Journal of Engineering and Applied Sciences 14 (21): 7828-7840, 2019
ISSN: 1816-949X

CPN-based Test Case Generation Approach for Testing BPEL-based Web Services Composition

[1,4]Hosney Jahan, [2,4]S.M. Hasan Mahmud, [3]Md Altab Hossin and [2]Sheak Rashed Haider Noori
[1]College of Computer Science, Sichuan University, Chengdu, China
[2]Faculty of Science and Information Technology, Daffodil International University, Dhaka, Bangladesh
[3]Department of Management Science and Engineering, University of Electronic Science and Technology of China, Chengdu, China
[4]SEEDSlab (Software Intelligence and Data Science Research Group), Dhaka, Bangladesh
hasan.swe@daffodilvarsity.edu.bd

Abstract: Business Process Execution Language (BPEL) is an up-and-coming language which depicts the composition of web services in the structure of business processes. However, the interaction among the participating services can make the BPEL code significantly complicated. Therefore, it is necessary to find the interaction inconsistencies among the BPEL processes. Testing provides a solution to improve the quality of the BPEL code. The formal method that can handle all the requirements for testing BPEL-based web service composition is Colored Petri Net (CPN) which provides a strong mathematical background for the modeling, verification and testing of the BPEL structures. This study presents an approach to generate test cases for testing BPEL-based web services composition using CPN in an effective and feasible manner. Our approach combines a reachability graph and a control flow graph to generate feasible test cases by reducing the associated time cost. A prototype tool has been implemented based on our proposed approach and its validity is empirically evaluated with two case studies. The effectiveness of the prospective approach is measured in terms of its fault detection capability. Furthermore, the results of the proposed approach are compared with state-of-the-art approaches which demonstrates that the approach is both effective and feasible than existing approaches.

Key words: BPEL, web service composition, CPN, test path generation, test data generation, test case generation

INTRODUCTION

Web services are self-governing and self-describing standard applications which can be invoked, published and located over the web. However, in many cases, a single service is not sufficient to respond to the user's request, it is desirable to develop new functionalities that integrate the existing web services in order to accomplish the business requirements. This limitation has triggered the notion of web services composition. BPEL is an industry standard language to model the behavior of web services composition to describe high level business processes (Aalst and Stahl, 1999). Since, the composition of web services is an error prone task, the BPEL structure may contain defects or faulty behavior which can cause inconsistencies from its expected behavior. Therefore, testing is mandatory to avoid the inconsistent interaction among the BPEL processes and ensure its quality. However, BPEL itself does not provide a proper formal description which makes it difficult to formally verify the composition of business processes accurately. The dynamic features related to BPEL processes also make the behavior analysis and testing task significantly complicated.

There are several formal methods Bernot *et al.* (1991), Stocks and Carrington (1996), Liu and Nakajima (2011) that have been used for software testing or more specifically testing BPEL processes. The popular modeling formalisms (Hierons *et al.*, 2009) include FSMs, LTS and Petri nets (Murata, 1989). In comparison with other modeling techniques, CPN (Jensen, 1996; Jensen and Kristensen, 2009) is more scalable and expressive. Moreover, some formal method such as FSM (Rao *et al.*, 2016) is typically only control-oriented (Xu *et al.*, 2015) and variants require a large computation cost (Watanabe and Kudoh, 1995) while CPN provides both control and data dependencies.

Corresponding Author: S.M Hasan Mahmud, SEEDSlab (Software Intelligence and Data Science Research Group), Dhaka, Bangladesh, hasan.swe@daffodilvarsity.edu.bd

Furthermore, CPN is capable of minimizing the computation cost and size of the test suite. CPN provides a strong formal semantics which is able to handle the dynamism, synchronization and concurrency of the BPEL-based web services composition. Colored Petri net's high expressive power and better formal capabilities to define and inspect complicated behaviors of a system motivate us to use CPN for the modeling of the BPEL processes. Moreover, a CPN Model has the ability to simulate dynamically, directed by the data-dependent control flow of system behaviors which provides a strong foundation in this research to generate test cases. This study presents a test case generation approach for testing BPEL-based web service composition applying colored Petri net and presents a prototype tool demonstrating the feasibility of the approach.

