修士学位論文

題目

Design and Implementation of Test Case Generation Tool for BPEL Unit Testing

(BPEL 単体テストのためのテストケース生成手法の提案と実現)

指導教員
井上 克郎 教授

報告者
Choy Kho Yee

平成 20 年 2 月 8 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻 ソフトウェア工学講座

平成 19 年度 修士学位論文

Design and Implementation of Test Case Generation Tool for BPEL Unit Testing
BPEL 単体テストのためのテストケース生成手法の提案と実現

Choy Kho Yee

Abstract

In order to create test cases for the unit testing of business process written in Web Services Business Process Execution Language (WS-BPEL or BPEL), developers have to prepare input data for the BPEL process under test (PUT) and corresponding verification conditions for output data from the PUT. This preparation of test-related data can be a tedious task due to the complexity of XML data used by the PUT. Furthermore, it is difficult for the developers to decide whether the created test cases are sufficient for testing the PUT.

In this thesis, the design and implementation of a test case generation tool employing a data dependency based approach is presented. In this approach, developers first define data dependencies using XPath expression. Type definitions in WSDL documents are then leveraged to automatically generate independent data which, together with the specified data dependencies, are then used to generate coherent input data and corresponding verification conditions. Finally, test cases are composed using these data. Besides, a platform independent method to collect execution information of the PUT is also presented. This can provide developers useful information for evaluating the adequacy of generated test cases. Experiments were carried out to verify that this tool indeed helps in the creation of test cases for BPEL unit testing.

Keywords

Unit testing
Test case generation
WS-BPEL
Data dependency

Contents

List of Figures

List of Tables

1 Introduction

Web Services Business Process Execution Language (WS-BPEL or BPEL) [18] is an XML-based language designed to compose Web services [6] in order to implement business processes. With support from many big companies such as IBM, Microsoft and BEA which have involved in its standardization work, BPEL is expected to be the de-facto standard in Web service composition and thus the core technology in realizing Service-Oriented Architecture (SOA).

As more and more Web service compositions are created with BPEL, it is important to ensure that the compositions function as expected. Unit testing has been known as an efficient way in program testing [15]. Therefore, it is believed to be effective in testing BPEL processes. Unit testing frameworks for BPEL has been proposed by Li et al. [26] and Mayer et al. [20]. While Li et al. provided some initial ideas on BPEL unit testing framework design and implementation, Mayer et al. went one step further by implementing a concrete framework, BPELUnit [5] and licensed it as an open source software.

Nevertheless, BPELUnit does not provide much support in test case creation and the monitoring of the process under test (PUT). As stated in [11], there are four phases in software testing: (1) modeling the software's environment, (2) selecting test scenarios, (3) running and evaluating test scenarios, and (4) measuring testing progress. While BPELUnit contributes to the first and the third phases, the other two are still left for further studies. In this thesis, the design and implementation of a test case generation tool employing a data dependency based approach is presented. This tool contributes to the second phase. At the same time, a method to collect execution information of the PUT using only standard BPEL functions is also proposed and implemented to facilitate the last phase.

To create a test case in BPELUnit, developers first need to specify a set of relevant operations which are expected to be invoked by the PUT in the test. Then, for this set of operations, they have to prepare input data for the PUT and corresponding verification conditions for output data from the PUT. While the first step is not very difficult, the preparation of these test-related data while maintaining data coherency can be a tedious task due to the complexity of XML data used in the communications between the PUT and multiple Web services it interacts with.

A data dependency based approach is proposed in this thesis to address this problem. In order to achieve data coherency, developers are required to describe the data dependencies in XML Path Language (XPath) [12]. In the actual generation of input data for the PUT, the data type definitions specified in the Web Services Description Language (WSDL) [9] documents of the PUT and its partner Web services are utilised. Based on the type definitions, independent data that do not depend on other data are automatically generated. Combining the data dependencies specified by developers in the first step and the automatically generated independent data, sets of coherent input data and corresponding verification conditions can be generated. Test cases are then composed automatically using these data.

On the other hand, test coverage such as statement coverage, branch coverage and path coverage are often used as a base measure of testing progress [11]. In order to obtain test coverage information, the execution information of the PUT must be available. Therefore, a method to capture execution information of the PUT using only standard BPEL functions is also proposed and implemented. In this method, standard BPEL activities which invoke an external logging service are weaved into the PUT. Logged information helps developers in determining the adequacy of generated test cases.

In this thesis, the design and implementation of a tool based on the above approaches are presented. The result of a two-stage evaluation process is presented as well. In the first stage, we carried out a case study to show that the implemented tool can actually be used in generating test cases for BPEL processes. Then, an experiment was conducted to compare this tool with the BPELUnit TestSuite Editor [5, 21], which is currently the only tool available to create BPELUnit test cases.

The rest of this thesis is organized as follows: Section 2 further explains the related technologies of this research. Section 4 covers the design and implementation of the test case generation tool based on the proposed approach. Section 5 presents the evaluation process and the results. Section 6 states the stance of this research with regard to other related work. Finally, Section 7 concludes the thesis with conclusion and future work.

2 Background Technologies

This section explains on the technologies involved in this research. First, we start by introducing the fundamental ideas such as service oriented architecture and Web service. Then, BPEL is introduced through an example BPEL process which we created from scratch using ActiveBPEL Designer [1]. This process will be used throughout this thesis as example and later in a case study conducted to show the usefulness of our proposed methods. Finally, the BPELUnit framework which is the sole unit testing framework for BPEL processes is introduced.

2.1 SOA and Web Service

The Service-Oriented Architecture (SOA) is an architectural paradigm which has gained great popularity in recent years. It is the latest approach to building, integrating and maintaining complex enterprise software systems [21]. According to [2, 17, 22], the basic building blocks in SOA are *services*, which are loosely coupled and mostly distributed. The interface of a service is described in an abstract interface language and can be invoked without knowledge of the underlying implementation. Services can be dynamically discovered and used. Furthermore, SOA supports integration, or composition, of services.

Web service is widely used in implementing SOA. According to the definition of World Wide Web Consortium, *a Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.* [6]

A WSDL document is an XML document which defines the interface of a Web service. It describes the operations provided by the Web service and the structure and types of XML data those operations receive and return. Type information of these data is often defined using XML Schema [19].

XML Schema is used to define data structures and datatypes of elements in XML documents. There are simple datatypes and complex datatypes in XML Schema. Simple datatypes represent scalar data while complex datatypes are built up by simple datatypes to represent more complex structured data. XML Schema provides 19 built-in primitive datatypes and 25 built-in derived datatypes. Developers may use these pre-defined simple datatypes and derive new datatypes by applying constraints called facets. For example, ISBN datatype can be derived from the string datatype by limiting the length of the string to exactly 10 characters in the following way:

```
<simpleType name="isbn">
    <restriction base="string">
        <length value="10" />
    </restriction>
</simpleType>
```

The above definition of ISBN datatype constrains data with this type to have an exact length of 10 characters. However, applicable facets depend on the base datatypes used in the derivation. A sample XML Schema document can be found in the appendix. This document is part of the implementation of our proposed tool.

2.2 BPEL

The Web Services Business Process Execution Language (WS-BPEL or BPEL) is an XML-based language for Web service composition. Its major strength is the simplicity to orchestrate Web services to deliver a combined service. BPEL uses existing XML specifications such as WSDL, XPath and XML Schema. It uses definitions in WSDL documents for composition, offers the composition as WSDL Web service, and allows handling of XML Schema based data with XPath expressions. We use the following terms in this thesis:

- A *BPEL process* is a process written in BPEL.
- A *partner Web service* or simply partner of the process is Web service that the process interacts with.
- A *client* is an application that invokes the BPEL process to use the service it provides.

Although the specification of BPEL does not consider graphical notation of a BPEL process, graphical editors such as ActiveBPEL Designer and NetBeans [16] are available. Figure 1 shows an example BPEL process, the BPEL Travel process, created using ActiveBPEL Designer. This process implements a tour package search service which returns additional information about the destination of choice together with the package results. In particular, it queries services provided by a travel agency and those provided by Amazon [3], a geographical information query service, an image search service and a currency exchange rate query service. However, all services are imaginary except that provided by Amazon.

First, the BPEL Travel process accepts search query from a client, the input message includes information like date and destination. The process then relay the query to other Web services parallelly in AmazonScope, TravelScope and GeoInfoScope. Books related

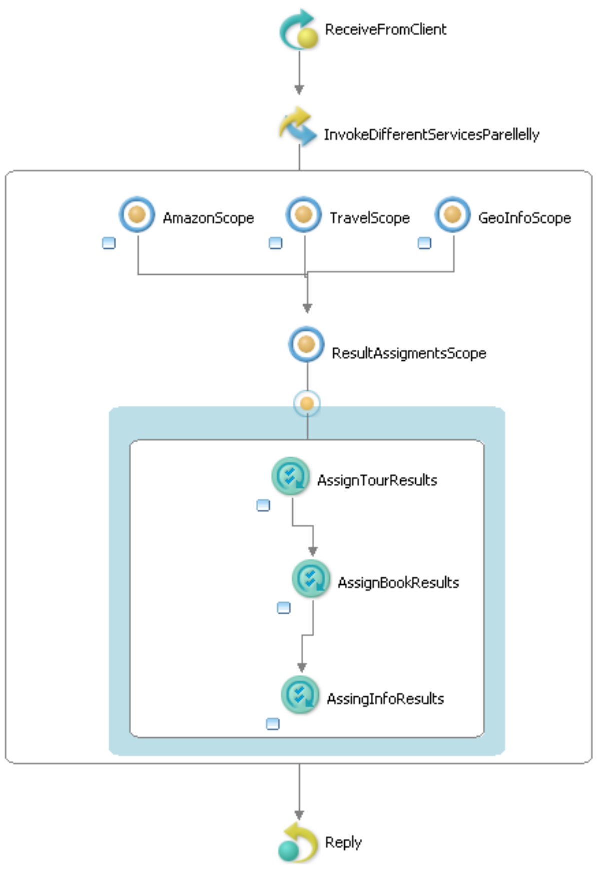


Figure 1: Flow of BPEL Travel Process

to the destination are returned by the Amazon Web service in the AmazonScope. In TravelScope, both services provided by the travel agent and the image search service are invoked. From the travel service, it gets tour package information and from the image search service, it gets images related to the destination search string. Then, geographical information like temperature and currency used are returned by the geographical information query service in GeoInfoScope.

These results are then assigned to the output message of the BPEL Travel process in the AssignTourResults, AssignBookResults and AssignInfoResults loops. In AssignInfoResults loop, the currency exchange rate query service is invoked as many times as the number of results returned by the geographical information query service, to convert one Japanese yen to the currency used in the locations returned by the geographical information query service.

A BPEL process is built up by basic activities and structured activities. In the above example, we have used four basic activities: receiving query from client (`receive`), invok-

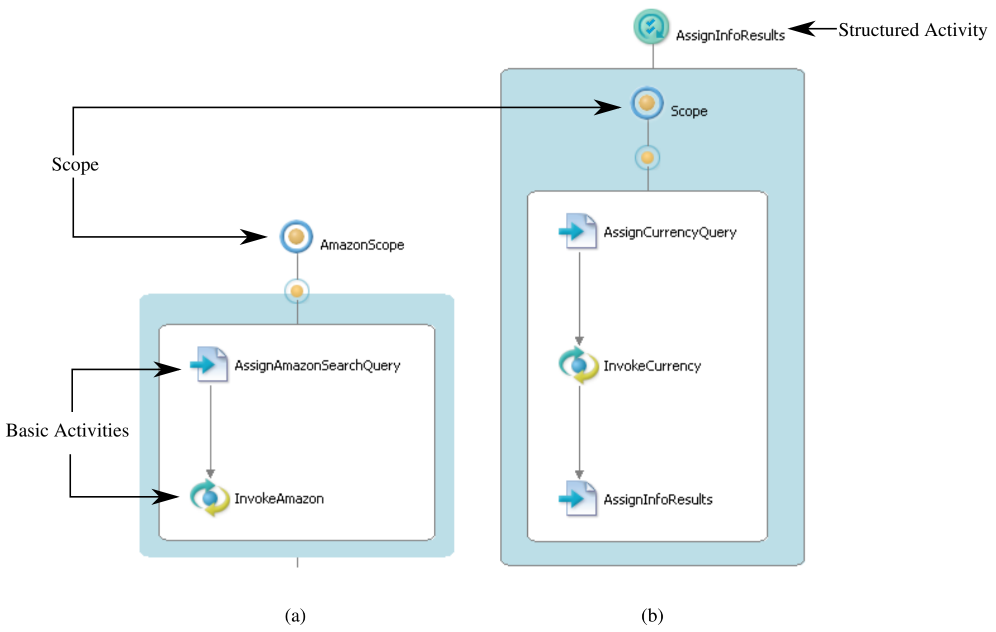


Figure 2: Types of Activity in BPEL

ing partner Web services (`invoke`), assigning appropriate values to messages (`assign`) and returning results to the client (`reply`). These basic activities serve as functions with single purposes. On the other hand, structured activities such as `sequence`, `if`, `while`, `forEach`, etc. are used to express more complex controls on how basic activities are executed. From the above example, parallel flow of activities (`flow`) and loop (`forEach`) are examples of structured activities.

Figure 2 shows more detailed examples of these activities in the BPEL Travel process. Figure 2 (a) shows the inside of AmazonScope. A scope is a container with which we can use to hold related activities inside it. As we can see, there are two basic activities, AssignAmazonSearchQuery and InvokeAmazon, in the scope. Activities in BPEL can have their own names. AssignAmazonSearchQuery is an `assign` activity, which assigns appropriate values to the request message of the service provided by Amazon. On the other hand, InvokeAmazon is called to actually send the request to Amazon. Then, Figure 2 (b) shows the inside of AssignInfoResults, which is a `forEach` structured activity. Inside it, there is one scope which holds three basic activities.

BPEL manipulates data with the `assign` activity, using values returned from query and expression language. Query language is used to retrieve a node or its data from XML data while expression language is used to return a value as a result of an expression. The default query and expression language adopted in BPEL is the XML Path Language (XPath).

