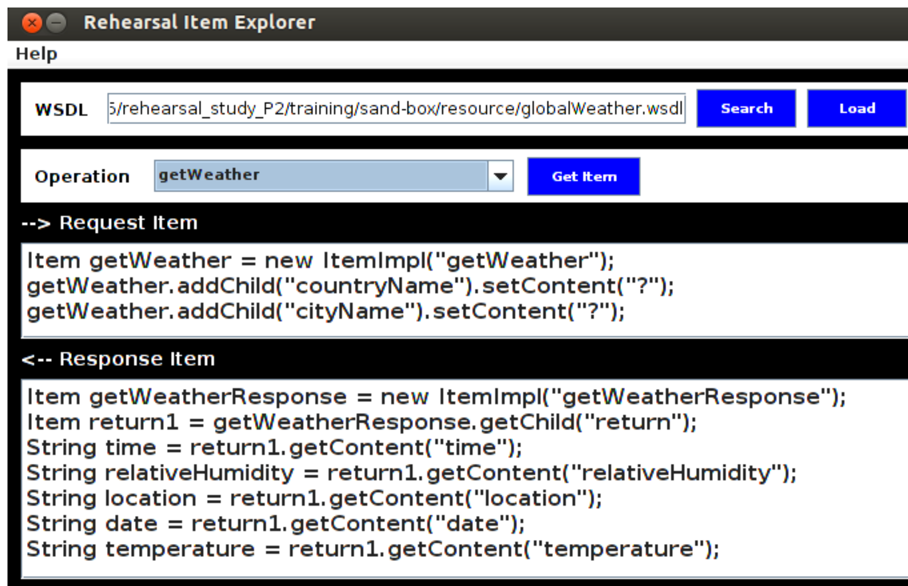


**Figure 8:** *Item Explorer*

### I.3 Message Interceptor

In the context of choreographies, integration tests can be applied by validating the messages exchanged among the services. During the choreography development, there are two levels of integration:

1. **Intra choreography roles.** As discussed previously, a choreography role can be implemented via orchestrations. At this level of integration, the messages exchanged internally inside one role are validated.

2. **Inter choreography roles.** In this level of integration, messages exchanged among the choreography roles are validated. These messages are those represented in the choreography global model.

Rehearsal supports both integration levels in an offline environment. The messages sent and received by this service are incrementally validated with the message interceptor at development time, after a service is added to the choreography. Figure 9 presents the basic workflow to use this feature.

In the first step of Figure 9, the developer provides the URL where the service WSDL is available, the port and the host name where the proxy will be published. In Step 2, the proxy is published automatically by Rehearsal. A proxy consists of an object that provides the same interface of the intercepted service Gamma et al. (1995). The proxy was built upon the SoapUI Mock API.

In Step 3, Rehearsal executes the test cases written by the developer. In this execution, the framework invokes the proxies (Step 4). At this point, the proxy stores and forwards the received message to the service that is being intercepted. Finally, in Step 7, the developer can extract and validate the intercepted messages. Messages are intercepted by a `MessageInterceptor` instance. Figure 10 presents an example of use. In lines 39-40, the interceptor is instantiated, configured to be available on port “4321", and to intercept the messages sent to the provided URL.

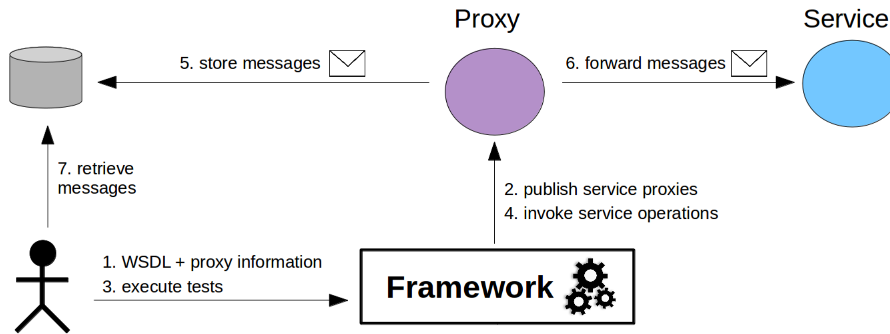


**Figure 9:** *Message Interceptor workflow*

```
38  public void shouldSendTheCorrectEndpoint() throws Exception{
39      MessageInterceptor interceptor = new MessageInterceptor("4321");
40      interceptor.interceptMessagesTo("http://supermarkets/registry?wsdl");
41
42      WSClient client = new WSClient("http://localhost:4321/registryProxy?wsdl");
43      client.request("registerSupermarket", "http://localhost:1234/store?wsdl");
44      Item  message = interceptor.getMessages().get(0);
45
46      assertEquals("http://localhost:1234/store?wsdl", message.getContent("endpoint"));
47  }
```

**Figure 10:** *Message Interceptor usage example*

In lines 42–43 of Figure 10, the proxy is invoked through the `registerSupermarket` operation. Then, the intercepted message is retrieved and validated (lines 44–47). This is a simple example since the intercepted message is the same message triggered in the tests. The goal of this example is only to show how this feature works.

### I.4 Service Mocking

Inter-organization integration of web services is an inherent characteristic of SOA. In spite of the advantages, this integration may also introduce problems for web service testing such as the absence of a testing environment to invoke the services. This way, web service non idempotent operations cannot be tested properly, preventing the application of complex and robust integration tests. To deal with such problems, Rehearsal provides a feature for mocking (emulating) service operations.

Figure 11 presents the basic workflow of this Rehearsal feature. In the first step, the developer provides the WSDL URL to the framework as well as the mock name. Then, Rehearsal, via SoapUI, creates and publishes a mock object (Step 2). It is also possible to define the mock host and port in this step, before or after the mock is running.

In the third step of Figure 11, the developer configures new messages (or updates existing ones) for the emulated service. These responses are applied in Step 4. To exercise different execution scenarios, the developer can define conditional messages. Figures 12 and 13 present a usage example of conditional messages.

As depicted in Figure 12, line 21, a mock object (called `smMock`) is created for the service available in the URL specified in the SM_WSDL_URI variable. The emulated service provides

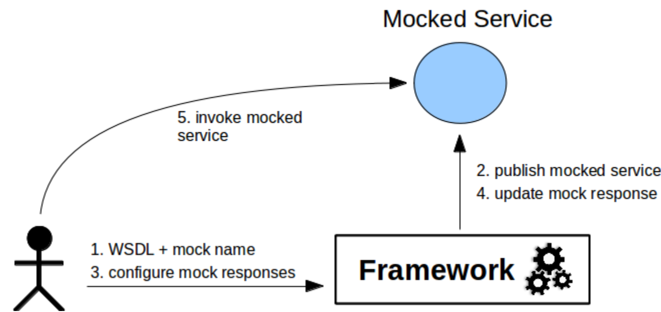


**Figure 11:** *Service mocking workflow*

```
19  @Test
20  public void shouldReturnTheCorrectPrice() throws Exception{
21    WSMock smMock = new WSMock("smMock", "4321", SM_WSDL_URI);
22
23    response1 = new MockResponse().whenReceive("coke").replyWith("3.50");
24    response2 = new MockResponse().whenReceive("beer").replyWith("4.00");
25    response3 = new MockResponse().whenReceive("*").replyWith("5.00");
26    smMock.returnFor("getPrice", response1, response2, response3);
27    smMock.start();
28
29    WSClient smClient = new WSClient(smMock.getWsdl());
30    Item response = smClient.request("getPrice", "beer");
31
32    assertEquals((Double)4.00, response.getContentAsDouble("price"));
33  }
```

**Figure 12:** *WSMock usage example*

the same contract (WSDL interface) of the real service. A conditional response is defined in line 23. If the request content matches the word “coke”, the mock response must show the price *3.50*. A similar conditional is defined in line 24. Finally, according to line 25, for all the other request conditions (not containing the previous words), the response must contain the price *5.00*.

In lines 26–27, the emulated service is configured to return the conditional messages when the `getPrice` operation is invoked. In lines 29–32, an assertion validates the mock response. Figure 13 presents the XML messages that are exchanged when line 29 is executed.

In this example, the conditional requests and response contain only one parameter that is a simple string. However, it is also possible to use complex objects in the conditionals. In this case, an `Item` object can be used to define such conditionals. Throught the `WSMock` object, the developer can configure multiple responses to support the validation of faulting scenarios. This object provides the `doNotRespond` and `crash` methods to simulate response absence and service crash, respectively.

