2008－SE－159 (21)
2008／3／18

Data Dependency based Test Case Generation for BPEL Unit Testing

チョイ コーイー† 石尾 隆† 松下 誠† 井上 克郎†
四野見 秀明‡ 湯浦 克彦‡

† 大阪大学 大学院情報科学研究科
〒 560-8531 大阪府豊中市待兼山町 1-3
‡ 株式会社日立コンサルティング
〒 108-0075 東京都港区港南 2-16-4

ウェブサービスビジネスプロセス実行言語 WS-BPEL で書かれたプロセスを対象とした単体テストのためのテストケースを作成する際，開発者はテスト対象に入力として送信するデータとテスト対象からの出力を検証する検証式を手作業で用意しなければならない．しかし，ウェブサービスが送受信する XML データは一般に複雑なため，これらのテストデータを作成するのは困難である．また，作成したテストケースが十分なのかを判断することも難しい．本稿では，テストケースの作成を支援するために，入出力データ間での依存関係を用いたテストケース生成手法を提案する．提案手法では，開発者はまず XPath 式でテスト対象の入出力データ間での依存関係を記述する．次に，システムは記述された依存関係情報と WSDL 文書から取得できるデータ型の情報を用いて一貫性を持つテストデータを生成する．最後に，生成されたテストデータを用いてテストケースを作成する．また，本稿では，プラットフォーム非依存な実行履歴記録手法についても述べる．BPEL プロセスの実行履歴は，テストケースが十分かどうかを開発者が判断する際に有用である．提案手法を実装したシステムを実際に運用してもらい，評価実験を行った．

Data Dependency based Test Case Generation for BPEL Unit Testing

Kho Yee CHOY† Takashi ISHIO† Makoto MATSUSHITA† Katsuro INOUE†
Hideaki SHINOMI‡ Katsuhiko YUURA‡

† Graduate School of Information Science and Technology, Osaka University
1-3 Machikaneyama-cho, Toyonaka, Osaka 560-8531, Japan
‡ Hitachi Consulting Co., Ltd.
2-16-4 Konan Minato-ku Tokyo 108-0075, Japan

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 verification conditions for output data from the PUT. This preparation of test data can be a tedious task due to the complexity of XML data used by the PUT. Furthermore, it is difficult for developers to decide whether the created test cases are sufficient for testing the PUT. In this paper, we propose a data dependency based test case generation approach. 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 test data. 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.

1 Introduction

Web Services Business Process Execution Language (WS-BPEL or BPEL) [8] is an XML-based language designed to compose Web services [2] in realizing Service-Oriented Architecture (SOA). BPELUnit is a unit testing framework implemented by Mayer et al. [6] to facilitate unit testing of these Web service compositions. Nevertheless, BPELUnit does not provide much support in test case creation and the monitoring of the process under test (PUT). Developers have to manually prepare large amount of coherent XML data and XPath expression to compose a test case. This is a painstaking task considering the complex structure of involved XML data.

In this paper, we propose a data dependency based approach to generate test cases for BPELUnit while preserving data coherency. Developers are required to describe the data dependencies in XML Path Language (XPath) [4]. Based on type information specified in Web Services Description Language (WSDL) [3] documents, independent data are randomly generated, while dependent data are generated according to data dependencies specified by developers. We also propose a method to capture execution information of the PUT using only standard BPEL functions to help developers determine the adequacy of generated test cases.

We implemented the approach as a tool and evaluated its usefulness through experiment. Four graduate students actually used the implemented tool to create test cases for sample BPEL processes. From the result, we found that the proposed method is indeed effective in solving existing problems.

The rest of this paper is organized as follows: Section 2 further explains the related technologies of this research. Section 3 presents the details of our proposed approach and Section 4 covers the implementation. Section 5 presents the experiment and its result. Section 6 states the stance of this research with regard to other related work. Finally, Section 7 concludes the paper with conclusion and future work.