XPath is used to find nodes in an XML document and retrieve useful information from the nodes. In XPath, location paths are used to locate nodes inside a XML tree structure. They are a subset of a more general concept called XPath expressions. Instead of just finding nodes, more operations can be done with XPath expressions, such as number calculations, string manipulation, etc. For example, with regard to the following XML fragment, which is the input message for the BPEL Travel process:

```
<query>
    <date>2007-11-13</date>
    <destination>Tokyo</destination>
    <periodOfDays>10</periodOfDays>
    <cost>200000</cost>
    <numOfTravellers>1</numOfTravellers>
    <numResults>20</numResults>
</query>
```

the location path `/query/destination/text()` returns the text node in `<destination>`, whose value is "Tokyo", while the XPath expression `count(/query/node())` counts the number of elements inside `<query>`, which evaluates to 6.

The following chunk of XML code is the syntax of BPEL for substituting an XML node with the return value of an XPath expression. This copies the value from the `destination` node of a variable called `requestMessage` to the node specified by `Request/Title` in the variable named `AmazonItemSearchRequestMsg`.

```
<assign name="AssignAmazonSearchQuery">
    <copy>
        <from part="body" variable="requestMessage">
            <query>destination</query>
        </from>
        <to part="body" variable="AmazonItemSearchRequestMsg">
            <query>Request/Title</query>
        </to>
    </copy>
</assign>
```

2.3 BPEL Unit Testing

In the BPEL unit testing framework proposed by Mayer et al. [20], the PUT is isolated from the real world. The PUT is actually deployed on server while the client and other partner Web services are simulated by the framework as client track and partner tracks

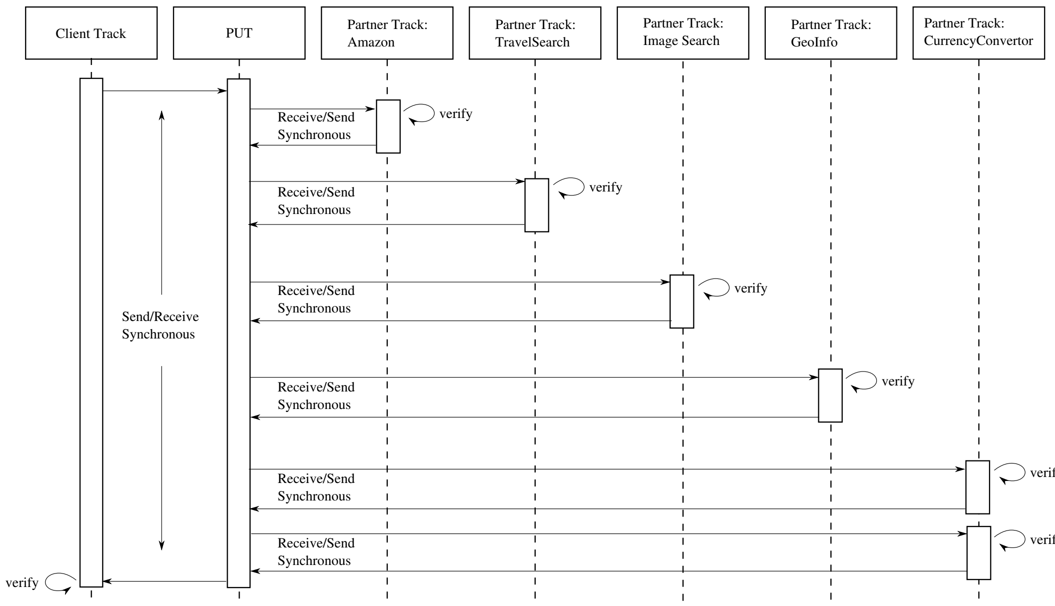


Figure 3: BPEL Unit Testing of BPEL Travel Process

respectively. All tracks are run as independent processes simultaneously when the test starts.

In these tracks, activities can be added. Each activity corresponds to an invocation of an operation in that track. However, in the client track, an activity corresponds to the operation provided by the PUT. There are six types of activities, i.e. "Send Asynchronous", "Receive Asynchronous", "Send/Receive Synchronous", "Receive/Send Synchronous", "Send/Receive Asynchronous" and "Receive/Send Asynchronous". These activities are different from the order and direction of the message flow, and whether there is synchronization involved. "Send Asynchronous" and "Receive Asynchronous" activities correspond to one-way interactions while the two synchronous activities are used in two-way synchronous interactions. The asynchronous pairs of receive and send are actually a pair of one-way operations. Figure 3 shows the examples of synchronous activities.

The proposed framework was implemented as BPELUnit, which is licensed as an open source software. Figure 3 shows a test process in BPELUnit. In this example, the PUT is the BPEL Travel process introduced in the previous section.

The creation of test case in BPELUnit involves two steps. Developers first select a set of involved operations in the test case and add them to corresponding tracks as activities. A track can have more than one activity as shown in the CurrencyConverter partner track in Figure 3. This means the operation will be invoked more than once. For each of these activities, developers then prepare the XML data that are to be sent to the PUT and verification conditions used to verify the data received from the PUT during the test.

Here, a verification condition is a pair made up of an XPath expression to be executed inside the received data and a value which the result of the XPath expression must match. Below we further elaborate on the preparation of test case using the BPEL Travel process as example.

We would like to test the correctness of the PUT when the TravelSearch partner returns zero result. First, we define the involved operations in the test case. Since there are no conditional branching in this sample process, all operations are selected and added as activities to each track. Then, XML data to be sent to the PUT during the test have to be defined for each activity. Below is the example "Send/Receive Synchronous" activity added into the client track. In the example, we order the client to send the XML data inside the `<data>` element to the PUT.

```
<clientTrack>
    <sendReceive service="put:BpelTravelService"
                 port="BpelTravelServicePort" operation="request">
        <send>
            <data>
                <query>
                    <date>2007-11-13</date>
                    <destination>Tokyo</destination>
                    <periodOfDays>10</periodOfDays>
                    <cost>200000</cost>
                    <numOfTravellers>1</numOfTravellers>
                    <numResults>20</numResults>
                </query>
            </data>
        </send>
        <receive />
    </sendReceive>
</clientTrack>
```

Finally, we define verification conditions used to verify the data received from the PUT during the test. For example, the PUT is supposed to send the following XML data to the TravelSearch service, corresponding to the request from the client above.

```
<query>
    <appid>TravelKey-ID1234</appid>
    <startDate>2007-11-13</startDate>
    <searchKey>Tokyo</searchKey>
</query>
```

In order to verify that the `<startDate>` and `<searchKey>` are properly set in the PUT and sent to the TravelSearch service, corresponding conditions can be defined in the activity in TravelSearch partner track.

```
<partnerTrack name="TravelSearch">
    <receiveSend service="trv:DummyTravelSearchService"
             port="DummyTravelSearchServicePort" operation="request">
        <send>
            <data />
        </send>
        <receive>
            <condition>
                <expression>startDate</expression>
                <value>'2007-11-13'</value>
            </condition>
            <condition>
                <expression>searchKey</expression>
                <value>'Tokyo'</value>
            </condition>
        </receive>
    </receiveSend>
</partnerTrack>
```

After preparing the test case, it can be run inside the BPELUnit framework. At the beginning of the test, the simulated client initiates the test by sending the prepared data to the PUT. The PUT then invokes the operations of its simulated partners as necessary along the test. These partners receive data from the PUT and verify them according to the given verification conditions. If everything is fine, the invoked partners return the prepared data to the PUT for further processing. If an error occurs along the way, the test is terminated immediately and the error is reported to the tester.

3 Approach to Support Test Case Generation

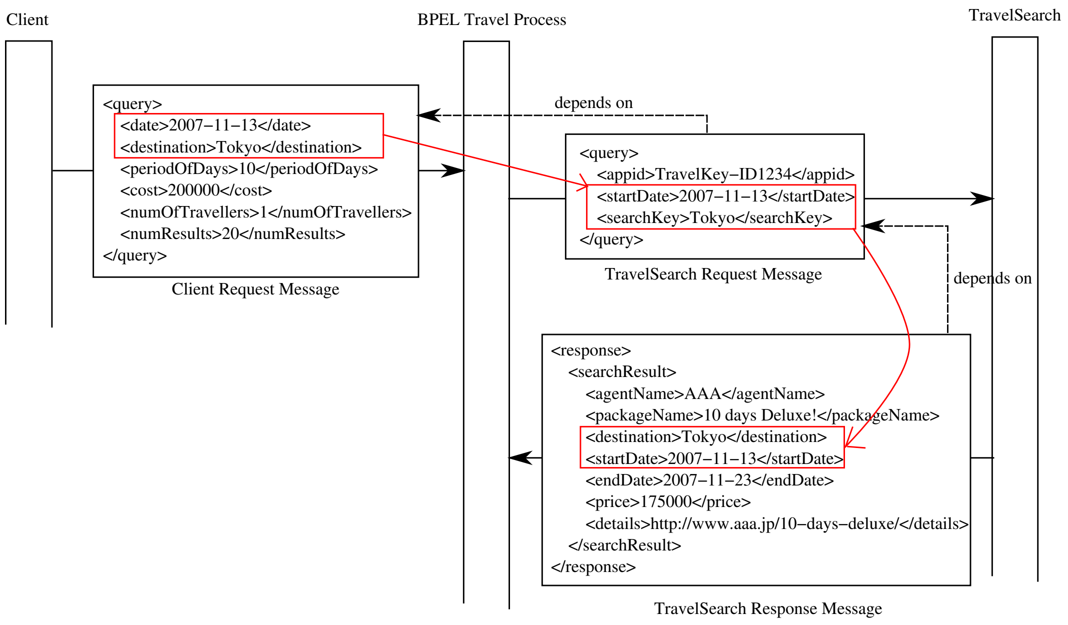
The main purpose of this research is to design and implement a tool to support BPEL developers in creating test cases for BPELUnit. We achieve this goal by proposing (1) a data dependency based method to ease the creation of coherent input data for the PUT and verification conditions in BPELUnit, and (2) a platform independent way of capturing execution information of the PUT which can help developers decide whether the created test cases are sufficient for testing the PUT. The following sections present the proposed approach in detail.

3.1 Data Dependency Based Test Case Generation

Data dependencies exist amongst the input and output data of the PUT. Figure 4 shows an example of data dependency that exists in BPEL Travel process which was used as example in Section 2.2. In the example, data are simply copied from message to message. As in the case of this example, we believe that these data dependencies are simple because BPEL only provides minimum functions needed to perform simple data manipulation necessary in defining business processes [10]. We aim to deploy this simplicity of data dependencies in test case generation.

We propose a test case generation method based on data dependencies. In this proposed method, developers first need to understand the specification of the PUT. This can be safely assumed as BPELUnit is designed for white box testing of the PUT. Then, developers select sets of relevant operations in the test case. For each operation set, developers list out data dependencies of its input and output data. A simple presentation of the data dependencies that exist in BPEL Travel process is shown in Figure 5. Note that the number of times an operation is invoked might depend on data, too. This dependency is shown as an arrow that leads from CurrencyExchage to data number ten in Figure 5. In order to process these data dependencies, they must be specified in a machine-processable format. We found XPath convenient in the specification of data dependency since it is the default query and expression language in BPEL.

Test-related data are then generated based on the data dependencies specified earlier. For the example shown in Figure 4, independent data in the client request message are first generated randomly. Verification conditions can then be added automatically to the corresponding activity in the TravelSearch track, to verify that the dependent data indeed have the same values as the data they depend on. These test-related data are later used to generate test case. Dependencies which indicate the number of times an operation is invoked are used in generating the appropriate number of activities in a track.



<query>
  <date>2007–11–13</date>
  <destination>Tokyo</destination>
  <periodOfDays>10</periodOfDays>
  <cost>200000</cost>
  <numOfTravellers>1</numOfTravellers>
  <numResults>20</numResults>
</query>

<query>
  <appid>TravelKey–ID1234</appid>
  <startDate>2007–11–13</startDate>
  <searchKey>Tokyo</searchKey>
</query>

<response>
  <searchResult>
    <agentName>AAA</agentName>
    <packageName>10 days Deluxe!</packageName>
    <destination>Tokyo</destination>
    <startDate>2007–11–13</startDate>
    <endDate>2007–11–23</endDate>
    <price>175000</price>
    <details>http://www.aaa.jp/10–days–deluxe/</details>
  </searchResult>
</response>

Figure 4: Example of Data Dependency

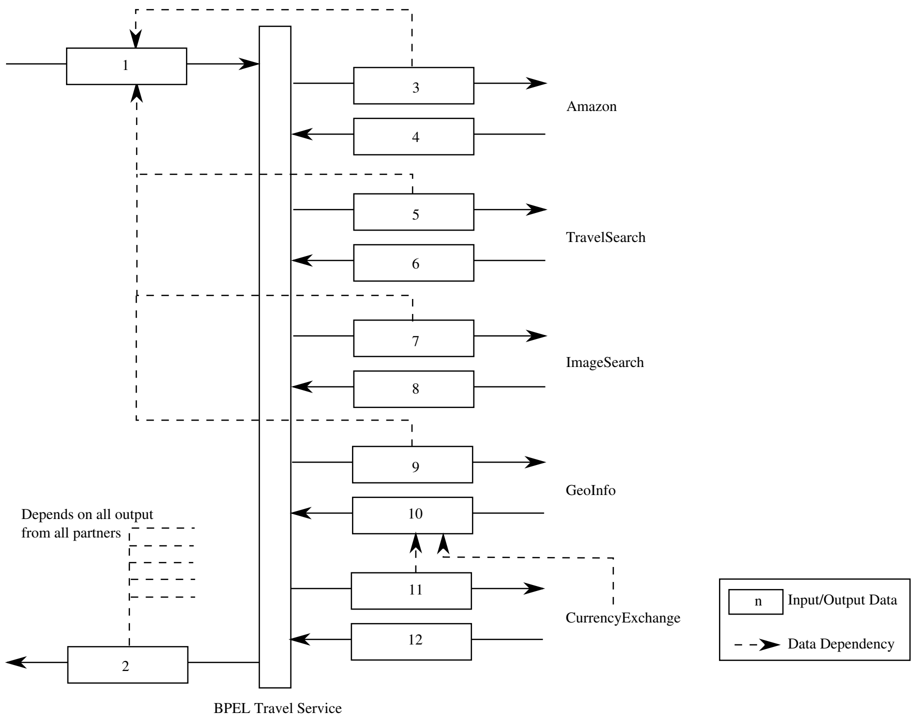


Figure 5: Data Dependencies in BPEL Travel Process

3.1.1 Data Dependency Specification

We chose to use XPath as the language to specify data dependencies in our work. Location path is used to specify the dependent node in an XML data while XPath expression is used to return the value this dependent node should have. An example is shown below. This example shows the pair of XPath expressions to specify the dependency between the TravelSearch request message and the client request message in Figure 4.

```
Dependent node: (TravelSearch Request Message) query/startDate
Value:          (Client Request Message) query/date
```

Note that `(TravelSearch Request Message)` and `(Client Request Message)` are inserted to clarify which message the XPath expressions operates on and not part of the XPath expressions.