Besides CPN, we also used the notion of two other approaches, Control Flow Graph (CFG) and Reachability Graph (RG). Both these approaches acquire some advantages and disadvantages. For example, CFG based techniques may contain infeasible paths which leads to extra work in order to handle the problem and makes the approach time consuming and expensive. On the other hand, though, reachability graphs contain only the feasible paths, the size of a reachability graph can become very large which may increase the required time to generate test cases. In this study, we combined these two approaches and taken only their advantages to generate feasible test cases by reducing the associated time cost. We firstly, constructed a Reachability Graph (RG) from the CPN Model of the BPEL for generating only the feasible paths and then constructed a CFG from the reachability graph, based on some proposed guideline in order to minimize its size and speed up the test case generation process.

Literature review: We divide the state-of-the-art approaches based on BPEL and CPN into three parts. The first part contains the related works on software testing using CPN, the second part contains the works done on testing BPEL-based web services composition and the third part contains the works done on testing BPEL based web services composition using CPN.

Related work on software testing using CPN: CPN based techniques can be successfully applied to various information processing systems such as asynchronous, parallel, concurrent, distributed, dynamical and so on (Murata, 1989). Many researchers have worked on this topic and several tools and techniques have been proposed to facilitate the test case generation process using CPN. By Xu *et al.* (2015) presented an automated test case generation tool MISTA for real world software systems based on reachability analysis. Watanabe and Kudoh (1995) proposed two conformance based test suite generation techniques for concurrent systems: CPN-tree and CPN-graph. The advantages of the two methods are shown through the evaluation of the test suite length reduction by equivalent markings. Zhu and He (2002) proposed a methodology of testing high level petri net on the general theory of concurrent systems. Four types of testing strategies are investigated in this study based on reachability analysis. Another simple test case generation method for model-based testing is proposed by Cai *et al.* (2011) where the test cases are generated directly from the state space of the CPN Model. Unlike these methods our approach firstly constructs a CFG from the state space to minimize its size and then generates test cases from the CFG which expedites the test case generation and reduces the required time.

Farooq *et al.* (2008) presented a control flow based test sequence generation method from the colored Petri net model. The proposed method converts the AD (Activity Diagram) activities to a CPN Model. Afterwards, a random walk algorithm is applied to create test sequences from the CPN Model. However, this approach tends to be inefficient for covering a large graph quickly. Besides, it could also generate redundant test cases due to randomly covering the model. Our proposed approach does not generate redundant test cases, since, it covers one path only once by using a depth first search algorithm. Reza and Kerlin (2011) proposed a method in which test cases are generated from various scenarios of a system. These scenarios are then converted into Constraint based Modular Petri Net (CMPN) and finally, test cases are generated from the CMPN. This method works fine with component based testing but when it comes to integration testing it shows some inefficiency with redundant tests. Liu *et al.* (2011) presented a technique where the ioco theory and CP-nets modeling technique is integrated to develop a CP-nets model based conformance test case generation approach.

Related work on testing BPEL-based web service composition: Yan *et al.* (2006) proposed an approach to generate test cases directly from BPEL. They modeled a BPEL process as an Extended Control Flow Graph (XCFG) where the XCFG edges contain BPEL activities and maintain the execution of activities. Afterwards, they generated test cases from the CFG by applying several coverage criteria. Similarly, Yuan *et al.* (2006) used another extension of CFG and named it BPEL Flow Graph (BFG) to represent a WS-BPEL program as a graphical model which contains both the control and data flows of a BPEL program. Test paths are then generated by traversing the BFG based on some coverage criteria and test data are generated by solving the constraints. Ni *et al.* (2013) presented a test case generation method

where a test case is represented as a sequence of messages received by the BPEL. They modelled the BPEL program as a Message-Sequence Graph (MSG) and generates test cases from the MSG.

All these methods are prone to generate redundant and infeasible paths and require further research to solve this issue, thus, the approaches become time consuming. Our approach, on the other hand could avoid the infeasible path problem by constructing a reachability graph from the CPN Model as the reachability graph contains only the feasible paths which makes our approach time effective and feasible.