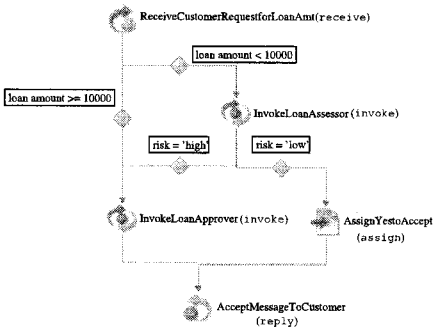

Figure 1: The Loan Approval Process

2 Background Technologies

2.1 BPEL

Web Services Business Process Execution Language (WS-BPEL or BPEL) is an XML-based language for Web service composition. BPEL uses existing XML specifications such as WSDL, XPath and XML Schema. It uses WSDL documents for Web service composition and handles XML Schema based data with XPath expression. We use the following terms in this paper:

- A *BPEL process* is a process written in BPEL.
- A *partner Web service* is the Web service invoked by a BPEL process.
- A *client* is an application or Web service that invokes the BPEL process.

Figure 1 shows the loan approval process provided as example in the BPEL specification [8]. This process will be used as example throughout the paper. This process first receives loan request from the client. If both the loan amount and the risk assessment result of the individual are low, approval is automatically granted. However, if the loan amount or the individual's risk is high, then further investigation is needed. The process invokes partner services to assess individual's risk level and to conduct further investigation. The two `invoke` activities indicates these invocations. Finally, the loan approval process replies the client with either approval or rejection.

A BPEL process is built up by basic activities and structured activities. The above example uses four basic activities: `receive`, `invoke`,

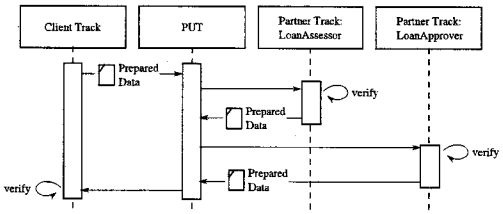

Figure 2: BPEL Unit Testing of the Loan Approval Process

`assign` and `reply`. 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. For example, a `sequence` structure is used to execute activities contained within it sequentially. Besides, scope can be used to hold related activities together.

XML Path Language (XPath) is the standard query and expression language used in BPEL to handle data in the `assign` activity. XPath is used to find nodes and extract information from XML documents.

2.2 BPELUnit Testing Framework

In BPELUnit, the PUT is deployed on server while the client and other partner Web services are simulated by the framework as client track and partner tracks. All tracks run as independent processes simultaneously when the test begins. Activities are added into these tracks to represent invocations of operations in the tracks. In the client track, an activity corresponds to the operation of the PUT. Figure 2 shows an example of the invocations of operations in a BPELUnit test case when the loan amount is low but the individual's risk is high.

Test case creation 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. 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. A verification condition is a pair of an XPath expression to be executed inside the received data and a value which the result of the

Input for the PUT:

```
<creditInfo>
  <id>12345</id>
  <name>BPEL</name>
  <amount>5000</amount>
</creditInfo>
```

Verification condition to check PUT's output in LoanAssessor track:

```
<condition>
  <expression>./creditInfo/amount</expression>
  <value>'5000'</value>
</condition>
```

Figure 3: Example of Test Data

XPath expression must match. Here, we call the XML data and verification condition *test data*. Example of test data is shown in Figure 3.

Test case is then 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.

3 Approach

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 the loan approval process in which data are simply copied from message to message. We believe that these data dependencies are simple because BPEL only provides minimum functions needed to perform data manipulation for business processes [5]. We deploy this simplicity of data dependency and propose a data dependency based test case generation method.

