BPEL-Unit: JUnit for BPEL Processes

Zhong Jie Li and Wei Sun

IBM China Research Lab, Beijing 100094, China
{lizhongj, weisun}@cn.ibm.com

Abstract. Thanks to unit test frameworks such as JUnit, unit testing has become a common practice in object-oriented software development. However, its application in business process programming is far from prevalent. Business process unit testing treats an individual process as the unit under test, and tests its internal logic thoroughly by isolating it from the partner processes. This types of testing cannot be done by current web service testing technologies that are black-box based. This paper proposes an approach to unit testing of Business Process Execution Language for Web services (BPEL4WS, or WS-BPEL as the new name), and introduces a tool prototype named BPEL-Unit, which extends JUnit. The key idea of this approach is to transform process interaction via web service invocations to class collaboration via method calls, and then apply object-oriented test frameworks. BPEL-Unit provides the following advantages: allow developers simulate partner processes easily, simplify test case writing, speed test case execution, and enable automatic regression testing. With BPEL-Unit, BPEL process unit testing can be performed in a standardized, unified and efficient way.

1 Introduction

Over the last decade, businesses and governments have been giving increasing attention to the description, automation, and management of business processes using IT technologies. This interest grows out of the need to streamline business operations, consolidate organizations, and save costs, reflecting the fact that the process is the basic unit of business value within an organization.

The Business Process Execution Language for Web Services [1] (BPEL4WS, or WS-BPEL as the new name, abbr. BPEL) is an example of business process programming language for web service composition. Other languages include: BPMN, WfXML, XPDL, XLANG, WSFL [2], etc. They all describe a business process by composing web services. Generally speaking, a business process is defined in terms of its interactions with partner processes. A partner process may provide services to the process, require services from the process, or participate in a two-way interaction with the process.

Mission-critical business solutions need comprehensive testing to ensure that it performs correctly and reliably in production. However, in current industrial practice, business process testing focuses on system and user acceptance testing, whereas unit testing [3] has not gained much attention. This is strange, given the fact that unit testing has been prevalent in object-oriented software development

[4]. We expect that business process, e.g. BPEL process, unit testing will draw more attention along with the maturation and adoption of SOA and BPEL specification. BPEL unit testing treats an individual BPEL process as the unit under test, and tests its internal logic thoroughly.

Current web service testing methods and tools like [5][6] (open source) and [7][8] (commercial) are not applicable to business process unit testing, as they are black-box based, and only deal with simple request-response interaction patterns between a web service and its client. [9] presents a BPEL unit test framework that uses a proprietary approach, specially, a self-made test specification format in xml and the associated test execution.

We show in this paper how to use, adapt and extend current object-oriented unit test frameworks (specially, JUnit [10] and MockObjects [11]) to support BPEL process unit testing. The key idea is to transform process interaction via web service invocations to class collaboration via method calls, and then apply object-oriented test framework and method. The proposed method has been implemented in a tool prototype named BPEL-Unit, an extension of JUnit.

This paper is organized as follows. Section 2 introduces some preliminary knowledge, including JUnit, MockObjects and a BPEL process example. Section 3 describes BPEL unit test method in an abstract view. Section 4 presents the BPEL-Unit tool implementation in detail. Section 5 concludes the paper with future work prediction.

2 Preliminaries

2.1 JUnit

JUnit is an open source Java testing framework used to write and run repeatable tests. Major JUnit features include: assertions for testing expected results, test fixtures for sharing common test data, test suites for easily organizing and running tests, graphical and textual test runners. For a quick tour, please go to http://junit.sourceforge.net/doc/faq/faq.htm.

2.2 MockObjects

MockObjects is a generic unit testing framework that supports the test-first development process. It is used to simulate the collaborator class or interface dynamically in order to test a class in isolation from its real collaborators. A mock implementation of an interface or class mimics the external behavior of a true implementation. It also observes how other objects interact with its methods and compares actual behavior with preset expectations. Any discrepancy will be reported by the mock implementation. EasyMock [12] is a specific MockObjects implementation that is adopted in BPEL-Unit.

