

Basis Path Test Suite and Testing Process for WS-BPEL

THEERAPONG LERTPHUMPANYA^{1,2} AND TWITTIE SENIVONGSE¹

¹Department of Computer Engineering, Chulalongkorn University

Phyathai Road, Pathumwan, Bangkok 10330

²Department of Software Engineering, Bangkok University

Rama IV Road, Klong Toey, Bangkok 10110

THAILAND

theerapong.l@bu.ac.th <http://tulip.bu.ac.th/~theerapong.l/>

twittie.s@chula.ac.th <http://www.cp.eng.chula.ac.th/~twittie/home.htm>

Abstract: - Web services technology offers a WS-BPEL language for business process execution. The building block of WS-BPEL is those Web services components that collaborate to realize a certain function of the business process. Applications can now be built more easily by composing existing Web services into workflows; each workflow itself is also considered a composite Web service. As with other programs, basis path testing can be conducted on WS-BPEL processes in order to verify the execution of every node of the workflow. This paper discusses the generation of the test suite for basis path testing of WS-BPEL and an accompanying tool that can be used by service testers. The test suite consists of test cases, stubs of the constituent services in the workflow, and auxiliary state services that assist in the test; these are deployed when running a test on a particular WS-BPEL. The paper presents also a testing process for service testers. A business process of a market place is discussed as a case study.

Key-Words: - Basis path testing, WS-BPEL, test cases, control flow graph, cyclomatic complexity

1 Introduction

Web services technology opens an opportunity to aggregate functionalities of various loosely-coupled software components, called services, in a software composition. This is due to the fact that service consumers' requirements cannot be fulfilled by a single Web service and hence several Web services are composed in order to answer the requirements. The composition can be defined by a Business Process Execution Language (WS-BPEL) which has become a standard for describing a workflow of collaborating Web services [1]. With WS-BPEL, the workflow can be seen as an internal process of a composite Web service which executes by the orchestration of an execution engine.

A service composer designs a composite Web service by defining Web service instances of particular tasks and how they collaborate in the workflow. Such a composite service can be used further as a component in the construction of other composite services. Before its deployment, the WS-BPEL service will be tested by a service tester to ensure a correct composition. Web services testing tools mostly focus on single Web services, treating them as black boxes and testing their functions or performance, while WS-BPEL tools support designing but not testing of the composition of the designed workflows.

As with other programs, basis path testing [2] can be conducted on WS-BPEL processes in order to verify the execution of every node of the workflow. This paper discusses the generation of the test suite for basis path testing of WS-BPEL and an accompanying tool that can be used by service testers. The test suite for a particular WS-BPEL process consists of test cases, stubs of the constituent Web services within the flow, and auxiliary state services that assist in the test; these are deployed when running a test on that WS-BPEL. Such a test is white box testing, considering the internal process of the composite service while performing black box testing on the constituent Web services through their corresponding stubs. The supporting tool creates a control flow graph for any WS-BPEL flow and determines its basis paths based on McCabe's cyclomatic complexity [3]. It generates test case data for all basis paths together with service templates, stubs for constituent Web services, and auxiliary state services with little human intervention. The paper describes also the basis path testing process for service testers.

Section 2 reviews related work on Web services testing. Section 3 describes basis path testing and introduces a market place case study. Details on each task realized by our supporting tool for the generation of the test suite are in Section 4, followed by the testing procedure to be taken by service

testers in Section 5. Section 6 gives implementation details of the tool. Section 7 discusses the approach and Section 8 concludes the paper.

2 Related Work

There have been research attempts to define frameworks or create testing tools for Web services. The work in [4] extends WSDL of a Web service with information necessary for testing the service including input-output dependency, sequence of method invocation, hierarchical functional description, and sequence specification. This information helps automate regression testing, data flow testing, and path testing. The work in [5] introduces a tool to measure CPU utilization and response time of a Web service by installing a sensor object at the service provider's site. The tool can also perform unit testing on the service by using the information in the extended WSDL of [4]. In [6], a test master is used to generate XML-based test cases from the WSDL of a Web service and a test engine is used to run the test cases for the service. In [7], a framework for generating Web service requests and analyzing subsequent request-response pairs is proposed. Based on the WSDL of a Web service, a client code is automatically generated to send requests to the Web service with extreme, special, and random input values. The response is then analyzed to see how robust the Web service is in response to the generated inputs.

On the commercial side, several testing tools are available in the market such as those in Table 1. They primarily target on testing certain aspects of a Web service and now some are extending to testing at business process level, e.g. SOATest and WebServiceTester support load testing for the entire workflow of Web services instead of load testing for individual service endpoints.

To summarize, current research and commercial tools mostly treat Web services as black boxes and test their functions or performance; none support basis path testing on the internal processes of composite services.

Table 1. Web services testing tools

Tool	By	Testing Capability
SOATest [8]	Parasoft	- Server Functional Testing - Load Testing - Client Testing - Regression Testing - Performance Testing - Workflow testing

Table 1. Web services testing tools (continued)

Tool	By	Testing Capability
WebService Tester [9]	Optimyz Software	- Functional Testing - Regression Testing - Load/Stress Testing - Performance Testing - Business Process Orchestration Testing - Secure Service Testing
Stylus Studio [10]	Progress Software Corporation	- Functional Testing
SilkPerformer [11]	Borland	- Load Testing - Performance testing
SOAPscope [12]	Mindreef	- Functional Testing - Regression Testing - Load/Stress Testing - Performance Testing - Client Testing
soapUI [13]	Eviware	- Functional Testing - Load Testing
TestMaker [14]	PushToTest	- Functional Testing - Load Testing

3 Basis Path Testing

Basis path testing is a white box testing that aims for deriving a logical complexity measure of a procedural design of a program and using the complexity measure as a guide for defining a basis set of execution paths. For WS-BPEL, a service tester will perform basis path testing to test whether all the loops and conditions within the workflow give proper outputs and whether existing data flows and control flows are correct. Test cases that exercise the basis set will execute every statement in the workflow at least once during testing.

This paper demonstrates basis path testing through the business process of a market place service, adapted from [15], as in Fig. 1. The market place service acts as a negotiator for a buyer and a seller on the sale of products. Given a particular product ID as an input, the market place negotiates by invoking a Seller Web service and a Buyer Web service to obtain the offered selling price and buying price and compare them. The negotiation repeats until the offered prices match. The commission is then charged to the client of the market place; the rate depends on the product category and the efforts put on the negotiation.

The rest of the paper focuses mainly on two aspects: the generation of a test suite for the market place service and a testing procedure for a service tester to utilize the test suite.

