Rehearsal: a framework for automated testing of web service choreographies

Felipe Besson, Paulo Moura, Fabio Kon, Dejan Milojicic

Rehearsal: a framework for automated testing of web service choreographies*

Felipe M. Besson[1], Paulo Moura[1], Fabio Kon[1], Dejan Milojicic[2]

[1]Institute of Mathematics and Statistics
Department of Computer Science - University of São Paulo

[2]Hewlett Packard Laboratories - Palo Alto, USA

{besson,pbmoura,fabio.kon}@ime.usp.br, dejan.milojicic@hp.com

Abstract. *Choreographies have been considered a scalable and decentralized solution for composing web services. Despite the advantages, choreography development, including the testing activities, is not disciplined. This work aims at applying Test-Driven Development (TDD) on choreographies to promote a more productive development. To achieve that, we present Rehearsal, a framework that supports TDD of web service choreography by automating unit, integration, and scalability testing.*

1. Introduction

Service-Oriented Architecture (SOA) consists of an architectural model that uses services as the building blocks of distributed applications. Web services can be combined recursively to create more complete services and then, to implement complex business workflows. Choreographies are a scalable and distributed approach for composing web services. Compared to orchestrations, which have centralized approach to distributed service management, the interaction among the choreographed services is collaborative with distributed coordination. Each participant plays a role specified in a global model that defines the messages exchanged in the collaboration [Peltz 2003].

Locally, each choreography participant is only concerned with the internal actions it must take to play the desired role. A few methodologies such as Savara [1] have been proposed for choreography development. However, up to now, none of them have experienced wide adoption. This results in choreographies implemented using ad hoc development process models. As a consequence, the choreography development, including testing, cannot be performed properly. Neither the functional behavior nor scalability of choreographies is assessed properly; hindering the scalability that is actually achieved.

The process of deploying services in a choreography is called enactment. During testing and development activities, it is used to say the choreography is being rehearsed. Thus, this paper describes Rehearsal: a framework to support Test-Driven Development (TDD) [Beck 2003] of choreographies. Our goal is to apply TDD to choreographies to facilitate their development and improve their adoption. To achieve this, Rehearsal provides features for automating multiple levels of testing: unit, integration and scalability.

*The research leading to these results has received funding from HP Brasil under the Baile Project and from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement number 257178 (project CHOReOS - Large Scale Choreographies for the Future Internet).

[1]Savara: `http://www.jboss.org/savara`

2. Related Works

SoapUI[2] 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. Besides, the tool provides a feature for mocking web services. Since Rehearsal uses SoapUI to build Soap envelopes at runtime, we classify SoapUI as an internal dependency of our work.

Pi4SOA[3] and CDLChecker [Wang et al. 2010] are tools for integration test, but they only provide mechanisms for validating the message exchange using simulation. We are interested in validating message exchange by invoking the real choreography. RoleCAST (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 pre-defined 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 roleCast.

Stress testing tools[4] such as LoadUI, and JMeter do not focus on scalability testing but a developer can use them combined with an execution strategy to perform scalability assessment of execution scenarios. Rehearsal may facilitate the definition of these execution scenarios.

3. Rehearsal features

Rehearsal aims at providing features to support automated testing in multiple levels during choreography development. Besides these features, Rehearsal supports notable TDD practices at the development-time such as web service mocking.

3.1. Dynamic generation of web service clients

Considering the service composition context, the smallest part (unit) of software consists of the web service. This way, in the beginning of choreography development, web services are tested in isolation. Each service operation can be invoked by using tools[5] such as Apache Axis and JAX-WS. With these tools, it is possible to create stub objects (also called clients) from a valid WSDL specification to interact with SOAP services. This process needs human intervention to create and use the stub objects. Besides, if the WSDL specification of the requested service changes clients need to be generated again.

To overcome these problems, Rehearsal provides the **WSClient**, a dynamic generator of web service clients. With this feature, the developer can interact with a service without creating stub objects. Given a WSDL specification, its operations can be requested dynamically. In Figures 1 and 2, we compared a unit test case using *WSClient* object with the same test case written with stubs generated by using JAX-WS.

As can be seen in these figures, test case A is four lines smaller than the test case written in B, that uses *WSClient*. However, the test case A uses the object *FlightResult* which is a stub object generated by the JAX-WS. Since the service under testing provides

[2]SoapUI: `http://www.soapui.org`

[3]Pi4 Technologies Foundation: `http://sourceforge.net/projects/pi4soa`

[4]Testing tools: `www.loadui.org` and `http://jmeter.apache.org`

[5]Stub tools: `http://axis.apache.org/axis` and `http://jax-ws.java.net`

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

    FlightResult flight = stub.getFlight(destination, date);

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

Figure 1. Stub version (test case A)

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

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

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

Figure 2. WSClient (test case B)

other operations, other stub objects have been created and must be known for testing those operations. Thus, in this case, the code needed for testing is longer than the code presented. The test case B is independent of stub generation. The code snippet presented is only what the developer needs for testing this operation. Besides, the test can be written before having a contract (e.g., WSDL specification) which favors the use of TDD to guide the contract development.