2.3 Example BPEL Process

Through this paper, we'll use the purchase process in the BPEL specification as the running example. It is shown graphically in Figure 1.

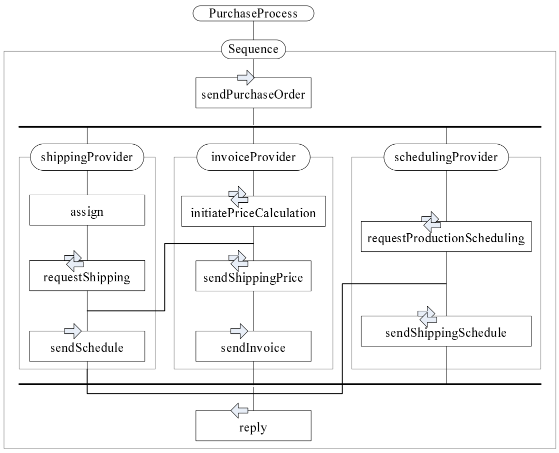

Fig. 1. Purchase process example

The purchase process runs as follows. On receiving a purchase order from a customer (*sendPurchaseOrder*), the process communicates with three partner processes - ShippingProvider, InvoiceProvider and SchedulingProvider - to carry out the work. It initiates three tasks concurrently: requesting for shipment (*requestShipping*), calculating the price for the order (*initiatePriceCalculation*), and scheduling the production and shipment for the order (*requestProduction-Scheduling*). While some of the processing can proceed concurrently, there are control and data dependencies between the three tasks. In particular, the shipping price is required to finalize the price calculation (as is indicated by the link between *requestShipping* and *sendShippingPrice*), and the shipping date is required for the complete fulfillment schedule (as is indicated by the link between *sendSchedule* and *sendShippingSchedule*). When the three tasks are completed, invoice processing can proceed and the invoice is sent to the customer (*reply*).

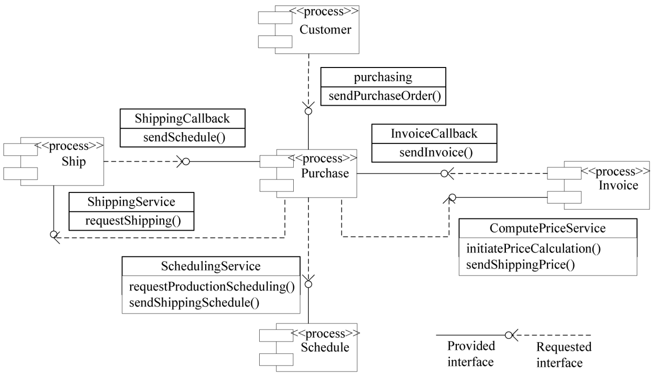

Fig. 2. Service contract between the purchase process and its partners

Interfaces that are provided by the purchase process and its partner processes are defined in WSDL documents as a set of portType definitions. They are visualized in Figure 2. The purchase process provides three interfaces: Purchasing, ShippingCallback and InvoiceCallback, each with one operation. Each partner process - Ship, Invoice and Schedule - provides one interface: ShippingService, ComputePriceService and SchedulingService, respectively.

3 BPEL Unit Test Method

A BPEL process could interact with several partner processes (partners, for short), which in turn interact with other partners, resulting in a network of processes. Figure 3a only shows a process and its direct neighbors. The Process Under Test is abbreviated as PUT, and its Partner Process is abbreviated as PP. A partner may be a simple stateless web service, or a complex process that choreographs several web services / processes and exposes one or several web service interfaces to the caller process. Treating a partner as a generic process makes the method applicable for general cases.