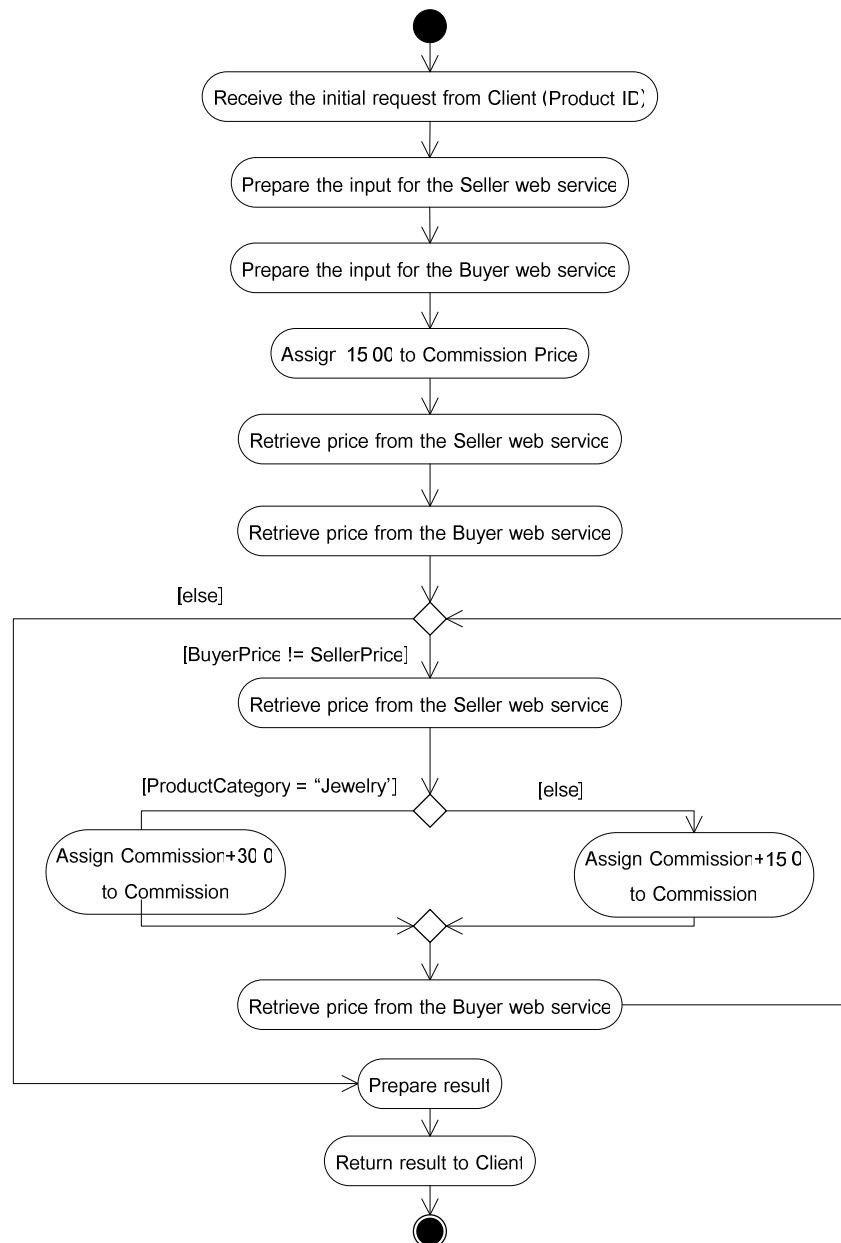


Fig. 1. Business process of market place service

4 Test Suite Generation

Fig. 2 depicts how a basis path test suite for a WS-BPEL is generated. These tasks are supported by our testing tool; each is described in the corresponding numbered subsections.

4.1 Receive Input Files

A WS-BPEL can be created by a designer tool. In this work, we use Oracle BPEL Designer [15] which produces a WS-BPEL file, a WSDL file for the WS-BPEL service, and a configuration file which contains PartnerLinks of the WS-BPEL (i.e. links to

WSDLs of constituent Web services). These files are uploaded to the tool as in Fig. 3.

4.2 Insert Node IDs to WS-BPEL

Basis path testing starts with creating a control flow graph (CFG) for the WS-BPEL. To do so, node IDs are first associated with WS-BPEL constructs, e.g. input receipt, output reply, assignment, sequence, repetition, condition, and invocation control within the workflow. We insert Node IDs by using Java embedding feature of Oracle BPEL designer such as node IDs 1 and 2 are associated with <receive> and <assign> nodes of the WS-BPEL in Fig. 4.

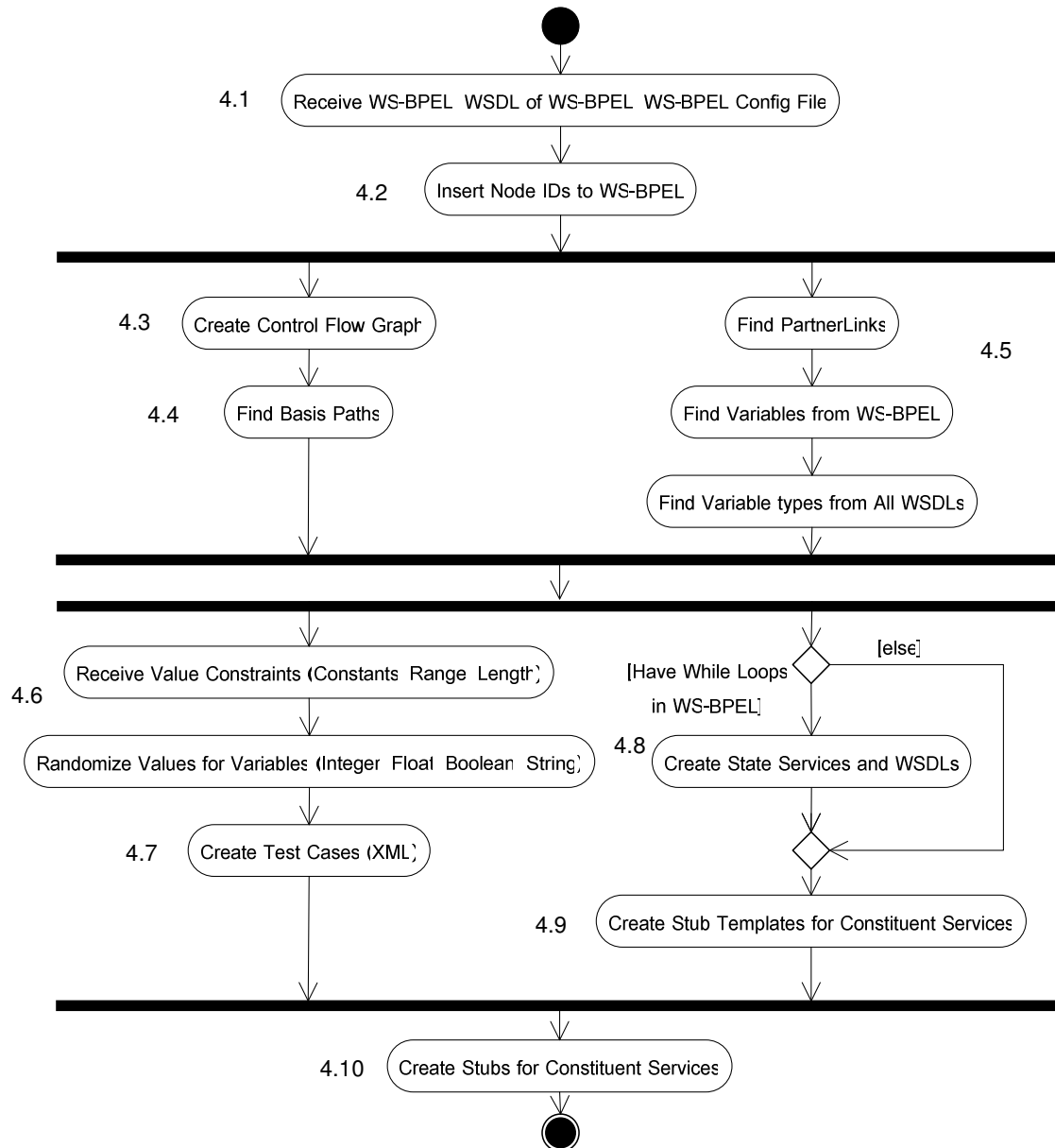


Fig. 2. Tasks in test suite generation

4.3 Create Control Flow Graph

The testing tool generates a CFG for the WS-BPEL with node IDs such as in Fig. 5. The complete CFG of the market place service is in Fig. 6.