Related work on testing BPEL-based web service composition using CPN: There are several methods that have been proposed for the transformation and verification of BPEL processes into CPN Model (Yang *et al*., 2005a, b; Kang *et al*., 2007; Dong *et al*., 2006). However, all these proposed approaches are mainly based on the composition and transformation of BPEL into CPN Model but gives less attention to the verification process. Our approach provides a solid foundation of transformation, verification and testing of the BPEL processes by using CPN. Several tools developed for the modeling and verification of the web services composition includes CPN Tools (Ratzer *et al*., 2003) (poses++ 2010) and (Schmidt, 2000), etc.

Dong *et al*. (2006) presented a systematic approach for testing web services from WSDL specification based on the fault coverage. A prototype system is developed for automatic test sequence generation using the existing tool poses++. Wang and Yang (2014) introduced a new approach of test case generation of web service composition which can deal with the BPEL features. The proposed method simplifies the CPN Model of the BPEL program as one which only has control flows and then generates test cases from it. Yi and Kochut (2004) proposed a unified model for the specification of the conversation protocol and process the composition based on CP-nets. The model enables a comprehensive analysis of the service process. The conversation protocol can be derived automatically where the verification is based on the CPN state space tool.

Among the above techniques, reachability analysis is the most common and effective one for test case generation. However, it suffers from a fundamental problem of state space explosion. On the other hand, CFG based test case generation methods may contain infeasible paths. To make the best of advantages of the reachability graph and control flow graphs in this research, we proposed an approach to generate test cases for BPEL-based web service composition using CPN by combining the reachability analysis and control flow graph together in order to avoid the infeasible path problem and reduce the time cost associated with the test case generation.

Test case generation approach: One of the most critical components of software testing (Myers, 2004; Miller *et al*., 1992; Ammann and Offutt, 2008) is the construction of test cases. In this research, we proposed a test case generation approach based on CPN which is able to generate test cases in a more formal and systematic way. This section presents our proposed test case generation approach.

Overview of the approach: As shown in Fig. 1, the approach consists of four steps which are described in details in the following subsections. Here, we give an overview of the approach.

Transformation of BPEL into CPN: The objective of this step is to unfold the complex structures of the BPEL code more explicitly. To do, so, we transformed the BPEL processes into CPN as CPN provides more formal and graphical notation.

Reachability graph construction and verification of the CPN Model: In order to ensure the generation of feasible test cases, the CPN model should be verified first. To achieve this goal, we constructed a reachability graph

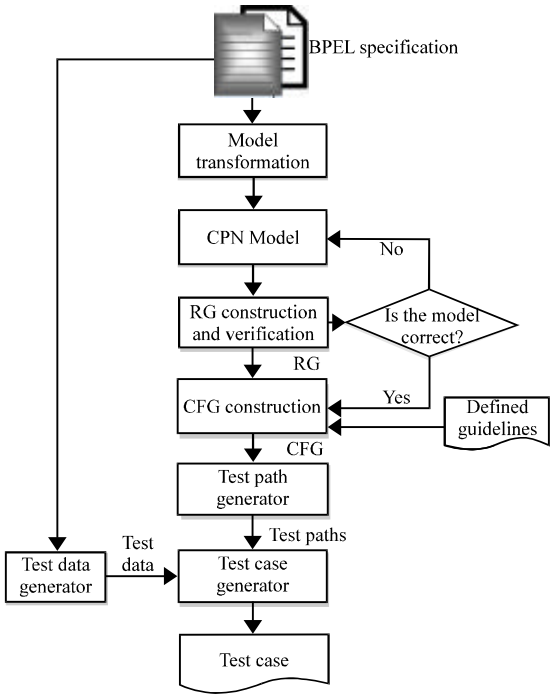

Fig. 1: Schematic diagram of the proposed approach

from the CPN Model and verified its properties to check and repair the design errors by using the CPN Tool (Ratzer *et al*., 2003).

CFG construction from the reachability graph: In this step, we propose four guidelines to construct a CFG from the reachability graph. These guidelines construct the CFG by eliminating some internal unimportant states and transitions from the RG which can avoid the state space explosion problem.

Generation of test cases: This step contains three parts; generation of test paths, generation of test data and generation of test cases. Firstly, we generate the test paths by traversing the CFG then generate the test data by solving the constraints specified in the BPEL code afterwards, the test cases are generated by combining the test paths and test data together. Since, the generated test cases are abstract we transformed them into a specific programming language to make them executable.