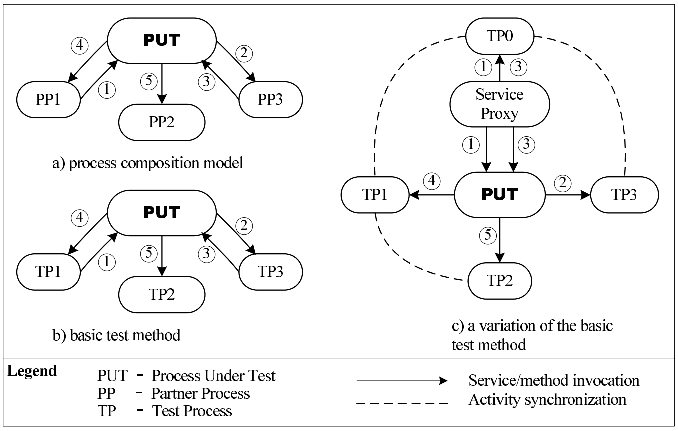

Fig. 3. Process composition model and test methods

A conversational relationship between two processes is defined in a partner link. Each partner link defines up to two *role* names and lists the web service interfaces that each role provides to the other in this conversation. Thus we use arrow lines to connect two processes, indicating the service invocation from the consumer to the provider. Note that the arrow lines do not specify the type of the web service operations, which may either be 1-way or 2-way. For the present, let's ignore the circled numbers beside the arrow lines.

Figure 3b shows a basic unit test method for the process composition model in Figure 3. A Test Process (TPi, i=1,2,3) is used to simulate the behavior of each

Partner Process (PPi). A variation of this method is used in [3] and implemented in a tool prototype named B2B, where test processes are specified in BPEL and executed in a BPEL engine.

Comparatively, this paper uses a different variation of the basic test method as shown in Figure 3c, wherein test processes are simulated by mock objects in Java code and executed in any Java runtime. In this method, each test process (TPi) only describes one direction of interactions - service invocation from the PUT to its partners. This fact can be seen from the direction of the arrow lines between the PUT and TPi. The other direction of invocation - service invocation from the partner processes to the PUT - is delegated to a Service Proxy. Thus the invocations of PUT services that are originally made in partner processes are now made in the Service Proxy to execute in testing. This decision is made based on the fact that a mock object can only specify how its methods are invoked, but not how it calls other methods. There is a special test process named TP0, which describes the expected service invocations into the PUT and its expected responses. The requests and responses are used by the service proxy to actually invoke PUT services and verify the actual responses. In addition, the dashed lines between test processes indicate activity synchronization between test processes. This will be explained in more detail later.

4 BPEL-Unit Implementation

The implementation of BPEL-Unit is centered around the following idea: transform process interaction using web service invocations to class collaboration using method calls, then apply object-oriented testing techniques.

Web service definitions are described in WSDL files and optionally XSD files. The structure of a typical WSDL file is shown in Figure 4 which contains the following elements: type, message, portType, binding, and service. The portType groups a set of operations that can be invoked by a service requester, type and message definitions provide the data model of the operation. Binding gives protocol and data format specification for a particular portType, and service declares the addressing information associated with defined bindings.

Figure 4 also shows the implementation details: how to map web service elements to Java equivalents, how to simulate partner interfaces with mock objects, how to use partner stubs to connect the PUT to the simulated partners, and how to write the test logic inside mock objects based on the PUT behavior. Each is described in a separate section below.

4.1 Web Service to Java Interface Mapping

For the purpose of writing Java tests, firstly the WSDL elements should be mapped to Java language equivalents. Specially, each web service interface definition of the involved processes is mapped to a Java interface definition. As Figure 4 shows, this consists of two parts: a web service interface (denoted as A) maps to a Java interface (denoted as C); and web service types and messages