4.4 Find Basis Paths

From the CFG, McCabe's cyclomatic complexity ($V(G)$) is computed to determine the number of basis paths which corresponds to the number of test cases needed for the flow. The $V(G)$ of the market place is computed by:

$$\begin{aligned}
 V(G) &= \text{no. of edges} - \text{no. of nodes} + 2 \\
 &= 15 - 14 + 2 = 3
 \end{aligned}$$

According to $V(G)$, there are 3 basis paths for this market place service, and therefore 3 test cases are needed to test these paths:

Path 1 : 1 – 2 – 3 – 4 – 5 – 6 – 7 – 13 – 14

Path 2 : 1 – 2 – 3 – 4 – 5 – 6 – 7 – 8 – 9 – 10 – 12 – 7 – 13 – 14

Path 3 : 1 – 2 – 3 – 4 – 5 – 6 – 7 – 8 – 9 – 11 – 12 – 7 – 13 – 14

The tool produces basis paths such as in Fig. 7.

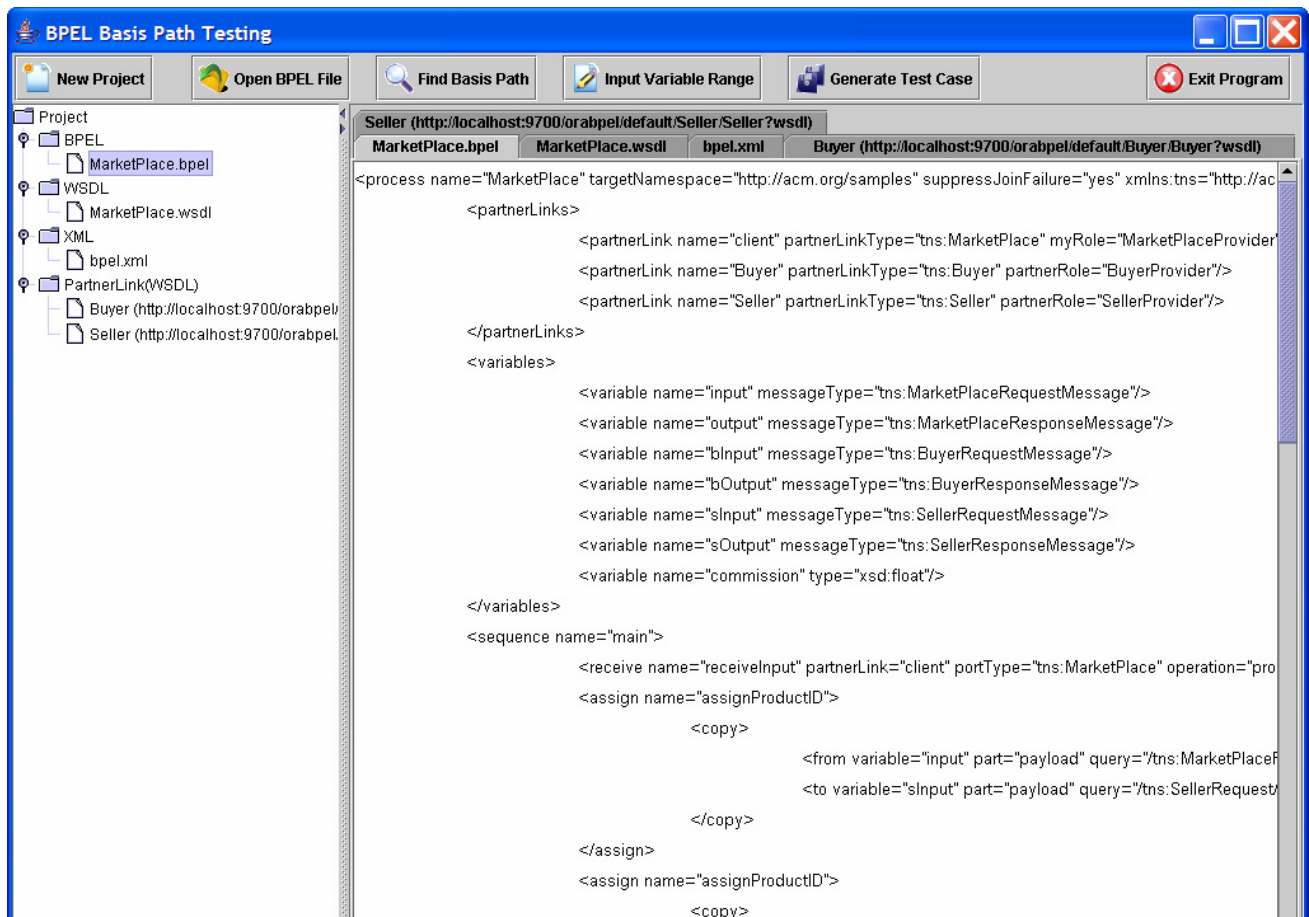


Fig. 3. Input WS-BPEL

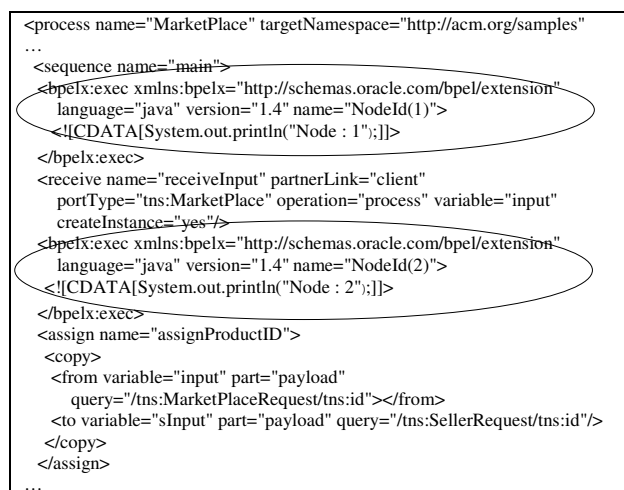


Fig. 4. WS-BPEL with embedded node IDs

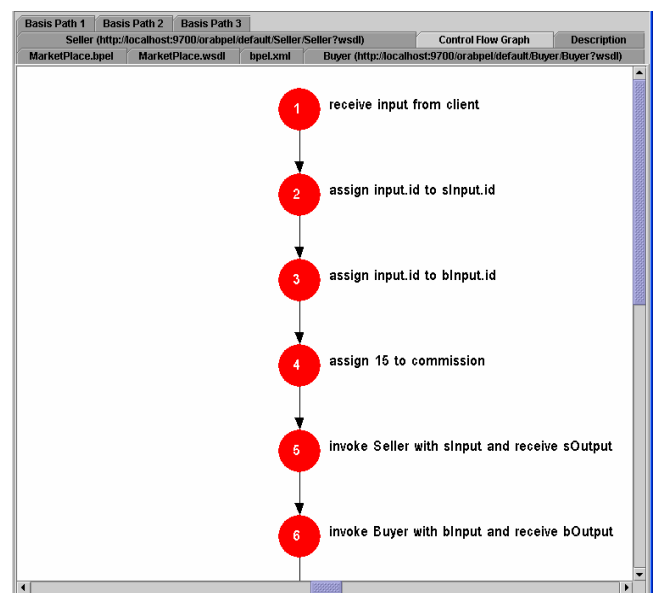


Fig. 5. CFG of market place produced by the tool

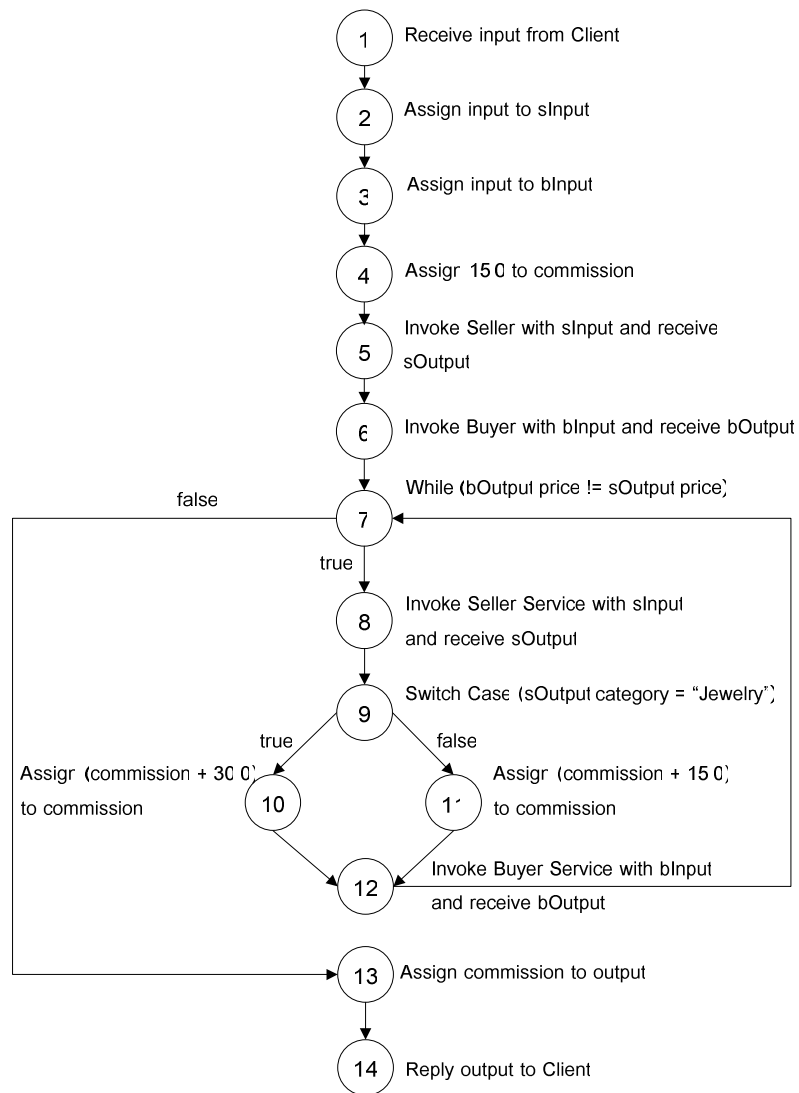


Fig. 6. Complete CFG of market place

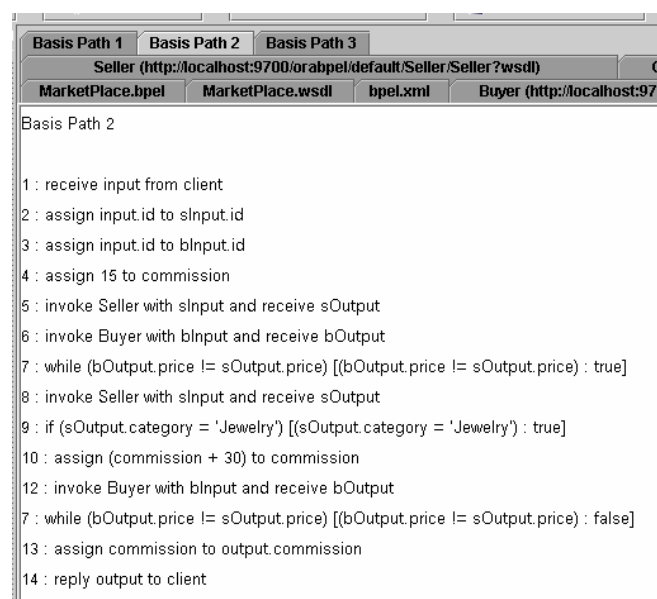


Fig. 7. Basis path 2 of market place

4.5 Find Variable Information

To create test data for the basis paths, the testing tool analyzes variables relevant to each path. The tool finds PartnerLinks from the uploaded configuration file in order to get to the WSDLs of constituent Web services and then determine input and output of these services. The WS-BPEL and its WSDL are also analyzed to determine input and output of the flow and other data that will be needed in the execution of each path.

4.6 Define Variable Values

From the previous step, some variables that determine branching of the path or are the result of service invocation will have their values generated automatically by the tool. On the other hand, those that are input or expected output will be constants and specified manually by the service tester. The tester specifies range (i.e. possible min and max values) and length of all relevant data values such as in Fig. 8.

Also, the tester specifies necessary constant values such as those for basis path 2 in Fig. 9. Here the tester specifies constants for the initial input product ID “2”, the product name “Product Name 2” to be returned by the Seller service, and the expected final commission ‘45.00’. Other data values are randomly generated according to their types. Supported data types are integer, float, boolean, and string. (Note that two sets of variable values will be defined for the Seller service because it would be invoked twice in path 2.)

