MODEL-BASED TESTING OF SERVICE-ORIENTED SOFTWARE

Kaushik Rana[1], Jalpa Ramavat[2], Durga Prasad Mohapatra[3]
[1,2]Computer Engineering Department, Vishwakarma Government Engineering College, Chandkheda, Gujarat, India,
[3]Department of Computer Science &Engineering, NIT Rourkela, Odisha, India

Abstract
Nowadays, many organizations are using SOA to increase the adaptability of their systems to react quickly to changes occurring in their environments. Due to the dynamic nature of SOA, its testing frequency is high. This results in an expensive process of test path construction and generation of test cases, each time SOA-based software is tested. While testing SOA-based software, we have to generate test paths and test data values for each of the services involved in the business workflow. In this scenario, generation, and identification of test paths and test cases have become challenging factors as the raw code for web services and the associated test scripts are proprietary to the service providers and invisible to the tester. This situation can be avoided, if we generate test paths and test cases from the models describing SOA-based software. This paper introduce a novel technique to test SOA-based software using BPMN diagram.
Index Terms: BPMN, Model, Service-Oriented Software, Testing.

I. INTRODUCTION

Nowadays, many organizations are looking forward to increase the adaptability of their systems to react quickly to changes occurring in their environments. This requirement of system adaptability is best addressed by service-oriented architecture (SOA), as it facilitates the availability of environmental services provided by the third parties. This feature of SOA rapidly increases the adoption of SOA by different organizations. Hence, service-oriented architecture (SOA) is being used in design, development, deployment and management of different business services. These business services in SOA can also be used by other composite applications through publishing and binding interfaces. SOA integrates heterogeneous services and enables them for effectively exchanging information among member enterprises to process remote orders, inquiries, payroll, billing, HR and information delivery etc.

Due to the dynamic nature of SOA, its testing frequency is high. This results in an expensive process of test path construction and generation of test cases, each time an SOA-based software is tested. While testing SOA-based software, we have to generate test paths and test data values for each of the services involved in the business workflow. The assignment of test data values uses six different assertion types such as existing data value from the same interface, a constant value, a set of alternate values, a value range, a concatenated value and a computed value [7]. To enhance the efficiency of testing, wrong/faulty data plays a greater role. The use of faulty data in testing SOA-based software increases the fault tolerance of SOA-based software.

In SOA, faulty data can be categorized into two sets with two different perspectives such as (i) to test the services in isolation and (ii) to test the service as a component in the system environment [8].

In this scenario, generation, and identification of test paths and test cases have become challenging factors as the raw code for web services and the associated test scripts are proprietary to the service providers and invisible to the tester. This situation can be avoided, if we

generate test paths and test cases from the *models* describing SOA-based software.

II. OUR APPROACH: TESTING OF SERVICE-ORIENTED SOFTWARE USING BPMN DIAGRAM

In this section, we propose a model-based approach for testing SOA-based software using BPMN diagram. The business process modeling notation (BPMN) is a graphical notation that depicts the steps in a business process. It also depicts the end to end flow of a business process. The notation has been specifically designed to coordinate the sequence of business processes or services and the messages that flow between different processes or services present in a related set of activities [3]. The main advantages of using a BPMN model for testing of service-oriented software are as follows:

1. It describes the services efficiently in order to analyze, and understand the SOA-based software.

2. It also identifies which test paths need to be run to execute a particular business flow.

This BPMN model is usually designed and created in the early stage of software development life cycle. However, test case data generation can be made parallel to software development in order to reduce time and effort. Thus, testers will have time to pay attention to test the software before delivery.

A BPMN diagram contains four basic element categories (i) *flow objects*: these are the events, activities and gateways (ii) *connecting objects*: these are the sequence flow, message flow and association (iii) *swim lanes*: these are the pools and lanes (iv) *artifacts*: these are the data objects, group and annotations. These four categories enable the creation of simple business process diagrams (BPDs). Even BPDs also permit making new types of flow objects or artifacts, to make the diagram more understandable. This specification also includes semantic information related to SOA-based software.

BPMN diagram is widely used in business process modeling (BPM). BPM is the activity of representing the enterprise processes so that, the current process may be analyzed and improved. BPMN model consists of simple diagrams constructed from a limited set of graphical elements such as flow objects, connecting objects, swimlanes, and artifacts. While BPMN shows the flow of data (messages), and the association of data artifacts to activities, it is not a data flow diagram. The BPMN diagram can also be used to test service-oriented software.

Our proposal for testing can be visualized as shown in Fig. 2.1.