Besides data-to-data dependency, there is also dependency of the number of invocations of an operation on data, such as in the case in which an operation have to be invoked as many times as there are search results. In this case, the dependency consists of an identifier of the operation and the XPath expression which specify the number of times it is invoked.

3.1.2 Test-related Data and Test Case Generation

Data are divided into four categories according to their direction (in or out) with regard to the PUT and their dependencies on other data. The characteristics of each category together with the way they are handled are presented below.

Independent Input Data

Independent input data are input data to the PUT which do not rely on other data, thus can be generated freely. To generate a set of coherent input data, independent input data should first be generated. An example of independent input data is the client request message shown in Figure 4.

A simple yet effective approach is used in this research to generate independent input data. Generators corresponding to each built-in primitive datatype in XML Schema are first prepared. These generators take as input parameters corresponding to the applicable XML Schema facets and generate random instance value within the specified constraints. For example, the float generator which generates data of `float` datatype accept arguments corresponding to facets applicable on `float` datatype, such as `enumeration`, `maxExclusive`, `maxInclusive`, `minExclusive`, `minInclusive`, `pattern`, etc. An example is shown in Figure 6. A set of similar generators for other predefined simple datatypes, which are derived from primitive datatypes can then be easily constructed based on the

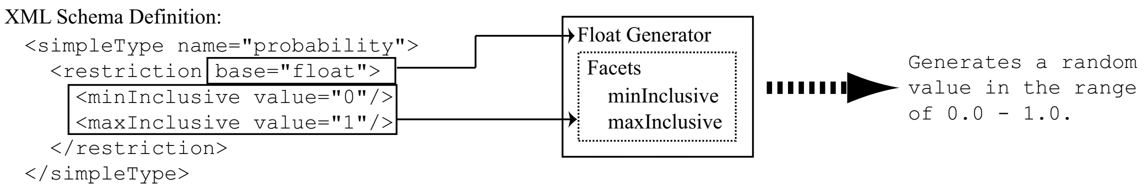

XML Schema Definition:
<simpleType name="probability">
<restriction base="float">
<minInclusive value="0"/>
<maxInclusive value="1"/>
</restriction>
</simpleType>
Float Generator
Facets
minInclusive
maxInclusive
Generates a random value in the range of 0.0 - 1.0.

Figure 6: Using Corresponding Generator to Generate Random Data within Constraints

above prepared generators. These generators can be refined to restrict the kind of data generated as far as the XML Schema allows.

Although these generators are only defined for simple datatype, they can be used to generate complex datatype with the help from a controller, which traverse the tree structure of the complex XML data and fill in values of simple datatype along the way.

Dependent Input Data

Dependent input data are input data to the PUT which rely on other data. An example of dependent input data is the TravelSearch response message shown in Figure 4. Dependent input data can be generated according to their relationship with other data specified in XPath expression.

Independent Output Data

These data come out from the PUT and do not rely on previous data. The corresponding verification conditions have to be independently specified. This data group includes dynamically generated or statically defined data in the PUT. For example, partner Web services often require the use of special access keys to identify callers, the value of these keys are often defined statically inside the PUT and do not depend on other data. An example of independent output data is the `<appid>` element in TravelSearch request message shown in Figure 4.

Dependent Output Data

Dependent output data are output data from the PUT which relies on other data. An example of dependent output data is the TravelSearch request message shown in Figure 4. To verify dependent output data, the verification condition which is made up of a location path pointing to the data and an expected value have to be generated. The location path to the element which is to be verified can be constructed easily since the structure of the overall output data is known. The expected value can be generated according to the relationship with other data specified in XPath expression.

Generating Test Cases

As explained in Section 2.3, a BPELUnit test case basically consists of a set of client track and partner tracks, which contains activities corresponding to involved operations respectively. These activities hold the data to be sent to the PUT and verification conditions used to verify output from the PUT. Amongst these necessary information in a BPELUnit test case, data and verification conditions have been generated in the previous stage. The set of involved operations is also defined by developers beforehand. Therefore, it is comparatively easy to compose a complete test case with these data. Refer Section 4.3.2 for a detailed explanation of the process.

3.2 Platform Independent Execution Logging

This section presents a method to capture execution information by weaving standard BPEL activities which invoke an external logging service into the PUT. From the log, execution information can be retrieved and analyzed to present useful information to developers.

The proposed approach is inspired by the work of L. Baresi et al. [14]. In their work, a non intrusive way of adding assertions as comments into BPEL processes was proposed for process monitoring purposes. These comments are then converted accordingly to BPEL activities at run time, without changing the functional behavior of the original process. Although their intention was not on execution logging, similar approach can be applied in this case. Since the purpose is simply to gather execution information, no human interaction is needed here. Standard BPEL activities used to gather such information can be added into the process automatically before compilation takes place. The proposed method is explained in the following subsections.

3.2.1 Augmentations of the PUT

The PUT is first augmented with additional `invoke` activities, with one attached before and another one after each basic activity in the PUT. This augmentation process is shown in Figure 7. These `invoke` activities all have the same single purpose: invoke an external logger Web service to notify that the activity to which it is attached is going to be, or has been executed. From the log output of the logger Web service and the source code of the PUT, execution information can be retrieved and analyzed to present useful information to developers.

The attachment of the pair of `invoke` activities to a basic activity is done by grouping the basic activity and the surrounding `invoke` activities inside a `sequence` structured activity. In Figure 7, this grouping is shown as boxes surrounding the tuples of basic activity and `invoke` activities. With the use of `sequence`, it is guaranteed that corresponding

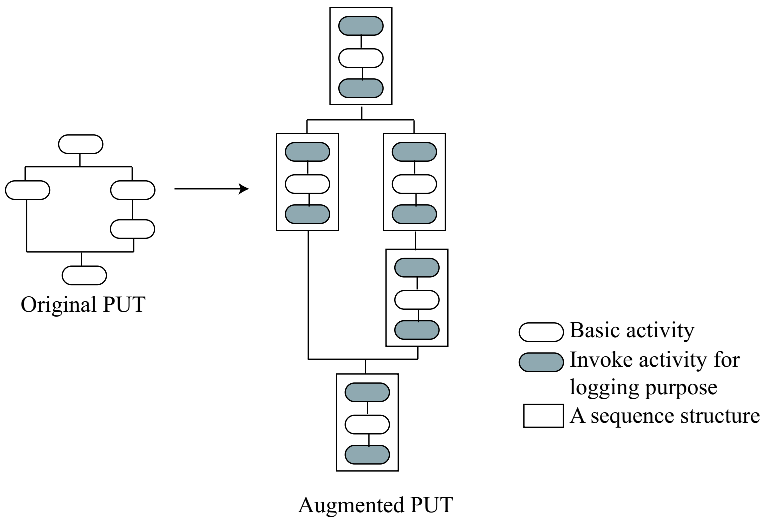


Figure 7: Augmentation of the PUT

`invoke` activities is executed right before and after the activity they are attached to.

3.2.2 The Logger Web Service

The logger Web service should at least provides the following operations:

- An operation which accepts log requests. Once invoked by the call from the PUT, this operation records the time of arrival of the message and the information contained in the request.
- An operation to control log profile so that log requests from different test cases can be identified and handled accordingly.
- An operation to retrieve logged information.

4 Design and Implementation

This section presents the design and implementation of a test case generation tool for BPEL unit testing based on the data dependency based approach proposed in Section 3. First, the overview of the whole system is presented. Then, the types of data dependency this tool supports and the way data dependencies are stored are discussed. This is followed by detailed explanations on both the back end and the user interface of the tool.

4.1 System Overview

Figure 8 shows the system overview of the test case generation tool. A dotted-line box represents a subsystem. There are two subsystems in the tool, one is the test case generation subsystem and the other one is the platform independent execution logging subsystem. As their names suggest, the test case generation subsystem aids developers in the creation of test cases while the execution logging subsystem gives feedbacks to developers after the generated test cases have been run.

The test case generation subsystem comprises three components. The BPEL Data Dependency Editor enables developers to group relevant operations into operation sets and then specify data dependencies for each set. This editor gets type information of XML data used by these operations from the WSDL files of involved Web services. It provides a graphical interface to help developers create the BPEL Data Dependency Description document. The Test Case Generator takes as input the BPEL Data Dependency Description document and the WSDL files and output test cases which can be run on the BPELUnit framework. The specification of the BPEL Data Dependency Description document is presented in Section 4.2 and the mechanism of the Test Case Generator is discussed in Section 4.3. Section 4.5 introduces the graphical interface of the BPEL Data Dependency Editor.

The platform independent execution logging subsystem is composed of three components. The BPEL Process Augmenter takes in a BPEL process and augment it with additional `invoke` activities as explained in Section 3.2.1. Developers then deploys the augmented BPEL process on the BPEL engine as usual. Test cases generated previously are then run on the BPELUnit framework and these `invoke` activities gets called and they send log requests to the Logger Web Service. The details of these components are discussed in Section 4.4. The BPELUnit Runnder Editor is a graphical front end provided to access all these components. It lets developers transform a BPEL process using the BPEL Process Augmenter and run BPELUnit test cases. After running the test cases, it invokes the Logger Web Service to retrieve the execution information. Comparison is made between the original BPEL process and the retrieved information to produce the final report which is presented to the developers as feedbacks. This editor is introduced

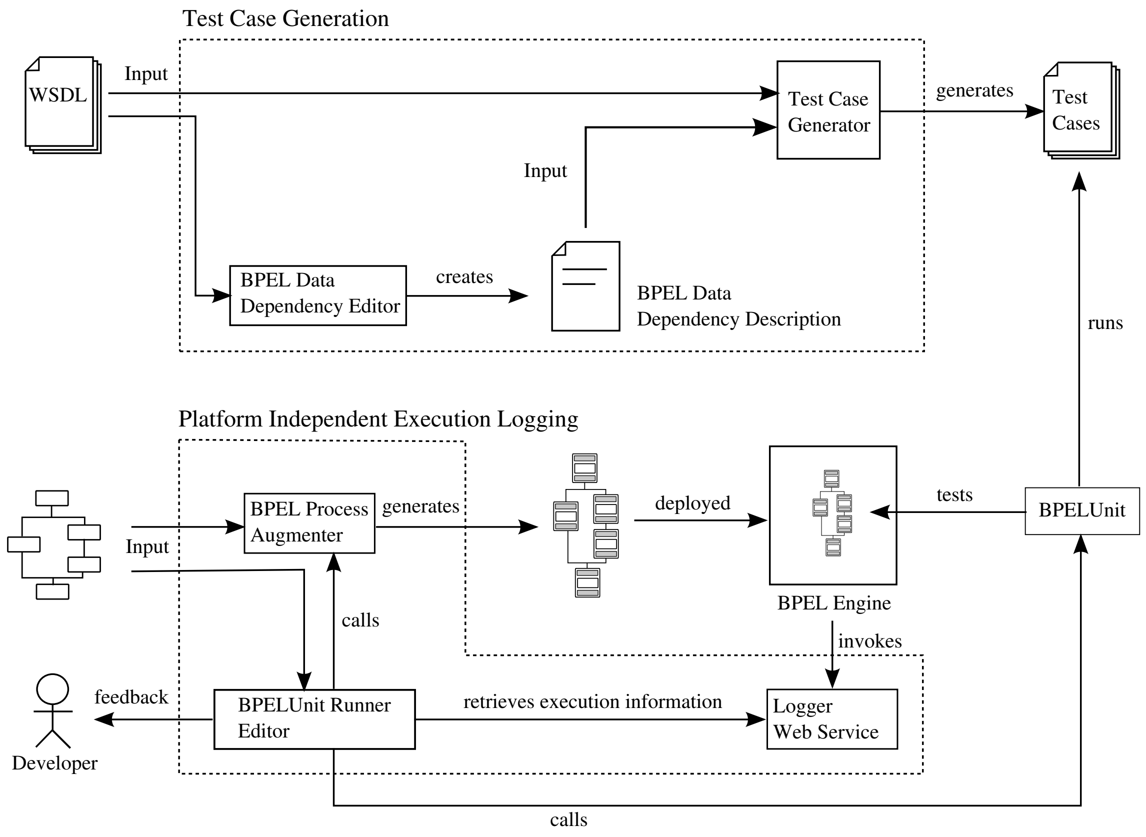
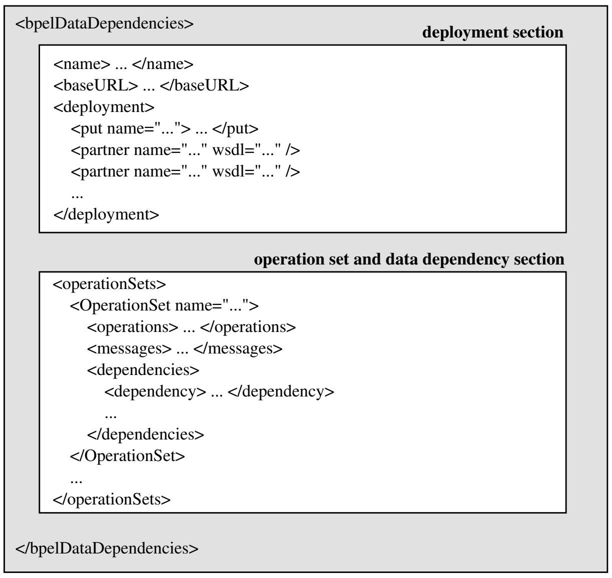


Figure 8: System Overview



```
<bpelDataDependencies>
    <name> ... </name>
    <baseURL> ... </baseURL>
    <deployment>
        <put name="..."> ... </put>
        <partner name="..." wsdl="..." />
        <partner name="..." wsdl="..." />
        ...
    </deployment>

    <operationSets>
        <OperationSet name="...">
            <operations> ... </operations>
            <messages> ... </messages>
            <dependencies>
                <dependency> ... </dependency>
                ...
            </dependencies>
        </OperationSet>
        ...
    </operationSets>
</bpelDataDependencies>
```

Figure 9: The BPEL Data Dependency Description Document

in Section 4.5.