4.7 Create Test Cases

According to the value constraints above, the testing tool generates variable values that have to be randomized (Table 2). These generated values and manually-specified constants form the test cases for the WS-BPEL. For example, test case 2 is for testing basis path 2 of section 4.4; it comprises data that are needed to execute the path. Given a constant product ID ‘2’ to the flow, the tool generates the output of the first invocation to the Seller service to return the category “Jewelry” and the price ‘40.80858’. The output of the first invocation to the Buyer service is also generated with the price ‘78.72712’. Since these two output prices do not match, the path would fall through the loop and the Seller and Buyer services would be re-invoked. For the second invocation, the tool generates the prices of equal value, i.e. 47.74675, so that the two prices would match and the loop would be exited. Finally the commission

charge would be returned. Other test cases trigger corresponding paths in the same way.

The testing tool represents the test cases for all basis paths in XML format as in Fig. 10.

4.8 Create State Services and WSDLs

When the WS-BPEL contains a loop as for the case of the market place, the testing tool additionally generates an auxiliary Web service called a BPEL state service, together with its WSDL, for each loop execution. The state service is used to determine the state of the visit to the loop by a relevant basis path. The path will be controlled to fall into the loop just one time on its first visit (i.e. the state service returns 1) and exits on the next visit (i.e. the state service returns 2). Fig. 11 describes the WSDL of the state service and Fig. 12 presents its Java implementation.

4.9 Create Stub Templates

The testing tool creates a template for each constituent Web service in the WS-BPEL. The template will be used for generating a stub for each constituent service which will be invoked during the test. For the market place service, the template for the stubs of the Seller and Buyer services is shown in Fig. 13. It is a WS-BPEL flow that merely receives the same input as the service and assigns certain data values from a particular test case as outputs. The placeholders that will be replaced by constants and generated data from a particular test case are shown in boldface type.

