A Business Protocol Unit Testing Framework for Web Service Composition

Jian Yu[1], Jun Han[1], Steven O. Gunarso[1], and Steve Versteeg[2]

[1] Faculty of Information and Communication Technologies
Swinburne University of Technology
Hawthorn, 3122, Melbourne, Victoria, Australia
{jianyu,jhan}@swin.edu.au, 7253702@student.swin.edu.au
[2] CA Labs
380 St. Kilda Rd, Melbourne, VIC 3004, Australia
steve.versteeg@ca.com

Abstract. Unit testing is a critical step in the development lifecycle of business processes for ensuring product reliability and dependability. Although plenty of unit testing approaches for WS-BPEL have been proposed, only a few of them designed and implemented a runnable unit testing framework, and none of them provides a technique to systematically specifying and testing the causal and temporal dependencies between the process-under-test and its partner services. In this paper, we propose a novel approach and framework for specifying and testing the inter-dependencies between the process-under-test and its partner services. The dependency constraints defined in the business protocol are declaratively specified using a pattern-based high-level language, and a FSA-based approach is proposed for detecting the violation of constraints. A testing framework that integrates with the Java Finite State Machine framework has been implemented to support the specification of both dependency constraints and test cases, and the execution and result analysis of test cases.

Keywords: unit testing, WS-BPEL, temporal patterns, Finite State Automata.

1 Introduction

In recently years, the service-oriented architecture (SOA) is steadily gaining momentum as the dominant technology in developing cross-organisational distributed applications with the estimation of its usage in more than 80% of the applications by the year 2015 [7,16]. SOA promotes creating applications by composing open, autonomous, and internet accessible software components in a loosely coupled manner. Currently, Web services [1] is the main implementation technology for SOA.

The Web Services Business Process Execution Language (WS-BPEL, or BPEL in short) [4] is the de facto industry standard for composing Web services. BPEL is a XML-based workflow language that facilitates the description of process logic and the message interactions between Web services. A BPEL composition/process is also exposed as a Web service. Another partner Web service (or BPEL process) may send messages to this process, receive messages from it, or participate in a two-way interaction with it.

Unit testing [8] has been recognised as an important step in the software development lifecycle to ensure software quality especially with the prevalence of Test-Driven Development methodology [2], and BPEL unit testing is gradually gaining the attention of the research community since 2005 [15]. Although dozens of studies have been made on BPEL unit testing, most of them focus on the issue of test case generation [21]. Only a few efforts are devoted to creating unit testing frameworks for BPEL [12,11,13]. In particular, all these efforts recognised that ensuring the correctness of the casual and temporal dependencies among the interactions between the Process Under Test (PUT) and its partner services/processes is an important part of the testing. But unfortunately, none of the frameworks provides necessary support to this issue: inter-process dependency testing is only implicitly supported by programming such dependencies in the test case, which is tedious, error prone, and lacks of maintainability.

In this paper, we present a BPEL unit testing approach for specifying and testing the inter-process dependencies among the PUT and its partner services. The dependency constraints defined in the business protocol are declaratively specified using a pattern-based high-level language called PROPOLS [18]. These constraints can be automatically translated to finite state automata. The inconsistency between the PUT and the constraints will be detected if an execution of the process drives an automaton to a non-accepting state, i.e., a dependency constraint is violated. We have implemented a BPEL unit testing framework that integrates the Java Finite State Machine Framework [5] to support the specification of both dependency constraints and test cases, and the execution and result report of test cases. To validate the viability and effectiveness of this approach, we have successfully applied it on testing the interaction protocol of a purchase business process in the e-commerce domain.

The main contribution of this paper is twofold: i) We propose a novel approach to specifying and testing the inter-process dependencies in BPEL unit testing based on temporal patterns and finite state automata; ii) We implement a framework for conducting the inter-process dependency testing in BPEL unit testing. The main functionality of this framework includes the specification of both dependency constraints and test cases, and the execution and result analysis of test cases.

The rest of the paper is organised as follows: In Section 2, we introduce the background of this research work by explaining some basic concepts in BPEL unit testing and introducing a motivating business scenario. In Section 3, we explain in detail our approach to specifying the inter-dependencies between the PUT and its partner services including the pattern-based declarative specification language and how to use it to define the inter-dependencies in the example scenario. Section 4 presents the overall process of conducting BPEL unit testing using our framework. Section 5 introduces the implementation details of the framework. Section 6 discusses related work, and Section 7 concludes the paper.

2 Background

In this section, we first introduce the basic concepts that are used in the context of this paper including unit testing, BPEL unit testing, and inter-process dependencies. Then we introduce a motivating scenario in the e-commerce domain to highlight the need for a systematic approach for the specification and testing of BPEL inter-process dependencies.

2.1 Unit Testing and BPEL Unit Testing

Unit testing is a software development process in which the smallest testable parts of an application, called units, are individually and independently scrutinised for proper operation [17]. Similar to a hardware unit, a software unit needs to have clearly defined interfaces, and testing is carried out around interfaces. In Java, a unit is usually a class, which may implement several interfaces. Because a BPEL process is exposed as a Web service and communicates with its partner services through standard Web service invocations, a BPEL process naturally becomes a unit with its interfaces defined in the WSDL descriptions of this process.