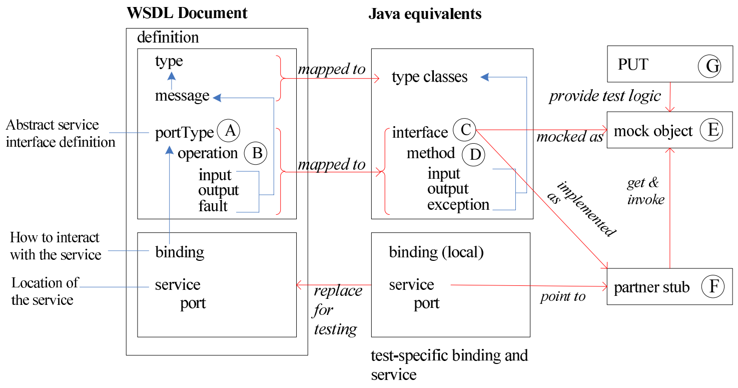

Fig. 4. Method details

map to Java data type classes. Java data type classes will be used to define test data. A Java interface will be used to generate a mock object (or simply called *mock*, denoted as E) of the interface, and a partner stub of the interface (denoted as F, introduced later). Each WSDL operation (denoted as B) will have a correspondent Java method (denoted as D).

The following code snippet illustrates the mapping between the ShippingService portType of the purchase process example and a Java interface.

```
<portType name="ShippingService">
    <operation name="requestShipping">
        <input message="wsdl:ShippingRequest"/>
        <output message="wsdl:ShippingInfo"/>
        <fault message="wsdl:ShippingFault" name="fault"/>
    </operation>
</portType>
-->
public interface ShippingService {
    public ShippingInfo requestShipping(ShippingRequest
    shippingRequest) throws java.lang.Exception;
}
```

4.2 Mock Objects

With a mapped Java interface (denoted as C in Figure 4), a mock implementation (a mock control and a mock object, denoted as E in Figure 4) can be created to simulate that interface, as exemplified in Section 2.2.

Each portType (defining a web service interface) of the PUT and its partner processes has a mock implementation. Therefore, a process may correspond to several mock objects, one for each portType. For the purchase process example, there will be six mock objects: three for the purchase order process (Purchasing, InvoiceCallback and ShippingCallback), one for each partner process - ShippingService, ComputePriceService and SchedulingService.

A mock object for a process sets the expected invocations on the process as well as the return values, and verifies at runtime that the expected invocations occur with the correct parameters. For example, say that *mockShip* is the mock object of ShippingService, we use *mockShip.requestShipping(shipRequest)* to set the expectation of the invocation of requestShipping with a parameter shipRequest, and use *setReturnValue("ShippingService", shipInfo)* to set the return value *shipInfo* of the invocation. Therefore, a mock object simulates the services provided by a non-existent partner process. The mock objects for partner processes will be called by the PUT at run time by relay of partner stubs.

However, mock objects for the PUT are handled differently. We use mock objects for the PUT not to simulate its behavior, but to tell the service proxy to make an invocation to the PUT and then check if the return value of the invocation is correct. Let's see the following example. *mockPurchasing* is the mock object of the PUT Purchasing interface. The first statement tells the service proxy that it should invoke the *sendPurchaseOrder* operation with the specified parameters, and the second statement tells the service proxy to check that the PUT should return invoice as the response.

```
mockPurchasing.sendPurchaseOrder(po, customerInfo);
setReturnValue("Purchasing", invoice);
```

The service proxy does so on behalf of the relevant partner process which makes the invocation originally, because the MockObjects framework does not allow specifying outgoing call behavior in a mock object. The service proxy also invokes the mock object when it invokes the PUT. This is a unified way of expressing test logic, allowing all the interactions to be verified by the MockObjects built-in verification function, no matter it is from the PUT to a partner process or reverse. Nevertheless, this may bring in some confusion on the semantics of mock objects. A simple cure for this problem is to treat the PUT as nonexistent process too in writing test cases.

4.3 Partner Stubs

In a process execution, how to interact with a service, and the address of that service are described in the WSDL binding and service elements. The original WSDL binding and service definitions of a partner process may be varied: SOAP, JMS, EJB, etc. For unit testing, all the partner processes will be simulated as local services implemented in Java. So we should define test-specific WSDL Java binding and service endpoints.