### I.5 Abstraction of choreography

To facilitate test writing and the usage of the above functionalities, Rehearsal provides a feature to abstract choreographies into objects. Thereby, choreography elements, such as services and roles, can be dealt with by accessing Java objects.

```
Request
<soapenv:Body>
   <ser:getPrice>
      <name>beer</name>
   </ser:getPrice>
</soapenv:Body>
Response
<soapenv:Body>
   <ser:getPriceResponse>
      <price>4.00</price>
   </ser:getPriceResponse>
</soapenv:Body>
```

**Figure 13:** *Soap envelopes*

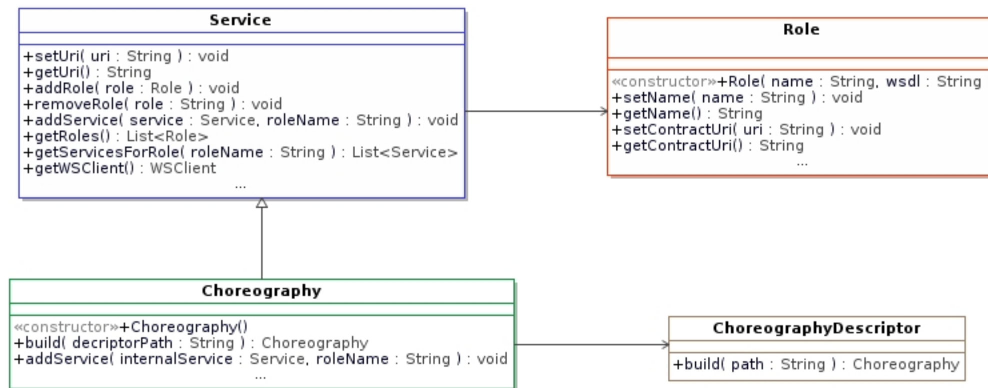


**Figure 14:** *Public interface of abstraction of choreography*

As can be seen in Figure 14, a `Choreography` object is created according to a descriptor file and its architecture follows the composite design pattern (Gamma et al., 1995). Via this composite, the developer can interact with all the services in the choreography. Each service plays one or more roles in the choreography. The developer can extract the URI of a service WSDL specification via the `Choreography` object. Due to the recursive nature of service composition, a `Service` object can be a composition of other services. As can be observed, a `Choreography` is, itself, a `Service`. Thus, a list of internal services can be extracted from a `Service` object.

A developer can use these objects to write tests before the choreography implementation. During development, tests would be written using `Service` objects instead of particular endpoints. Rehearsal extracts endpoint information in a descriptor file which is a YAML (YAML ain't a Markup Language)[11] document containing the services' endpoints and roles. Rehearsal extracts the endpoints to use in the `Service` objects at run-time. With this feature, tests written in a development environment will work with production services if the endpoints in the choreography descriptor are updated.

Figure 15 shows how a `Choreography` object is created from a descriptor file and Figure 16 brings examples of how to use the abstraction to create `WSClient`, `MessageInterceptor`, and `WSMock`. As can be seen, no real hard-coded endpoints must be used when writing tests, so the

[11] http://www.yaml.org

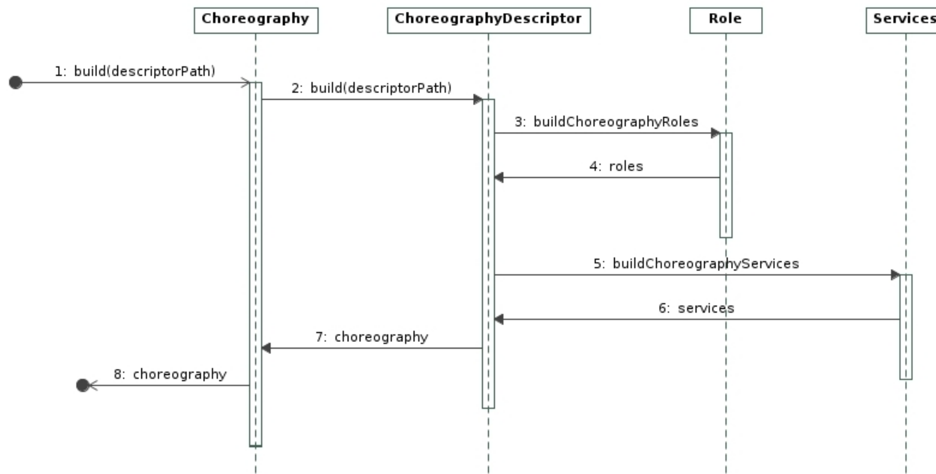


**Figure 15:** *Choreography object creation*

test code becomes more flexible.

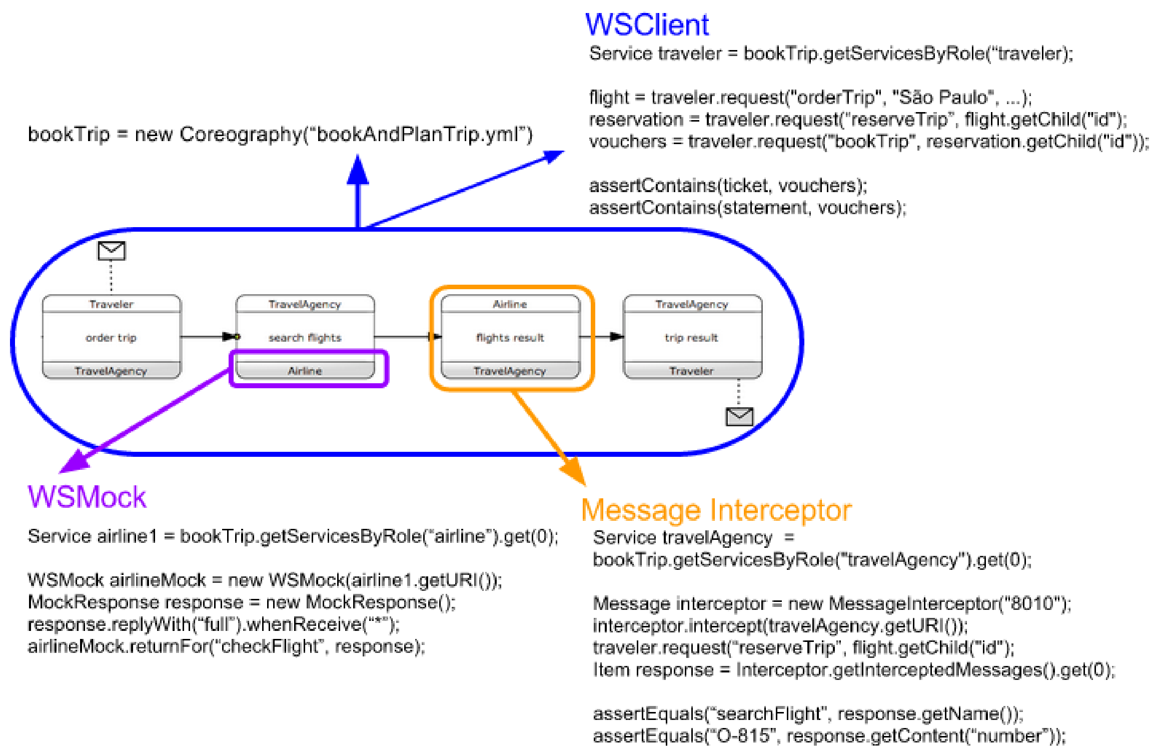


**Figure 16:** *Examples of Choreography Abstraction usage*

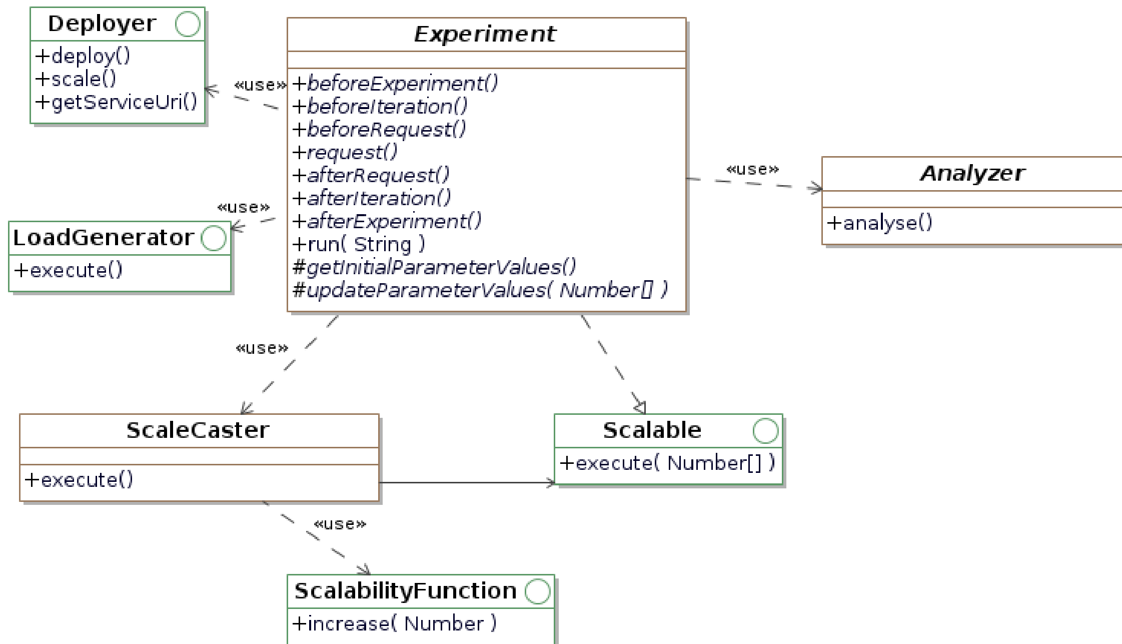


**Figure 17:** *Scalability Explorer Architecture*

### I.6 Scalability Explorer

Rehearsal provides the Scalability Explorer component to assist developers in verifying the choreography scalability, serving as a guide for improvements in its implementation and design. Its core is mainly composed of interfaces and abstract classes integrated as shown in Figure 17. Implementations for each component of this architecture are included in the framework, hence a scalability test can be built by plugging a set of them and writing a little code specific to the interaction with a particular system that the user wishes to analyze. Notwithstanding, the user can create new customized implementations for framework interfaces to adapt the experiments to his or her needs.