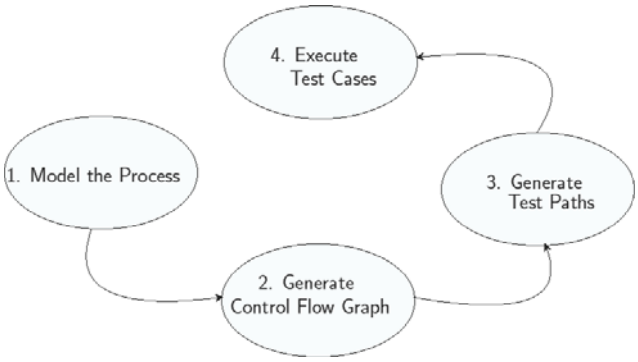

Figure 2.1 Proposed testing approach

Below, we present our proposed approach as an algorithmic pseudo-code.

Algorithm 1	**Testing of Service-Oriented Software Using BPMN Diagram (TSOSBD)**
1.	Model the Process: design the BPMN diagram of the service-oriented software.
2.	Model the Process: design the BPMN diagram of the service-oriented software.
3.	Generate Test Paths: apply depth first search (DFS) method for generating the test paths.
4.	Execute Test Cases: apply test cases on generated test paths.

Let us follow the steps (represented by circles) one by one. The first step portrayed is *model the process*. This is an external task, in which a general task process, like travel reservation, shopping item etc. is being modeled using business process modeling notation (BPMN). Then, we export the BPMN file in *.bpmn* format. The XML DOM parser reads it and transforms it's relevant data into a directed graph, this stage

is called, as *Generate Control Flow Graph*. Then, the test paths are generated from CFG using depth first search (DFS) method. Finally, the test cases are executed on the generated test paths.

1. Model the Process

The service call specific control flow, leads us to choose a graph-based modeling approach for the business process. Each service or service call request in our model will be represented as an individual node. In the case of service-oriented software, where a service may call other services can be represented as another node and an edge represents the control flow from one service node to another service node. This inherent nature of service-oriented software leads us to choose a graph-based modeling approach to represent the business processes. This also has to do with the increased usage of business process modeling languages such as BPMN [3] and WS-BPEL [4].

Most companies have already modeled their business processes with any one of these two languages, which makes easy the model generation, often reduces execution costs and effectively communicates the business processes in a standard manner. A BPMN defines a business process diagram (BPD), based on a flowcharting technique tailored for creating graphical models of business processes. The web service business process execution language (WS-BPEL) [4], is an OASIS's standard executable language for specifying actions within business processes with web services. A business process model does not necessarily have to be implemented as an automated business process in a process execution language. Even by design, there are some limitations on the process topologies that can be described in WS-BPEL, so it is possible to represent processes in BPMN that cannot be mapped to WS-BPEL. Moreover, there are concepts, such as *ad-hoc sub-proceses*, that BPMN can represent that may not be implemented with any technology [3]. Since there is a serious advantage of BPMN and as it supports convenient graph-based modeling, we decided to use BPMN with the help of Bizagi Modeler, a freeware tool [1]. The bizagi modeler is a modeling tool much like Microsoft Visio Professional [2] or Sparxsystems Enterprise Architect 12.1 [6], but more targeted towards the creation of BPMN models. It supports BPMN model exchange through export or import with *.bpmn* format.

1.1 A BPMN Example: Online Shopping System (OSS)

The online shopping system (OSS) described in Section 5.1 can be represented as a set of business services in a BPMN diagram as shown in Fig. 2.2. We have used Bizagi Modeler tool [1] to design OSS. It shows various *service tasks* such as *product registration*, *courier company registration*, *login*, *sign up*, *third-party login*, *search product*, add_to_cart, *make_payment* and *make_courier*. These service tasks are being provided by a *pool* or more precisely entities such as *customer*, *payment gateway provider*, *online retailer*, *third-party login provider*, *product seller*, *shipping and courier company*. Other tasks are *product registration request*, *courier company registration request*, *login request*, *sign up request*, *third-party login request*, *search product request*, *make_payment request* and *make_courier request*. The information seen on the edges might represent one of the following two different things depending on it's source. If it is a service or task, it represents it's output data, on the other hand if it is a gateway, it contains the condition that makes the control flow in that direction.