BPEL interfaces are described using port types and operations in WSDL. An operation could be asynchronous (one-way) or synchronous (two-way), and an asynchronous operation could either receive a data flow, or send a data flow. Based on the interfaces defined in the corresponding WSDL description, unit testing of a BPEL process is performed by providing a series of inputs and observing the outputs. The following three types of errors in the process under test can be detected [12]:

1. Incorrect output message content
2. Output absence, i.e., an expected output is not produced by the PUT
3. Output surplus, i.e., an unexpected output is produced by the PUT

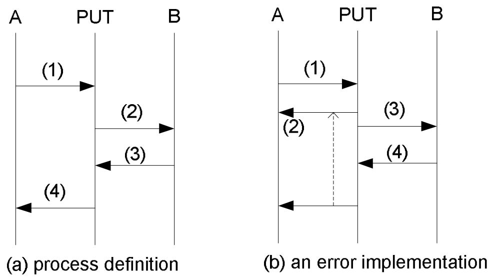


Fig. 1. An example of process dependencies

However, WSDL syntax lacks the ability to describe the actual business protocol, or dependencies, involved in the interaction between the PUT and the partner services, e.g., which operation must be invoked after or before which other operations [13]. Such violation of protocol error may not be detected by just observing the outputs. For example, in [12], the authors described the following case as shown in Figure 1, which shows an interaction segment between the PUT and two partner processes A and B. The numbers denote the sequencing of the message flows. Figure 1(a) shows a correct PUT definition according to the requirements, and Figure 1(b) shows a wrong implementation that moves Message#4 in Figure 1(a) up to next to Message#1 (as the dashed line indicates). A test process that simulates A and B running in parallel and without synchronisation may not detect such an error in situations that both the send-receive behaviour of A and the receive-send behaviour of B complete their own logic successfully. It is necessary for a BPEL testing framework to provide a way for the software engineering to specify the interaction protocol between the PUT and its partner services, and to follow it in the testing [13].

2.2 Motivating Business Process Scenario

Next we describe a purchase business process in the e-commerce domain as a motivating business scenario. This process will be used to demonstrate our approach throughout this paper. Suppose a manufacturer wants to provide an online purchasing service. The key requirements to this service are sketched as follows:

1) The customer may log into the system and place orders online. Login can only be tried three times. After three unsuccessful login, the process will abort. Any order should be checked before being processed. For a valid one, the customer will get a confirmation. For an invalid order, the customer will get a rejection notification.

2) The transactions between customers and the manufacturer follow a hard credit rule. That is, on the one hand, the customer pays to the manufacturer only when the ordered product has been received. On the other hand, the manufacturer processes the order only when it is confirmed that the customer will pay if the order is fulfilled. A third party, the bank, is introduced as the mediator. For the manufacturer to start processing the order, the customer must deposit the payment to the bank in the first place. After that, the bank will notify the manufacturer that the payment for the order has been deposited in the bank. When the order is fulfilled, the bank ensures that the payment is transferred to the manufacturer.

Figure 2 shows a possible process design of the above requirements, which includes three parties: the customer, the manufacturer, the bank, and their interacting messages. We assume that the manufacturer process is the process under test.

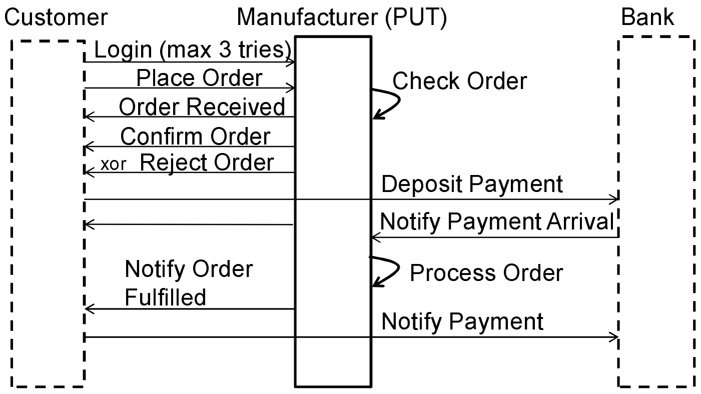


Fig. 2. The motivating business process

3 Specification of Inter-process Dependencies

From the example in Figure 1, we can see that the inter-process dependencies determine the sequence of interacting messages between processes. One approach to checking the dependency is to hard-code the testing logic in a test case. For example, a test case may check whether Message B appears after Message A but before Message C to ensure the correct sequence of the three messages. But such approach is rather rudimentary and lacks of the capability to formally and systematically specify the logical relationships between process messages that are derived from the actual requirements of the business process. To properly address this issue, we adopt the PROPOLS (Property Specification Pattern Ontology Language for Service Composition) language [18], which is a high-level declarative language with formal semantics defined in FSA (Finite State Automata). Next we give a brief introduction to PROPOLS, and also give some examples of how to use it to specify some inter-process dependencies in the motivating business process.

3.1 The PROPOLS Language

The PROPOLS language is based on property specification patterns [3], which include a set of patterns that represent frequently used temporal logic formulae. The main feature of the PROPOLS language is that it is a high-level declarative language that facilitates common users (such as software engineer or business analyst) to define the temporal and causal relationships between messages. Because it also has a formal semantics, relationships (or constraints) defined using PROPOLS can be automatically verified.