Table 2. Randomized values for all test cases

Test Case No. (Path No.)	Input			
	Variable Name	Service Name	Type	Value
1	sOutput.price	Seller	Float	6.9283094
	bOutput.price	Buyer	Float	6.9283094
2	sOutput.category	Seller	String	“Jewelry”*
				“Jewelry”**
	sOutput.price	Seller	Float	40.80858*
	bOutput.price	Buyer	Float	78.72712*
3				47.74675**
	sOutput.category	Seller	String	“u6GP5L8ydd”*
				“BShX34mreq”**
	sOutput.price	Seller	Float	60.880894*
	bOutput.price	Buyer	Float	87.276184**
				58.134274*
				87.276184**

* generated for 1st invocation ** generated for 2nd invocation

Expected Output (Basis Path 2)		Expected Output (Basis Path 3)				
Basis Path 2	Basis Path 3	Input Variable Range		Expected Output (Basis Path 1)		
Seller (http://localhost:9700/orabpel/default/Seller/Seller?wsdl)				Control Flow Graph	Description	Basis Path 1
MarketPlace.bpel	MarketPlace.wsdl	bpel.xml	Buyer (http://localhost:9700/orabpel/default/Buyer/Buyer?wsdl)			
Service Name	Variable Name	Variable Type	Min Value	Max Value	Length	
MarketPlace	input.id	string	0	0	10	
MarketPlace	output.price	float	0	100	0	
MarketPlace	commission	float	0	100	0	
Buyer	blInput.id	string	0	0	10	
Buyer	bOutput.price	float	0	100	0	
Seller	sInput.id	string	0	0	10	
Seller	sOutput.name	string	0	0	10	
Seller	sOutput.category	string	0	0	10	
Seller	sOutput.price	float	0	100	0	

Fig. 8. Specify range and length for data values

Expected Output (Basis Path 2)		Expected Output (Basis Path 3)			
Basis Path 2	Basis Path 3	Input Variable Range		Expected Output (Basis Path 1)	
Seller (http://localhost:9700/orabpel/default/Seller/Seller?wsdl)				Control Flow Graph	Description
MarketPlace.bpel	MarketPlace.wsdl	bpel.xml	Buyer (http://localhost:9700/orabpel/default/Buyer/Buyer?wsdl)	Basis Path 1	
Service Name	Variable Name		Variable Type	Value	
MarketPlace	input.id		string	2	
Seller	sOutput.name		string	Product Name 2	
Seller	sOutput.category		string	(Random)	
Seller	sOutput.price		float	(Random)	
Buyer	bOutput.price		float	(Random)	
Seller	sOutput.name		string	Product Name 2	
Seller	sOutput.category		string	(Random)	
Seller	sOutput.price		float	(Random)	
Buyer	bOutput.price		float	(Random)	
MarketPlace	Expected Output (output.commission)		float	45.00	

Fig. 9. Specify constants (input and expected output) for basis path 2 of market place

Expected Output (Basis Path 3)	All Test Case	Test Case 1	Test Case 2	Test Case 3	
Basis Path 2	Basis Path 3	Input Variable Range	Expected Output (Basis Path 1)	Expected Output (Basis Path 2)	
Seller (http://localhost:9700/orabpel/default/Seller/Seller?wsdl)			Control Flow Graph	Description	Basis Path 1
MarketPlace.bpel	MarketPlace.wsdl	bpel.xml	Buyer (http://localhost:9700/orabpel/default/Buyer/Buyer?wsdl)		

```
<testsuite service="MarketPlace">
  <testcase no="1">
    <input>
      </variable>
      <variable name="input" service="MarketPlace">
        <element name="id" type="string" value="1">
      </variable>
      <variable name="sOutput" service="Seller">
        <element name="name" type="string" value="Product Name 1">
        <element name="category" type="string" value="Sgk08JqT0o">
        <element name="commission" type="float" value="6.9283094">
      </variable>
      <variable name="bOutput" service="Buyer">
        <element name="commission" type="float" value="6.9283094">
      </variable>
    </input>
    <output>
      <variable name="output" service="MarketPlace">
        <element name="commission" type="float" value="15.00">
      </variable>
    </output>
  </testcase>
  <testcase no="2">
    <input>
      <variable name="input" service="MarketPlace">
```

Fig. 10. Generated test cases


```

<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
  targetNamespace=http://localhost:8080/axis/BPELStateService.jws
  ...
  <wsdl:message name="getStateIDResponse">
    <wsdl:part name="getStateIDReturn" type="xsd:int"/>
  </wsdl:message>
  <wsdl:message name="getStateIDRequest">
    <wsdl:part name="strFile" type="xsd:string"/>
  </wsdl:message>
  <wsdl:portType name="BPELStateService">
    <wsdl:operation name="getStateID" parameterOrder="strFile">
      <wsdl:input message="impl:getStateIDRequest"
        name="getStateIDRequest"/>
      <wsdl:output message="impl:getStateIDResponse"
        name="getStateIDResponse"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="BPELStateServiceSoapBinding"
    type="impl:BPELStateService">
    <wsdlsoap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="getStateID">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="getStateIDRequest">
        <wsdlsoap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://DefaultNamespace" use="encoded"/>
      </wsdl:input>
      <wsdl:output name="getStateIDResponse">
        <wsdlsoap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://localhost:8080/axis/BPELStateService.jws"
          use="encoded"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="BPELStateServiceService">
    <wsdl:port binding="impl:BPELStateServiceSoapBinding"
      name="BPELStateService">
      <wsdlsoap:address
        location="http://localhost:8080/axis/BPELStateService.jws"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>

```

Fig. 11. WSDL of state service

```

import java.io.*;
public class BPELStateService {
  public int getStateID(String strName) throws IOException {
    int stateID=1;
    String strFile = "C:/"+strName+".txt";
    File inFile = new File(strFile);
    if (inFile.exists()) {
      FileInputStream inFileStream = new FileInputStream(inFile);
      DataInputStream inDataStream = new DataInputStream(inFileStream);
      stateID = inDataStream.readInt();
      inDataStream.close();
      if (stateID >= 2) {
        stateID = 1;
      } else {
        stateID = 2;
      }
    }
    File outFile = new File(strFile);
    FileOutputStream outFileStream = new FileOutputStream(outFile);
    DataOutputStream outDataStream = new DataOutputStream(outFileStream);
    outDataStream.writeInt(stateID);
    outDataStream.close();
    return stateID;
  }
}

```

Fig. 12. Implementation of state service

```

<process name="_ServiceName_" targetNamespace=http://acm.org/samples
  ...
  xmlns:nsxml0="http://localhost:8080/axis/BPELStateService.jws">
  <partnerLinks>
    <partnerLink name="client" partnerLinkType="tns:_ServiceName_"
      myRole="_ServiceName_Provider"/>
    <partnerLink name="BPELStateService"
      partnerLinkType="nsxml0:BPELStateServiceLink"
      partnerRole="BPELStateServiceProvider"/>
  </partnerLinks>
  <variables>
    <variable name="input"
      messageType="tns:_ServiceName_RequestMessage"/>
    <variable name="output"
      messageType="tns:_ServiceName_ResponseMessage"/>
    <variable name="bpelInput" messageType="nsxml0:getStateIDRequest"/>
    <variable name="bpelOutput"
      messageType="nsxml0:getStateIDResponse"/>
  </variables>
  <sequence name="main">
    <receive name="receiveInput" partnerLink="client"
      portType="tns:_ServiceName_" operation="process" variable="input"
      createInstance="yes"/>
    <assign name="assignService">
      <copy>
        <from expression="_ServiceName_"></from>
        <to variable="bpelInput" part="strName"/>
      </copy>
    </assign>
    <invoke name="invokeBPELStateService"
      partnerLink="BPELStateService" portType="nsxml0:BPELStateService"
      operation="getStateID" inputVariable="bpelInput"
      outputVariable="bpelOutput"/>
    <switch name="checkState">
      <case condition="(bpws:getVariableData('bpelOutput','getStateIDReturn')
        = 1)">
        <sequence>_ReturnOutput1_</sequence>
      </case>
      <otherwise>
        <sequence>_ReturnOutput2_</sequence>
      </otherwise>
    </switch>
    <reply name="replyOutput" partnerLink="client"
      portType="tns:_ServiceName_" operation="process"
      variable="output"/>
  </sequence>
</process>

```

Fig. 13. Template for stubs of Seller and Buyer

4.10 Create Stubs

Test data are obtained from each test case to replace the placeholders in relevant templates to create service stubs for that test case. For test case 2 of the market place, the stubs for the Seller and Buyer services are created. Fig. 14-15 show the Seller stub and Buyer stub respectively.