4.2 BPEL Data Dependency Description Specification

The BPEL Data Dependency Description document is implemented as an XML document. In this document, the following information needs to be stored in order to provide enough information for the Test Case Generator to generate test cases later.

- Information needed in the BPELUnit test suite document.
- Sets of relevant operations, each composes a test case.
- Data dependency information in each of the operation sets.

The basic structure of a BPEL Data Dependency Description document is shown in Figure 9. In the following sections, how each of the above is stored in the document is discussed. The full XML Schema of the BPEL Data Dependency Description document is provided in the appendix.

4.2.1 BPELUnit Test Suite

BPELUnit test suite binds several test cases together. It contains two sections, i.e. the deployment section and the test case section. The deployment section contains information it needs to set up a testing environment, such as the location of WSDL files of the PUT

```
<operations>
   <operation id="1" partner="..." service="..." port="..." operation="..."/>
</operations>

<messages>
   <message id="1" operationId="1" messageType="input"/>
   <message id="2" operationId="1" messageType="output"/>
</messages>
```

Figure 10: Operations and Messages

and the partners. Although the Test Case Generator's main job is to generate test cases, it combines these test cases into a BPELUnit test suite for easier management. Therefore, the generator needs information required to produce a BPELUnit test suite. This piece of information is stored in the deployment section in the BPEL Data Dependency Description document as shown in Figure 9. Although the BPELUnit can take many different parameters to set up the testing environment, here some of these parameters take the default values to simplify the tool's interface. More flexibilities can be introduced in the future versions of the tool.

4.2.2 Operation Set

In BPELUnit, each test case corresponds to a set of operations. Therefore, the BPEL Data Dependency Description document must contain information on the set of operations involved in each test case.

Operation set information is stored in the second section of the BPEL Data Dependency Description document, shown in Figure 9. Inside the `<operationSets>` element, each operation set is given a name, a set of operations, messages used by those operations and a list of data dependencies.

The name of the operation set corresponds to the name of the generated test case. Each operation added to the set and the messages it uses are given identification numbers for simpler reference later in the dependency section. An example of how operations and messages are stored is given in Figure 10.

4.2.3 Data Dependency

The most important information in an operation set is the data dependency information. In order to specify a dependency, the dependent particle have to be specified, together with an XPath expression from which the intended value can be computed as explained in Section 3.1.1. The dependent particle can be a simple datatype element, an attribute or an operation. However, when the dependent particle is an operation, only dependency of `multiplicity` type is applicable. More information on dependency type

```
<dependency type="..." targetMsgId="..." targetOpId="..." dependsOnMsgId="... ... ..." iteration="..." >
   <target> ... </target>
   <dependsOn> ... </dependsOn>    OR    <fixedValue> ... <fixedValue>
</dependency>
```

Figure 11: Data Dependency Specification

later.

The XML structure shown in Figure 11 is used to accommodate necessary information. Nevertheless, not all fields are compulsory. Refer to Table 1 for the meaning of each field.

Further explanation on the types of data dependency and multiple invocations of an operation is provided below.

Types of Data Dependency

The following three types of data dependency are supported for the moment.

- **Substitution**
  The value of the dependent simple datatype element or attribute can be either a fixed value or the result of an XPath expression. If it is the latter, then it depends on other elements or attributes.
- **Multiplicity**
  The number of occurrence of the dependent element or operation can be either a fixed value or the result of an XPath expression. If it is the latter, then it depends on other elements or attributes.
- **Verification**
  This data dependency is introduced to support arbitrary verification of the output data from the PUT. It does not influence the values of the generated data but merely add a verification condition to the generated test case.

Some limitations exist in the specification of data dependency using XPath expression based on different data dependency type. In both substitution dependency and multiplicity dependency, the location path specified in the `target` field must only return one node. Also, the final data dependency graph must be acyclic. Furthermore, in the multiplicity dependency, the XPath expression specified in the `dependsOn` field must always returns a numeric value. For verification dependency, no limitations apply as they are copied verbatim to the resulting test case and have no influence on the data generation process.

Table 1: Fields in Data Dependency XML Structure

Field	Meaning
type	Type of data dependency. Supported values are `substitution`, `multiplicity` and `verification`. More explanation in the later part of this section.
targetMsgId / targetOpId	The identification number of the dependent data or operation, as assigned in the `<messages>` or `<operations>` section.
dependsOnMsgId	A space separated list of identification numbers of messages being dependent on. Each number takes the form of `m` or `m-n`, where `m` is the identification number and `n` the iteration of invocation. More explanation on multiple invocation in the later part of this section.
iteration	Specify an invocation of an operation when it is invoked more than one time. If this attribute is omitted, the dependency is applied to all invocations. More explanation on multiple invocation in the later part of this section.
target	The XPath expression to locate the dependent particle, when this particle is a simple datatype element or an attribute.
dependsOn	The XPath expression to compute the intended value for the dependent particle.
fixedValue	A fixed value.

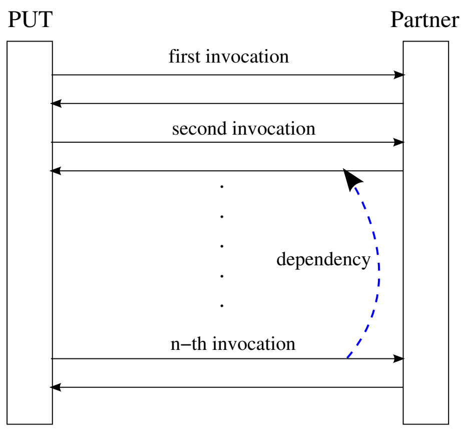


Figure 12: Data Dependency in a Multiple Invocation Scenario

Multiple Invocations of Operation

At times, operations in the operation set are invoked multiple times, for example, when an operation has to be invoked once for each result returned from a search engine, as shown in Figure 12. It would be convenient to be able to specify dependency on a specific invocation basis. However, identification number of message alone does not facilitate this type of specification since the same message is used no matter how many times the operation is invoked. Therefore, another identification scheme is introduced, which includes the iteration field and a special notation of identification number in the form of `m-n` in dependsOnMsgId field.

For the example shown in Figure 12, let assume that the input message of the Partner has the identification number m_i and the output message has the identification number m_o. To specify dependency shown in the figure, we set the iteration field of the dependency to n since it is the n-th iteration of invocation. Then, m_o-2 is to be added to the dependsOnMsgId list, to specify that the message used in the second invocation of the method is what we want this dependency to depend on.

However, if the iteration field is omitted, then this dependency applies for all iterations. On the other hand, if the "-n" part in dependsOnMsgId field are omitted, then they refer to the current iteration being processed. This is useful in specifying dependency of the output message of a partner on its input in the same iteration.

4.3 Test-related Data and Test Case Generation

This section discusses the algorithm used to generate test case corresponding to an operation set. In comparison to test cases used to test programs written in traditional languages like Java, the structure of a test case for BPEL unit testing is much simpler.

Basically it is composed of a series of operation invocations for the client and each involved partners, together with data sent back from the client or partners to the PUT and some verification conditions used to verify the output from the PUT.

The test case generation process is broken down to two steps. First, XML data used in all operation invocations are generated based on the data dependency information specified in the BPEL Data Dependency Description document and the datatype information available in WSDL files. Then, these generated data are used in the generation of test case. The details of each step are further discussed in the following sections.

4.3.1 XML Data Generation

All messages are put into a dependency map where their message identification numbers are mapped to a list of the message identification numbers of those messages that they depend on. Independent messages are mapped to an empty list. All messages in the dependency map are looped through during the generation process. Figure 13 shows what happened when a specific message is processed.

First, if the list that this message is mapped to is empty, then it is an independent message, the whole structure of the XML data is created, with optional elements and attributes generated and filled with random values of the appropriate types. The message is then removed from the map indicating that it has been generated. On the other hand, if it is a dependent message, then the map is checked to see if all messages that it depends on have been generated, if that is the case, random data are filled in the generated structure as in the case of independent message. However, data dependencies specified are iterated and resolved to replace certain random data in this case. It is then removed from the map. Nevertheless, if the messages that it depends on are not yet ready, this message is skipped. The above steps are repeated until the dependency map is empty. Therefore, skipped dependent messages are guaranteed to be generated at the end.

Table 2 shows the change of dependency map during the data generation process for the first six messages involved in the BPEL Travel process used as example in Section 2.2. Data dependencies that exist in this example are shown in Figure 5. In this example, messages 1, 4 and 6 are independent messages while the rest are dependent messages. Therefore, data for these messages are generated in the following sequence.

$$1 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 2$$

In the first loop, data for message 1 is generated as it is an independent message. Then, message 2 is skipped in this loop as the messages it depends on, i.e. message 4 and 6 have not been generated. Data for message 3 onwards are generated since these messages are either independent or all messages they depend on have been generated. Messages with data generated are removed from the map, leaving message 2 only in the map at the end of

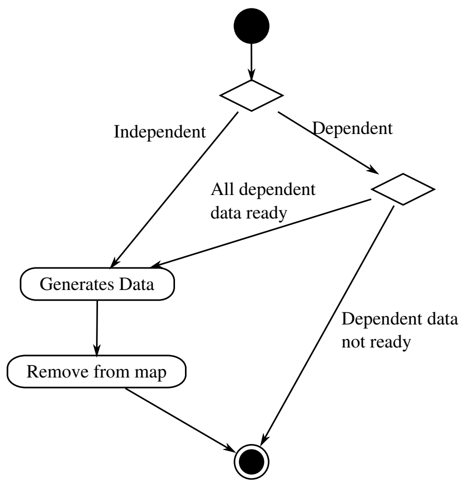


Figure 13: Flow of XML Data Generation

the first loop. In the second loop, data for message 2 are generated since the dependencies are all ready.

In fact, this is just a simplified explanation regarding XML data generation because data dependency might be specified across multiple invocations as explained in Section 4.2.3. More details on this topic will be discussed in the later part of this section. Below the implementation of XML data random generators is first presented.

Random Data Generation

In order to generate a random simple datatype XML data, classes shown in Figure 14 are implemented.

Supported XML Schema facets are all grouped under a general class named Facet. Default values for specific primitive and predefined datatypes provided by XML Schema

Table 2: Example of Data Dependency Map

Message ID	Dependencies
1	[]
2	[4,6]
3	[1]
4	[]
5	[1]
6	[]

First Loop

$\longrightarrow$

Message ID	Dependencies
2	[4,6]

Second Loop

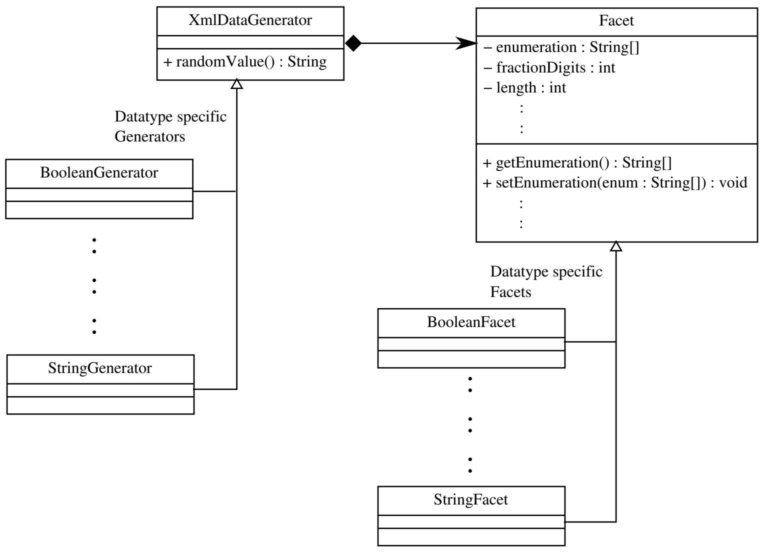


Figure 14: XML Data Generator Class

are then encapsulated in datatype specific facet classes, i.e. BooleanFacet, StringFacet, etc. which inherent class Facet.

The interface of XML data generator class is abstracted by class XmlDataGenerator. It holds an instance of class Facet so that the random data it generates, returned by the method randomValue(), conforms to the constraints set by various facets encapsulated in class Facet. This conformity is necessary as data not conforming to the constraints are blocked by some BPEL engines before they reach the PUT. Datatype specific generators, Boolean, StringGenerator, etc. inherent this class and generate data with the proper datatype.

A controller class is needed for the generation of a complex XML data structure since the above generators only generate simple datatype XML data. In this research, a sample tool provided by the XMLBeans project [24] is used to control the generation of complex XML data since XMLBeans is used in processing XML documents throughout the implementation of the tool.

Resolution of Data Dependendency Across Multiple Invocations

Data dependency might occur across multiple invocations of an operation as explained in Section 4.2.3. Figure 15 shows the table used to store generated message data according to a specific iteration of invocations. For each message identification number (Message ID), multiple instances of data are generated as necessary. A cell in the table represents an instance of data for the specific message corresponding to a specific invocation. A shaded cell in the table means that the instance has been generated.