The web service response is specified in an **Item** object, a recursive data structure to represent XML. Thus, differently from SoapUI, the developer interacts with this object instead of manipulating XML. To help building and interpreting these *Item* objects, Rehearsal comes with a supporting tool called **Item Explorer**.

3.2. Message Interceptor

After testing atomic web services properly, these units must be connected to generate the choreography. At this point, integration tests can be applied by validating the messages exchanged between the services in the composition. **Message Interceptor** provides mechanisms to intercept, store and then forward these messages to their destination.

```
38  public void shouldSendTheCorrectEndpoint() throws Exception{
39      MessageInterceptor interceptor = new MessageInterceptor("4321");
40      interceptor.interceptTo("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 3. Message Interceptor Example

As depicted in Figure 3, in lines 39–40, Message Interceptor is instantiated to intercept the messages sent to a real service. During this process, a proxy providing the same WSDL interface of the real service will be automatically published on the port *4321*. Then, through the proxy, the operation *registerSupermarket* belonging to the real service is invoked (lines 42–43). At this point, this message is intercepted, stored, and then, forwarded to the real service. Finally, in lines 44 to 46, the first intercepted message is retrieved and its content validated.

3.3. Service Mocking

During the integration phase, some services may not be available in testing (offline) mode. The integration of inter-organizational services is one of the benefits of SOA. However, it may bring difficulties for testing such as the absence of a testing environment, some service operations, for instance the non-idempotent ones, cannot be tested completely. To deal with such constraints, Rehesarsal provides **WSMock**, a feature for mocking services. With this feature, real services can be easily simulated by mock objects.

Figure 4 shows a concrete example of how to use WSMock. In line 21, a mock is created for a real service that is deployed on the URI specified in the *SM_WSDL_URI*. The mock is deployed on *http://localhost:4321/smMock* and provides the same contract of the real service. Line 23 states that if the request message content contains the word **coke**, the message response will be **3.50**. A similar return condition is defined on line 24. Finally, line 25 defines that when the request message does not contain the words **coke** or **beer**, the returned price will be **5.00**.

```
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 4. WSMock example

```
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 5. Soap envelopes

In lines 26–27, the mock is configured to return the defined messages when *getPrice* is invoked. In lines 29–32, the test invokes the mock and validates the response. Figure 5 presents the XML Soap Envelopes that are exchanged when the line 29 is executed. In this example, the mock responses are primitive types (String). However, complex types, which are specified by *Item* objects, can also be used in the *WhenReceive* and *replyWith* methods. Moreover, this feature can assist in fault scenarios simulation. Using the methods WSMock, it is possible to configure the mock to not respond or simulate a crash behavior.

3.4. Scalability Explorer

After the services integration, the choreography scalability should be assessed. According to [Quinn 1994], an application is scalable if it achieves the same performance when increasing the architecture capacity with the same proportion that the problem size grows. A common scalability test for a Web Services composition is to increase the workload (problem size) and replicate some service (architecture capacity) to check the response time (performace). Scalability Explorer aims to support automated scalability tests by providing features for executing the choreography in different scenarios, collecting performance metrics, manipulating resources and reporting execution results.

Scalability tests must be methods receiveing some sort of parameters and returning a number. They are annotated with *@ScalabilityTest* which can receive three arguments.