The main constructs of the PROPOLS language is shown in the class diagram in Figure 3. As we can see, every PROPOLS statement is composed of a *Pattern* and a *Scope*. The pattern specifies what must occur and the scope specifies when the pattern must hold.

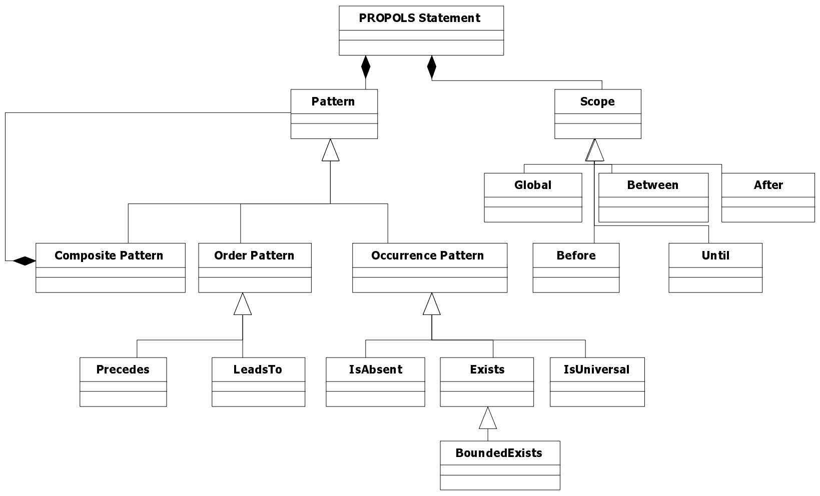


Fig. 3. Main constructs of the PROPOLS language

Patterns are classified into *order patterns*, *occurrence patterns*, and *composite patterns*, where composite patterns are the composition of patterns. Below we briefly describe the meaning of each pattern below (the symbol P or Q represents a given message).

– *P IsAbsent*: P does not occur within a scope
– *P IsUniversal*: P occurs throughout a scope
– *P Exists*: P must occur within a scope
– *P Bounded Exists*: P occurs at most k times within a scope
– *P Precedes Q*: P must always precede Q within a scope.
– *P Leadsto Q*: P must always be followed by Q within a scope
– *Composite Pattern*: combining two patterns using one of the following Boolean logic operator: *And, Or, Xor, Imply*

It is worth noting the difference between *P Precedes Q* and *P Leadsto Q*: if P Precedes Q, then whenever Q occurs, P must occur preceding Q, but P may exist in a scope without the occurrence of Q. On the other hand, if P Leadsto Q, then whenever P occurs, Q must occur after P, but Q may exist in a scope without the occurrence of P.

A scope defines a starting and an ending message for a pattern, and a pattern is not applicable outside its scope. There are five types of scope:

– *Globally*: the pattern must hold during the entire system execution
– *Before P*: the pattern must hold up to the first occurrence of a given P
– *After P*: the pattern must hold after the first occurrence of a given P
– *Between P And Q*: the pattern must hold from an occurrence of a given P to an occurrence of a given Q
– *After P Until Q*: the same as *between-And*, but the pattern must hold even if Q never occurs

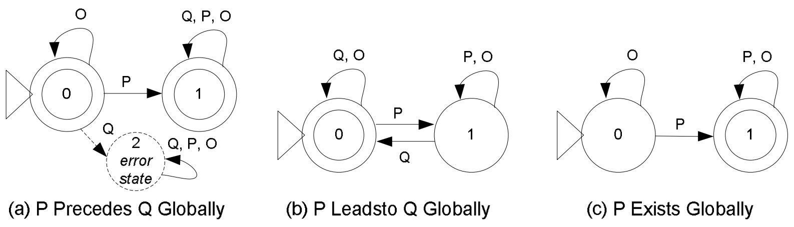
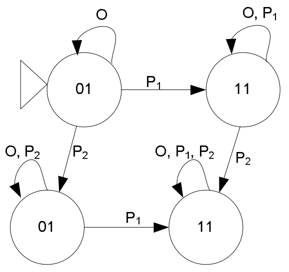


Fig. 4. FSA semantics of three types of statements

Usually for a process that has a single globally defined protocol, the *Globally* scope is used on all the patterns. But other scopes may be used when there are local protocols inside a business process.

The semantics of a basic statement is defined in FSA. For example in Figure 4 we illustrate the FSA semantics for three types of statements: *Precedes*, *LeadsTo*, and *Exists*. In the Figure, symbol O denotes any message other than P and Q. Figure 4(a) indicates that before P occurs, an occurrence of Q will drive the FSA to a non-final state, and this non-final state can never reach a final state. We call such a state an error state and error states are omitted in the FSA graphical representations for brevity's sake. Figure 4(b) states that if Q has occurred, an occurrence of P is necessary to drive the FSA to a final state. Finally, Figure 4(c) says that only the occurrence of P can drive the FSA to a final state. A complete FSA semantics of all the basic statement types can be found in [19].