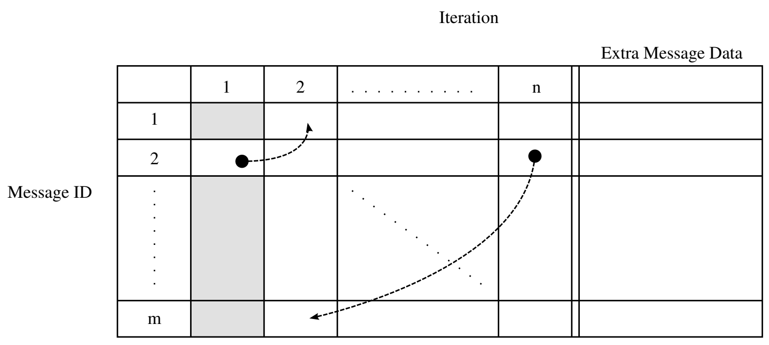


Figure 15: Message Data Table

Data dependency across multiple invocations is handled in the following way, which is an extension to the algorithm explained in the beginning of this section. The message data table is initialized to have m rows and n columns, where m is the total number of messages and n being the maximum value of the iteration field specified in the dependencies. In this extended algorithm, multiple dependency maps, each for a specific iteration, are used. Dependency maps are looped through to generate message data.

Message data are basically generated as explained before in an iteration. If cross-invocation dependency exists as shown in Figure 15, where message 2 in the first iteration depends on message 1 in the second iteration, message 2 in the first iteration cannot be generated at this point since the message it depends on is not available yet. Therefore, the generation of message 2 in the first iteration has to be skipped. This causes the dependency map of iteration 1 not empty. The algorithm moves on to the generation of the next iteration, which is iteration 2. All data in this iteration should be generated successfully as there is no dependency in this iteration. This causes the dependency map of iteration 2 to be empty after all message data are generated. This continues until the n-th iteration is reached. Again, since the dependency in this iteration is resolvable, the dependency map for this iteration will be empty at the end. After all dependency maps are processed, empty dependency maps are discarded to prevent further processing. Non-empty dependency maps are then re-processed, so that previously unresolvable dependencies can be processed again. However, when there is only one non-empty dependency map, the table is expanded to include another new iteration to prevent indefinite loop caused by dependency on further iterations. The above are repeated until all dependency maps are empty. In other words, all message data are by then generated.

Extra Message Data

The message data table shown in Figure 15 is only filled in as far as needed to resolve data dependency across multiple invocations. In the above example, only n iterations of message data are generated. Since multiplicity dependency applies on operation, certain operations might be invoked more than n times. Therefore, extra messages might need to be created specifically for that operation. When an operation is to be invoked $(n + 1)$ times, extra message data for that operation have to be generated, filling in the right part of the table shown in Figure 15. Since all dependencies should have been resolved by now, the generation of these extra message data is comparatively simple.

4.3.2 Test Case Generation

A BPELUnit test case is made up of one `clientTrack` and several `partnerTrack`s corresponding to the client and partners of the PUT respectively. Inside a track, activities which correspond to operation invocations can be defined.

Although different types of invocation are supported in BPELUnit, only "Send/Receive Synchronous" and "Receive/Send Synchronous", which corresponds to an operation with both input and output are implemented in this tool for the moment. In contrast to "Send/Receive Synchronous" which sends data before getting reply, "Receive/Send Synchronous" first receive data before sending back reply. Therefore, "Send/Receive Synchronous" is often used in the `clientTrack` while "Receive/Send Synchronous" is mostly used in `partnerTrack`. The other types of invocations are "Send Asynchronous", "Receive Asynchronous", "Send/Receive Asynchronous" and "Receive/Send Asynchronous".

Composing BPELUnit test case from the above generated message data is a comparatively straight-forward task. First, the generated test case is given the name of the operation set. Then, a `clientTrack` with a "Send/Receive Synchronous" activity and `partnerTrack`, with one or more "Receive/Send Synchronous" activities, for each operation defined in the operation set are added to the test case. The number of activities in each `partnerTrack` depends on the multiplicity dependency of the corresponding operation.

During the creation of tracks in the test case, data used in the `send` section which corresponds to the data that will be sent to the PUT are taken from the message data table explained in Section 4.3.1 while the pair of verification condition and the expected value are taken from the specified verification dependencies in the operation set.

One test case is generated for each operation set. Test cases generated for a BPEL Data Dependency Description document is grouped together as a BPELUnit test suite using deployment information specified in the BPEL Data Dependency Description document.

4.4 Platform Independent Execution Logging

The platform independent execution logging subsystem of the tool is made up of three components, i.e. the BPEL Process Augmenter, the Logger Web Service and the BPELUnit Runner Editor. The first two components are introduced in this section while the BPELUnit Runner Editor is explained in Section 4.5

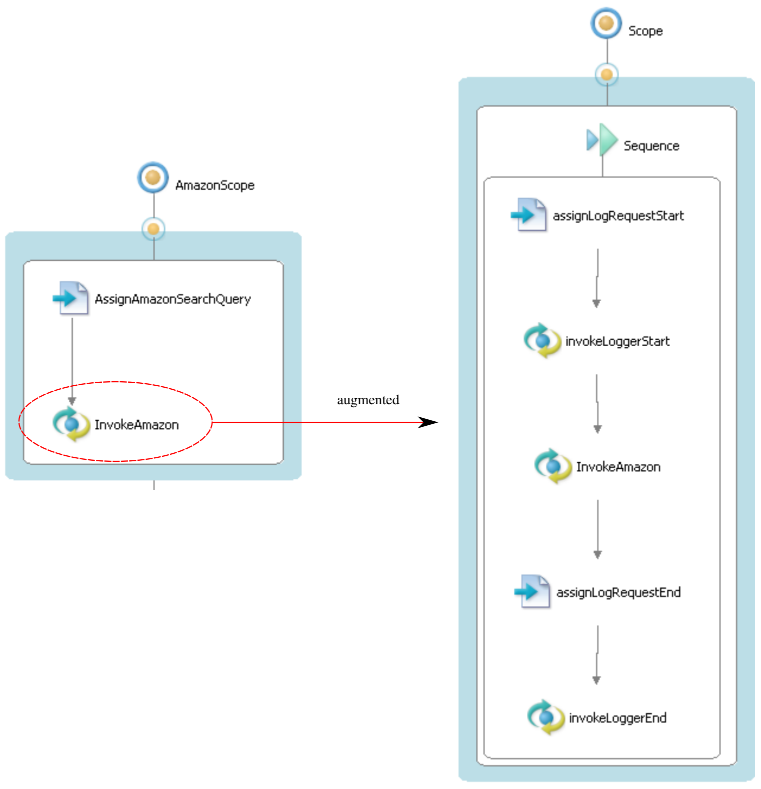


Figure 16: Sample of Augmentation

4.4.1 BPEL Process Augmenter

The BPEL Process Augmenter takes in a BPEL process source file, augment the basic activities within it with a pair of surrounding `invoke` activities as presented in Section 3.2.1 and output an augmented BPEL process file. Here, basic BPEL activities refer to `invoke`, `assign`, `throw`, `wait`, `empty` and `rethrow`. The PUT source can be parsed to locate all these basic activities easily by using XPath expression. Figure 16 shows the result of augmentation for the InvokeAmazon activity.

After the location of basic activities, a new `scope` is created for each of the basic activities. A `scope` is simply a container used to group related activities in BPEL. By using a `scope`, local variables can be used, and the resulting BPEL process will be easier to understand. This scope takes over the place of the basic activity in the BPEL process. Two local variables are created inside the `scope`. These variables are used in the `invoke` activities that will be added before and after the basic activity as log request and log response.

A `sequence` is then created in the `scope`. The name of the basic activity and the status message “start” to indicate that the activity is going to be executed are set in the log request using an `assign` activity. The first `invoke` activity is then added to the `sequence`. Next, the basic activity is copied to the sequence. Lastly, the status message "end" to indicate that the activity has bee executed successfully is assigned to the log request using another `assign` activity, and the second `invoke` activity is added to the `sequence`.

4.4.2 Logger Web Service

The Logger Web Service is implemented in the Java language using the Axis framework [4]. It provides the following three operations.

Operation: log

This operation takes the name of the basic activity and a status message which can be "start" or "end" as input and returns nothing. It writes an entry in the internal log file which records the time of arrival of the request, the activity name and whether it is entered or left. Below are the sample entries used in the actual implementation.

```
2008-01-07 18:17:56 JST,InvokeAmazon entered
2008-01-07 18:17:57 JST,InvokeAmazon left
```

Operation: setLogProfile

This operation takes the name of test suite, the name of test case and a status message which can be "start" or "end" as input and returns nothing. It writes an entry in the internal log file which records the time of arrival of the request, the name or the test suite and the name of the test case if the status message is "start". Below is a sample entry used in the actual implementation. The activity log entries below this line all belong to the specified test case.

```
[ TestSuite1 >>> TestCase1 (2008-01-07 18:17:54 JST) ]
```

Operation: getLog

This operation takes the name of a test suite and the name of a test case as input and returns a list of activity execution information, which contains the name of the activity, the total number of execution, the number of successful execution and the number of failed execution. It obtains the execution information by analyzing the internal log file.

First, it locates the start of the specified test case by looking for the log entry supposedly written by setLogProfile in the log file. It then breaks the activity log entries written by the log operation under the specified test case into two groups, i.e. "entered" and "left". Number of activities with the same name is counted and recorded along the way.

Finally, the operation builds the list of activity execution information by looping through activities grouped under “entered”. For each activity execution information, the name of the activity is first set. Then, the total number of execution is set to be the number of times this activity is put into the “entered” group. The number of successful execution equals to the number of times this activity is put into the “left” group and the number of failed execution is the difference of the former two numbers.

4.5 Graphical User Interface

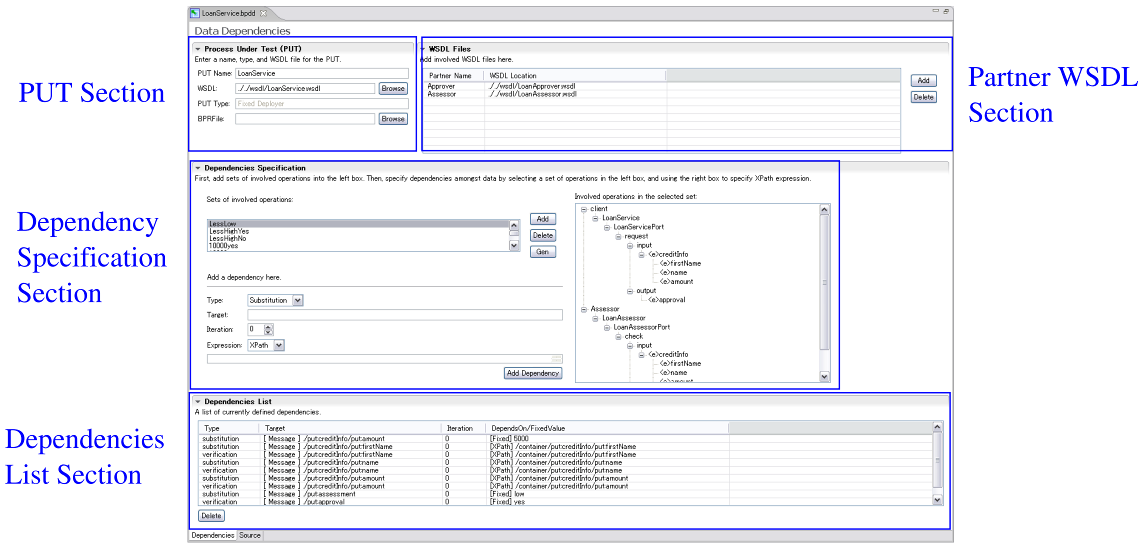
There are two components with graphical user interface implemented to provide graphical access to the aforementioned functions. The BPEL Data Dependency Editor helps developer create BPEL Data Dependency Description file and the BPELUnit Runner Editor provides an integrated tool to run generated test cases and get feedback on the test. These tools are implemented as plug-ins for the Eclipse IDE [8]. In this section, the user interfaces of these components are presented.

4.5.1 BPEL Data Dependency Editor

Figure 17 shows the main interface of the BPEL Data Dependency Editor. It consists of four different sections, i.e. the PUT section, the partner WSDL section, the dependency specification section and the dependencies list section. The PUT section and the partner WSDL let developer specify basic information needed to produce test cases for BPELUnit, such as the name as well as the location of WSDL files of the PUT and partners. The dependencies list section simply shows the data dependencies that are added so far. Dependencies can be deleted from here if so desirable.

The main part of the editor is the dependency specification section, which is enlarged and shown in Figure 18. On the upper left of the section, operation sets can be added by clicking on the “Add” button. Figure 19 shows the dialog box which appears when the “Add” button is clicked. It shows the list of available operations in all the services. Multiple operations can be selected here to add operations to the operation set. When the “OK” button is clicked, a simple input dialog box shows up to ask for the name of this operation set. Added operation sets can be discarded with the “Delete” button. Finally, when the “Gen” button is clicked, the test case generator will be invoked and one test case will be generated for each operation set. These generated test cases are grouped into a BPELUnit test suite which is saved inside the same folder as the current BPEL data dependency document.

Data dependencies are defined and added at the bottom left of the section. First, developer needs to specify the type of the dependency, which can be substitution, multiplicity and verification. Then, the target of the dependency is specified. This can be done by right-clicking on an element, attribute or operation name shown in the tree on the



Process Under Test (PUT)
Enter a name, type, and WSDL file for the PUT.

PUT Name: LoanService
WSDL: ./../wsdl/LoanService.wsdl
PUT Type: Fixed Deployer
BPRFile:

WSDL Files
Add involved WSDL files here.

Partner Name	WSDL Location
Approver	./../wsdl/LoanApprover.wsdl
Assessor	./../wsdl/LoanAssessor.wsdl

Dependencies Specification
First, add sets of involved operations into the left box. Then, specify dependencies amongst data by selecting a set of operations in the left box, and using the right box to specify XPath expression.

Sets of involved operations:

```
LessLow
LessHighYes
LessHighNo
10000yes
```

Involved operations in the selected set:

```
client
  LoanService
    LoanServicePort
      request
        input
          <e>creditInfo
            <e>firstName
            <e>name
            <e>amount
        output
          <e>approval
Assessor
  LoanAssessor
    LoanAssessorPort
      check
        input
          <e>creditInfo
            <e>firstName
            <e>name
```

Add a dependency here.