5 Testing Procedure

When the test cases, stubs, and state service are all created, the test suite is ready to use for basis path testing. The service tester then follows the procedure depicted in Fig. 16. A test case is first selected and the tester deploys relevant service stubs, state service (if any), and the WS-BPEL with embedded node IDs. In our case, we deploy the WS-BPEL and stubs by using Oracle BPEL designer and executing them by Oracle BPEL Process Manager engine [15],

while the state service is an Axis Web service [16]. With the initial input, the WS-BPEL starts and lists the execution path as its control goes through node IDs. The tester then compares the listed path with the basis path given by the tool (such as Fig.7).

```
<process name="Seller" targetNamespace=http://acm.org/samples
...
xmlns:nsxml0="http://localhost:8080/axis/BPELStateService.jws">
<partnerLinks>
<partnerLink name="client" partnerLinkType="tns: Seller"
myRole="SellerProvider"/>
<partnerLink name="BPELStateService"
partnerLinkType="nsxml0:BPELStateServiceLink"
partnerRole="BPELStateServiceProvider"/>
</partnerLinks>
<variables>
<variable name="input"
messageType="tns: SellerRequestMessage"/>
<variable name="output"
messageType="tns: SellerResponseMessage"/>
<variable name="bpelInput" messageType="nsxml0:getStateIDRequest"/>
<variable name="bpelOutput"
messageType="nsxml0:getStateIDResponse"/>
</variables>
<sequence name="main">
<receive name="receiveInput" partnerLink="client"
portType="tns: Seller" operation="process" variable="input"
createInstance="yes"/>
<assign name="assignService">
<copy>
<from expression="Seller"></from>
<to variable="bpelInput" part="strName"/>
</copy>
</assign>
<invoke name="invokeBPELStateService"
partnerLink="BPELStateService" portType="nsxml0:BPELStateService"
operation="getStateID" inputVariable="bpelInput"
outputVariable="bpelOutput"/>
<switch name="checkState">
<case condition="(bpws:getVariableData('bpelOutput', 'getStateIDReturn')
= 1)">
<sequence>
<assign name="assignoutputname">
<copy>
<from expression="Product Name 2"></from>
<to variable="output" part="payload"
query="/tns: SellerResponse/tns: name"/>
</copy>
</assign>
<assign name="assignoutputcategory">
<copy>
<from expression="Jewelry"></from>
<to variable="output" part="payload"
query="/tns: SellerResponse/tns: category"/>
</copy>
</assign>
<assign name="assignoutputprice">
<copy>
<from expression="40.80858"></from>
<to variable="output" part="payload"
query="/tns: SellerResponse/tns: price"/>
</copy>
</assign>
</sequence>
</case>
<otherwise>
<sequence>
<assign name="assignoutputname">
<copy>
<from expression="Product Name 2"></from>
<to variable="output" part="payload"
query="/tns: SellerResponse/tns: name"/>
</copy>
</assign>
<assign name="assignoutputcategory">
<copy>
<from expression="Jewelry"></from>
```

Fig. 14. Stub of Seller service

```
<to variable="output" part="payload"
query="/tns: SellerResponse/tns: category"/>
</copy>
</assign>
<assign name="assignoutputprice">
<copy>
<from expression="47.74675"></from>
<to variable="output" part="payload"
query="/tns: SellerResponse/tns: price"/>
</copy>
</assign>
</sequence>
</otherwise>
</switch>
<reply name="replyOutput" partnerLink="client" portType="tns: Seller"
operation="process" variable="output"/>
</sequence>
</process>
```

Fig. 14. Stub of Seller service (continued)

```
<process name="Buyer" targetNamespace=http://acm.org/samples
...
xmlns:nsxml0="http://localhost:8080/axis/BPELStateService.jws">
<partnerLinks>
<partnerLink name="client" partnerLinkType="tns: Buyer"
myRole="BuyerProvider"/>
<partnerLink name="BPELStateService"
partnerLinkType="nsxml0:BPELStateServiceLink"
partnerRole="BPELStateServiceProvider"/>
</partnerLinks>
<variables>
<variable name="input"
messageType="tns: BuyerRequestMessage"/>
<variable name="output"
messageType="tns: BuyerResponseMessage"/>
<variable name="bpelInput" messageType="nsxml0:getStateIDRequest"/>
<variable name="bpelOutput"
messageType="nsxml0:getStateIDResponse"/>
</variables>
<sequence name="main">
<receive name="receiveInput" partnerLink="client"
portType="tns: Buyer" operation="process" variable="input"
createInstance="yes"/>
<assign name="assignService">
<copy>
<from expression="Buyer"></from>
<to variable="bpelInput" part="strName"/>
</copy>
</assign>
<invoke name="invokeBPELStateService"
partnerLink="BPELStateService" portType="nsxml0:BPELStateService"
operation="getStateID" inputVariable="bpelInput"
outputVariable="bpelOutput"/>
<switch name="checkState">
<case condition="(bpws:getVariableData('bpelOutput', 'getStateIDReturn')
= 1)">
<sequence>
<assign name="assignoutputprice">
<copy>
<from expression="78.72712"></from>
<to variable="output" part="payload"
query="/tns: BuyerResponse/tns: price"/>
</copy>
</assign>
</sequence>
</case>
<otherwise>
<sequence>
<assign name="assignoutputprice">
<copy>
<from expression="47.74675"></from>
<to variable="output" part="payload"
query="/tns: BuyerResponse/tns: price"/>
</copy>
</assign>
</sequence>
</otherwise>
</switch>
<reply name="replyOutput" partnerLink="client" portType="tns: Buyer"
operation="process" variable="output"/>
</sequence>
</process>
```