The semantics of a composite statement is derived by composing the FSAs of its component statements. For example, Figure 5 shows the logical composition of two basic statements: P_1 *exists globally* and P_2 *exists globally*. The states in the FSA are the Cartesian product of the FSAs of the two basic statements. The first number in a state label represents the state of the first FSA, while the second represents the state of the second FSA. The final states of the composite pattern are determined by the logic operator used. For example, the pairing of one final state *And* one non-final state is a non-final state, and one final state *Xor* one non-final state is a final state. The final states for different compositions are also described in Figure 5. The full definition of the semantics of composite statements can be found in [19].



Logic Operator	*Final States*
and	11
or	01,10,11
xor	01,10

Fig. 5. Composition of two *Exists* statements

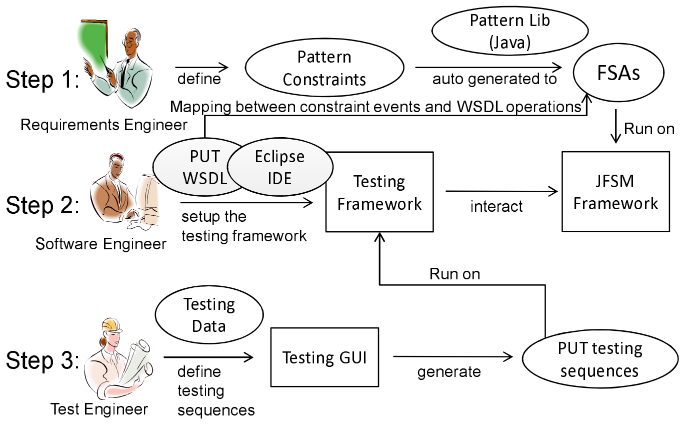


Fig. 6. Overview of the testing approach

3.2 Specification of the Inter-process Dependencies

In the following we give a list of PROPOLS statements that specify the causal and temporal inter-dependencies (or dependency constraints) among the three parties/processes in the motivating scenario based on the requirements, from the PUT (Manufacturer) perspective.

(S1) Manufacturer.Login *boundedexist*(3) *Globally*
(S2) Manufacturer.Login *precedes* Manufacturer.PlaceOrder *Globally*
(S3) Manufacturer.PlaceOrder *leadsto* Customer.OrderReceived *Globally*
(S4) (Customer.OrderConfirmed *exists Globally*) *xor* (Customer.OrderRejected *exists Globally*)
(S5) Manufacturer.CheckOrder *precedes* Manufacturer.confirmOrder *Globally*
(S6) Manufacturer.CheckOrder *precedes* Manufacturer.RejectOrder *Globally*
(S7) Bank.DepositPayment *leadsto* Manufacturer.NotifyPaymentArrival *Globally*
(S8) Customer.NotifyOrderFulfilled *leadsto* Bank.NotifyPayment *Globally*

In the list of statements, each message is preceded by the receiver, or the service provider, of the message. For example, S_1 is supposed to be received and processed by the manufacturer. Among the list of statements, S_1 is an occurrence pattern that specifies that login can only be tried maximally three times. We also have three Precedes statements that specify the precondition of some messages. For example, S_1 specifies that `Login` must be successfully performed to enable the `PlaceOrder` message/operation. Three *Leadsto* statements are defined in the list. For example, S_3 specifies that if the order is successfully placed, the customer must receive a notification that the order has been received by the manufacturer. Finally, we use a composite statement S_4 to specify that the customer either receive an order confirmation or order rejection, but not both. It is worth noting that we apply the scope *Globally* on all the statements because all the statements are used to specify a global interacting protocol among the participating processes.

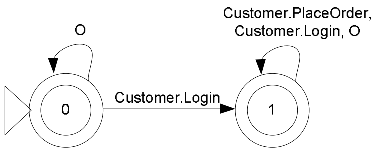


Fig. 7. FSA for *Customer.Login precedes Customer.PlaceOrder Globally*

4 Testing of Inter-process Dependencies

We give an overview of the testing approach in Figure 6. In the rest of this section we discuss each step in detail.

In Step 1, first the requirements engineer defines the inter-process dependencies using PROPOLS. The details of this step and the example PROPOLS statements defined based on the motivating scenario have been discussed in Section 3. After that, every statement is automatically translated to a FSA. For basic statements, there is a one-to-one mapping between it and a FSA template. So what we need to do is to parameterize the corresponding FSA template. For example, the statement *Customer.Login precedes Customer.PlaceOrder globally* can be represented by the *precedes* FSA template as shown in Figure 7. Similarly for a composite statement, we first compose the FSA templates based on the definition given in [19], and then parameterize the composed FSA template.

In Step 2, we need to solve the issue of how to create a testing environment for the PUT to interact with, while the partner services of the PUT are not available or not implemented at all. As illustrated in Figure 8(a), a BPEL process under test usually needs to interact with several partner services to implement a business process such as the purchase process specified in Section 2.2. Because the partner services are usually located and managed by the partner organisations, they are out of the control of the organisation that owns the PUT. Such situation brings difficulties to testing the PUT as the availability of the partner services cannot be guaranteed, and also using external partner services may bring extra cost. To solve this issue, a basic test model (as illustrated in Figure 8(b)) may be adopted to use a test process to either serve as a mock object for a partner service, or emulate the behaviour of a real partner service [11]. Alternatively, as illustrated in Figure 8(c), if we use the emulation based approach, we may also use a composite test process to emulate the behaviour of all the partner services of the PUT. In our approach, we adopt the composite test process model. The benefit of this approach is that we can focus on the behaviour of the PUT to provision a single emulated test environment (the composite test process) instead of focusing on the behaviour of individual partner services.