In testing, each service endpoint of a partner process should be a Java class. As we know, mock objects are dynamically created Java artifacts and cannot serve this purpose. So we decide to define a separate stub Java class for each web service interface and name it "partner stub". A partner stub class (denoted as F in Figure 4) implements an interface (C). The implementation of each method (D) in a stub class is simple: it dynamically gets the mock object (E) that simulates the service and calls the mock object's correspondent method (D), collects the return value (if not void) and returns that value. In this way, the partner stub is essentially a simple wrapper of the real service provider implementation in mock (E), and doesn't contain any test logic. The exact behavior of the mock objects can be defined dynamically in test cases. The partner stubs are stateless and independent of test behaviors, and can be automatically generated. For the purchase process example, the ShippingServiceStub is shown below.

```
public class ShippingServiceStub implements ShippingService{
    public ShippingInfo requestShipping(ShippingRequest
    shippingRequest) {
     ShippingService service = MockUtil.getMockObject("ShippingService");
     ShippingInfo result = service.requestShipping(shippingRequest);
     return result;  }
}
```

The address of the partner stubs will be referenced in the correspondent WSDL service endpoint definition so that the invocation of a web service operation (B) will go to the correct method (D) of a correct partner stub (F). In this way, a partner stub and its associated mock object collectively implement a simulated partner process. For the purchase process example and the ShippingServiceStub, the service endpoint information is shown below.

```
<service name="ShippingServiceJavaService">
 <port binding="ShippingServiceJavaBinding" name="ShippingServiceJavaPort">
    <java:address className="ShippingServiceStub"/>
 </port>
</service>
```

The java:address specifies that ShippingServiceStub is the service endpoint. Note that this binding and service information should replace the original one in deploying the process under test to test it. Therefore, these artifacts should be taken as part of the test resource and thus managed as such in the project.

This is different from current use of stub processes in that stub processes contain the real test logic, and are connected to the process under test directly, so that we have to write and maintain a lot of stub processes, redeploy and restart the processes for each test scenario. Through a separation of responsibilities onto a partner stub and a mock object, only one partner stub is needed, and also dynamic changing of test logic without redeploying and restarting is supported.

4.4 Test Logic Specification

Test logic specifies the behavior of the process under test and the simulated partner processes. As aforementioned, test logic will be written in the mock objects that simulate the partner processes.

The first question is where to get the behavior of each partner process. The answer is the process under test. It may have many execution scenarios that are resulted from different decision-making in the control flow. Each of these execution scenarios consists of a set of activities, which are either internal or external. The external activities are those related to service invocation, including invoke, receive, reply and so on. These external activities form a service invocation sequence, which can be used as a test scenario. From a test scenario, interactions with different partners can be separated and used to specify the partner behaviors in mock objects.

Then in a test case, in each mock object of a partner, we record a sequence of calls that the correspondent partner process is expected to receive from the PUT, and prescribe the return values. If the PUT makes a wrong invocation at runtime (including method call with wrong parameters, wrong call numbers or sequencing), the verification mechanism of MockObjects will report the error.

Concurrency and synchronization. In a BPEL process, the service composition follows certain sequencing, which is expressed using programming control structures. BPEL defines the following control structures among others: *sequence*, *flow*, *while* and *switch*. A *flow* construct creates a set of concurrent activities directly nested within it. It further enables expression of synchronization dependencies between activities that are nested within it using *link*.

Therefore, in test processes that simulate real processes, we need similar control structures to express the original activity ordering constraints. Note that complex test logic is not encouraged in unit testing, whereas fast-written, simple, correct, and easy to read/maintain test logic is favored. Applying this principle in BPEL unit testing, a piece of test logic should simply describes an execution path of the PUT; complex behaviors like branching should be avoided as far as possible. However, concurrency and synchronization is a common kind of ordering constraints put on an execution path, so it's unavoidable and must be supported in test behavior description. In Figure 3c, activity synchronization is denoted by dashed lines. It only shows synchronization dependencies between test processes. Actually, the synchronization can also occur inside a test process. Figure 5 shows both cases. Figure 5a specifies: op1, op3, op6 are concurrent activities; op1 must be invoked before op4; op5 must be invoked after op2 and op4. Figure 5b specifies: op1 must be invoked before op5; otherwise is a violation.