The only concrete class among those in Figure 17 is `ScaleCaster`, which iteractively invokes `Scalable.execute(Number[])` passing values updated by the`ScalabilityFunction`. The latter implements a simple function that receives one value and returns another, following a pattern. Currently the Scalability Explorer includes functions for linear and exponential increase.

`Experiment` is a Template Method (Gamma et al., 1995) for a scalability test, implementing `Scalable` and using `ScaleCaster`. Its execution flow is depicted in Figure 18. An experiment is a sequence of iterations where, for each iteration, scalability parameters related to workload and/or the system architecture vary. In each iteration, a `LoadGenerator` sends a number of requests to the target system. `Experiment` can use a `Deployer` to update the system architecture before each iteration. After the iterations, an `Analyzer` receives experiment data and produces an statistical analysis of the results.

`Experiment` includes a list of methods that can be overridden to define how to interact with the system under test. One of these methods, `execute()`, is invoked to trigger the requests that cause the system behavior that is under evaluation. The others can be used to include required actions before and after each request, iteration, or experiment.

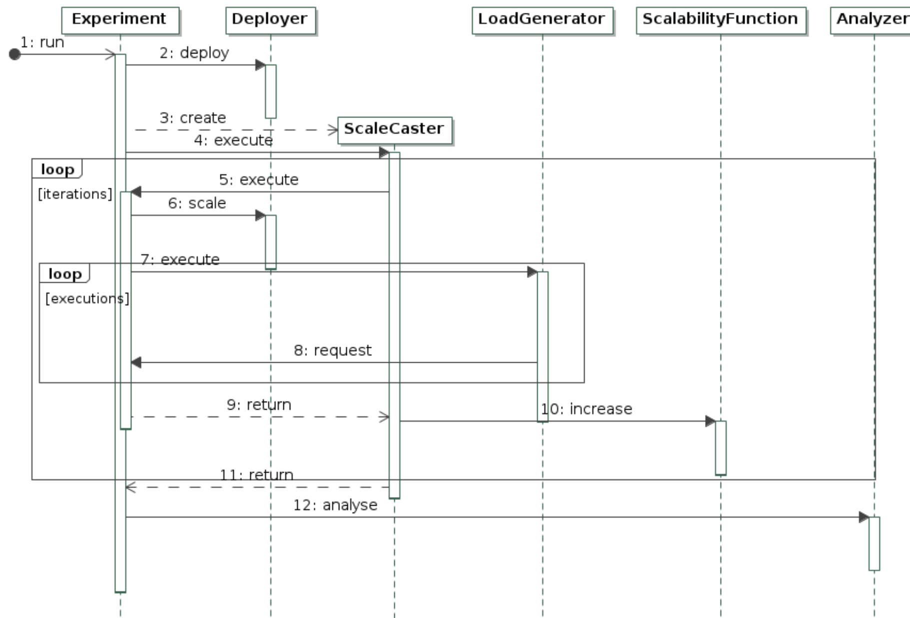


**Figure 18:** `Experiment` *execution flow*

Three extensions of `Experiment` are currently provided:

- `ArchitectureExperiment`: an experiment that varies one parameter used to specify some aspect of the system architecture, such as number of service instances or hardware resources available in a virtual machine. If a `Deployer` is used, `scale` receives this parameter.
- `WorkloadExperiment`: varies one parameter used to define the interval between requests made to the system. This parameter is used to configure the `LoadGenerator`.
- `ScalabilityExperiment`: this experiment uses both scaled parameters described above and they vary simultaneously.

The `Analyzer` is called at the end of the experiment to support examining the results of a scalability test. It handles the collected metrics and scalability parameter values, generating a meaningful output. Currently, the Scalability Explorer provides the following analyzers:

- `RelativePerformance`: displays a chart that shows how the performance varied in comparison with the first iteration. It can be used to create speedup or degradation charts;
- `AggregatePerformance`: aggregates the measurements made in each iteration in a single value, using operations such as arithmetic mean or percentile, and plots a chart with the aggregated performance obtained in each iteration.
- `ANOVA`: performs hypothesis test (Casella and Berger, 2002) to verify if the mean performance obtained in all iterations are equivalent;
- `SampleSizeEstimation`: estimates the minimum number of request that should be performed per iteration to make the `ANOVA` test meaningful.
- `ComposedAnalysis`: allows the usage of more than one `Analyzer` in an experiment.

`Deployer` is an interface for a component used to deploy services during the experiment. `Deployer`'s methods `deploy()` and `scale(int)` are used by `Experiment`. The former is called at the beginning of the experiment to set the system up and the latter is called before each iteration, passing the architectural scalability parameter as argument. Another method defined by this interface is `getServiceUri(String)`, which can be used to retrieve the URIs of a given service during the experiment.

The `EEDeployer` is an abstract implementation of `Deployer` that encapsulates the interaction with the Enactment Engine (Leite et al., 2014), a middleware system that provides a platform for automation of the distributed deployment of Web service compositions in cloud computing environments. `EEDeployer` can be extended by overriding the `enactmentSpec()` and `scaleSpec(int)` methods to return the specification to be sent to the Enactment Engine when `enact()` and `scale(int)` are called, respectively.

Thus, to write a scalability test, the programmer can choose one extension of `Experiment`, override required methods, and set auxiliary components for load generation (`LoadGenerator`), updating parameter values (`ScalabilityFunction`), and experiment analysis (`Analyzer`). If needed, an automated deployment can be created using `EEDeployer`. Also, if some behavior is not yet provided by the framework, it is possible to introduce it by implementing the available interfaces.

## VI. Test-Driven Development methodology for choreographies

Based on Rehearsal features (see Section V), the proposed methodology focuses on applying TDD principles and benefits (Section II) to assist in the design, coding, and testing of choreography services and roles. Depending on the development scenario, not all of the proposed activities belonging to the methodology phases must be fully executed. Choreographies can be implemented by partners of different organizations, thus, from a partner point of view, a developer can be involved in different development scenarios.

In some scenarios, TDD does not help with the design of the contracts (because they are defined previously). However, as outlined below, in such cases, the methodology still applies key TDD practices. In particular, the methodology provides features for maintaining an extensive test suites that gives the developer confidence to make business changes and detect immediately (or in a short time) potential problems introduced into the code (services implementation). In the next sections, we present the proposed development phases and scenarios.

### Development Phases

The proposed phases are depicted in Figure 19 and presented below. All internal activities belonging to each phase are performed in a testing/development environment, which can be the developer computer or a cloud infrastructure when the activities demand a large amount of resources.

#### Phase 1: Creation or adaptation of atomic web services

During the implementation of the choreography roles, new web services need to be created or existing ones must be adapted to implement the role requirements. Normally, a traditional approach is used for creating or adapting these services. In such approach, the service contract is defined and then the service is implemented. In this case, the service operations have already been defined in a contract, and based on it, the service is coded.