In Step 3, the test engineer specifies the test cases including testing data and testing sequences. Each testing sequence specifies a behaviour of the composite test process that is used to exercise the PUT. For example, a testing sequence could be:

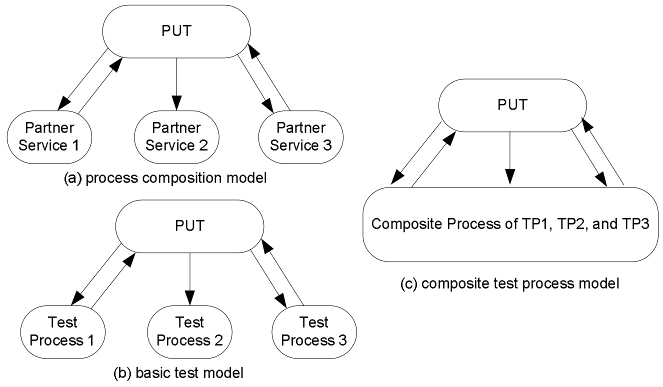


Fig. 8. Process composition and test models

Seq1: Login(incorrect identity) →Login(incorrect identity) →Login(incorrect identity) →Login(correct identity)

If this sequence is accepted by the PUT without throwing any exceptions, then the PUT violates the requirement that `Login` can only be tried three times. But this fault will be captured by the test framework as this sequence will drive the FSA of S_1 (a bounded exist statement specifying that `Login` can only be tried three times) to an error state.

Another example is as follows:

Seq2: Login(correct identity) →PlaceOrder →OrderReceived →DepositPayment

If this sequence is accepted by the PUT without throwing any exceptions, then the PUT violates S_4 that it fails to either confirm or reject the order. As such, this fault will also be captured by the FSA of S_4.

5 Testing Framework Implementation

The BPEL unit testing framework contains four main functions: *PUT to Java mapping*, *mock objects setup*, *test cases definition* and *test case execution.* In the rest of this section, we discuss the technical details of each function one by one.

5.1 PUT to Java Mapping

In order for the PUT to interact with the testing framework, we map the web service interface (WSDL) of the PUT into several Java interfaces. These Java interfaces then serve as a bridge between the PUT web service and the testing framework.

Figure 9 shows how the mapping is done. In this figure, the Web Service block represents the WSDL structure of the PUT, while the Java Client block

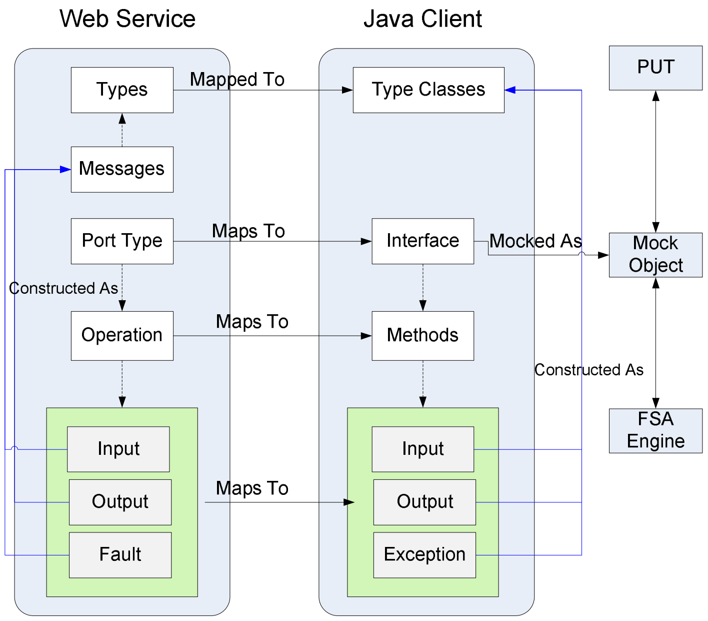


Fig. 9. Map Web service interface to Java interface

represents the results of the mapping. As seen in Figure 9, the mapping is done for each port type in the WSDL. I.e., each port type is mapped to a Java interface. Because each port type may contain a number of operations, and each operation has its own name and parameters, we also map these operations into Java methods to enable access to them. Also, similar to a Java class, each operation requires inputs and produce outputs or exceptions (Faults), which are represented as WSDL Messages. These messages themselves may contain several variables, each with their own data types (such as int, float, string). In the mapping process, these messages and data types are also represented as custom Java classes and variables.

5.2 Mock Objects Setup

Mock objects are the instantiation of the Java interfaces mapped from the PUT. As discussed in the previous section, the PUT to Java interface mapping results in Java interface classes. These interface classes are the core of the mock/emulation capabilities of our testing framework. By instantiating these interfaces, the testing framework is able communicate with the specified web service and access the operations contained inside it via the generated methods. In this section, we use a *Login* operation from our case study as an example to demonstrate the setup of the mock objects and their interaction with the PUT.