Transformation of BPEL-based web service composition into CPN: A colored Petri net is a tuple CPN = < Σ, P, T, A, E, M_0> (Jahan *et al*., 2016) where:

- Σ is a finite set of non-empty types, called color sets. The color sets are associated with the term token where each token value belongs to a type
- P is a finite set of places, denoted by the circles
- T is a finite set of transitions, denoted by the rectangles
- A is a finite set of directed arcs which connects places to transitions and transitions to places, i.e., A⊆ P×TUT×P
- E is an arc expression function
- M_0 is the initial marking, i.e., the initial allotment of tokens

Several tools have been developed to practically construct a CPN Model from the BPEL-based web services composition and formally verify the constructed CPN Models. Among them, in this research, we used the CPN tool for the construction and verification of the CPN Model from the BPEL. The transformation process is summarized into the following three steps.

Step 1: Firstly, the data types or color sets of the tokens are determined based on the part types specified in the WSDL file of the BPEL specification. For example, the color sets used to model the loan approval process are defined as follows:

```
colset amount = with<10000|>10000
var s: amount
colset name = string
colset request = product name*Name
var req : request
colset state = bool
var t, m, n:state
colset risk:with low|high
var risk: risk
colset approval = with yes|no
var approve, a:approval
```

Here, the color set "Amount" is declared as the enumerated type which includes two values in order to denote the two different inputs (i.e., one input is <10000 and other one is ≥10000) of the process. The color set "Request" denotes the product of the color set "Name" where, the first color set represents the information of the customer and the second one represents the amount being requested. a"State" is a color set of bool type. "t", "m" and "n" are the variables of type "State" which represents the flow of resources within the model. The other color sets are determined accordingly; however, we also used some low level data types in order to simplify the transformation and model the characteristics of the BPEL program more accurately.

Step 2: In the second step, we constructed CPN Models for each of the activities of the BPEL code based on their input-output message relations i.e., the input message of one activity is came from another activity. In these models, places related a to a transition denotes the states before and after executing the corresponding activity and firing of a transition represents the execution of its corresponding activity. For example, after firing the transition "approver" (Fig. 2), it sends a "yes" or "no" message to its output place "decision". Same way, transition "sendmessage" sends a token to its output place "message". Though both these places carry the output messages of the transitions "approver" and "sendmessage", respectively, at the same time they carry the input message for the transition "reply".

Step 3: Finally, all the CPN Models are combined together into a single model according to the activity relationship defined in the BPEL specification. The details of the CPN Model of the loan approval process are shown in Fig. 2.

Construction of reachability graph: The state space or reachability graph is a directed graph which contains a set of nodes and a set of edges where each node denotes a particular state and each edge denotes a transition firing. It consists of all the reachable states and all the possible transition firings of a system. Each state can be presented by the distribution of tokens in the places of the CPN Model and firing of a transition denotes the change of a state from one to another. The state space can be defined as a tuple <S_0, S, T>where the elements are defined as follows:

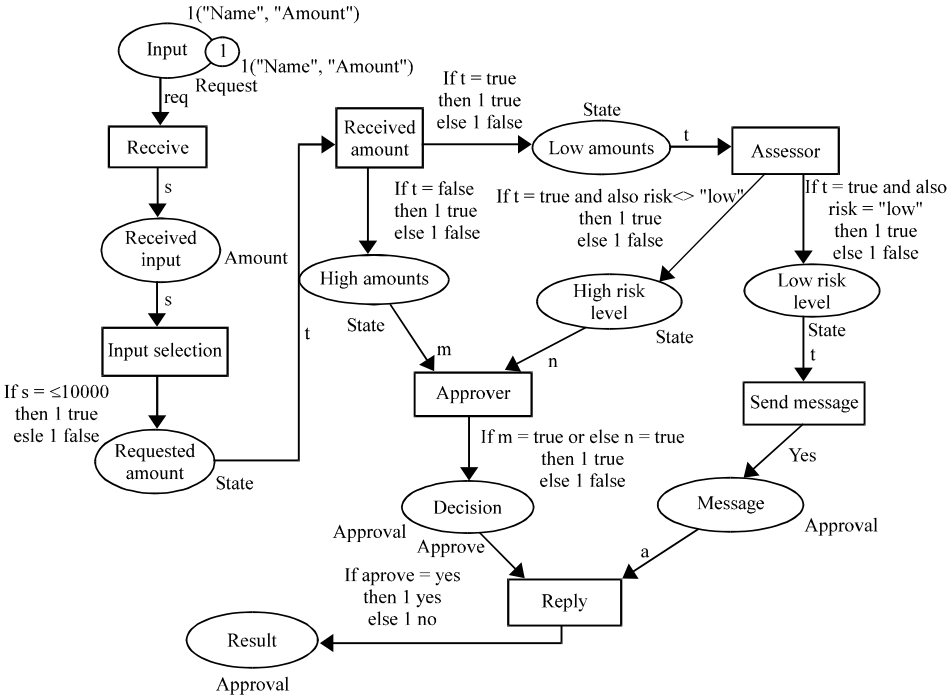

Fig. 2: CPN Model of loan approval approch

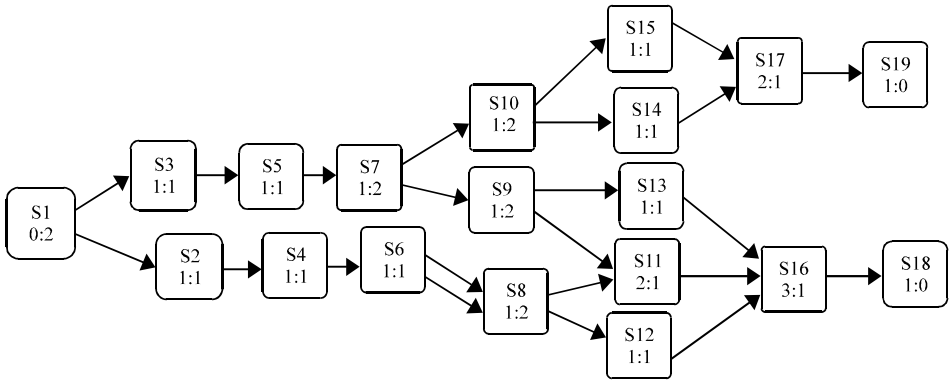

Fig. 3: State space of the CPN Model of the loan approval process

- $S_0 \in S$ is the initial state
- S is the set of all the reachable markings or states from the initial state S_0
- T is the set of transitions

Figure 3 presents the generated reachability graph of the CPN Model of Fig. 2. In the reachability graph, each node represents three values: state number (topmost value), predecessor (the value before the colon) and successor (the value after the colon) states. A number of behavioral properties (Kang *et al*., 2007; Dong *et al*., 2006) (reachability, boundness, deadlock-freedom, safeness, liveness, fairness, etc.,) of the system can also be verified from this reachability graph. Using the state space tool of the CPN tool, we automatically constructed the reachability graph and verified the properties of the CPN Model.

Construction of CFG: Abstraction is important for improving the performance of the test case generation process. In this research, the reachability graph constructed in subsection 3.3 is abstracted into a control flow graph. The abstraction does not hinder the model simulation or system functionalities but helps to speed

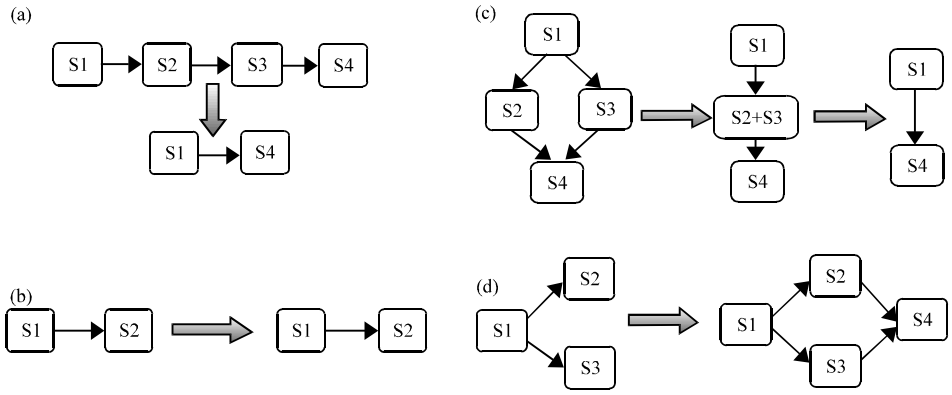

Fig. 4: Guidelines for constructing a CFG: a) Guideline 1; b) Guideline 2; c) Guideline 3 and d) Guideline 4

up the test case generation process. The model reduction can mitigate the state space explosion problem inherent of test case generation techniques. In this research, we proposed some guidelines to construct a CFG which is able to expedite the test case generation approach and improve its performance. The constructed CFG contains both the control flow and data flow information of the BPEL process under test. The control flow information is used to generate a set of test paths from the CFG and the data flow information facilitates the test data generation. TheCFG is defined as a four tuple (N, E, s, e), where:

- N is a set of nodes
- E is a set of edges
- s is the start node where s∈N
- e is the end node where e∈N

Based on some scenarios, we proposed four model formations as the basic elimination entity together with the guidelines for elimination which are described as follows: when there is a sequence of nodes in the RG, the internal nodes are eradicated. Therefore, the sequence is represented by the start node and end node of the sequence, connected by a single edge. Figure 4a presents an example. The parallel nodes are tied together as a single node if their initial node and final node are same. The RG is further reduced by applying the guideline 1. Figure 4b describes this situation. If two nodes are connected with more than a single edge, the edges are considered as a single edge. Example is shown in Fig. 4c. A CFG usually consists of a start node and an end node. If the RG contains more than one terminal nodes in that case, the terminal nodes are attached with a new node which is then considered as the end node of the CFG. Figure 4d presents an example.

We considered both the requirements and sequence of functionalities specified in the BPEL code while applying the proposed guidelines. Moreover, the

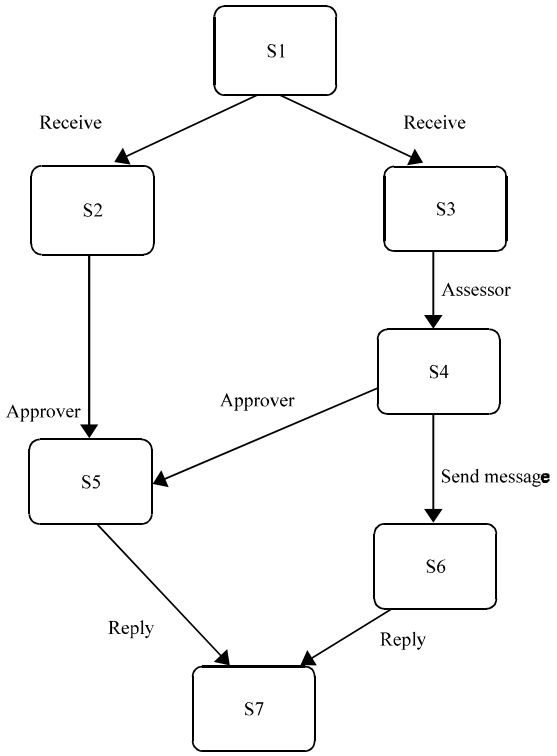

Fig. 5: CFG of the loan approval process

application of these guidelines does not cause any harm to the successive model simulation because they are applied to the RG i.e., all the transitions have already been fired. We applied the guidelines iteratively until all the internal states and transitions are being taken care of. Currently we applied these guidelines manually which is shown in Fig. 5. We marked the nodes of the CFG with different numbers and the edges with the associated transition names which represents the transition sequence of the RG and activities of the BPEL.

Test case generation: A test case is a composition of test paths and test data. These two elements can be generated from graphs (control flow graph, data flow graph), diagrams (UML activity diagram, class diagram, sequence diagram etc.) or specification of a system. In this research we generated the test cases by traversing the CFG constructed from the reachability graph.

Generation of test paths: A test path is any path from the initial state to the final state of a CFG. Generating test paths is the first step of test case generation. A single test path may correlate to an immense number of test cases (Myers, 2004) it represents a model of a test case in the abstraction captured by a graph. We applied a simple Depth First Search (DFS) algorithm on the CFG of Fig. 5 and generated the test paths based on two coverage criteria: state coverage and transition coverage (Ammann and Offutt, 2008). These two coverage criteria are also known as node coverage and branch coverage, respectively. State coverage generates test paths in a way that all the states of the graph is covered by at least one path whereas in transition coverage all the transitions are covered by at least one path. The DFS algorithm generates three paths by traversing the CFG (Fig. 5) which are:

- Path 1 (S1, S2, S5, S7)
- Path 2 (S1, S3, S4, S5, S7)
- Path 3 (S1, S3, S4, S6, S7)

Generation of test data: Test data (Offutt, 2003) are the inputs that have been prepared to test the system. It can be generated manually, automatically or semi automatically. It can be generated in two ways: test data that serves only input messages and test data that provides both input and output messages. In the former one, the output messages are provided manually and for the latter one they could be the after values of an executed event or a post condition associated with an action. There are many existing methods for automatic test data generation. One of the most common technique is constraint solving. Other techniques include random test data generation, goal oriented test data generation, path oriented test data generation and so on (Offutt, 2003).