With such synchronization capabilities provided, the service interaction ordering indicated in Figure 3a is supported in the test logic as Figure 3c shows. For example, the logic "firstly a PUT service is invoked, then a TP3 service is invoked" could be supported.

Test logic support in MockObjects implementations. Usually a MockObjects implementation provides some flexible behavior description and verification mechanism. For example, EasyMock has three types of MockControl. The *normal* one will not check the order of expected method calls. Another *strict* one will check the order of expected method calls. For these two types, an unexpected method call on the mock object will lead to an AssertionFailedError. The remaining *nice* one is a more loose version of the *normal* one, it will not check the

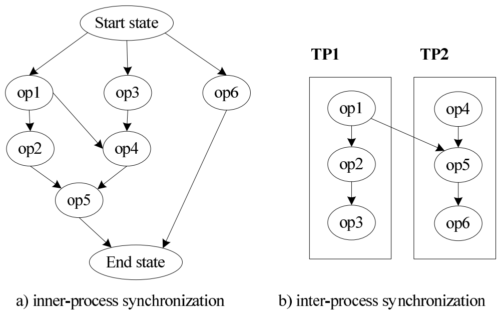
Start state
op1
op2
op3
op4
op5
op6
End state
a) inner-process synchronization
TP1
TP2
op1
op2
op3
op4
op5
op6
b) inter-process synchronization

Fig. 5. Activity synchronization

order of expected method calls, and an unexpected method call will return an empty value (0, null, false).

These preset and special MockControl types could be used to express two basic types of control logic / method call ordering: sequence and random (unordered). Take the purchase process as an example. If we want to ensure that several service invocations from the PUT to another process occur in the right sequential order as specified, the strict MockControl could be used to create the mock implementation of that process. Besides sequence and random, there is generic control logic such as alternative (*switch*), *timer* operations (start, cancel, timeout), and concurrency that probably haven't been supported by existing MockObjects implementations. Ideally, the testing of business processes requires the MockObjects implementation to support generic control logic. Practically, the MockObjects implementation should support the concurrency and synchronization logic described previously.

For this purpose, extension to current MockObjects implementation may be necessary. For example, a possible extension is to allow testers specify a successive relation on several methods, say, by an API *syncMethods(m1, m2, ...)* that specifies the occurrence order of the methods m1, m2, etc. This extension has been implemented on EasyMock in BPEL-Unit.

Then for inner-process concurrency and synchronization shown in Figure 5a, the logic could be expressed as follows: use normal MockControl to create the mock implementation so that the method calls will be unordered, then the ordering constraints are expressed by the *syncMethods()* API like this: *syncMethods(op1, op2, op5)*; *syncMethods(op3, op4, op5)*; *syncMethods(op1, op4)*.

For inter-process concurrency and synchronization shown in Figure 5b, the logic could be expressed as follows: use strict MockControl to create the mock implementations for TP1 and TP2 so that the method calls on each mock object will be checked for their correct ordering, e.g. op1 before op2 before op3, then use *syncMethods(op1, op5)* to designate the synchronization between TP1 and TP2. Note that different mock objects are independently invoked at run time so their behaviors are pure concurrent unless explicit synchronization is specified.

4.5 BUTestCase

BUTestCase extends JUnit TestCase class. It is implemented to add business process testing specific APIs and override JUnit APIs to facilitate business process unit test case design. For example, the tearDown() method is overrode to include MockObjects verification logic. For each test method of a test case, tearDown() will be automatically called after the test method is run, thus saving the tester's effort to write verification logic in each test method.