As discussed in the motivating scenario, the customer is the one who initiates the entire Order Processing by logging in to the system. To emulate this behaviour, we use the generated *login* method inside the UnitTestingMain class (as shown in the following code snippet). By calling the *login*() method, we can send emulated login credentials to the PUT and observe the returning values.

```
<portType name="UnitTesting_Main">
    <operation name="Login">
            <input message="tns:LoginRequest"></input>
            <output message="tns:LoginResponse"></output>
        <fault name="fault" message="tns:ProcessOrderFault
            "></fault>
    </operation>
</portType>
```

$\downarrow$*mapsto*

```
public interface UnitTestingMain {
    public LoginResponse login(LoginRequest payload) throws
       ProcessOrderFault_Exception;
}
```

As discussed in the previous subsection, each method requires specific messages for both the parameter and the return. In our *login* example, the method uses *LoginRequest* message as parameter and *LoginResponse* message as return type. The structure of the *LoginRequest* messages is shown in the code snippets below.

```
<element name="LoginRequest">
    <complexType>
        <sequence>
            <element name="username" type="string" />
            <element name="password" type="string" />
            </sequence>
        </complexType>
</element>
```

$\downarrow$*mapsto*

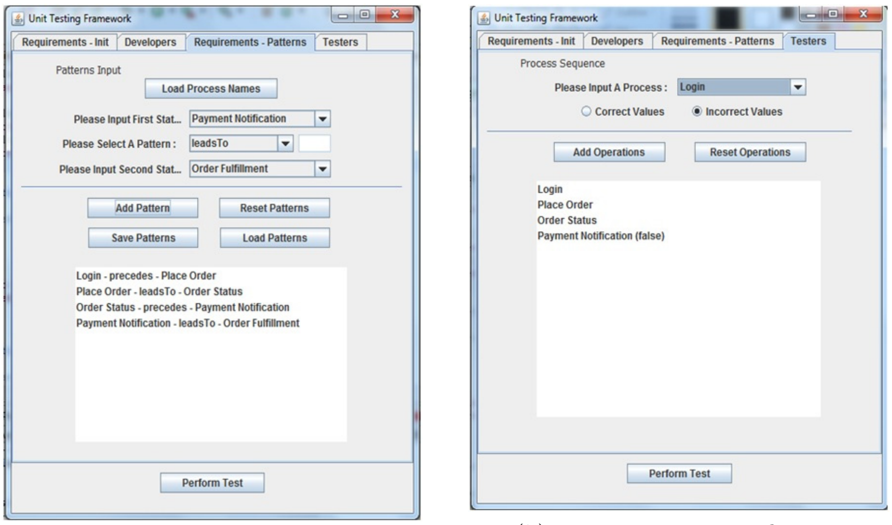
```
public class LoginRequest {
    protected String username;
    protected String password;
}
```

By manipulating the values inside these message classes and using them in conjunction with the Java client methods, we can emulate the behaviour of the partner services and perform the unit testing based on the returned results. The following code snippet shows the manually defined emulation code for the *login* method.

```
try {
  UnitTestingMain_Service service1 = new
      UnitTestingMain_Service();
  UnitTestingMain port1 = service1.getUnitTestingMainPort()
      ;

  LoginRequest request = new LoginRequest();

  request.setUsername(input1);
  request.setPassword(input2);

  LoginResponse response = port1.login(request);
  return response;
catch (Exception e) {
  return "Process Fault";
}
```

5.3 Test Case Definition

In the current implementation of the testing framework, we do not support automatic test case generation. Instead, several graphical user interfaces have been implemented to facilitate the definition of test cases.



Please Input First Stat... Payment Notification
Please Select A Pattern : leadsTo
Please Input Second Stat... Order Fulfillment

Login - precedes - Place Order
Place Order - leadsTo - Order Status
Order Status - precedes - Payment Notification
Payment Notification - leadsTo - Order Fulfillment

Please Input A Process : Login
Correct Values [✓ Incorrect Values]

Login
Place Order
Order Status
Payment Notification (false)

(a) Patten specification

(b) Test sequence specification

Fig. 10. GUI for test case definition

There are mainly two steps in defining a test case: the first step is to specify the set of constraints that are supposed to hold for this test case (it is worth noting that there could be different sets of constraints depending on testing purposes), and the second step is to define a running sequence of the PUT that will be tested against the defined constraints.

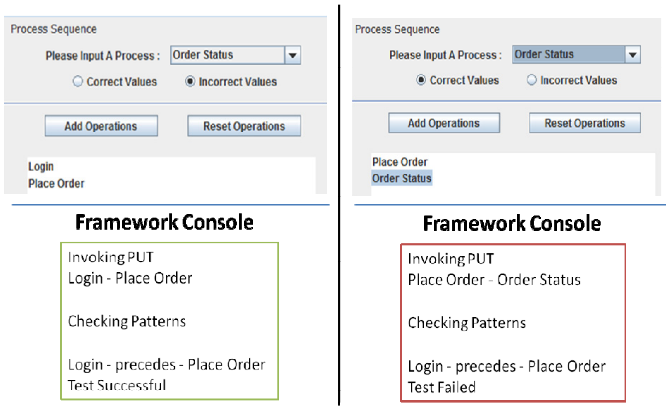
The graphical interface for specifying the dependency constraints is shown in Figure 10(a): a process definition can be loaded and then the operations in this process will be automatically extracted and put in two drop-down lists for selection (one for the first parameter of a constraint, and the other for the second parameter). All the patterns are also put in another drop down list for selection. Based on the three drop-down list, the set of dependency constraints can be defined one by one, and the defined constraints will be displayed in the information area in the centre bottom. When the set of constraints are saved, they will automatically be converted to FSAs that will be executed by the Java FSM framework.