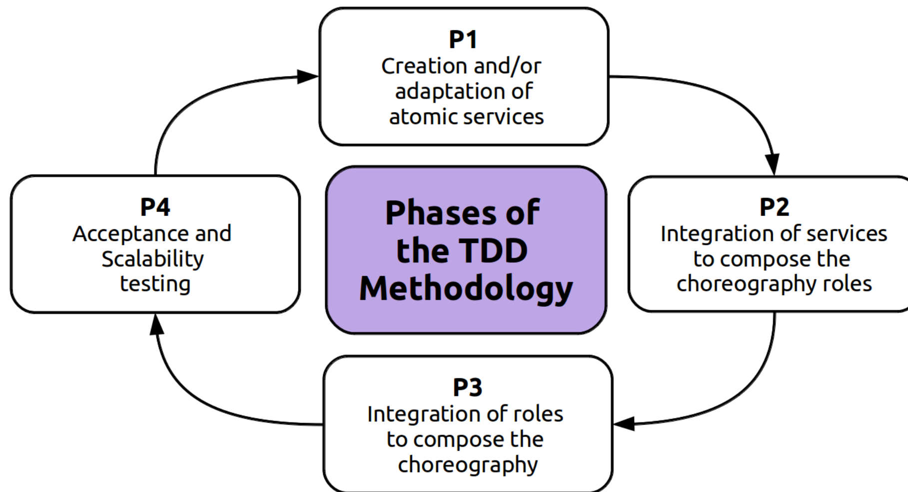


**Figure 19:** *Methodology Phases*

Rehearsal provides 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 as depicted in Figure 20.

```
@Test
public void testname() throws Exception {
   String wsdlUri = "http://localhost:1234/storeWS?wsdl";

   WSClient client = new WSClient(wsdlUri);
   Item response = client.request("getPrice", "milk");

   assertEquals((Double)2.50, response.getChild("price").getContentAsDouble());
}
```

**Figure 20:** *Example of dynamic request to a Soap service using the WSClient*

In Figure 20, the service under test does not exist yet. But, using Rehearsal, one can apply a test-driven approach for implementing it. Thus, in the test, the developer can specify the service endpoint (WSDL URI), the operation name (`getPrice`), and signature (receive a `String` and return a `Double` object). This test can be considered an executable specification of the service under development. This artifact may help other developers to understand web service operations in future maintenances. After writing the tests, the developer must code the service to make the tests pass, and refactor the solution to improve the software architecture.

#### Phase 2: Integration of services to compose the choreography roles

After the services are created (or adapted) and tested properly, they are integrated to compose a choreography role. As explained in Section II, at this point, a role consists of an executable process defined by a service or a set of services. When a set of services is needed for the composition, third-party services may not be available at development-time. To solve this integration problem, Rehearsal provides a service mocking feature where real services (e.g., third-party ones) can be simulated.

After all missing dependencies have been mocked, the services can be integrated to compose a choreography role. To assess the messages exchanged among the services within the executable process, Rehearsal provides a message interceptor feature. Using this feature, tests to validate this message exchange can be written before the developer performs the real integration.

**Phase 3: Integration of roles to compose the choreography**

After implementing an executable process by following the steps defined in Phase 2, the developer can assess the integration of the developed service with the rest of the choreography. Regarding the development scenarios S1 and S2, the services playing the other choreography roles may not be available at development-time. However, since the service contracts (WSDL interfaces) are defined in the choreography models, through these contracts, the unavailable service can be mocked.

After all dependency problems have been overcome, the integration among the services is performed and the external messages exchanged are intercepted and validated. However, this time, the messages that must be intercepted and validated are the ones specified in the choreography global model, and not the internal ones.

**Phase 4: Acceptance and scalability testing**

To complete the overall process, the choreography must be assessed taking into account properties that affect directly the end user. Thus, acceptance and scalability assessments are applied. The former focus on executing the choreography features to validate its functional behavior from the end user point of view. The latter aims at investigating the choreography performance at scale.

**Acceptance testing**

Differently from other testing strategies, acceptance tests verify the behavior of the entire system or complete functionality. From the point of view of an end user, the choreography is available as an atomic service. Thus, the acceptance test validates the choreography as a unit service, testing a complete functionality. In such context, the tools used for this type of test are similar to those used for unit tests. This way, WSClient (see Figure 20) can also be used for testing the choreography.

**Scalability testing**

According to Law (1998), an application is scalable if improvements in its architectural capacity implies the system capacity grows in direct proportion, maintaining the performance. However, all these dimensions are multifaceted: the system architecture comprises available memory, number of processor cores, processor frequency, storage space, network bandwidth etc.; the system capacity can be considered in term of input size, number of simultaneous request and the request frequency, for instance; and the system performance could be measured in terms of latency, throughput, resource usage, reliability or availability. Usually, it is not feasible to consider so many variables, therefore Law proposes to work with a single facet of each dimension. Therewith, a scalability test could go through the following steps:

1. Choose the variables that define the system capacity, the performance metrics, and the architecture capacity.
2. Define the functions of the complexity size of the problem, performance metric, and the architecture capacity.
3. Choose initial values for these variables.

4. Execute the application multiple times, increasing the variable values and collecting the performance metrics for each execution.

5. Analyze the performance metrics.

Given the choices made in Steps 1 to 3, the Rehearsal Scalability Explorer (Section I.6) can support the implementation of scalability tests, automating their execution and metrics collection (Step 4), and presenting charts for analyses, as the one shown in figure 21 (Step 5).

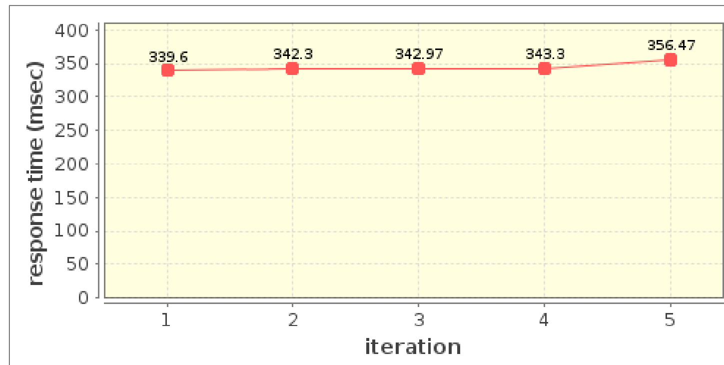


**Figure 21:** *Resulting chart of a scalability test*

After applying this assessment, the developer can have a good idea of how scalable the choreography is according to the aspects and metrics he/she chooses. If the result is not suitable, the developer can refactor the choreography architecture and analyze again with the same scalability tests to verify whether the desired scalability was achieved. Another possibility is to implement new tests, considering other variables.

## Development Scenarios

Depending on the development stage, business needs, and even industry cultural aspects, the previous phase can be conducted differently. In the next sections, we present the development scenarios addressed by the proposed methodology.

### Scenario 1: Design and implementation of choreography roles

In the first proposed scenario (S1), the developer is responsible for defining a new role (see Section I) or a set of roles for an existing choreography. In this case, the developer also acts as a choreography architect or designer. Table 3 presents internal activities that the developer may apply, for each methodology phase, when facing this scenario.

Following the methodology phases, the developer starts by defining a new choreography role, based on the software requirements, and the role dependencies (other choreography roles). Then, he or she develops a service, or a set of services, to implement the new role. At this point, the developer may apply a test-driven approach using the Rehearsal framework to guide the service interface design and implementation via unit tests. Since the tests invoke the service interface (WSDL file), after making all tests pass, the developer will have created the service WSDL. Later on, the developer may create integration tests to validate the role's internal message exchange. In these tests, external dependencies, such as other choreography roles, are emulated (mocked). In Phase 3, a similar integration test approach may be conducted. However, in this case, the messages

| | Scenario 1 (S1) |
|---|---|
| **Phase 1** | - test-driven approach<br>- unit testing<br>(service interface) |
| **Phase 2** | - service mocking<br>- integration testing<br>(internal message exchange) |
| **Phase 3** | - role mocking<br>- integration testing<br>(external message exchange) |
| **Phase 4** | - choreography testing<br>- scalability assessment |

**Table 3:** *Activities belonging to Scenario 1*

exchanged with other roles (dependencies) are the messages that must be monitored and validated. Finally, in Phase 4, if applicable, the functional and scalability behavior of the entire choreography (with the new role(s) integrated) are assessed.

**Scenario 2: Implementation of choreography roles**

In the second scenario (S2), the role interface had already been defined and the developer is only in charge of developing one, or a set of, web services to implement this role. Table 4 presents some activities that the developer may apply to implement the required services.

| | Scenario 2 (S2) |
|---|---|
| **Phase 1** | - traditional approach<br>- unit testing<br>(service interface) |
| **Phase 2** | - service mocking<br>- integration testing<br>(internal message exchange)<br>- compliance testing |
| **Phase 3** | - role mocking<br>- integration testing<br>(external message exchange) |
| **Phase 4** | - choreography testing<br>- scalability assessment |

**Table 4:** *Activities belonging to Scenario 2*

Similar to the previous scenario, the developer may use TDD to design and implement the choreography role. However, in this case, the developer focus on creating or adapting existing services to implement the choreography role requirements. Differently from the previous scenario, in Phase 2, the choreography role contract may be validated through compliance testing. In such tests, the developer can use the contract as an oracle and then validate his/her implementation. The idea is that the role implemented must have the same interface and behavior of the oracle. Rehearsal provides a feature for applying compliance tests that aim at verifying whether a service is playing the role properly based on the interface of this oracle contract.