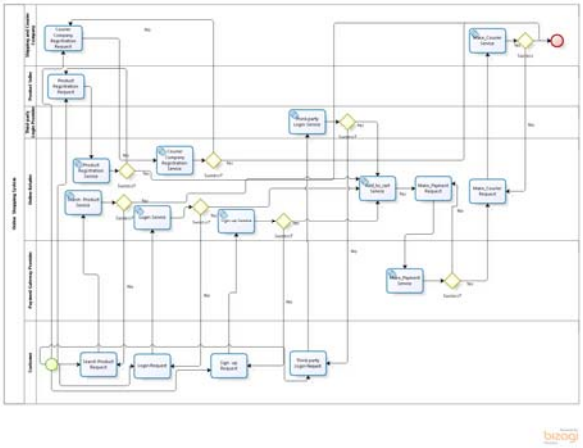

Figure 2.2 A BPMN diagram for ``online shopping system (OSS)"

Once the business process modeling is completed, it is exported in .bpmn format as shown in Fig. 2.3.

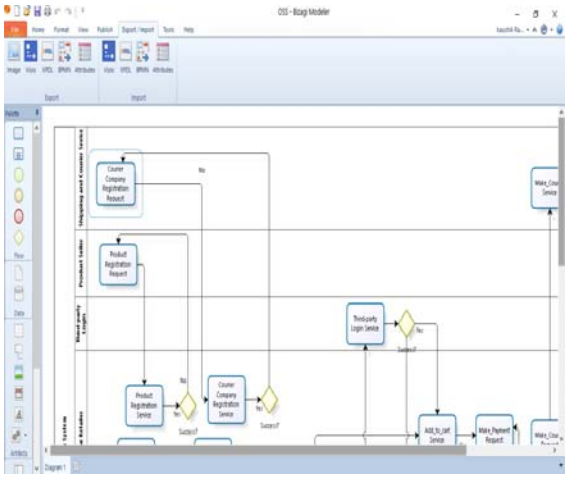

Figure 2.3 Exporting BPMN with bizagi modeler

2. Generate Control Flow Graph (CFG)

First, we briefly describe our CFG generation algorithm. Then, we present the pseudo-code of the algorithm. Subsequently, we discuss the complexity of our algorithm.

2.1.1 Overview of the CFG Generation Algorithm

Before execution of a business process, the CFG of a business process is constructed statically. It describes the sequence in which different *tasks* or *service tasks* of a business process get executed. Stating, in other words, a control flow graph (CFG) describes how the control flows through the business processes. Before presenting our proposed algorithm, few more definitions that would be used in our algorithm are introduced in this section.

Definition 2.1 Unique Node Number Assignment:

A unique node number is an incremental assignment of numeric numbers to each task of BPMN diagram.

Definition 2.2 BPMN Version (V):

Version (V) of a BPMN diagram comes into picture if an evolution happens in the BPMN diagram causing a change in the business flow of the system. This evolution may take effect due to the changes in the business service specification or the changes to the customer requirements. Once the changes have been incorporated, then the corresponding task of the BPMN diagram are updated and the BPMN diagram is redrawn resulting in a new version

(say V1 to V2). For simplification of the proposed approach, we assume that the BPMN has initial version V1.

In the algorithm, we first create two special nodes *start* and *stop* corresponding to nodes *startEvent* and *endEvent* of BPMN model to define the start and end of business flow. Then after, we extract the root node *process* and it's subchild nodes *task* or *serviceTask* and create nodes in CFG and assign them unique node numbers and attributes. We add a control flow edge if *textnode* value of *outgoing* element of subchild *ni* equals *textnode* value of *incoming* element of subchild *nj*. We now present our CFG generation algorithm for our business process in the form of pseudo-code as below:

Algorithm 2	**Control Flow Graph (CFG) Generation Algorithm.**
Input :	A BPMN model // in .bpmn format
Output :	Control Flow Graph
1.	Node Construction
(a)	Create two special nodes *start* and *stop* corresponding to nodes *startEvent* and *endEvent*
(b)	For each root node *process* do the followings For each subchild node *task* or *serviceTask* node do the following i. Create node *ni* ii. Assign the node *ni* with unique node number.
2.	Add control flow edges
(b)	For each root node *process*, do the following For each subchild node *task* or *serviceTask* node do the following
i.	Add control flow edge from node (*ni*,*nj*), if *textnode* value of *outgoing* element of subchild *ni* equals *textnode* value of *incoming* element of subchild *nj* .