The graphical interface for specifying the testing sequence is shown in Figure 10(b): in this step, the test engineer is able to specify which operations should be inserted into the testing sequence. Aside from the operations sequence, the testing engineer can also specify whether an operation is a valid or invalid one using the GUI. When a valid operation is chosen, the framework will fill the operation with valid values of the defined type classes. Otherwise, the framework simply leaves the value of the type classes inside the operation as blank. Invalid operations are necessary for testing certain logic/execution paths of the PUT. For example, to test the constraints that `Login` can only be tried three times, the testing engineer may specify a sequence with three invalid `Login` operations followed by one valid `Login` operation, if the PUT still accepts the fourth `Login` operation and proceed to the Place Order operation, then it violates the constraint and such fault will be captured by the testing framework.

5.4 Test Case Execution

To execute a test case, the framework sends the specified operations to the PUT according to the sequence defined in the test case, and also examines the PUT responses. The framework also drives the FSAs in the FSA Engine whenever there is a match between the interaction messages and an enabled event in any FSA. For example, when a `Login` message is sent from the mock object client to the PUT, because it matches the enabled event of the FSA defined in Figure 7, the framework will drive this FSA from state 0 to state 1.

A dependency error will be detected by the framework whenever any of the FSAs is driven to an error state, or any of the FSAs is in a non-final state when the test case finishes execution. For example as shown in Figure 11, if a PUT executes the `PlaceOrder` operation without a preceding `Login` operation, then this violates the prescribed constraints and will be reported as an error by the testing framework.



Process Sequence
Please Input A Process : Order Status
Correct Values [✓ Incorrect Values]
Add Operations | Reset Operations
Login
Place Order

Framework Console
Invoking PUT
Login - Place Order

Checking Patterns

Login - precedes - Place Order
Test Successful

Process Sequence
Please Input A Process : Order Status
[✓ Correct Values] Incorrect Values
Add Operations | Reset Operations
Place Order
Order Status

Framework Console
Invoking PUT
Place Order - Order Status

Checking Patterns

Login - precedes - Place Order
Test Failed

Fig. 11. Snapshot of the testing results

6 Related Work

Testing of Web service composition is still a new and immature area compared to other Web service research areas such as service discovery, selection, and composition [15]. In general, Web service composition testing approaches can be divided into two categories: test case generation and unit testing framework. Clearly, our work falls in the latter category. In the rest of this section, we first briefly discuss the related work in BPEL test case generation, and then discuss the related work in BPEL unit testing framework and also compare our work with these approaches.

6.1 Related Work in BPEL Test Case Generation

According to [15], there are mainly three categories of approaches to BPEL test case generation: the model-checking approach, the graph search algorithm approach, and the path analysis constraint solver approach. The model-checking approach is a model-based testing method in which model checkers are applied to the abstract model of the Web service process under test in order to generate test cases. For example, Garcia-Fanjul et al. [6] transform a BPEL process into PROMELA, the input language of model checker SPIN, and then use SPIN to generate both positive and negative test cases. The graph search algorithm approach transforms the composition specification into graph models and test paths are derived by traversing the model. For example, Lallali et al. [10] transform a BPEL process into an Intermediate Format (IF) model which is based on timed automata, and then test cases are generated from the IF model. The path analysis constraint solver approach generates test cases by analysing the test paths which are constructed from graph models derived from depth-first or breadth-first traverse of the composition specification. For example in [20], the authors transform a BPEL process into an extension of Control Flow Graph called BPEL Flow Graph (BFG), and concurrent test paths are generated by traversing the BFG using depth-first search, while test data for each path are generated by using constraint solvers.

6.2 Related Work in BPEL Unit Testing Framework

To the best of our knowledge, there are only three published approaches to BPEL unit testing framework. In [12], Li et al. introduce an implementation of a BPEL unit testing framework. They use BPEL as the test specification language, and require test engineers to create a BPEL test process for each partner service of the PUT, as well as a central process as the coordinator of testing. However, this work does not discuss the issue of how to actually run the tests as BPEL itself does not allow user interactions. In [11], Li and Sun propose a new BPEL unit testing framework based on their previous work in [12]. In this work, the authors extend the object-oriented unit testing framework especially JUnit [9] and MockObjects [14] to support BPEL unit testing. Process interaction via Web service invocations are transformed to class collaboration via method calls, and then the object-oriented test framework and methods are applied. In [13], Mayer and Lubke propose a layer-based approach and framework for BPEL unit testing. They use a specialised XML-based BPEL-level testing language to describe interactions with the PUT in a test case. In the test case, literal XML data are used as the data specification format and the interaction between the PUT and its partner services is specified as testing sequences.