Compliance tests assure that the implemented service plays the choreography role properly. Thus, these tests give confidence and encourage the developer: (1) to integrate the new role into the production choreography; (2) to refactor the code; and (3) to adapt the software in case of changes in the requirements. In the next phases, the activities belonging to this scenario are the same of scenario S1.

**Scenario 3: Design and implementation of a new choreography**

In the third scenario (S3), a partner (e.g., an organization) is beginning to develop a new choreography. Similar to scenario S1, the developer may also act as a choreography architect or designer. Each internal activity of this scenario, and its correlation with the presented phases, is presented in Table 5.

| | **Scenario 3 (S3)** |
|---|---|
| **Phase 1** | - test-driven approach<br>- unit testing<br>(service interface) |
| **Phase 2** | - service mocking<br>- integration testing<br>(internal message exchange) |
| **Phase 3** | - integration testing<br>(external message exchange) |
| **Phase 4** | - choreography testing<br>- scalability assessment |

**Table 5:** *Activities belonging to Scenario 3*

In this scenario, the roles do not exist at first. Thus, the developer can apply all the proposed activities, similarly to the scenario S1, to build the roles, and the respective service, iteratively. During this process, since the services are under the developer control, the real services can be used in the integration assessment. However, due to resource constraints, such as available memory and CPU, may not be feasible to run the entire choreography in the developer machine. In such cases, mocks can be used to emulate choreography roles that have already been validated.

## VII. Exploratory Study

To assess Rehearsal and the TDD methodology proposal, we have conducted an exploratory study in two phases with advanced Computer Science students at the Institute of Mathematics and Statistics at the University of São Paulo, where our research was mostly carried out. The first phase was conducted with eight Masters and PhD students, which had more than five years of experience with software development. All students had a very good knowledge of web service compositions and Test-Driven Development (TDD) that were acquired either on university courses or in professional jobs in large companies and startups. The first phase focused on assessing the efficacy and adequacy of Rehearsal and the proposed methodology in the context of web service choreography development. To achieve this goal, the following research questions were defined:

### Phase 1

**Research Question 1 (RQ1):** Does the Rehearsal features aid in the application of the proposed methodology steps?
**Research Question 2 (RQ2):** Does the proposed methodology provide adequate guidelines for developing a choreography?

After the first validation phase, Rehearsal and the methodology was refined based on the feedback received. Then, in the second phase, a similar exploratory study was conducted. In this second phase, the goal was to assess the Rehearsal and the TDD methodology effectiveness when they are used by non-expert developers. To achieve this, the subjects of this phase were eight undergraduate students, three masters, and one PhD student. They were also from the same Institute and most of them had less than five years of experience with software development,

mostly in university projects and in a few professional jobs such as internships. All students had basic or no knowledge of SOA. For the second phase, the following research questions were defined:

| **Phase 2** |
|---|
| **Research Question 1 (RQ1):** How easy is to use the Rehearsal features to apply the methodology steps? |
| **Research Question 2 (RQ2):** How easy is to follow the methodology steps to develop a choreography? |

## I. Organization

Both phases were designed following the guidelines of similar scientific studies (Seaman, 1999; Kitchenham, 1996). Since Rehearsal and the proposed methodology introduced novel concepts in our research theme, we decided to apply an *exploratory study* to identify benefits and problems in the use of both artifacts. The major part of our results is qualitative data, which, in our case, we consider more adequate to identify these benefits and problems from the developer perspective.

Before each phase of our study, we developed a systematic protocol that defined the study steps depicted in Figure 22. This protocol as well as all elements (questionnaires, produced code, interview audio) belonging to both phases are organized in a study package[12], enabling the replication of the study by other groups. In the training part, the participants received a practical course of TDD, web service compositions, Rehearsal, and related topics. In the development part, the participants worked in pairs (teams of two) to develop a choreography in four development tasks. Each task corresponded to each TDD methodology step. Finally, we collected the results through questionnaires. These results are presented in the next section.

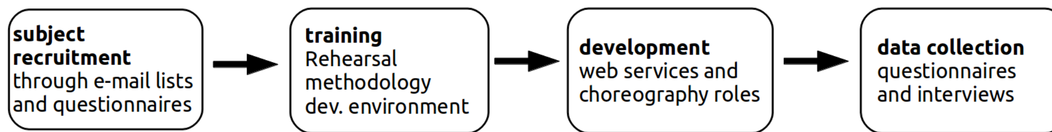


**Figure 22:** *Exploratory study steps*

We now present a detailed description of the results we obtained. The reader that is not interested in the details obtained in the result may jump to Section V where we summarize the study conclusions.

## II. Qualitative Results

Our qualitative results have been presented in box plots using a percentile function to extract the numbers of the positions representing 25% and 75% of the ordered data set. These charts also contain the Whisker points (Easton and McColl, 1997) that represent the minimum and the maximum values present in the sample (data set). This way, we can easily visualize the highest and lowest answers as well as how dispersed they are. As defined in the study protocols, we proposed to analyze groups of similarity in isolation. For each group, we present, in a table, the corresponding studied questions and charts. In these questions, **QRx** means "Questions to evaluate Rehearsal", while **QMx** means "Questions to evaluate the methodology".

[12]Study package: http://ccsl.ime.usp.br/baile/VandV/rehearsal-study/

### II.1 Dynamic generation of web service clients

| ID | Questions |
|---|---|
| **Phase 1** | |
| QR1 | It was easy to learn how to use the WSClient feature. |
| QM3 | The use of the WSClient feature is useful in Task 1. |
| QM13 | The use of the WSClient feature is useful in Task 4. |
| **Phase 2** | |
| QR1 | It was easy to learn how to use the WSClient feature. |
| QM3 | It was easy to use the WSClient feature in Task 1. |
| QM13 | It was easy to use the WSClient feature is useful in Task 4. |

**Table 6:** *Questions to study WSClient feature*

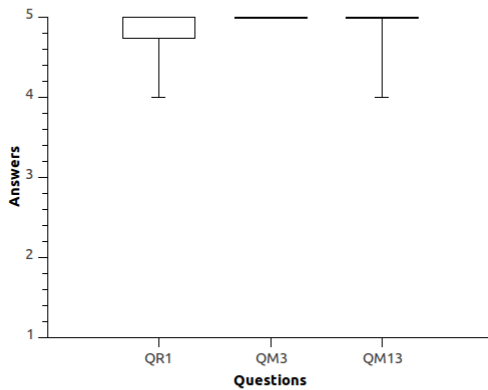


**Figure 23:** *WSClient - Phase 1*

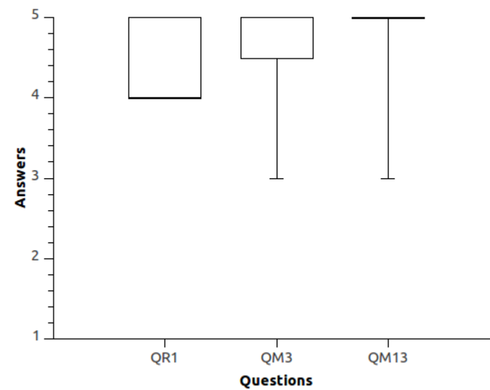


**Figure 24:** *WSClient - Phase 2*

In the first phase, as can be seen in Figure 23, all answers were positive (options 4 or 5). According to the box plot depicted in Figure 23, all subjects strongly agreed that the WSClient is a useful tool for applying the Phase 1 (QM3 question), and almost all of them, strongly agreed that it is also efficient in the acceptance testing, which is presented in the fourth phase of our methodology (QM13 question). As can be seen in Figure 24, for Phase 2, we had similar results. Although we had some answers for option 3 ("indifferent"), most part of the answers were the options 4 and 5.

### II.2 Web service mocking

| ID | Question |
|---|---|
| **Phase 1** | |
| QR2 | It was easy to learn how to use the WSMock feature. |
| QM6 | The use of the WSMock feature is useful in Task 2. |
| **Phase 2** | |
| QR2 | It was easy to learn how to use the WSMock feature. |
| QM6 | It was easy to use the WSMock feature in Task 2. |

**Table 7:** *Questions to study the web service mocking feature*

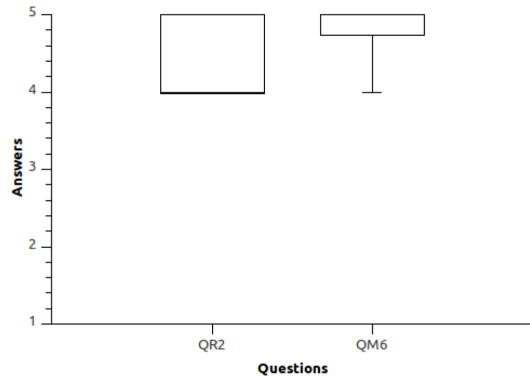


**Figure 25:** *WSMock - Phase 1*

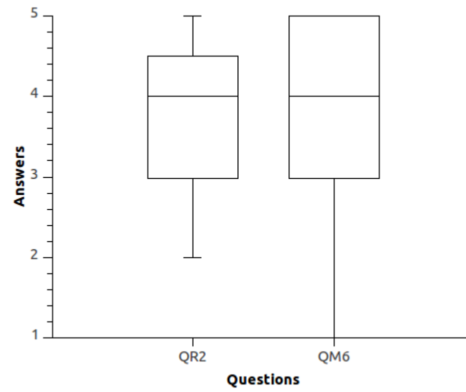


**Figure 26:** *WSMock - Phase 2*

In the first phase, even though almost all of the subjects strongly agreed that this feature is useful for applying the Task 2 (QM6 question), they apparently had problems to learn this feature. As can be seen in Figure 25, the median for the QR2 question was 4, which means that learning this feature was not so easy for the most part of the subjects. In the second phase, half of students had no problems in learning how to use and applying this feature in the Task 2. According to the students, in the interviews, learning the feature for its own is not a problem but to use it in real situations is not trivial.