The code below shows a test case for the example purchase process, named PurchaseTest, which extends BUTestCase. In the test case, firstly variables for mock objects and test data objects are declared. Then the variables are initialized in the setUp() method. There can be many test methods defined in a test case, one for each test scenario. The example test method testNormal() checks a complete execution of the purchase process: from the submission of a purchase order to the reply of an invoice. In this method, firstly we set the process input and predict the output. Note that *mockPurchasing* is a mock object of the PUT. The *sendPurchaseOrder*() operation tells the service proxy to start the process, and the *invoice* specified in *setReturnValue*() is used to verify the response of the PUT. Then *mockShip*, *mockPrice* and *mockSchedule* object will receive method calls in parallel. If the method has a return, the return value is set using the *setReturnValue*() method. Finally, the possible synchronization relationship between activities are expressed using the *syncMethods*() API.

```
public class PurchaseTest extends BUTestCase {
  // variables for mock objects and data objects
  public void setUp() {
    // get mock objects & read Process Data Objects
  }
  public void testNormal() throws Exception {
    // Process Input/Output
    mockPurchasing.sendPurchaseOrder(po, customerInfo);
    setReturnValue("Purchasing", invoice);
    // Interaction with Shipping Provider
    mockShip.requestShipping(shipRequest);
    setReturnValue("ShippingService", shipInfo);
    mockShipCallBack.sendSchedule(scheduleInfo);
    //Interaction with Invoice Provider
    mockPrice.initiatePriceCalculation(po, customerInfo);
    mockPrice.sendShippingPrice(shipInfo);
    mockInvoiceCallBack.sendInvoice(invoice);
    // Interaction with Scheduling Provider
    mockSchedule.requestProductionScheduling(po, customerInfo);
    mockSchedule.sendShippingSchedule(scheduleInfo);
     // Synchronization
    MethodSynchronizer.syncMethods(new String[] {
        "ShippingService.requestShipping(ShippingRequest)",
        "ShippingCallback.sendSchedule(ScheduleInfo)" });
    ...}
}
```

5 Conclusion and Future Works

With the increasing attention to business processes in the e-business age, business process testing is becoming more and more important. Lack of unit test tools has resulted in inefficient practices in developing, testing and debugging of automated business processes, e.g. BPEL processes. To address this problem, this paper proposed a BPEL test framework - BPEL-Unit that extends JUnit.

BPEL-Unit has several advantages in supporting BPEL process unit testing.

1. Does not rely on the availability of partner processes. BPEL-Unit provides an easy way to simulate partner processes using mock objects. Thus a single process can be easily tested in isolation.
2. Simplify test case writing. Most developers are already familiar with the JUnit test framework. BPEL-Unit allows process interaction to be programmed in object-oriented flavor. With this tool, developers will no longer be concerned with XML document manipulation, interface, binding and service details in testing.
3. Speed test execution. BPEL-Unit allows "one-deploy, multiple tests". The process under test is deployed only once to run all the test cases associated with this process. This is compared to those methods using stub processes to simulate the partner processes, in which any modification of the stub processes mandates the process redeployment and server restart.
4. Enable automatic regression testing. Process testing is automated by encapsulating all the required test logic and data in formal JUnit test cases. Each time the process under test is modified, its test cases can be re-run (after possible modification) to detect potential function break due to modification.

Currently, we are working on automatic unit test case generation for BPEL processes, which includes searching various execution scenarios, and giving proper test data to enable the execution scenario. The generated BPEL tests can be concretized into BUTestCase format and run in BPEL-Unit.

References

1. http://www.ibm.com/developerworks/library/ws-bpel
2. Process Markup Languages. http://www.ebpml.org/status.htm
3. Z. J. Li, W. Sun, Z. B. Jiang, and X. Zhang. Bpel4ws unit testing: framework and implementation. ICWS2005, volume 1, pages 103 C 110, 11-15 July 2005.
4. Test Driven Development. http://www.testdriven.com
5. WS-Unit. The Web Service Testing Tool. https://wsunit.dev.java.net/
6. ANTEater. Ant based functional testing. http://aft.sourceforge.net/
7. WebServiceTester. http://www.optimyz.com
8. SOAPtest. http://www.parasoft.com/soaptest
9. Philip Mayer and Daniel Lubke. Towards a BPEL unit testing framework. TAV-WEB'06, Pages: 33-42.
10. JUnit. http://www.junit.org
11. MockObjects. http://www.mockobjects.com
12. EasyMock Projects. http://www.easymock.org/EasyMock1_2_Java1_3_Documentation.html