In our method, developers first need to understand the specification of the PUT. This is a safe assumption as BPELUnit is designed for white box testing purpose. Then, developers select a set of operations to be invoked in the test case. For this operation set, developers list out

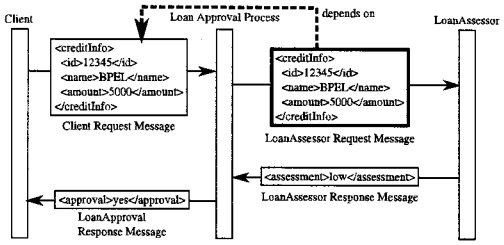

Figure 4: Example of Data Dependency

data dependencies of its input and output data. Note that the number of times an operation is invoked might depend on data, too. 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 standard query and expression language in BPEL.

Test-related data are then generated based on the data dependencies specified earlier. For the example 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 LoanAssessor track to verify that the dependent data indeed have the same values as the data they depend on. These test data are later used to generate test case. Number of activities in a track, i.e. the number of invocations of an operation might depend on data, too. This dependency is handled appropriately during test case generation.

3.1.1 Data Dependency Specification

We use XPath expression 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 one of the dependencies between the client request message (CMSG) and the LoanAssessor request message (LMSG) in Figure 4.

```
Dependent node:  (LMSG) creditInfo/id
Value          :  (CMSG) creditInfo/id
```

Note that (LMSG) and (CMSG) are inserted to clarify which message the XPath expressions operate on and not part of the XPath expressions.

Besides, data might affect the number of invocations of an operation. For example, for n results returned by a search engine, an operation have to be invoked n times in a loop. 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 Data Generation

Independent input data are input data to the PUT which do not rely on other data, thus can be generated freely. An example of independent input data is the client request message shown in Figure 4. The generation of a set of coherent test data starts with generating independent input data randomly within the constraints set by XML Schema facets.

Dependent input data are input data to the PUT which rely on other data. For example, partner Web services might copy part of the request message that they have received from the PUT into their own response messages. Dependent input data can be generated according to their relationship with other data specified in XPath expression.

Independent output data come out from the PUT and do not rely on previous data. The corresponding verification conditions have to be independently specified. This includes statically defined data in the PUT. For example, partner Web service often requires special access key to identify the caller, the key is often embedded inside the PUT and do not depend on other data.

Dependent output data are output data from the PUT which rely on other data. An example of dependent output data is the LoanAssessor request message shown in Figure 4. To verify dependent output data, the location path to the element which is to be verified is specified during dependency specification and the expected value is generated according to the specified XPath expression.

3.1.3 Test Case Generation

The set of client track and partner tracks as well as activities within them are created from the operation set specified by developers. Test data

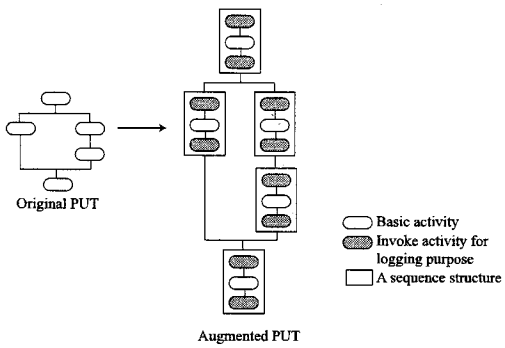

Figure 5: Augmentation of the PUT

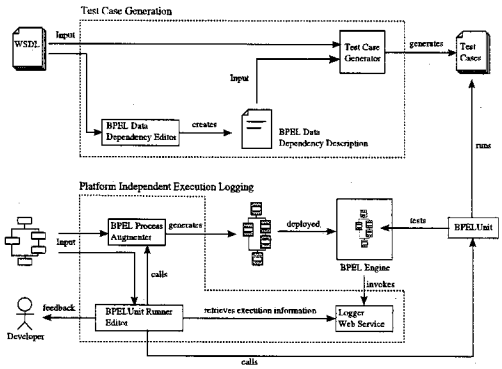

Figure 6: System Overview

generated in the previous stage are then filled into these activities accordingly to generate complete test case.