In all the above works, the authors unanimously mentioned the important issue of testing the inherent business protocol between the PUT and the partner services. [12] raised the issue but did not provide any solution. [11] proposed to extend their implementation with a `syncMethods()` API to specify the occurrence order of method invocations. [13] also allowed test engineer to specify sequences of the interactions in their test case specification language. It is clear that existing support to testing message sequences, or inter-dependencies, stops at the programming level. Also, there is a lack of a specification language and approach to systematically conduct such test. This issue is actually what our work aimed to solve.

7 Conclusion

In this paper, we have proposed a novel approach and framework to specify and test the causal and temporal inter-dependencies between a BPEL process under test and its partner Web services. A high-level declarative pattern language is used to specify the interaction dependencies and a comprehensive framework has been implemented for specifying the dependencies and the test cases, and also for executing the test cases. Instead of using a basic test model in which each partner service of the PUT is specified separately, we use a composite test process to emulate the behaviour of all the partner services. To the best of our knowledge our work is the first BPEL unit testing framework that is able to systematically specify and test the inherent business protocol between the PUT and its partner services.

In the future, we intend to investigate into the business protocol related test case generation issue. We also intend to integrate our work with the existing BPEL unit testing frameworks, such as those discussed in [13] and [11], to provide a comprehensive testing tool and then conduct case studies in real life Web service composition projects.

References

1. Alonso, G., Casati, F., Kuno, H.A., Machiraju, V.: Web Services - Concepts, Architectures and Applications. Springer (2004)
2. Beck, K.: Test Driven Development: By Example. Addison-Wesley Professional (2002)
3. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property Specification Patterns for Finite-State Verification. In: Proc. of the 2nd Workshop on Formal Methods in Software Practice, pp. 7–15 (1998)
4. Evdemon, J., Arkin, A., Barreto, A., Curbera, B., Goland, F., Kartha, G., Khalaf, L., Marin, K., van der Rijn, M.T., Yiu, Y.: Web Services Business Process Execution Language Version 2.0. In: BPEL4WS Specifications (2007)
5. eVelopers Corporation. Java Finite State Machine Framework (2007), `http://unimod.sourceforge.net/fsm-framework.html`
6. Garca-Fanjul, J., Tuya, J., de la Riva, C.: Generating Test Cases Specifications For BPEL Compositions Of Web Services Using Spin. In: Proc. of the International Workshop on Web Services Modeling and Testing (WS-MaTe 2006), pp. 83–94 (2006)
7. Georgakopoulos, D., Papazoglou, M.P.: Service-Oriented Computing. The MIT Press (2008)
8. Hamill, P.: Unit Testing Frameworks. O'Reilly (2004)
9. JUnit, `http://www.junit.org`
10. Lallali, M., Zaidi, F., Cavalli, A.: Transforming BPEL Into Intermediate Format Language For Web Services Composition Testing. In: Proc. of the 4th International Conference on Next Generation Web Services Practices, pp. 191–197 (2008)
11. Li, Z.J., Sun, W.: BPEL-unit: JUnit for BPEL processes. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 415–426. Springer, Heidelberg (2006)
12. Li, Z.J., Sun, W., Jiang, Z.B., Zhang, X.: Bpel4ws unit testing: Framework and implementation. In: ICWS, pp. 103–110 (2005)
13. Mayer, P., Lübke, D.: Towards a bpel unit testing framework. In: TAV-WEB, pp. 33–42 (2006)
14. MockObjects, `http://www.mockobjects.com`
15. Rusli, H.M., Puteh, M., Ibrahim, S., Hassan, S.G.: Comparative Evaluation of State-of-the-Art Web Service Composition Testing Approaches. In: Proc. of the 6th International Workshop on Automation of Software Test (AST 2011), pp. 29–35 (2011)
16. Vaughan, J.: Gartner: SOA Will Be Like Electricity For Architects Looking Toward Cloud Computing. SOA News (2010), `http://searchsoa.techtarget.com/news/article/0,289142,sid26_gci1523670,00.html`
17. Wikipedia. Unit testing (2002), `http://en.wikipedia.org/wiki/Unit_testing`

18. Yu, J., Manh, T.P., Han, J., Jin, Y., Han, Y., Wang, J.: Pattern Based Property Specification and Verification for Service Composition. In: Aberer, K., Peng, Z., Rundensteiner, E.A., Zhang, Y., Li, X. (eds.) WISE 2006. LNCS, vol. 4255, pp. 156–168. Springer, Heidelberg (2006)
19. Yu, J., Phan, M.T., Han, J., Jin, J.: Pattern based Property Specification and Verification for Service Composition. Technical Report SUT.CeCSES-TR010. CeCSES, Swinburne University of Technology (2006), http://www.it.swin.edu.au/centres/cecses/trs.htm
20. Yuan, Y., Li, Z., Sun, W.: A Graph-Search Based Approach to BPEL4WS Test Generation. In: Proc. of the International Conference on Software Engineering Advances, ICSEA 2006 (2006)
21. Zakaria, Z., Atan, R., Ghani, A., Sani, N.: Unit Testing Approaches for BPEL: A Systematic Review. In: APSEC, pp. 316–322 (2009)