Type: Substitution
Target:
Iteration: 0
Expression: XPath

Dependencies List
A list of currently defined dependencies.

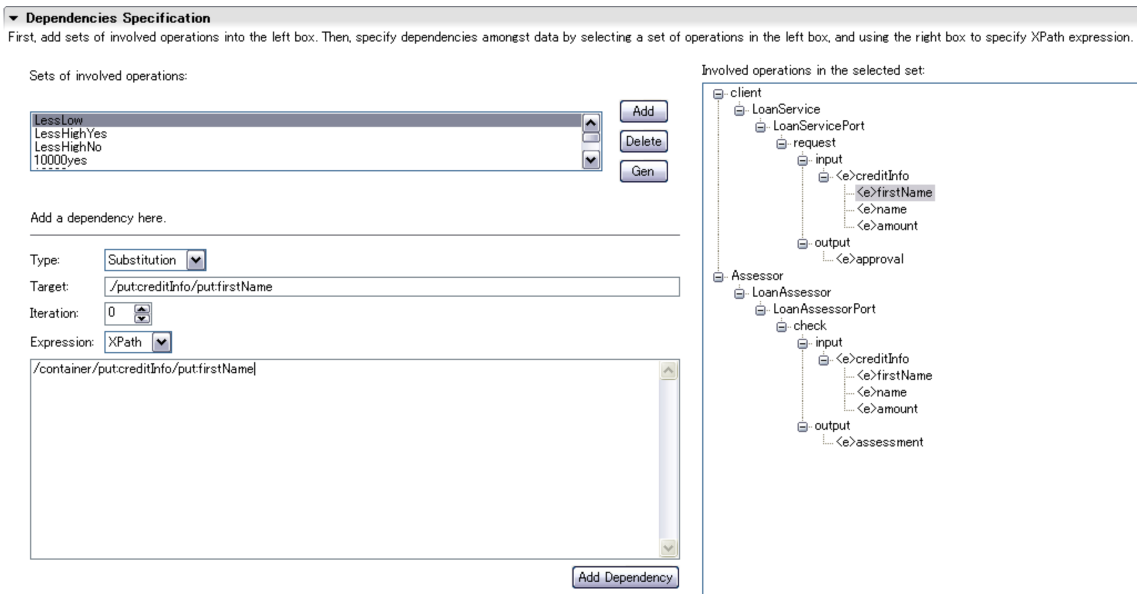
Type	Target	Iteration	DependsOn/FixedValue
substitution	[Message] /put:creditInfo/put:amount	0	[Fixed] 5000
substitution	[Message] /put:creditInfo/put:firstName	0	[XPath] /container/put:creditInfo/put:firstName
verification	[Message] /put:creditInfo/put:firstName	0	[XPath] /container/put:creditInfo/put:firstName
substitution	[Message] /put:creditInfo/put:name	0	[XPath] /container/put:creditInfo/put:name
verification	[Message] /put:creditInfo/put:name	0	[XPath] /container/put:creditInfo/put:name
substitution	[Message] /put:creditInfo/put:amount	0	[XPath] /container/put:creditInfo/put:amount
verification	[Message] /put:creditInfo/put:amount	0	[XPath] /container/put:creditInfo/put:amount
substitution	[Message] /put:assessment	0	[Fixed] low
verification	[Message] /put:approval	0	[Fixed] yes

Figure 17: BPEL Data Dependency Editor

right part of the section, and select "Use selected as target" from the context menu. The XPath expression pointing to the element will be automatically generated and set. Target iteration of the message can be set using the spin box below the target field. The default value is zero which means this dependency is effective for all iterations of the specified message. Finally, the value this message depends on can be set either as a fixed value or an XPath expression. However, it is necessary to specify this using the expression drop-down list. The fixed value or the XPath expression can then be inputted into the text box at the bottom. XPath expression to select a node in other messages can be done by double-clicking on any element or attribute shown in the data structure tree. Whenever a node in the tree is double-clicked, a simple dialog shows up to ask for the iteration of the message the target depends on. Same as the target, this is defaulted to have a value of one. After everything is specified, developer clicks on the "Add Dependency" button and the dependency will be added to the list. However, when the target is a output from the PUT and the dependency type is either substitution or multiplicity, a equivalent verification dependency will be automatically added.

4.5.2 BPELUnit Runner Editor

BPELUnit Runner Editor provides an integrated user interface to access the BPEL process augmentation function provided by the BPEL process augmenter, to call the command-line version of BPELUnit to run the test cases generated by the test case generator and to display the execution information retrieved from the logger Web service. Figure 20 shows the interface of the editor.



▾ Dependencies Specification

First, add sets of involved operations into the left box. Then, specify dependencies amongst data by selecting a set of operations in the left box, and using the right box to specify XPath expression.

Sets of involved operations:

LessLow
LessHighYes
LessHighNo
10000yes

Add a dependency here.

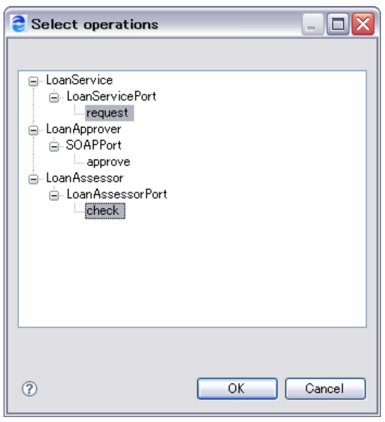
Type: Substitution
Target: ./put:creditInfo/put:firstName
Iteration: 0
Expression: XPath

```
/container/put:creditInfo/put:firstName
```

Involved operations in the selected set:

```
client
  LoanService
    LoanServicePort
      request
        input
          <e>creditInfo
            <e>firstName
            <e>name
            <e>amount
        output
          <e>approval
Assessor
  LoanAssessor
    LoanAssessorPort
      check
        input
          <e>creditInfo
            <e>firstName
            <e>name
            <e>amount
        output
          <e>assessment
```

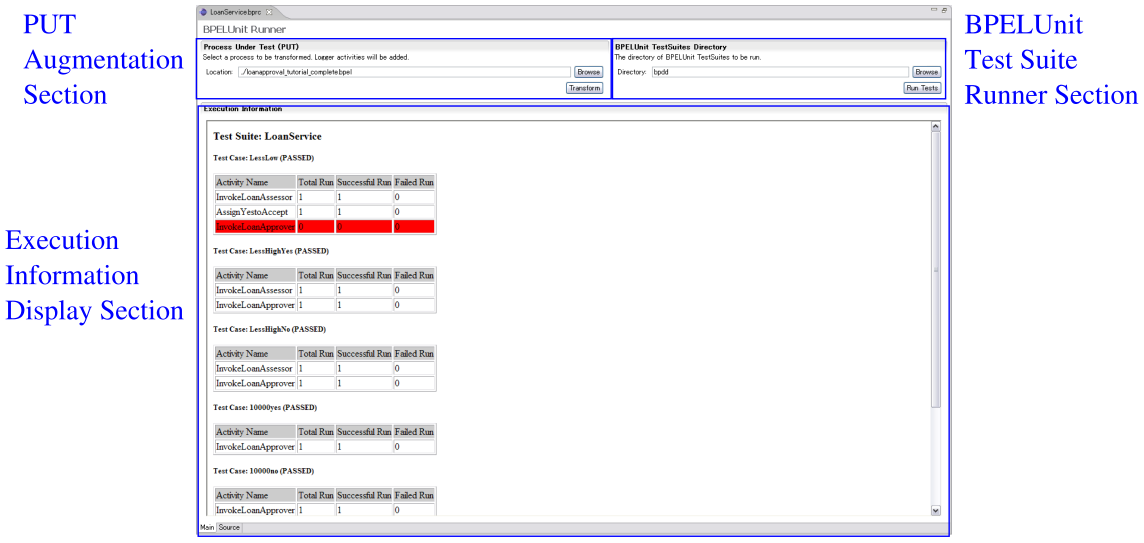
Figure 18: Dependency Specification Section in BPEL Data Dependency Editor



Select operations

```
LoanService
  LoanServicePort
    request
LoanApprover
  SOAPPort
    approve
LoanAssessor
  LoanAssessorPort
    check
```

Figure 19: Selecting Related Operations in Operation Set



BPELUnit Runner

Process Under Test (PUT)
Select a process to be transformed. Logger activities will be added.
Location: ./loanapproval_tutorial_complete.bpel

BPELUnit TestSuites Directory
The directory of BPELUnit TestSuites to be run.
Directory: bpdd

Execution Information

Test Suite: LoanService

Test Case: LessLow (PASSED)

Activity Name	Total Run	Successful Run	Failed Run
InvokeLoanAssessor	1	1	0
AssignYestoAccept	1	1	0
InvokeLoanApprover	0	0	0

Test Case: LessHighYes (PASSED)

Activity Name	Total Run	Successful Run	Failed Run
InvokeLoanAssessor	1	1	0
InvokeLoanApprover	1	1	0

Test Case: LessHighNo (PASSED)

Activity Name	Total Run	Successful Run	Failed Run
InvokeLoanAssessor	1	1	0
InvokeLoanApprover	1	1	0

Test Case: 10000yes (PASSED)

Activity Name	Total Run	Successful Run	Failed Run
InvokeLoanApprover	1	1	0

Test Case: 10000no (PASSED)

Activity Name	Total Run	Successful Run	Failed Run
InvokeLoanApprover	1	1	0

Figure 20: BPELUnit Runner Editor

Developers can access the above three functions through the three sections in the editor. After specifying the location of the target BPEL process in the PUT augmentation section, developer clicks on the "Transform" button and a new augmented BPEL process will be generated in the same directory as the original process. Support to deploy the augmented process to the BPEL engine is not available at the current version of the tool. Therefore developers have to deploy the augmented process manually.

After deploying the BPEL process, developers come back to the editor, specify the directory in which generated BPELUnit test suites are stored and click on the "Run Tests" button in the BPELUnit test suite runner section. The command-line version of BPELUnit is called to run all the BPELUnit test suites stored under the specified directory.

When BPELUnit exits after all the tests are complete, the editor sends request to the logger Web service to retrieve execution information of the tests. It then parses the original BPEL process to create a list of all basic activities. Combined with the execution information retrieved from the logger Web service, this list of activities is used to create a table for each test case, showing the total number of execution, the number of successful execution and the number of failed execution for each activity. Activities that have not been executed in a test case are highlighted to alert the developer.

5 Evaluation

Evaluation has been carried out to demonstrate the usefulness of the tool. This section describes the process of evaluation and its results. First, a case study was performed to show that this tool can actually be used in generating test cases for BPEL processes. Then, an experiment was done to compare this tool with the BPELUnit TestSuite Editor [5, 21], which is currently the only tool available to create BPELUnit test cases.

5.1 Case Study: Testing the BPEL Travel Process

In the case study, we use the BPEL Travel process introduced as example in Section 2.2. We have tested the following aspects of the BPEL Travel process.

- **Normal operation**. This test case tests the process under normal condition in which all partners work as expected. Sixty one dependencies were added to generate the test case. All output data from the process are checked. No error is expected in this test.

- **Partner failure.** This test case tests the fault handling ability of the process when some of the partner services are not available due to network problems. The creation of this test case is almost effortless as no dependency is needed. Operations of partner services that fail are simply omitted from the operation set. Since there are no fault handler implemented in the process, this test should fail with error.

- **Zero result**. This test case tests the process in situation where some of the partner return zero results. Since there are five partner services which return results, the maximum number of dependencies needed in a single test case is just five. Here the travel agent service is set to return no result. Again, this test case should fail as this situation is not considered during the creation of the process.

- **Correct number of loops.** This test case tests whether the number of invocations of the currency exchange rate query service equals to the number of results returned by the geographical information query service. This test case can be defined in two ways. The number of results returned by the geographical information query service can be given a fixed value, or it can be given a random positive value by depending on an independent data which is restricted to have such value. Two dependencies are needed here, one to specify the number of results returned by th geographical information query service and one for the multiplicity of operation provided by the currency exchange rate query service. This test should pass as long as the number of results is not zero.

This case study shows that the tool is indeed flexible enough to handle common testing scenario. It enhances developer's productivity especially when only a small part of the process is to be tested as demonstrated in the last three cases. Although more dependencies are needed in the full test of the system, the specification of dependencies only needs to be done once and many different sets of data can be generated randomly to test the system thoroughly. Moreover, further enhancement to the implementation, such as direct comparison of similar XML data structures can help minimize the number of needed data dependencies in the future.

5.2 Experiment

An experiment is conducted to find out how the implemented tool benefits developers, if any, compared to the existing tool in creating BPELUnit test cases. The following are the points of consideration in this experiment.

- **Time measure.** Is there any difference in the time needed in creating the same amount of test cases using the proposed tool and the existing one?
- **Test case characteristics.** Is there any difference in the characteristics of test cases created with different tools?
- **Common mistakes.** What are the common mistakes made by the subjects in the experiment? Does this change depending on the tool being used?

The following sections present the experiment setup, the results of the experiment and finally the validity of the experiment.

5.2.1 Experiment Setup

Subjects. Subjects of the experiment are all first year master students majoring in software engineering. Most of them have little or no experience to Web services and related technologies. Self-evaluated skill levels of the subjects are shown in Table 3.

Software environment. The software environment used in the experiment was Eclipse version 3.2 with BPELUnit version 1.0 plug-in and the BPEL Data Dependency Editor plug-in installed. This environment was prepared by the experimenter and distributed to the subjects at the beginning of the experiment.

Hardware environment. All subjects used an 12-inch laptop computer with external mouse attached during the experiment.

Procedure. In this experiment, subjects were required to create two sets of test cases for two different BPEL processes, process A and process B. Specifications of these processes and the involved partner services were provided to the subjects in print. Besides, samples of all XML data involved are provided as clear text file. Test specifications which describes which aspect of the processes should be tested in each test case were provided as well. Questions were allowed throughout the experiment. The experiment was conducted in the following steps.

1. A brief tutorial was given to the subjects. This tutorial introduced basic ideas about Web services, XML, XPath and BPEL. The usage of each tool was also explained through demonstration in which test cases for a sample process were created using the different tools.