### II.3 Message interceptor

| ID | Question |
|---|---|
| **Phase 1** | |
| QR3 | It was easy to learn how to use the Message Interceptor feature. |
| QM7 | The use of the Message Interceptor feature is useful in Task 2. |
| QM10 | The use of the Message Interceptor feature is useful in Task 3. |
| **Phase 2** | |
| QR3 | It was easy to learn how to use the Message Interceptor feature. |
| QM7 | It was easy to use the Message Interceptor feature in Task 2. |
| QM10 | It was easy to use the Message Interceptor feature in Task 3. |

**Table 8:** *Questions to study the message interceptor feature*

According to the box plot presented in Figure 27, all of the subjects strongly agreed that the message interceptor is useful for validating messages exchanged among the choreography roles (QM10 question). Almost all of them strongly agreed that this feature is also useful for validating the messages exchanged inside the role, which was applied in the Task 2 of our study (QM7 question). Regarding the QR3 question, we had similar results to those we had for the WSMock feature. In the second phase, we had also good results. Although we had some bad results (answer 2 for all questions), more than 75% of the students strongly agreed or agreed that it was easy to use the Message Interceptor.

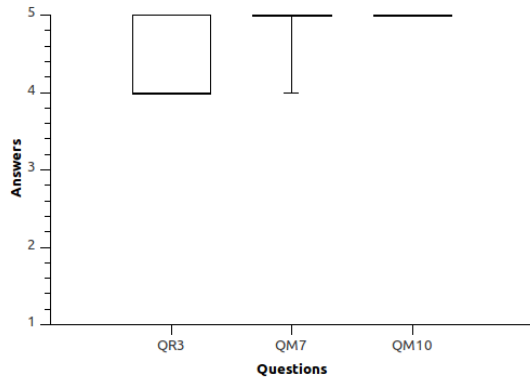


**Figure 27:** *Message Interceptor - Phase 1*

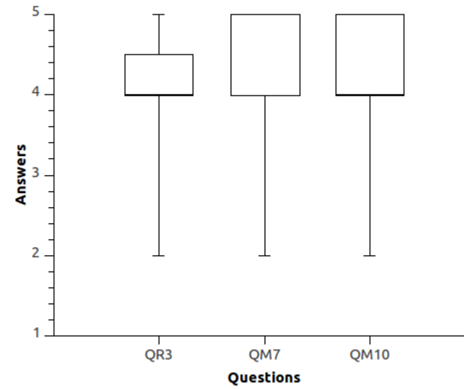


**Figure 28:** *Message Interceptor - Phase 2*

| ID | Question |
|---|---|
| **Phases 1 and 2** | |
| QR4 | It was easy to learn how to use the Abstraction Choreography feature. |
| QR5 | The Abstraction Choreography feature helped me to use other Rehearsal features. |
| QR6 | The Abstraction Choreography feature helped me to write the test cases. |

**Table 9:** *Questions to study the abstraction of choreography feature*

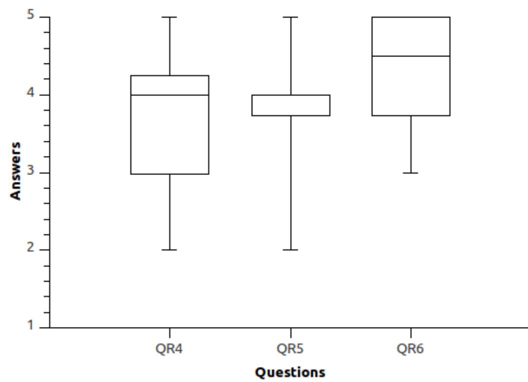


**Figure 29:** *Abstraction of Choreography - Phase 1*

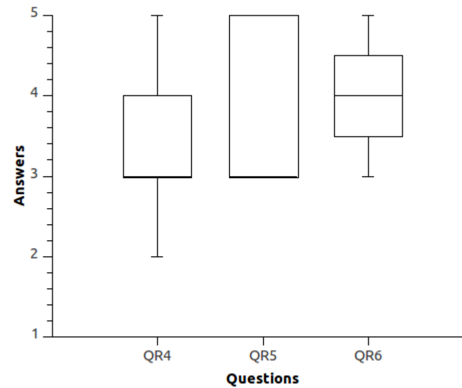


**Figure 30:** *Abstraction of Choreography - Phase 2*

### II.4 Abstraction of choreography

In both study phases, for this feature, we had negative results. In the first phase (Figure 29), some of the subjects had problems in learning how to use this feature (QR4 question). Although this feature has not always helped to use the other Rehearsal features (QR5 question), it has helped the subjects to write the tests in almost all cases (QR6 question). For this last question, the median was 4 in the second phase, and around 4 in the first phase; we had as the lowest value the option 3 (“indifferent”), but we had no negative results. Thus, although somewhat difficult to learn and use, we can conclude that the abstraction of choreography is useful for choreography development.

### II.5 Methodology and Rehearsal acceptance

We had good results for questions related to the adequacy and easiness in following the methodology steps. In the rest phase of the study, we investigated the methodology acceptance in terms of its suitability for SOA projects. The questions related to this topic are presented in Figure 10. The answers obtained for these questions are presented in Figure 31.

| ID | Question |
|---|---|
| **Phase 1** | |
| QM14 | I think the use of the methodology and Rehearsal would be useful for projects I have participated. |
| QM15 | I would use the methodology and Rehearsal in future projects I may participate. |

**Table 10:** *Questions to study the acceptance of the proposed methodology*

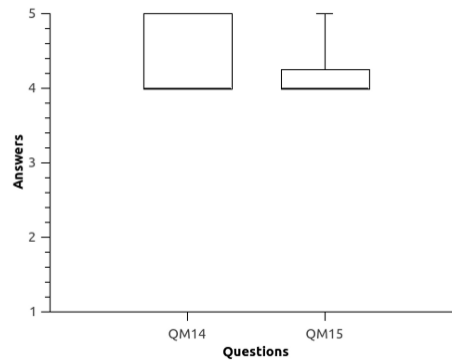


**Figure 31:** *Answers obtained for the questions of Table 10*

According to the data depicted in Figure 31, we had only positive results. Even though only one subject strongly agreed to the QM15 statement, the most part of subjects would use the methodology hereafter. In the case of QM14 question, we had similar results which means that Rehearsal and methodology steps can be adequate to solve problems that the developers have been faced.

## III. Quantitative Results

Besides the qualitative results presented above, we calculated the linear correlation between two variables: time spent by each team to complete all development tasks ($v2$) and the team experience in traditional software development ($v1$). In Table 11, we present the time and experience of the teams selected for this analysis.

Given the low number of samples, we applied a non-parametric test, which is recommended in these cases. In this kind of test, there are no presumptions about the population distribution. In this linear correlation, we applied the Kendall's methodology to the vectors $x = c(3, 3, 2, 1, 1, 1)$, classes of Table 11, and $y = c(140, 155, 221, 274, 275, 334)$, that represents the time spent by each team. Then, the result was **-0.700649** with p-value **0.10**. In this case, we can say that there exists a strong linear correlation between the studied variables ($v1$ and $v2$), and this correlation is inverse. The higher the experience, the less time taken to complete the tasks.

| Team | Experience (in years) | Class | Time (in minutes) |
|---|---|---|---|
| **Phase 1** | | | |
| team 1 | more than 5 | 3 | 140 |
| team 2 | more than 5 | 3 | 155 |
| **Phase 2** | | | |
| team 2 | 3-5 | 2 | 221 |
| team 3 | 1-3 | 1 | 274 |
| team 4 | 1-3 | 1 | 275 |
| team 5 | 1-3 | 1 | 334 |

**Table 11:** *Experience and time needed for the development*

This analysis is not complete and its goal consists of supporting the qualitative analysis conducted in the study. Given the novelty of this research topic, we first explored the benefits of using TDD for the development of web service choreographies. As future work, a more complete quantitative analysis must be conducted including, for example, experiments comparing the use of TDD versus traditional methodologies.