3.2 Platform Independent Execution Logging

It is important to know whether generated test cases are adequate to test the PUT. Here, we propose a method to capture execution information of the PUT by weaving standard BPEL activities which invoke an external logging service into the PUT. From the log, execution information is retrieved and analyzed to present useful information to developers.

The PUT is first augmented with `invoke` activities, with one attached before and another one after each basic activity in the PUT. This augmentation process is shown in Figure 5. These `invoke` activities invoke an external logger Web service to notify that the activity to which it is attached is going to be, or has been executed.

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 5, this grouping is shown as boxes surrounding the tuples of basic activity and `invoke` activities. The `sequence` activity guarantees that corresponding `invoke` activities is executed right before and after the activity they are attached to.

4 Implementation

Figure 6 shows the system overview of the implemented test case generation system. 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.

The test case generation subsystem comprises three components. The BPEL Data Dependency Editor is an Eclipse plug-in which enables developers to group relevant operations into operation sets and then specify data dependencies for each set. This editor gets XML data type information from the WSDL files of involved Web services. It provides a graphical interface shown in Figure 7 to help developers create the BPEL Data Dependency Description document, which is an XML document used to record data dependency information. The Test Case Generator takes as input the BPEL Data Dependency Description document and the WSDL files and output test cases for BPELUnit.

The platform independent execution logging subsystem comprises three components. The BPEL Process Augmenter takes in a BPEL process and augment it with additional `invoke` activities. 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 get called and they send log requests to the logger Web Service. The BPELUnit Runner Ed-

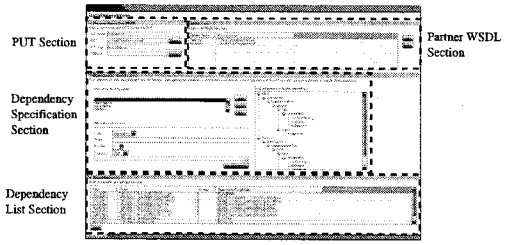

Figure 7: BPEL Data Dependency Editor

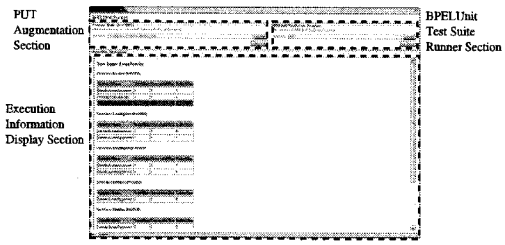

Figure 8: BPELUnit Runner Editor

itor shown in Figure 8 is a graphical front end provided to access all these components. After running the test cases, it invokes the Logger Web Service to retrieve execution information. Comparison is made between the original BPEL process and the retrieved information to produce the final report which shows the list of activities that have been successfully run, failed or not been run at all.

5 Evaluation

An experiment is conducted to find out how the implemented tool benefits developers compared to the existing tool in creating BPELUnit test cases. We compared the following points:

- **Test case characteristics.** Is there any difference in the characteristics of test cases created with the different tools?
- **Common mistakes.** What are the common mistakes made by the subjects in the experiment? Does this depend on tool being used?

5.1 Experiment Setup

Subjects. Subjects of the experiment were first year master students majoring in software engineering and have little or no experience in Web services and related technologies.

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

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

Procedure. Subjects were required to create two sets of test cases for two different BPEL processes, process A and process B, using different tools. Specifications of these processes and the involved partner services were provided in print. Besides, samples of all XML data involved are provided as clear text files. 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 on related technologies and tool usage was given to the subjects.
(2) Subjects were given time to comprehend the specifications of the services and test specifications.
(3) Subjects were required to create test cases for the two processes individually in different orders. The order in which they work was decided randomly.

Target processes and test specifications. Process A is the loan approval process used as example in previous sections. Subjects were required to create seven test cases, with varied loan amounts, risk levels and approval messages. On the other hand, process B is the Meta Search process presented in [7]. This process relays search string from the client to two search engines and returns search results after eliminating duplicates. Subjects were required to create six test cases corresponding to those presented in [7].