```
13 @ScalabilityTest(scalabilityFunction=Linear.class,  steps = 5)
14 public List<Long> test(@Scale int requests, @Scale int nodes){
15    List<Long> responseTimes = new ArrayList<Long>();
16    instantiateNode(nodes);
17    responseTimes = makeRequestsPerSecond ( requests ) ;
18
19    return responseTimes;
20 }
```

Figure 6. Scalability test

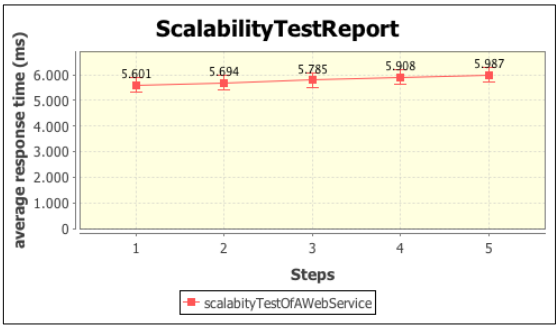

Figure 7. Graph of a scalability test

The first two, as shown in Figure 6, define how many times to run the test (*steps*) and a growing pattern (*scalabilityFunction*). *@ScalabilityTest* can also receive a *maxMeasurement* argument to specify a maximum allowed return value, stopping the tests when it is surpassed. Furthermore, the *@Scale* annotation is used to indicate the parameters to be increased according to the *scalabilityFunction*.

The scalability explorer also inlcudes facilities to support load generation, statistics and chart generation, as shown in Figure 7. As opposed to regular tests, a scalability test will not suceed or fail. It should be used to collect information to analysis by an expert.

4. Rehearsal architecture

Rehearsal is a test harness that must be combined with other frameworks such as JUnit or TestNG to execute the test cases. In Figure 8, we present the Rehearsal architecture.

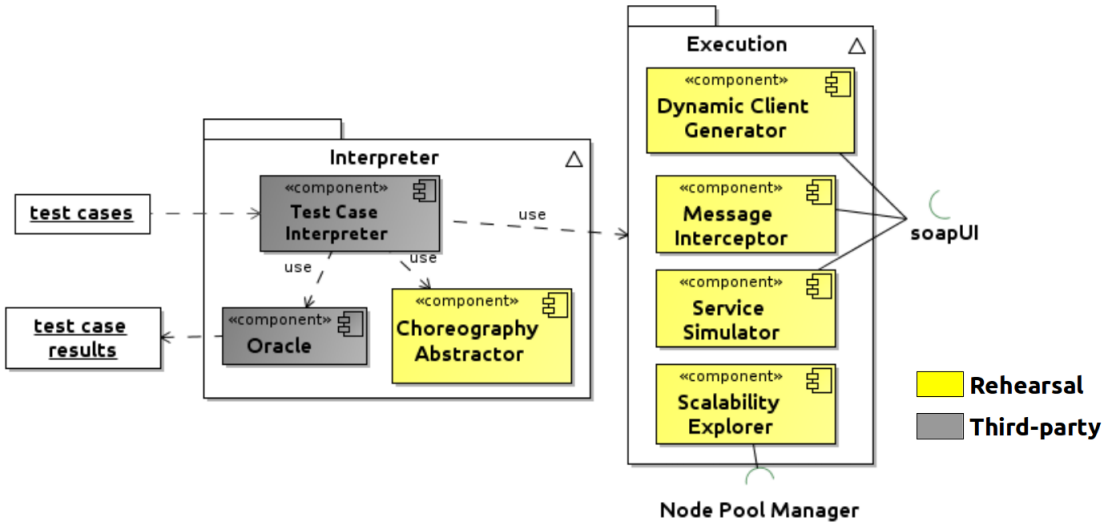

Figure 8. Rehearsal architecture

4.1. Interpreter

The Interpreter package contains the components for "processing" test cases. This process is guided by the *Test case interpreter* component which corresponds to the third-party components for executing test cases. In the case of JUnit, this component corresponds to

the JUnit Test executor. During the execution, service endpoints and Rehearsal commands can be found in the test specification. These elements are processed by the *Choreography Abstractor*. For each command defined in the test cases, the interpreter delegates their execution to the responsible execution component.

For instance, when an *interceptTo* command is found by the interpreter, this component invokes the *Message Interceptor* component, providing all needed information (extracted from the *Choreography Abstractor* component) to intercept and store the desired message. Each assert statement in the test case is processed by the *Oracle* component. In TDD, the expected result values are specified by the developer during the test case written. Thus, we can consider the developer as the oracle. Since in our framework the test cases are specified using third-party frameworks (e.g., JUnit), the process of matching the expected values with the actual ones and presenting the results is performed by these third-party frameworks.

4.2. Execution

The Execution package contains the components for supporting the execution of Rehearsal features. The *Dynamic client generator* is responsible for providing the dynamic generation of web service clients. The *Message Interceptor* components are in charge of intercepting, collecting, and storing the messages. Through the *Service Simulator*, services can be mocked. These three components use SoapUI to provide these features. *Scalability Explorer* supports scalability testing. Since, execution scenarios of a scalability assessment consisted of different choreography configurations and workloads, cloud infrastructure can be used to support these activities. In the future, Rehearsal will be integrated with Node Pool Manager [6], a component to create, destroy, and deploy new service instances automatically in the cloud.

5. Final remarks

Rehearsal is available under the LGPL license on: `http://ccsl.ime.usp.br/baile/VandV`. Apart from the Scalability Explorer, all Rehearsal features have been completely developed. An exploratory study to assess Rehearsal has been conducted. The goal of this study was to qualitatively investigate the benefits and difficulties when Rehearsal is used to support TDD in choroegraphies. Based on the results, we are refining Rehearsal and a methodology proposal to apply TDD in choreographies.

References

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

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

Peltz, C. (2003). Web Services Orchestration and Choreography. *Computer*, 36:46–52.

Quinn, M. J. (1994). *Parallel computing (2nd ed.): theory and practice*. McGraw-Hill.

Wang, Z., Zhou, L., Zhao, Y., Ping, J., Xiao, H., Pu, G., and Zhu, H. (2010). Web services choreography validation. *Serv. Oriented Comput. Appl.*, 4.

[6]Node pool manager: `https://github.com/choreos/choreos_middleware`