## IV. Threats to Validity

In the first phase, the proposed study may have limitations in its sample since the subjects of this study are Masters and PhD students in Computer Science and some of them have little experience in the industry. However, at the time of the study, the subjects worked with the development of web service choreographies and had good experience contributing to open source projects. Moreover, almost all of them have attended an "eXtreme Programming (XP) Laboratory" course. In this one-semester course, the students had the opportunity to try the concepts of eXtreme Programming (XP) such as pair programming, TDD, and automated testing in a real software project. In this phase, the most relevant result was the feedback about the usefulness of the tool and methodology, which was used to refine them, leading to the design of its subsequent version. Thus, since the feedback obtained was useful, we do not consider that the limitations of subjects sample impose a very relevant threat for the first phase.

In the second phase of the study, however, our sample included 12 subjects, all of them students in the same institute of the University of São Paulo, which hosts one of the best Computer Science programs in the country, thus the diversity of the set of subjects might be compromised. The results could be different if the subjects were students from other institutions or if they were professional programmers from the industry. In addition, the study participants knew personally the researchers involved in the project and this might have biased their answers. To mitigate this problem, we conducted the experiment in a way that only the first author of this paper, a graduate student, would have contact with the participants during the study; in addition, we asked them to be as honest as possible and do not hesitate in criticizing the work if needed. Finally, the study was not carried out as part of a course, so there were no grading involved that could also bias the results.

## V. Study conclusions

In both phases, the participants completed all development tasks correctly. In terms of the functional code, all teams developed the choreography correctly. In terms of the test cases, a few errors were detected. In the first phase, only 4 out of the 44 test cases (11 for each team)

were implemented incorrectly. In the second phase, only 3 of the 55 test cases were implemented incorrectly.

Regarding the questionnaire results, except for the choreography abstraction functionality, we had very positive results for all Rehearsal features. As a future work, the choreography abstraction will be improved to make the process of describing a choreography easier. In the case of the proposed TDD methodology for choreography development, we can conclude that we had good results in terms of its efficacy and adequacy.

## VIII. Industrial validation

To validate the use of Rehearsal in an industrial setting, the framework was used for Verification & Validation in two use-cases of the CHOReOS project, carried out by an industrial-academic consortium supported by the European Commission. The framework was used in two large-scale choreographies whose development were led by two industrial partners of the project:

- **Adaptive Costumer Relationship Booster (ACRB)** – a system for planning and executing marketing strategies, composed of a large number of services of different types including personal assistants, in-store totems, smart carts, back-end marketing management system, and external services provided by business partners (Mazza et al., 2012).
- **DynaRoute** – a system for managing the interaction among a fleet of taxis, their users, and other businesses to provide adaptive itineraries to citizens when traveling (Veranis et al., 2012).

A questionnaire was administrated to people who worked with Rehearsal to evaluate qualitatively its usage with relation to supporting the development of large scale choreographies for the Future Internet (Autili and Ruscio, 2011). In the following, we list the challenges identified as being addressed by Rehearsal, as well as how it contributes to overcoming them, as described in the CHOReOS final technical assessment report (Mazza et al., 2013):

- **Compositionality & Incrementality** – ultra-large systems could hardly be implemented in a traditional (waterfall) fashion that requires the overall picture of the system before implementation. Those systems demand incremental implementation and compositionality at runtime. The proposed TDD-based methodology enables the incremental development of web services and choreographies; Service Mocks and Message Interceptors provide support for assessing the compositionality of the choreography by testing service integration in a continuous development scenario.
- **The Future Internet scale** – in the Future Internet scenario, the focus is on the ultra-large size of distributed systems handling very to ultra-large workload. The Rehearsal Scalability Explorer enables automated experiments with very large workloads and analysis of measurements made during execution, supporting the assessment of system behavior in that condition before its deployment.
- **Adaptation** – large-scale systems need to evolve continuously, due to requirement and context changes. Despite taking no part at runtime adaptation, Rehearsal can provide assistance through functional tests, to verify whether a service can play a certain role in the choreography, and scalability tests, to identify how the system should be adapted when the workload changes.

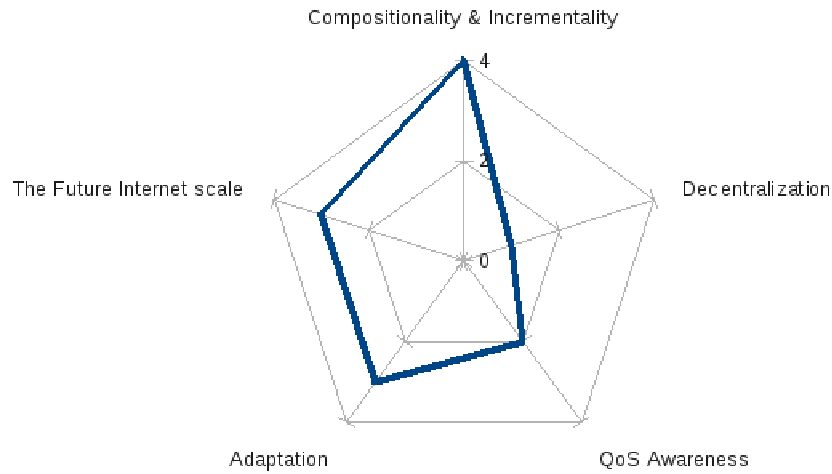


**Figure 32:** *Rehearsal's level of achievement of objectives*

- **QoS Awareness** – manual intervention for dealing with failures and performance issues in ultra-large systems is not a viable solution. Thus, Future Internet systems must incorporate QoS Awareness and support self-adaptation. Using the Rehearsal Scalability Explorer, one can identify the scalability profile of the choreography, pointing how to scale it to maintain QoS parameters within acceptable ranges as the workload changes.

- **Decentralization –** the scale and dynamics envisioned in the Future Internet hinders centralized coordination. Rehearsal support identification of bottlenecks caused by centralization through automated tests.

The level of achievement of these objectives, according to the opinion of the developers involved in the validation, is shown in Figure 32. As can be seen, Rehearsal is considered to provide good (3) to very good (4) support to issues concerning design and development of large-scale choreographies and, although used mostly at development time, it provides insights about how to deal with questions related to the self-adaptation of the compositions at runtime. The respondents considered that Rehearsal does not help much with matters related to Decentralization and QoS Awareness.

Additionally, the use-case teams performed their own evaluation of the usage of Rehearsal during their development process (Coletti et al., 2013; Parathyras et al., 2013). Both teams remark the usefulness of the framework, but identify a difficulty: the fact that the usage of a Message Interceptor requires the instrumentation of the choreography so that services invoke the proxy, instead of the actual service, could require some additional work to use the framework.

This industrial validation can be seen as an evidence that the results described in the exploratory study presented in Section VII could also be valid in an industrial setting. However, as the industrial developers involved in the validation were members of the consortium responsible for the development of Rehearsal, we cannot consider this as a scientific, unbiased result. Unfortunately, such a completely unbiased industrial study is not yet possible as the development of large-scale choreographies is not yet common in the industry so the costs for setting up an unbiased experiment would be very high. However, with the development of the technologies related to the Future Internet, we believe that this kind of systems will become increasingly more frequent within the next decade.

## IX. Conclusions

The activities involved in choreography development do not form yet a well-understood, solid process. Rehearsal and the development methodology proposed in this paper aim at providing mechanisms to enable a more robust and disciplined approach for developing choreographies. Based on the exploratory study results, we conclude that Rehearsal and the proposed methodology have a good potential to facilitate and bring more quality to choreography development. Besides, both artifacts are not coupled to specific choreography languages, which facilitates the adoption of Rehearsal and the methodology.

The set of features and architecture of the proposed tool and the steps of the proposed methodology were the result of a three-year careful research, which involved a comprehensive study of the literature and existing tools, interactions with tens of developers, and the implementation of several preliminary versions of the framework, which were incrementally assessed and refined. The research findings point out the effectiveness of the proposed tool and the methodology in helping developers to build complex decentralized systems based on collections of web services.

As future work, the Rehearsal features can be improved to be more flexible. In particular, message intercepting and service mocking could be adapted to include non-functional aspects. The scalability explorer component is currently being enhanced to cover more robustness scalability metrics and operation modes. Finally, the TDD methodology can be improved with its application to more large-scale practical case studies as well as its extension to cover other choreography life-cycle activities, such as requirement elicitation and choreography modeling.

Nevertheless, the feedback from users and the open source community, the personal experience of our research group members and of collaborators in other universities, research centers and companies, and the results we obtained with the scientific exploratory study are very encouraging. These results show that Rehearsal can be used in a variety of situations to support the effective testing of complex web service compositions.