5.2 Result and Discussion

5.2.1 Test Case Characteristics

Table 1 shows the number of verification conditions created by each subject using different tool in the test cases for Process A. From Table 1, we can see that numbers of verification condi-

tions created using BPEL Data Dependency Editor are in all cases higher than those created using BPELUnit TestSuite 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 tend to include these checks. This is believed to be the effect of showing the whole data structure of relevant data to developers in the implemented tool, making it easy to create conditions to check those data. This is supported by comments from the subjects that said the presentation of data structure was useful.

5.2.2 Common Mistakes

Since all subjects had not much experience in BPEL and related technologies, mistakes were unavoidable.

For BPELUnit TestSuite Editor, many mistakes were made regarding XML namespace and XPath expression in test data. These include syntax errors, such as using multiple, undefined or unnecessary namespace prefixes and omission of the needed ones. Besides, skipping of elements within XPath expression and spelling mistakes in elements' names were noticed as well. Moreover, a mistake was made in which the loan approver service was asked to send the result of the risk assessor.

On the other hand, the above mistakes were almost non-existent when BPEL Data Dependency Editor was used. This is believed to be the effect of automatic insertion of location paths by the tool. However, some necessary data dependencies were left out, making the resulting test cases incomplete. This is believed to be caused by insufficient understanding of the proposed method and can be improved through practice.

Table 1: Number of Verification Conditions Created Based on Tool Being Used

Test Case	Number of conditions created			
	Existing Tool		Proposed Tool	
	Sub. 1	Sub. 4	Sub. 2	Sub. 3
1	2	1	4	4
2	3	3	7	8
3	3	3	7	8
4	2	2	4	5
5	2	2	4	5
6	2	2	4	5
7	2	2	4	5

5.3 Validity of Result

5.3.1 Strength of the Experiment Design

We randomly assigned tasks to four participants in order to avoid the influence of order and human factors. Test cases for each process was created twice using each tool so that the results are less individual dependent. Then, enough time to understand the specifications was given as that is not related to the purpose of the experiment. Questions were allowed throughout the experiment to minimize the time taken in researching about related technologies, which might negatively influence experiment result. 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.

5.3.2 Internal Validity Issues

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

5.3.3 External Validity Issues

The subjects were all beginners and the group was small. There is no firm evidence that the results of this experiment can be generalized to professional BPEL developers. Further studies which involve subjects from the professional field are needed to further evaluate the effectiveness of the proposed approach.

6 Related Work

In the work of Yan et al. [9] and Yuan et al. [10], 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 [6], the use of a common API across BPEL engine vendors was 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. We have benefited from the work of Baresi et al. [1], which proposed a non intrusive way of adding assertions as comments into BPEL processes for process monitoring purposes.

7 Conclusion

This paper presented a data dependency based approach to alleviate the effort needed in generating test cases with coherent input and output data for BPEL unit testing. A platform independent method to obtain execution information using only standard BPEL features was also proposed. We implemented our approach as a test case generation tool for BPEL unit testing and a tool to run generated test cases while collecting execution information. We evaluated the test case generation tool and found that it is useful in minimizing common mistakes made when existing tool is used.

In the future, we would like to improve the user interface of the implemented tool to be more intuitive. Also, we plan to leverage methods proposed by Yan et al. [9] and Yuan et al. [10] so that operation sets can be extracted from the PUT automatically.

Acknowledgments

This research was supported by Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Young Scientists (B) (No.18700027).

References

[1] 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.

[2] 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`.

[3] 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`.

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

[5] Frank Leymann, Dieter Roller and Satish Thatte, Goals of the BPEL4WS Specification.

[6] 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.

[7] Philip Mayer, Design and Implementation of a Framework for Testing BPEL Compositions, September 11, 2006.

[8] OASIS WSBPEL Technical Committee, Web Services Business Process Execution Language Version 2.0, 11 April 2007.

[9] 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.

[10] 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.