2. Subjects were required to study the specifications of the services and test specifications for the two processes, process A and process B carefully.

3. Subjects were required to create test cases for the two processes individually in different orders and record the time needed in each task. The order in which they work was decided randomly and is shown in Table 4. For example, Subject 1 was required to create test cases for process A first using tool 1, and then test cases for process B using tool 2. Tool 1 was BPELUnit TestSuite Editor and tool 2 was BPEL Data Dependency Editor.

4. Subjects were required to fill in a questionnaire after they completed all tasks.

Target processes and test specifications. Process A is the loan approval process shown as example in the BPEL specification [18]. It is commonly used as example in various other work [13, 25]. Figure 21 shows the basic flow of the process. This process first receives loan request from the client. If both the loan amount and the result of risk assessment of the individual are low, the request is automatically approved. However, if the loan amount is high, or the result of risk assessment of the individual is high, then a further investigation is needed before the request is approved. The subjects were required to create seven test cases to test the process under the following conditions. Names of the test cases are written in brackets.

Table 3: Skill Level

Subject	Skill Level (0-5)			Unit Testing Experience
	XML	XPath	BPEL	
1	3	2	0	No
2	4	3	1	Yes
3	0	0	0	No
4	1	0	0	Yes

* Skill level of 0 means no experience at all, 5 means highly experienced.

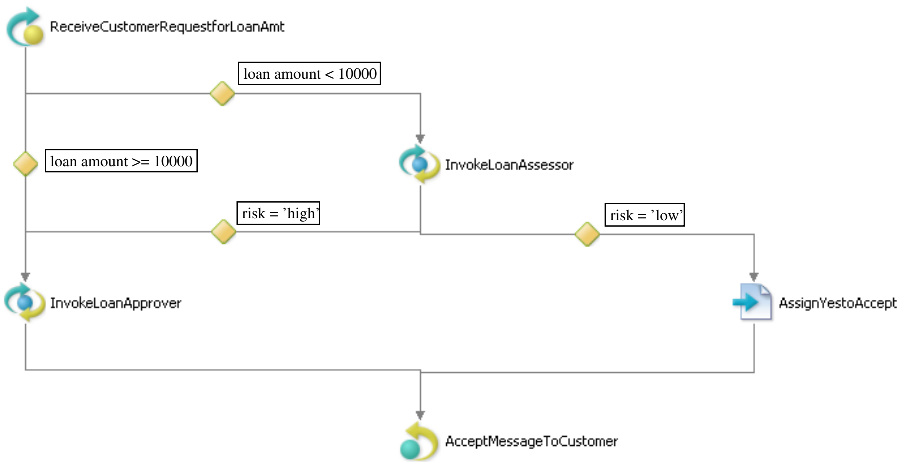


Figure 21: The Loan Approval Process

- Loan amount is less than 10000 (<10000) and the risk assessment service returns 'low'. (LessLow)
- Loan amount is less than 10000 (<10000) and the risk assessment service returns 'high'. Both situations in which the loan approver service returns 'yes' and 'no' must be tested. (LessHighYes and LessHighNo)
- Loan amount is exactly 10000. Both situations in which the loan approver service returns 'yes' and 'no' must be tested. (10000yes, 10000no)
- Loan amount is more than 10000. Both situations in which the loan approver service returns 'yes' and 'no' must be tested. (MoreYes, MoreNo)

On the other hand, process B is the Meta Search process presented in [21] as example. Figure 22 shows the basic flow of the process. This process first receives search string from the client and relays the search string to two search engines, Google and MSN Search. If any results are returned, it further processes them eliminating duplicates along the way and then returns them to the client. However, search engines which do not response or return no result at all will be ignored. Maximum number of results can be set as parameter in the request to the process. The subjects were required to create six test cases to test the process under the following conditions, which are presented in [21] as well. Names of the test cases are written in brackets.

- Only Google returns results. (GoogleOnly)
- Only MSN Search returns results. (MSNOnly)

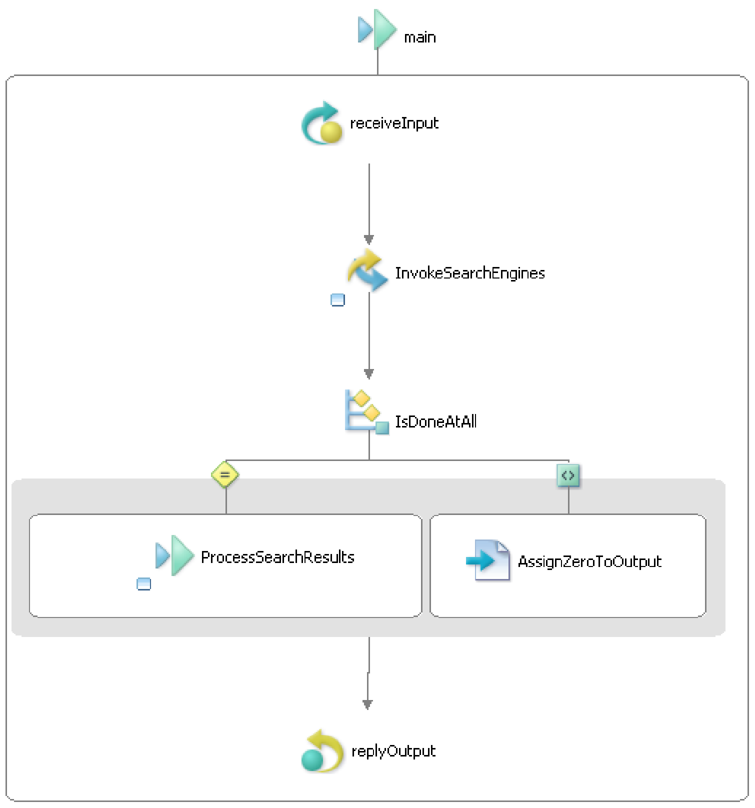


Figure 22: The Meta Search Process

- The same number of results are returned by the two search engines. However, all results are distinct. (DistinctResultsSameLen)
- Different number of results are returned by the two search engines. However, all results are distinct. (DistinctResultsDiffLen)
- The Meta Search process does not return results more than specified. (MaxResults)
- There are overlapping results. (OverlappingResults)

5.2.2 Results and Discussion

The Time Measures

The time needed to create each test case using a specific tool is shown in Table 4. The time varies greatly amongst subjects. This is because of the different skill levels of the subjects.

In most cases, the time taken is close independent on the target process or the tool used for a specific subject, except for Subject 1. Since the complexities of the two processes are about the same, this close time measure suggests that while not being able to greatly improve the productivity of developers, the suggested tool at least did not pose an initial burden on developers. Considering the fact that once the data dependencies are defined, infinite number of test cases with different combinations of data can be automatically generated, this is an important positive result. For Subject 1, it is found out later in an interview that this subject has spent most of the time taken to create test cases for process B on figuring out the unspecified details of the process, such as the order in which results are returned. Since this is not much related to the tool being used, this time measure is therefore ignorable.

Test Case Characteristics

Table 5 and 6 show the number of dependencies created during the experiment for each process using the BPEL Data Dependency Editor.

Table 5 shows the respective number of dependencies defined based on whether an XPath expression is used or a fixed value is used. Since data dependencies defined using fixed values put limitation on the variety of generated data, data dependencies defined with XPath expression are therefore preferable. From the table, we can see that Subject 2 and 3 have a more balanced usage pattern. In all test cases, there are more dependencies defined using XPath expression than using fixed values. On the other hand, Subject 1 and 4 tend to use fixed values exclusively. From the above observations, we believe that Subject 2 and 3 had a better understanding of the proposed method while Subject 1 and 4 did not fully understand it. This belief is supported by the fact that Subject 2 had a comparatively higher skill level in related technologies while Subject 3 asked more

Table 4: Experiment Results

Subject	Process	Tool	Time (min.)
1	A	1	73
	B	2	174
2	B	1	37
	A	2	30
3	A	2	73
	B	1	99
4	B	2	53
	A	1	56

Table 5: Number of Dependencies Created During Experiment Grouped by XPath Expression (XP) or Fixed Value (FV)

Process	Test Case	Subject					
		2			3		
		XP	FV	Total	XP	FV	Total
A	LessLow	3	3	6	3	3	6
	LessHighYes	7	3	10	7	4	11
	LessHighNo	7	3	10	7	4	11
	10000yes	4	2	6	4	3	7
	10000no	4	2	6	4	3	7
	MoreYes	4	2	6	4	3	7
	MoreNo	4	2	6	4	3	7

Process	Test Case	Subject					
		1			4		
		XP	FV	Total	XP	FV	Total
B	GoogleOnly	0	6	6	1	1	2
	MSNOnly	0	6	6	1	1	2
	DistinctResultsSameLen	0	6	6	3	0	3
	DistinctResultsDiffLen	0	9	9	0	1	1
	MaxResults	1	0	1	1	0	1
	OverlappingResults	0	6	6	1	0	1

Table 6: Number of Dependencies Created During Experiment Grouped by Substitution (S), Multiplicity (M) and Verification (V)

Process	Test Case	Subject							
		2				3			
		S	M	V	Total	S	M	V	Total
A	LessLow	2	0	4	6	2	0	4	6
	LessHighYes	3	0	7	10	3	0	8	11
	LessHighNo	3	0	7	10	3	0	8	11
	10000yes	2	0	4	6	2	0	5	7
	10000no	2	0	4	6	2	0	5	7
	MoreYes	2	0	4	6	2	0	5	7
	MoreNo	2	0	4	6	2	0	5	7

Process	Test Case	Subject							
		1				4			
		S	M	V	Total	S	M	V	Total
B	GoogleOnly	1	2	3	6	0	0	2	2
	MSNOnly	1	2	3	6	0	0	2	2
	DistinctResultsSameLen	3	0	3	6	1	0	2	3
	DistinctResultsDiffLen	6	0	3	9	0	0	1	1
	MaxResults	0	0	1	1	0	0	1	1
	OverlappingResults	3	0	3	6	0	0	1	1

Table 7: Number of Verification Conditions Created During Experiment Based on Tool Being Used

Process	Test Case	TestSuite Editor		Data Dependency Editor	
		Subject 1	Subject 4	Subject 2	Subject 3
A	LessLow	2	1	4	4
	LessHighYes	3	3	7	8
	LessHighNo	3	3	7	8
	10000yes	2	2	4	5
	10000no	2	2	4	5
	MoreYes	2	2	4	5
	MoreNo	2	2	4	5

questions during the experiment.

On the other hand, Table 6 shows the respective number of dependencies defined based on data dependency type. From this table, we notice that there are cases in which Subject 1 and 4 only defined verification dependencies in a test case. This causes all data to be generated completely randomly thus making it impossible to generate valid test case. From this, we can further confirm that Subject 1 and 4 did not understand the proposed method.

We chose test cases created for Process A as the base to compare the created test cases. This is because all subjects created comparatively valid test cases for this process, except for minor mistakes. We compare the number of verification conditions created in each test case since verification conditions actually verify the correctness of the PUT and it is a concept that exists in both methods. Table 7 shows the number of verification conditions created by each subject using different tool in the test cases for Process A.

From Table 7, we can see that numbers of verification conditions created using BPEL Data Dependency Editor are in all cases higher than those created using BPELUnit Test-Suite Editor. By closer observation, test cases created using BPEL TestSuite Editor tend to skip some unimportant output data from the PUT. On the other hand, test cases created using the proposed BPEL Data Dependency Editor tends to include these checks. This is believed to be the effect of showing the whole data structure of relevant data to developers, making it easy to create conditions to check those data. This is supported by the result of the questionnaire which gave high credits to the visibility of data structure.

In conclusion, although our proposed method is more difficult to learn compared to existing tool for total beginners, we believe that developers who have basic knowledge in Web services related technologies can pick up fast as demonstrated by Subject 2 and 3. Moreover, resulting test cases created with our proposed tool tend to verify more output data from the PUT.

Common Mistakes

Since all of the subjects have not much experience in BPEL and its related technologies as shown in Table 3, mistakes were unavoidable. It is interesting to study the common mistakes made depending on the tool being used.

In BPEL TestSuite Editor, most mistakes were related to XML namespace and XPath specification needed in specifying data to be sent by partner services to the PUT and verification conditions to verify output from the PUT. These mistakes include the syntax error in using XML namespace, such as using multiple prefixes, the insertion of unnecessary namespace prefix, omission of needed namespace prefix and use of undefined namespace prefix. Moreover, skipping of elements within XPath expression and spelling mistakes in elements' names were noticed as well. Besides, although only once, a mistake was made in which the loan approver service was specified to send the result of the risk assessor's result. The reason for this mistake is believed to be due to a wrong copy-and-paste since the two results look similar.

On the other hand, in BPEL Data Dependency Editor, the above mistakes were almost non-existent. This is believed to be the effect of automatic insertion of location paths by the tool. However, this tool is also not without problem. Since the subjects of the experiment are all beginners and this is the first time they tried out the proposed data dependency based approach, some necessary data dependencies were omitted in the specification, making the resulting test cases incomplete. Other implementation related mistakes were noticed, too. For example, although location paths are automatically inserted, developers still need to manually modify them to suit their needs in this implemented tool. Therefore, trivial errors such as wrong capitalizations in fixed values were unavoidable. Use of unimplemented functions, such as direct comparison of complex data and partial XPath expression like "> /location/path" exist, too. These implementation related mistakes provides some hints for further refinement of the tool.

In conclusion, the BPEL Data Dependency Editor has successfully solved most of the problems in existing tool. However, since it is still a preliminary implementation so there is still room for improvement to further assist developers to make less mistakes. In fact, from the responses collected by questionnaire, most of the problems the subjects faced in this experiment are originated from the imperfect implementation.

5.2.3 Validity of the Results

This section discusses the various validity issues of this experiment.

Strength of the Experiment Design

First, combinations of order in which tasks are carried out is designed so that the influence of order can be avoided. These orders are then assigned to subjects randomly to minimize the human factors. Test cases for each process is created twice using each tool so that the results are less individual dependent.