Fig. 15. Stub of Buyer service

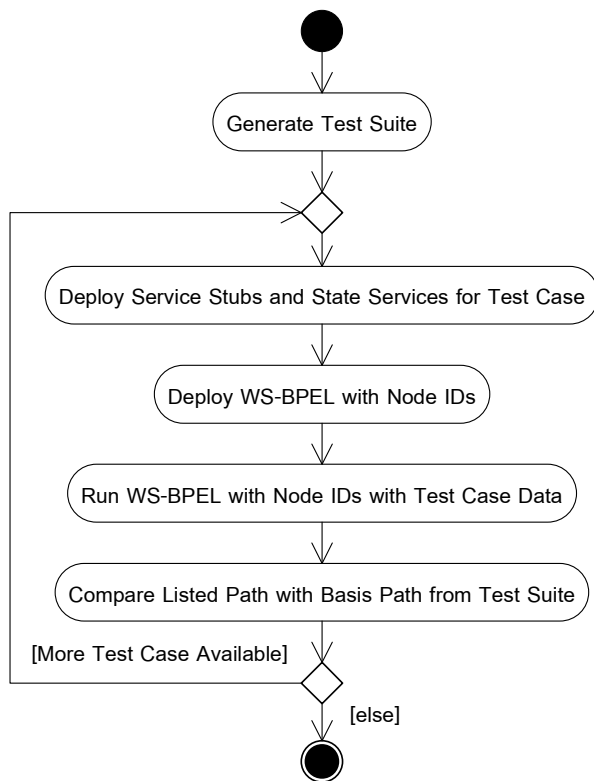


Fig. 16. Testing procedure

6 Implementation Details

The testing tool is implemented with Java (J2SDK 1.4.2_09). Its design is depicted in a class diagram in Fig 17. The classes are contained in three packages:

1. UI package comprises the following classes:

- Class MainProgram is the main class that calls class MainFrame.
- Class MainFrame builds the user menu for the tool and displays results. It controls the upload of all input files, creation of the control flow graph, discovery of basis paths, input and validation of variable information, generation of test cases, service template, service stubs, and state services. It extends Java's class JFrame.

2. Parser package comprises the following classes:

- Class InsertMarkNode analyzes the input WS-BPEL file to annotate WS-BPEL tags with node IDs according to Java embedding feature of Oracle BPEL designer.
- Class NodeParser extends Java's class SAXParser. It analyzes the WS-BPEL file with embedded node IDS from the class

InsertMarkNode in order to extract node IDs and details of the corresponding WS-BPEL tags, and determine the association between node IDs. This is for discovering the basis paths. The analyzed information is kept in the corresponding Node objects.

- Class PartnerParser extends Java's class SAXParser. It finds PartnerLinks of the WS-BPEL (i.e. links to WSDLs of constituent Web services). Since Oracle BPEL Designer also produces a configuration file that contains PartnerLinks to accompany with the WS-BPEL file, PartnerParser therefore analyzes this configuration file to determine locations of constituent Web services WSDLs.
- Class WSDLParser extends Java's class SAXParser. It analyzes all WSDLs (i.e. WSDLs of the WS-BPEL and constituent Web services) to extract information related to input/output variables of the WS-BPEL and the relevant Web service operations. This is for further generation of test cases. The extracted information is kept in the Node objects that correspond to <receive>, <reply>, and <invoke> tags of WS-BPEL.

3. Utility package comprises the following classes:

- Class Node maintains information of each node. The information includes node IDs, tag detail, node status (whether it is while or switch node), related variables and expressions, and associations with neighboring nodes. Each Node object is part of the NodeArray object.
- Class NodeArray maintains all Node objects of the flow.
- Class GraphDraw draws the control flow graph according to node information in the NodeArray object.
- Class Arrow draws arrow heads on the edges in the control flow graph. It is part of GraphDraw.
- Class BasisPath builds basis paths for the control flow graph. The information in the NodeArray object will be used to discover basis paths. The algorithm starts by identifying an associated node (i.e. the next node) for each current node. Table 3 lists all pairs of associated nodes for the control flow graph in Fig. 6. Then, for each node pair, paths leading to/from the current node are identified. For example, for the pair no.2, the path leading to node 2 is 1-2 and the path leading from node 2 is 3-4-5-6-7-13-14. These two paths join to form a complete path 1-2-3-4-5-6-7-13-14. Table 4 lists path details of all associated nodes pairs in Table 3. The algorithm then

removes duplicate paths and finally validates the resulting paths. That is, they are checked whether they start with the start node (node 1), end with the end node (node 14), and the path is valid to traverse. In our example, the resulting three paths as listed previously in Section 4.4 pass the validation and become the basis paths.

- Class `InputVariableRange` gets constants and constraints on data variables from the user (service tester). The input information is kept in `VarTable`.
- Class `VarTable` maintains a table of constants and constraints on data variables.
- Class `Expression` parses mathematical expressions associated with the nodes to extract relevant operators, operands, and any constant values in the expression. This is used when randomizing test case values.
- Class `GenerateData` calls the classes `GenerateTestCase`, `GenerateTemplate`, `GenerateStub`, and `GenerateStateService` to generate relevant XML documents.
- Class `GenerateTestCase` generates test cases in XML. It gets constants relevant to particular test cases from the class `VarTable` and uses the class `RandomData` to generate other necessary data values at random. Randomized data values have to obey their constraints and if they are part of any conditional expression, they have to satisfy the designated condition for such a test case. For

the case of basis path 2 of the market place (see Section 4.4), three randomized data values are involved: `sOutput.price`, `bOutput.price`, and `sOutput.Category` (see Fig. 9). The randomized values of `sOutput.price` and `bOutput.price` have to make the condition at node 7 of Fig. 6 “true” at the first invocation to the Seller and Buyer services. Therefore these two price values will be randomized until they can make the condition “true”. In our example as in Table 2, ‘40.80858’ is assigned to `sOutput.price` and ‘78.72712’ to `bOutput.price` at random. For `sOutput.Category`, “Jewelry” is assigned to make the condition at node 9 “true” for this basis path.

- Class `RandomData` randomizes data values for the test cases according to the constraints, i.e. data types and possible min and max values (for numeric data) or size of value (for string data).
- Class `GenerateTemplate` creates templates in WS-BPEL format for constituent Web services.
- Class `GenerateStub` creates sets of stubs in WS-BPEL format for constituent Web services; each set is for each test case.
- Class `GenerateStateService` creates a WSDL and Java code of a state service which will be called by a service stub that involves in a loop within the control flow graph. In our experiment, the state service is an Axis 1.1 Web service hosted by Apache Tomcat 5.0 application server.