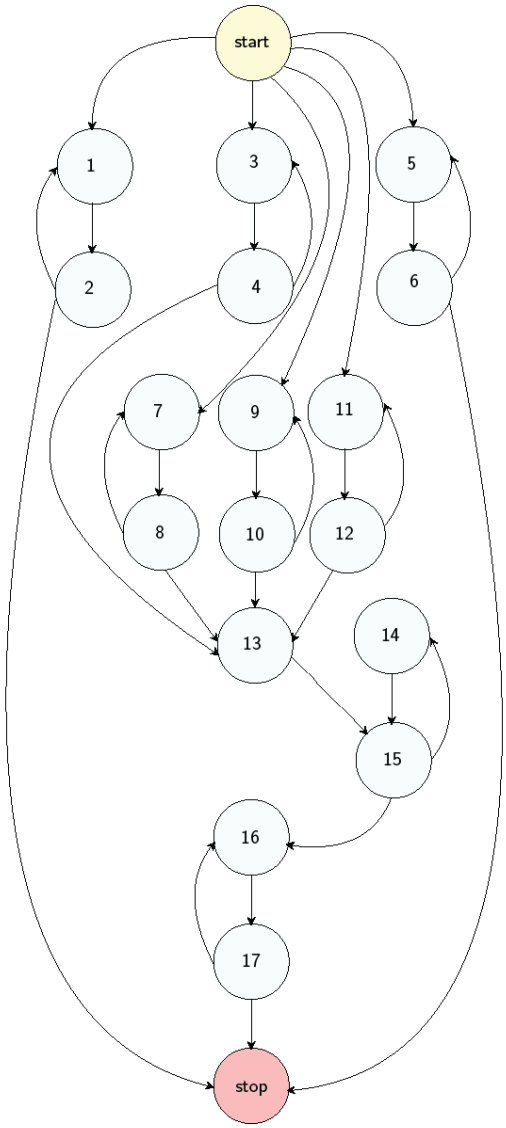

Figure 2.4 Control flow graph of ``Online Shopping System (OSS)"

Fig. 2.4 shows the control flow graph of Fig. 2.2. The different numbered tasks serve as the nodes of the control flow graph. An edge from one node to another exists if the execution of the task representing the first node can result in the transfer of control to another node.

2.1 Complexity Analysis of CFG Generation Algorithm

(1) Time complexity

To compute the time complexity of our CFG generation algorithm, we consider each step of the algorithm. Step *1(a)* requires O(1) constant time. Step *1(b)i* requires O(1) constant time, Step *1(b)i}* requires $O(n^2)$ time and finally the Step *2(a)i* requires $O(n^2)$ time. Hence the time complexity of our algorithm is $O(n^2)$, where *n* is the input size.

(2) Space complexity

The space complexity of our CFG generation algorithm would be O(n), since XML DOM parser maintains a hash table for CFG, and the space complexity of every reasonable hash table is O(n), where *n* is the input size.

2.2 Implementation of CFG Generation Algorithm

As we said in the previous section, Bizagi Modeler exports a file with all the information we need to create a control flow graph. The XML DOM parser extracts and stores the BPMN elements in a data structure in such a way that it facilitates control flow graph generation, graph searches, and further test case generation. A BPMN file fragment is shown in Fig. 2.5. In order to search a service used in the BPMN diagram, the XML DOM parser traverses the DOM tree to find *serviceTask* element node. While traversing the services or tasks it also stores the attribute values of *id*, *name* and other elements such as *incoming*, *outgoing*, which are the element nodes needed for generating CFG. Fig. 2.6 shows the internal data structure maintained by XML DOM parser. The unique node number is assigned by the parser.

The parser compares each *outgoing* element with *incoming* element to determine the possible control flows among the services or tasks. To easily describe the control flow from *outgoing* to *incoming* elements, we highlight the common *id*s with same color. The parser uses these elements and builds the CFG.

```
<serviceTask id="Id_25d82d6f-2809-4678-965f-882a0b64ebc9" name="Product Registration Service">
  <documentation />
  <extensionElements>
    <bizagi:BizagiExtensions xmlns:bizagi="http://www.bizagi.com/bpmn20">
      <bizagi:BizagiProperties>
        <bizagi:BizagiProperty name="bgColor" value="#ECEFFF" />
        <bizagi:BizagiProperty name="borderColor" value="#03689A" />
        <bizagi:BizagiProperty name="runtimeProperties" value="{&amp;quot;cost&amp;quot;:0,&amp;quot;prior
      </bizagi:BizagiProperties>
    </bizagi:BizagiExtensions>
  </extensionElements>
  <incoming>Id_8a28e5d1-5a85-4f1b-9230-9338c60ad610</incoming>
  <outgoing>Id_0c07e221-1be7-4703-b604-302ec8210737</outgoing>
</serviceTask>
```