## X. Acknowledgments

This research has received funding from HP Brasil under the Baile Project and from the European Community's

## References

Arikan, S., Kabzeva, A., GÃűtze, J., MÃijller, P., 2012. A generic testing framework for the internet of services. In: The Seventh International Conference on Internet and Web Applications and Services. ICIW. Stuttgart, Germany.

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

Autili, M., Ruscio, D. D., 2011. CHOReOS Perspective on the FI and initial conceptual model (D 1.2). Available at http://www.choreos.eu/bin/download/Share/Deliverables/CHOReOS%2DPerspectiveontheFIandinitialconceptualmodel%2DVA.0.pdf.

Barker, A., Besana, P., Robertson, D., Weissman, J. B., 2009. 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. ACM, pp. 1–10.

Bartolini, C., Bertolino, A., Elbaum, S., Marchetti, E., 2009a. 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. ACM, New York, NY, USA, pp. 161–170.

Bartolini, C., Bertolino, A., Elbaum, S., Marchetti, E., 2011. Bringing white-box testing to service oriented architectures through a service oriented approach. Journal of Systems and Software 84 (4), 655–668.

Bartolini, C., Bertolino, A., Marchetti, E., Polini, A., 2009b. WS-TAXI: A WSDL-based Testing Tool for Web Services. International Conference on Software Testing, Verification, and Validation, 326–335.

Beck, K., 2003. Test-driven development: by example. Addison-Wesley, Boston.

Beck, K., November 2004. Extreme Programming Explained : Embrace Change (2nd Edition). Addison-Wesley Professional.

Bertolino, A., Angelis, G. D., Polini, A., 2011. (role)CAST: A Framework for On-line Service Testing. In: 7th Internation Conference on Web Information Systems and Technologies. WEBIST. Noordwijkerhout, Netherlands.

Bhat, T., Nagappan, N., 2006. Evaluating the efficacy of test-driven development: Industrial case studies. In: ISESE'06 - Proceedings of the 5th ACM-IEEE International Symposium on Empirical Software Engineering. pp. 356–363.

Bucchiarone, A., Melgratti, H., Severoni, F., 2007. Testing Service Composition. In: 8th Argentine Symposium on Software Engineering (ASSE'07). Mar del Plata, Argentina.

Canfora, G., Penta, M. D., 2009. Service-Oriented Architectures Testing: A Survey. In: Software Engineering. Vol. 5413 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 78–105.

Casella, G., Berger, R., 2002. Statistical Inference, 2nd Edition. The Wadsworth & Brooks/Cole Statistics/Probability series. Wadsworth & Brooks/Cole Advanced Books & Software.

Chakrabarti, S. K., Kumar, P., nov. 2009. Test-the-REST: An Approach to Testing RESTful Web-Services. In: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. Computation World. pp. 302–308.

Coletti, L., Lavazza, L., Mazza, R., et al., 2013. "adaptive customer relationship booster - acrbÂť'Âť' use case assessment and demonstration (D 7.5). http://choreos.eu/bin/Download/Deliverables.

Easton, V. J., McColl, J. H., 1997. Statistics glossary v1.1. Available at http://www.stats.gla.ac.uk/steps/glossary.

Eler, M. M., Delamaro, M. E., Maldonado, J. C., Masiero, P. C., 2010. Built-In Structural Testing of Web Services. In: Proceedings of the 2010 Brazilian Symposium on Software Engineering. SBES '10. IEEE Computer Society, pp. 70–79.

Endo, A. T., Simão, A. d. S., Souza, S. d. R. S. d., Souza, P. S. L. d., 2008. Web Services Composition Testing: A Strategy Based on Structural Testing of Parallel Programs. In: Proceedings of the Testing: Academic and Industrial Conference - Practice and Research Techniques. TAIC-PART '08. IEEE Computer Society, pp. 1–12.

Erdogmus, H., Morisio, M., Torchiano, M., March 2005. On the effectiveness of the test-first approach to programming. IEEE Transactions on Software Engineering 31 (3), 226–237.

Erl, T., 2007. SOA Principles of Service Design. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Eviware, 2010. SoapUI, Web Services Functional Testing Tool. Available at http://www.soapui.org.

Fowler, M., 2011. Test-Driven Development. Available on: http://www.martinfowler.com/bliki/TestDrivenDevelopment.html.

Freeman, S., Pryce, N., 2009. Growing Object-Oriented Software, Guided by Tests, 1st Edition. Addison-Wesley Professional.

Gamma, E., Helm, R., Johnson, R. E., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

Greiler, M., Gross, H.-G., van Deursen, A., 2010. 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. ACM, pp. 36–42.

Guimaraes, F. P., Kuroda, E. H., Batista, D. M., 2012. Performance Evaluation of Choreographies and Orchestrations with a New Simulator for Service Compositions. In: The International Workshop on Computer-Aided Modeling Analysis and Design of Communication Links and Networks. CAMAD. IEEE.

Hwang, S.-Y., Hsieh, W.-F., Lee, C.-H., 2011. Verifying web services in a choreography environment. In: International Conference on Service-Oriented Computing and Application (SOCA). IEEE, pp. 1–4.

Issarny, V., Georgantas, N., Hachem, S., Zarras, A., Vassiliadis, P., Autili, M., Gerosa, M. A., Ben Hamida, A., 2011. Service-oriented Middleware for the Future Internet: State of the Art and Research Directions. Journal of Internet Services and Applications 2 (1), 23–45.

Jeffries, R., 2011. What is extreme programming ? Available on: http://xprogramming.com/xpmag/whatisXP#test.

Kitchenham, B., 1996. Desmet: A method for evaluating software engineering methods and tools. Technical Report TR96-09, University of Keele - Department of Computer Science.

Laranjeiro, N., Vieira, M., Madeira, H., 2012. A Robustness Testing Approach for Soap Web Services. Journal of Internet Services and Applications 4, 215–232.

Law, D. R., dec 1998. Scalable means more than more: a unifying definition of simulation scalability. In: Winter Simulation Conference Proceedings. Vol. 1. pp. 781 –788.

Leite, L., Moreira, C. E., Cordeiro, D., Gerosa, M. A., Kon, F., 2014. Deploying large-scale service compositions on the cloud with the choreos enactment engine. In: 13th IEEE International Symposium on Network Computing and Applications.

Madurai, B. K., 2009. Getting serious about enterprise architecture (whitepaper on alignment between testable architectures and TOGAF). Available on: http://docs.jboss.org/savara/whitepapers.

Mayer, P., Lübke, D., 2006. Towards a BPEL unit testing framework. In: Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications. ACM, New York, NY, USA, pp. 33–42.

Mazza, R., Lavazza, L., et al., 2013. CHOReOS final technical assessment report (D 10.3). http://choreos.eu/bin/Download/Deliverables.

Mazza, R., et al., 2012. Mobile-enabled coordination of people âĂŞ requirements, specification and use case definition (D 7.1). http://www.choreos.eu/bin/download/Share/Deliverables/CHOReOS_WP07D7.1Mobile%2Denabled_coordination_of_people_requirements_specification_and_use_case_definitionVB.pdf.

Mega, G., Kon, F., 2004. Debugging distributed object applications with the eclipse platform. In: Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology eXchange. ACM, pp. 42–46.

Parathyras, T., et al., 2013. Assessment of the "dynarouteÂť'Âť' pilot deployment and demonstration (D 8.4). http://choreos.eu/bin/Download/Deliverables.

Peltz, C., October 2003. Web Services Orchestration and Choreography. Journal Computer 36, 46–52.

Pi4 Technologies Foundation, 2010. Pi4soa - pi calculus for SOA. Available on: http://sourceforge.net/projects/pi4soa/.

Seaman, C. B., July 1999. Qualitative Methods in Empirical Studies of Software Engineering. IEEE Trans. Softw. Eng. 25, 557–572.

Tselentis, G., Galis, A adb Gavras, A., Krco, S., Lotz, V., Simperl, E., Stiller, B., Zahariadis, T. (Eds.), 2010. Towards the Future Internet - Emerging Trends from European Research. IOS Press, Amsterdam.

Veranis, G., Lockerbie, J., Parathyras, T., Papadakis-Pesaresi, A., 2012. "DynarouteÂť'Âť' scenario specification and requirements analysis (D 8.1). http://www.choreos.eu/bin/download/Share/Deliverables/CHOReOSWP08D8.1DynaroutescenariospecificationandrequirementsVA.pdf.

Wang, Z., Zhou, L., Zhao, Y., Ping, J., Xiao, H., Pu, G., Zhu, H., 2010. Web Services Choreography Validation. Service Oriented Computing Applications 4, 291–305.

Zhang, J., 2011. A Mobile Agent-Based Tool Supporting Web Services Testing. Wireless Personal Communications 56, 147–172.