Then, enough time to understand the specifications is given as that is not related to the purpose of the experiment. Since all the subjects are inexperience in related technologies, questions regarding those technologies are allowed throughout the experiment to minimize the time taken in researching about the technologies which can have a bad effects on the results of the experiment.

Finally, considering that simple XML data generation tools are freely available, samples of involved XML data are provided in digital form to better imitate the real development process.

Internal Validity Issues

Although many considerations have been made and the experiment has been carefully planned, there are some issues which affect the validity of the results.

First of all, most of the subjects are beginners and inexperienced in related technologies. Therefore, mistakes were unavoidable in the resulting test cases. It is hard to assess the quality of test cases created in the experiment since they vary too much depending on individuals. This makes time measure comparison difficult without having a concrete test case quality assessment scheme in place. In this experiment, subjects had unlimited time to complete their tasks, this might make them less efficient in the use of time. A clear evidence of this is that not many questions were asked during the experiment as they preferred to think their way out given the ample time. In future experiments, time might need to be limited to avoid this problem.

Besides, developers are expected to actually run the created test cases while creating them in the real world environment. This is not possible in this experiment since it was thought to be a burden for the subjects to learn about yet another tool. Mistakes might be less if they could run the test cases while creating them in the experiment.

Finally, from the fact that fixed values were favored over XPath expressions in data dependency specifications, subjects might not have fully understood the concept of the proposed approach. They were practicing the same method of creating test cases despite that the tool at hand had changed.

External Validity Issues

The subjects were all beginners and the group was small with only four persons. There is no firm evidence that the results of this experiment can be generalized to professional BPEL developers, who are the target audience of the proposed method. Further studies which involve subjects from the professional field are needed to further evaluate the effectiveness of the proposed approach and implemented tool.

6 Related Work

The proposed XML data generation technique is based on the technique proposed in [23]. They use a knowledge base to store association information of each built-in simple datatype with default facets definition and sets of candidate values based on test strategies. However, no further information is provided about who is responsible for the preparation of the knowledge base and how it should be done. In the proposed approach, the use of this knowledge base is abandoned. Values are generated randomly to free developers from the need to create such knowledge base.

Although not specifically designed for test data generation, ToXgene was presented by D. Barbosa et al. [7] to generate large, consistent collections of synthetic XML documents. With ToXgene, a user specifies certain properties of the intended data, such as probability distributions with ToXgene Template Specification Language (TSL) which is a subset of the XML Schema notation augmented with annotations. While this gives users good control over the generated data, users need to learn another language and spend time on writing good templates. This imposes additional burden on developers and thus avoided in this work.

In the work of Yan et al. [13] and Yuan et al. [25], a BPEL process is first analyzed and translated into extended control flow graphs from which test paths are extracted. Test data are then generated using constraint solving tools or methods. While this might help creating test cases that cover more paths, constraint solving is known to be hard and not applicable at all time. However, test paths extracted with this method can be used to identify operation sets which can then be used with BPEL Data Dependency Editor to generate test cases which cover more parts of the process.

In [20], the use of a common API across BPEL engine vendors is suggested to provide execution information to developers. While this suggestion sounds feasible, it might need tedious work in the creation of such API. Therefore, in this research this problem is tackled from another aspect, which involves only standard BPEL features.

On the other hand, Baresi et al. [14] proposed a non intrusive way of adding assertions as comments into BPEL processes for process monitoring purposes. These comments are then converted accordingly to BPEL activities at run time, without changing the functional behavior of the original process. We have benefited from their work in the design of the platform independent logging of execution information in our work.

7 Conclusion

In this thesis, an approach to alleviate the effort needed in generating test case with coherent input and output data for BPEL unit testing is first presented. This is followed by the proposal of a method to obtain execution information using only standard BPEL features. In the proposed approach, developers are presented with structures of involved XML data and required to define the relationship amongst them using XPath expression, which BPEL developers should be familiar with. Coherent test-related data and test cases are then generated automatically. To obtain execution information which helps developers understand the effectiveness of their test cases, the BPEL process is first augmented to include activities necessary for reporting execution information without its behavior being changed. This augmented BPEL process is then compiled and deployed as usual. A logger Web service is used to handle calls from the BPEL process to log the executed activities. Execution information is then reproduced and presented to developers based on this log and the source code of the BPEL process.

In the later part of the thesis, the design and implementation of a test case generation tool for BPEL unit testing based on the proposed approach is presented. This tool is then evaluated in two stages. First, a case study was performed to show that the tool can actually be used in generating test cases for BPEL processes. Then, an experiment was conducted to compare this tool with the existing one. We found that the time needed to define necessary data dependencies to create a test case in the proposed approach was about the same to the time needed to create a test case with existing tool. Considering the fact that once the data dependencies are defined, infinite number of test cases with different combinations of data can be automatically generated, this is an important positive result.

In the future, we would like to improve the user interface of the implemented tool to be more intuitive to minimize the chance of making mistakes. Also, we plan to leverage methods proposed by Yan et al. [13] and Yuan et al. [25] so that operation sets can be extracted from the PUT automatically.

Acknowledgments

I am most indebted to my supervisor Professor Katsuro Inoue for his continuous support and supervision over the years. Without his help, experience and advice, this thesis would never have reached completion.

I am very grateful to Associate Professor Makoto Matsushita and Assistant Professor Takashi Ishio for their valuable comments and helpful criticism which have helped guide and shape the development of this thesis.

I would also like to express my gratitude to Doctor Katsuhiko Yuura and Doctor Hideaki Shinomi from Hitachi Consulting Company, Limited for their valuable comments and advices on this thesis.

I would like to express my gratitude to all members of the Department of Computer Science for their guidance.

Thanks are also due to many friends in the Department of Computer Science, especially students in Inoue Laboratory.

References

[1] ActiveBPEL Designer. Available at `http://www.active-endpoints.com/active-bpel-designer.htm`.

[2] Gustavo Alonso, Fabio Casati, Harumi Kuno, Vijay Machiraju, Web Services: Concepts, Architectures and Applications. Springer, Berlin, 2004.

[3] Amazon Web Services. Available at `http://aws.amazon.com/`.

[4] WebServices - Axis. Available at `http://ws.apache.org/axis/index.html`.

[5] BPELUnit. Available at `http://www.bpelunit.org/`.

[6] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris and David Orchard, Web Services Architecture, 11 February 2004. Available at `http://www.w3.org/TR/ws-arch`.

[7] Denilson Barbosa, Alberto Mendelzon, John Keenleyside and Kelly Lyons, ToXgene: A template-based data generator for XML. In *Proceedings of the Fifth Workshop on the Web and Databases (WebDB'02)*, pages 49-54, June 6-7, 2002.

[8] Eclipse.org. Avaiable at `http://www.eclipse.org/`.

[9] Erik Christensen, Francisco Curbera, Greg Meredith and Sanjiva Weerawarana, Web Services Description Language (WSDL) 1.1, 15 March 2001. Available at `http://www.w3.org/TR/wsdl`.

[10] Frank Leymann, Dieter Roller and Satish Thatte, Goals of the BPEL4WS Specification. Available at `http://www.oasis-open.org/committees/download.php/3249/Original%20Design%20Goals%20for%20the%20BPEL4WS%20Specification.doc`.

[11] James A. Whittaker, What Is Software Testing? And Why Is It So Hard? In *IEEE Software*, pages 70-79, January/February 2000.

[12] James Clark and Steve DeRose, XML Path Language (XPath) Version 1.0, 16 November 1999. Available at `http://www.w3.org/TR/xpath`.

[13] Jun Yan, Zhongjie Li, Yuan Yuan, Wei Sun and Jian Zhang, BPEL4WS Unit Testing: Test Case Generation Using a Concurrent Path Analysis Approach. *17th International Symposium on Software Reliability Engineering (ISSRE'06)*, pages 75-84, 2006.

[14] Luciano Baresi, Carlo Ghezzi and Sam Guinea, Smart Monitors for Composed Services. In *Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC'04)*, pages 193-202, November 15-19, 2004.

[15] Michael Ellims, James Bridges and Darrel C. Ince, Unit Testing in Practice. In *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)*, pages 3-13, November 2-5, 2004.

[16] NetBeans Enterprise Pack 5.5. Available at `http://developers.sun.com/jsenterprise/nb_enterprise_pack/`.

[17] Eric Newcomer, Greg Lomow, Understanding SOA with Web Services. Addison-Wesley Professional, 2004.

[18] OASIS WSBPEL Technical Committee, Web Services Business Process Execution Language Version 2.0, 11 April 2007. Available at `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.

[19] Paul V. Biron and Ashok Malhotra, XML Schema Part 2: Datatypes Second Edition, 28 October 2004. Available at `http://www.w3.org/TR/xmlschema-2/`.

[20] Philip Mayer and Daniel Lübke, Towards a BPEL unit testing framework. In *Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications (TAV-WEB'06)*, pages 33-42, July 17, 2006.

[21] Philip Mayer, Design and Implementation of a Framework for Testing BPEL Compositions. Available at `http://www.se.uni-hannover.de/documents/studthesis/MSc/Philip_Mayer-Design_and_Implementation_of_a_Framework_for_Testing_BPEL_Compositions.pdf`, September 11, 2006.

[22] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, Donald F. Ferquson, Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More. Prentice Hall PTR, April 1, 2005.

[23] Xiaoying Bai, Wenli Dong, Wei-Tek Tsai and Yinong Chen, WSDL-Based Automatic Test Case Generation for Web Services Testing. In *Proceedings of the 2005 IEEE International Workshop on Service-Oriented System Engineering (SOSE'05)*, pages 207-212, October 20-21, 2005.

[24] XMLBeans. Available at `http://xmlbeans.apache.org/`.

[25] Yuan Yuan, Zhingjie Lie and Wei Sun, A Graph-search Based Approach to BPEL4WS Test Generation. In *Proceedings of the International Conference on Software Engineering Advances (ICSEA'06)*, page 14, 2006.

[26] Zhongjie Li, Wei Sun, Zhong Bo Jiang and Xin Zhang, BPEL4WS unit testing: framework and implementation. In *Proceedings of the IEEE international Conference on Web Services (ICWS'05)*, pages 103-110, July 11-15, 2005.

Appendix

A. XML Schema of BPEL Data Dependency Description

A XML Schema of BPEL Data Dependency Description

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://ddeditor4bpelunit.plugins.eclipse.k_choy. ↓
sel.ist.osaka_u.ac.jp/BpddSchema"
  xmlns:tns="http://ddeditor4bpelunit.plugins.eclipse.k_choy.sel.ist. ↓
osaka_u.ac.jp/BpddSchema"
  elementFormDefault="qualified">
  <complexType name="deploymentType">
    <sequence>
      <element name="put">
        <complexType>
          <sequence>
            <element name="property" maxOccurs="unbounded"
              minOccurs="0">
              <complexType>
                <simpleContent>
                  <extension base="string">
                    <attribute name="name"
                      type="string" />
                  </extension>
                </simpleContent>
              </complexType>
            </element>
            <element name="wsdl" type="string" />
          </sequence>
          <attribute name="name" type="string" />
          <attribute name="type" type="string" />
        </complexType>
      </element>
      <element name="partner" maxOccurs="unbounded"
        minOccurs="0">
        <complexType>
          <attribute name="name" type="string" />
          <attribute name="wsdl" type="string" />
        </complexType>
      </element>
    </sequence>
  </complexType>
  <complexType name="operationSetsType">
    <sequence>
      <element name="OperationSet" maxOccurs="unbounded"
        minOccurs="0">
        <complexType>
          <sequence>
            <element name="operations">
              <complexType>
                <sequence>
                  <element name="operation"
                    maxOccurs="unbounded" minOccurs="0"
                    type="tns:operationType" />
                </sequence>
              </complexType>
            </element>
            <element name="messages">
              <complexType>
                <sequence>
                  <element name="message"
                    maxOccurs="unbounded" minOccurs="0"
                    type="tns:messageType" />
                </sequence>
              </complexType>
            </element>
            <element name="dependencies">
              <complexType>
                <sequence>
                  <element name="dependency"
                    maxOccurs="unbounded" minOccurs="0"
                    type="tns:dependencyType" />
                </sequence>
              </complexType>
            </element>
          </sequence>
          <attribute name="name" type="string"></attribute>
        </complexType>
      </element>
    </sequence>
  </complexType>
  <complexType name="operationType">
    <attribute name="id" type="int" />
    <attribute name="partner" type="string" />
    <attribute name="service" type="QName" />
    <attribute name="port" type="string" />
    <attribute name="operation" type="string" />
  </complexType>
  <complexType name="messageType">
    <attribute name="id" type="int" />
    <attribute name="operationId" type="int" />
    <attribute name="messageType" type="tns:messageTypeType" />
  </complexType>
  <simpleType name="messageTypeType">
    <restriction base="string">
      <enumeration value="input" />
      <enumeration value="output" />
    </restriction>
  </simpleType>
  <complexType name="dependencyType">
    <sequence>
      <element name="target" type="string" minOccurs="0" />
      <choice>
        <element name="dependsOn" type="string" />
        <element name="fixedValue" type="string" />
      </choice>
    </sequence>
    <attribute name="targetMsgId" type="int" />
    <attribute name="targetOpId" type="int" />
    <attribute name="dependsOnMsgId">
      <simpleType>
        <list itemType="tns:depOnMsgIdType" />
      </simpleType>
    </attribute>
    <attribute name="iteration" type="int" />
    <attribute name="type" type="tns:dependencyTypeType" />
  </complexType>
  <simpleType name="depOnMsgIdType">
    <restriction base="string">
      <pattern value="\d+(-\d+)?"></pattern>
      </restriction>
    </simpleType>
    <simpleType name="dependencyTypeType">
      <restriction base="string">
        <enumeration value="substitution" />
        <enumeration value="verification" />
        <enumeration value="multiplicity" />
      </restriction>
    </simpleType>
    <element name="bpelDataDependencies">
      <complexType>
        <sequence>
          <element name="name" type="string" />
          <element name="baseURL" type="string" />
          <element name="deployment" type="tns:deploymentType" />
          <element name="operationSets"
            type="tns:operationSetsType" />
        </sequence>
      </complexType>
    </element>
  </schema>
```