Figure 2.5 A BPMN file fragment

Figure 2.6 Internal data structure of XML DOM parser

3. Generate Test Paths

Once we have generated the CFG, it is the time to generate test paths. Test paths can be generated by applying depth first search (DFS) method or breadth first search (BFS) method. In DFS, the path is generated by starting with the root node and exploring as far as possible along each branch before backtracking. In BFS, the path is generated by exploring all the nodes at one level completely and then moving into the next level and so on. The limitation of BFS is that it can lead to generate an exponential number of test paths which will increase the time complexity. The paths to be tested are coming out in exponential, out of which very few are useful, even there is a need of testing only basic paths. With this problem, and considering the efficiency of DFS method to generate test path, we use DFS method to generate test paths from BPMN diagram. By using this approach, the useless test paths are eliminated which in turn reduces the time complexity. The generated test paths for the CFG given in Fig. 2.4 using DFS are given below:

1. start →1 → 2 → stop.
2. start →3 → 4 → 13 →15 → 16 → 17 → stop.
3. start →7 → 8 → 13 →15 → 16 → 17 → stop.
4. start →9 → 10 → 13 →15 → 16 → 17 → stop.
5. start →11 → 12 → 13 →15 → 16 → 17 → stop.
6. start →5 → 6→ stop.
7. start →3 → 4 → 13 →15 → 14 →15 → 16 → 17→ stop.
8. start →7 → 8 → 13 →15 → 14 →15 → 16 → 17→ stop.
9. start →9 → 10 → 13 →15 → 14 →15 → 16 → 17→ stop.
10. start →11→ 12 → 13 →15 → 14 →15 → 16 →17→ stop

4. Execute Test Cases

The main goal of this step is to generate test case data, and execute the test cases for each business process based on the generated test paths.

We choose a path coverage-based test case generation strategy where test cases are designed such that all *linearly independent paths* in the business process are executed at least once. A linearly independent path is any path through the program that introduces at least one new edge that is not included in any other linearly independent path [5].

Our goal is to provide semi-automatically input values which will exercise each linearly independent path previously created. To test a specific path, a number of service tasks or tasks need to be executed. In regular cases, most service tasks or tasks require some type of input data. The only way to do it is by whitening up a bit for testing. In other terms, the service providers will have to give a description of what are the possible inputs and outputs of a service tasks or tasks. Let the service provider declares inputs, data types and restrictions for each of the service or task as shown in Table II.I. When we have sufficient information of tasks or service tasks, we can discover what inputs will fit, i.e, what input data will produce results matching those restrictions and corresponding test paths.

Once we have such information as provided in Table II.II we can supply test cases in such a way that each identified test path is covered. For example, we can exercise *test path 1* by supplying test case set {*product name=mi3,*

quantity=1, price=12000, seller name=xiaomi, contact address=china} as input, and observe the test case status as *valid*. This test path covers the unique node numbers 1 and 2 of CFG. Similarly, we can test all the test paths by executing appropriate test cases and check the output. These test cases along with their status are shown in Table II.II.

TABLE II.I: Inputs, data types and restrictions of service or task

Unique Node Number	Element Name	Data Type	XSD Constraints	XSD Constraints Value
1 & 2	product name	xs:string	minOccurs	1
1 & 2	quantity	xs:int	minOccurs	1
1 & 2	price	xs:int	minOccurs	1
1 & 2	seller name	xs:string	minOccurs	1
1 & 2	contact address	xs:string	minOccurs	1
3 & 4	product name	xs:string	minOccurs	1
3 & 4	pincode	xs:int	minOccurs	1
3 & 4	quantity	xs:int	minOccurs	1
5 & 6	shipping company name	xs:string	minOccurs	1
5 & 6	address	xs:string	minOccurs	1
5 & 6	charges	xs:int	minOccurs	1
7 & 8	username	xs:string	minOccurs	1
7 & 8	password	xs:string	minOccurs	1
9 & 10	mobile no	xs:int	minOccurs	1
11 & 12	tusername	xs:string	minOccurs	1
11 & 12	tpassword	xs:string	minOccurs	1
13	product name	xs:string	minOccurs	1
13	quantity	xs:int	minOccurs	1
13	customer name	xs:string	minOccurs	1
13	address	xs:string	minOccurs	1
13	contact no	xs:int	minOccurs	1
14 & 15	card type	xs:string	minOccurs	1
14 & 15	card no	xs:int	minOccurs	1
14 & 15	expiry date	xs:date	minOccurs	1
14 & 15	PIN	xs:int	minOccurs	1
14 & 15	OTP	xs:int	minOccurs	1
16 & 17	customer name	xs:string	minOccurs	1
16 & 17	address	xs:string	minOccurs	1
16 & 17	product name	xs:string	minOccurs	1
16 & 17	contact no	xs:int	minOccurs	1