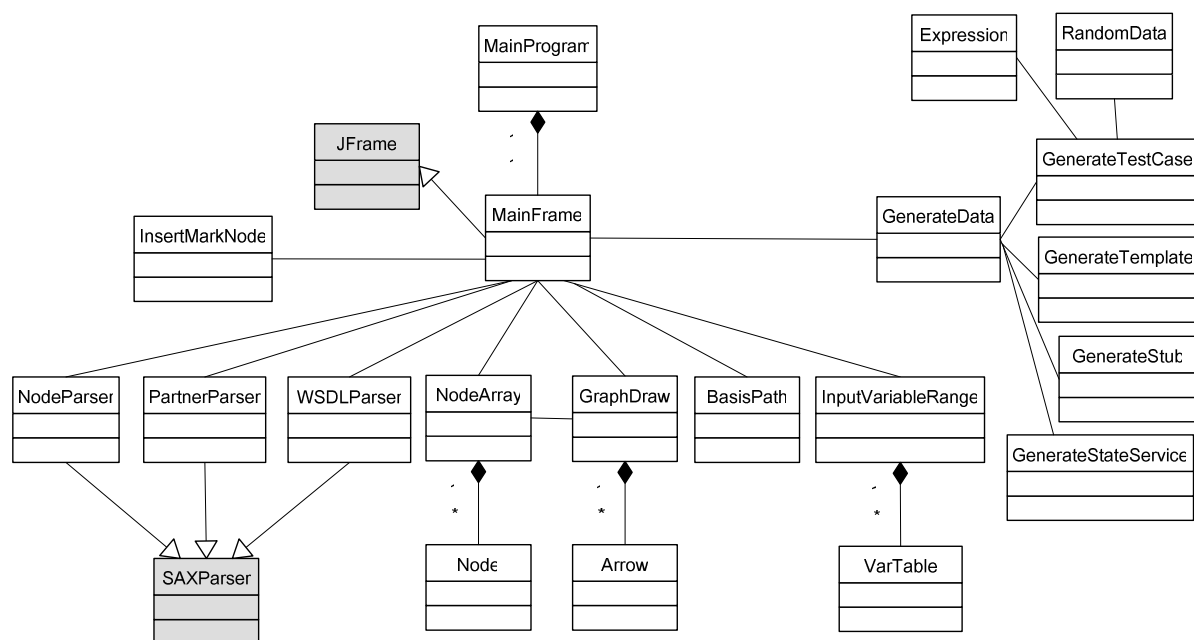


Fig. 17. Class diagram of testing tool

Table 3. Pairs of associated nodes for market place

Pair No.	Current Node	Next Node	Description
1	1	2	Start node
2	2	3	
3	3	4	
4	4	5	
5	5	6	
6	6	7	
7	7	13	In the case of false
8	7	8	In the case of true
9	8	9	
10	9	10	In the case of true
11	9	11	In the case of false
12	10	12	
13	11	12	
14	12	7	
15	13	14	
16	14	-	End node

Table 4. Path details of associated nodes pairs for market place

Pair No.	Path to	Path from	Complete Path
1	1	2-3-4-5-6-7-13-14	1-2-3-4-5-6-7-13-14
2	1-2	3-4-5-6-7-13-14	1-2-3-4-5-6-7-13-14
3	1-2-3	4-5-6-7-13-14	1-2-3-4-5-6-7-13-14
4	1-2-3-4	5-6-7-13-14	1-2-3-4-5-6-7-13-14
5	1-2-3-4-5	6-7-13-14	1-2-3-4-5-6-7-13-14
6	1-2-3-4-5-6	7-13-14	1-2-3-4-5-6-7-13-14
7	1-2-3-4-5-6-7	13-14	1-2-3-4-5-6-7-13-14
8	1-2-3-4-5-6-7	8-9-10-12-7-13-14	1-2-3-4-5-6-7-8-9-10-12-7-13-14
9	1-2-3-4-5-6-7-8	9-10-12-7-13-14	1-2-3-4-5-6-7-8-9-10-12-7-13-14
10	1-2-3-4-5-6-7-8-9	10-12-7-13-14	1-2-3-4-5-6-7-8-9-10-12-7-13-14
11	1-2-3-4-5-6-7-8-9	11-12-7-13-14	1-2-3-4-5-6-7-8-9-11-12-7-13-14
12	1-2-3-4-5-6-7-8-9-10	12-7-13-14	1-2-3-4-5-6-7-8-9-10-12-7-13-14
13	1-2-3-4-5-6-7-8-9-11	12-7-13-14	1-2-3-4-5-6-7-8-9-11-12-7-13-14
14	1-2-3-4-5-6-7-8-9-10-12	7-3-14	1-2-3-4-5-6-7-8-9-10-12-7-13-14
15	1-2-3-4-5-6-7-13	14	1-2-3-4-5-6-7-13-14
16	1-2-3-4-5-6-7-13-14	-	1-2-3-4-5-6-7-13-14

7 Discussion

We consider our approach to basis path testing of WS-BPEL simple and straightforward as it follows steps for basis path testing on programs in general. The supporting tool facilitates the testing procedure and requires little tester intervention. Although WS-BPEL and WSDL are supposed to be standards, in practice cross-vendor deployment of WS-BPEL processes and WSDLs is not yet well-supported. That is, they are specific to the environments and tools that create them. Our testing tool may therefore be considered an add-on to Oracle's BPEL development environment but the approach can be

followed to develop a basis path test suite for other platforms.

The tool is currently at a prototype stage as it does not yet support all XML schema data types in the generation of test data; only integer, float, boolean, and string are supported. Also, only sequence, condition, and repetition patterns of control are allowed. Nonetheless, these represent a set of data types and workflow constructs that are typically used for setting up business workflows. Since there is correspondence between parallel and sequence patterns [17], it may be possible to transform a WS-BPEL with parallel control into one

with sequence. The tool does not consider infeasible paths that cannot be accessed either.

8 Conclusion

The contribution of this paper is an approach to the generation of basis path test suite and testing procedure for basis path testing on WS-BPEL services. The procedure is assisted by a testing tool which can automatically generate test cases, stubs of constituent services, and state services. As with other testing tool, our tool requires small intervention from service testers to specify constraints on those data values to be generated for test cases. Through this testing, service composers can be assured of the composition flow before deploying WS-BPELs.

We expect to enhance the tool to support WS-BPELs and WSDLs from different vendors. Shortcomings as discussed in the previous section will be dealt with in the next version of the tool. It is also possible to extend the approach to accommodate other kinds of structural testing, e.g. branch coverage and path coverage testing.

Acknowledgment:

This research is part of the Engineering New Paradigm Software for Enterprises with Service-Oriented Architecture Project, supported by Thailand's Software Industry Promotion Agency (Public Organization).

References:

- [1] OASIS, WS-BPEL, <http://www.oasis-open.org/committees/wsbpel>
- [2] P. C. Jorgenson, *Software Testing: A Craftman's Approach*, 2nd edition, CRC Press, 2002.
- [3] T. J. McCabe, A Complexity Metric, *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, December 1976, pp. 308-320.
- [4] W. T. Tsai, R. Paul, Y. Wang, C. Fan, and D. Wang, Extending WSDL to Facilitate Web Services Testing, *Proceedings of 7th IEEE International Symposium on High Assurance System Engineering (HASE'02)*, 2002.
- [5] T. T. Cheng and C. H. Fu, On the Development of Software Tools for Testing Web Services, *Proceedings of International Conference on Internet Computing*, 2004.
- [6] W. T. Tsai, R. Paul, W. Song, and Z. Cao, Coyote: An XML-Based Framework for Web Service Testing, *Proceedings of 7th IEEE International Symposium on High Assurance System Engineering (HASE'02)*, 2002.
- [7] E. Martin, S. Basu, and T. Xie, Automated Testing and Response Analysis of Web Services, *Proceedings of 2007 IEEE International Conference on Web Services (ICWS 2007)*, Salt Lake City, Utah, 9-13 July 2007.
- [8] Parasoft, SOAtest, <http://www.parasoft.com/soatest>
- [9] Optimyz Software, WebServiceTester, <http://www.optimyz.com>
- [10] Progress Software Corporation, Stylus Studio, http://www.stylusstudio.com/ws_tester.html
- [11] Borland, SilkPerformer, <http://www.borland.com/us/products/silk/silkperformer/index.html>
- [12] Mindreef, SOAPscope, <http://home.mindreef.com/products/soapscope-server/products.html>
- [13] Eviware, soapUI, <http://www.soapui.org/>
- [14] PushToTest, TestMaker, <http://www.pushtotest.com/>
- [15] Oracle, Oracle BPEL Designer and Process Manager, <http://www.oracle.com>
- [16] Apache, Web Services – Axis, <http://ws.apache.org/axis/>
- [17] M. Klusch, A. Gerber, and M. Schmidt, Semantic Web Service Composition Planning with OWLS-Xplan, *Proceedings of 1st International AAAI Fall Symposium on Agents and the Semantic Web*, Arlington, VA, USA, 2005.