TABLE II.II: Inputs, data types and restrictions of service or task

Unique Node Number	Element Name	Data Type	XSD Constraints	XSD Constraints Value	Input	Test Case Status
1 & 2	product name	xs:string	minOccurs	1	mi3	Valid
1 & 2	quantity	xs:int	minOccurs	1	mi3	Invalid
1 & 2	price	xs:int	minOccurs	1	12000	Valid
1 & 2	seller name	xs:string	minOccurs	1	1	Invalid
1 & 2	contact address	xs:string	minOccurs	1	china	Valid
3 & 4	product name	xs:string	minOccurs	1	1	Invalid
3 & 4	pincode	xs:int	minOccurs	1	384001	Valid
3 & 4	quantity	xs:int	minOccurs	1	xiaomi	Invalid
5 & 6	shipping company name	xs:string	minOccurs	1	DHL	Valid
5 & 6	address	xs:string	minOccurs	1	1	Invalid
5 & 6	charges	xs:int	minOccurs	1	5000	Valid
7 & 8	username	xs:string	minOccurs	1	ABC	Valid
7 & 8	password	xs:string	minOccurs	1	123	Valid
9 & 10	mobile no	xs:int	minOccurs	1	9725577940	Valid
11 & 12	tusername	xs:string	minOccurs	1	XYZ	Valid
11 & 12	tpassword	xs:string	minOccurs	1	test123	Valid
13	product name	xs:string	minOccurs	1	mi3	Valid
13	quantity	xs:int	minOccurs	1	1	Valid
13	customer name	xs:string	minOccurs	1	KKR	Valid
13	address	xs:string	minOccurs	1	india	Valid
13	contact no	xs:int	minOccurs	1	9726677988	Valid
14 & 15	card type	xs:string	minOccurs	1	CREDIT	Valid
14 & 15	card no	xs:int	minOccurs	1	112233	Valid
14 & 15	expiry date	xs:date	minOccurs	1	2020-05-05	Valid
14 & 15	PIN	xs:int	minOccurs	1	112233	Valid
14 & 15	OTP	xs:int	minOccurs	1	1334	Valid
16 & 17	customer name	xs:string	minOccurs	1	KKR	Valid
16 & 17	address	xs:string	minOccurs	1	india	Valid
16 & 17	product name	xs:string	minOccurs	1	mi3	Valid
16 & 17	contact no	xs:int	minOccurs	1	9726677988	Valid

III. CONCLUSION

In this paper we illustrated the testing approaches for SOA-based software using BPMN diagrams. First, we described our testing algorithm for service-oriented software using BPMN diagram. In that we presented a CFG generation algorithm and its implementation. Finally, we tested the test paths of CFG by applying various test cases.

REFERENCES

[1] Bizagi Modeler Tool, [online]. Available: http://www.bizagi.com, [accessed on 18/03/2014].

[2] Microsoft Visio Professional 2016, [online]. Available: https://www.microsoft.com/en-in/evalcenter/evaluate-visio-professional-2016, [accessed on 18/03/2014].

[3] Object Management Group (OMG) Business Process Model and Notation (BPMN), [online]. Available: http://www.bpmn.org/, [accessed on 18/03/2014].

[4] OASIS Web Services Business Process Execution Language (WS-BPEL), [online]. Available: https://www.oasis-open.org/, [accessed on 18/03/2014].
[5] Rajib Mall, Fundamentals of Software Engineering, PHI Learning Private Limited, second edition, February 2009.
[6] Sparxsystems Enterprise Architect 12.1, [online]. Available: http://www.sparxsystems.com/products/ea/12.1/index.html, [accessed on 18/03/2014].
[7] Sneed, H. M., and Huang, S., "WsdlTest- A Tool for Testing Web Services", In Proceedings of Eighth International Symposium on Web Site Evolution, 2006.
[8] Zhang, J., and Qiu, R., "Fault Injection-based Test Case Generation for SOA-oriented Software", In Proceedings of IEEE International Conference on Service Operations and Logistics, and Informatics, Pages 1070-1078, 2006.