In this research, we applied the constraint solving approach to generate the test data. To do so, we gathered and solved the constraints related to the test paths. A generator is then prepared corresponding to the data type of the solution. Whenever the generator is called, a random instance value is generated within the specified constraint. For example, if the constraint set of the BPEL program is defined as:

getVariableData ('request', 'amount')≥ = 10000

This means that the requested amount is an integer value which should be >10000. A solution for this could be:

getVariableData ('request', 'amount') = 11200

Test case generation has another significant concept which is the test oracle or expected result. Test oracles are sometimes provided manually and sometimes the after values of a triggering events or the post conditions of the transitions are used as the oracle. In this research, we used the second approach to generate the expected results.

Generation of test cases: A test case can be represented as a sequence of triple (input, triggering event, expected result) where inputs are represented by the test data to execute an event and the expected results are defined by the post-condition of the executed event. Therefore, we collected the input values and after values of each of the events according to the system requirements and functionalities of the BPEL specification. Finally, all these information along with the test paths are combined together to make complete test cases. The generated test cases for the three test paths are represented as follows:

- tc_1 = {(null, receive, 22500), (22500, approver, false), ((false, false), reply, "Loan Rejected")}
- tc_2 = {(null, receive, 8250), (8250, assessor, "high"), (7600, approver, false), ((false, false), reply, "Loan Rejected")}
- tc_3 = {(null, receive, 3340), (3340, assessor, "low"), ("low", sendmessage, true), ((true, true), reply, "Loan Approved")}

However, the generated test cases are abstract. Therefore, they need to be converted into some executable code. For this research, we select Java to convert the generated test cases into executable code. In order to do the conversion, we create a method in Java for each of the transitions (which is denoted by the path between two nodes) of the CFG. From the generated test paths it is clearly visible that a test path can also be represented as a sequence of transitions. Same way, an executable test case can be represented as a sequence of method call including some additional information such as inputs, variables, values, conditions, expected result, etc.

Case study and prototype tool: In order to evaluate the feasibility and effectiveness of our approach we use two classic BPEL case studies (Anonymous, 2008). In this study, we provide a brief introduction of the case studies. In addition, the architecture of our prototype tool is also discussed which is implemented to evaluate the fault detection capability of the proposed approach.

Case studies

Loan approval process: The loan approval process contains three web services assessment, approval and the loan service itself which is shown in Fig. 6. In this process, firstly, the loan service receives loan request from the customers including the requested loan amount. For lower amounts, the risk assessment service performs a quick evaluation of risk associated with the customer. If the amount is <$10,000 and associated risk is "low", the request will be approved automatically and it will send a "loan approved" message to the customer. But if the amount is >$10,000 or the risk is "high" it will send a "loan rejected" message to the customer.

A slight modification is done by us for the amounts of <$10,000 to acquire the risk assessment result which is: for the amounts of <$5,000, the associated risk will be "low" otherwise it will be "high". The business scenarios of the other two web services (Assessment and approval) including the modifications are shown in Fig. 7.

Purchase order process: The purchase order process includes four web services: purchasing, scheduling, invoicing and shipping. Figure 8 shows the business scenario of the purchase order service. The purchasing service starts by receiving a purchase order request PO from the customer. The process then performs three tasks concurrently: firstly, it sends a shipping request which is assigned with PO to the shipping service and then it gets the shipping info and shipping schedule from it. Secondly, the process sends the PO and shippinginfo to the invoicing service in order to calculate the order's price and shipping price and then it gets the invoice from the invoicing service. Thirdly, it sends the PO and shipping schedule to the scheduling service for the scheduling and shipment information of the ordered product. After completing all these three tasks the final invoice is sent to the customer.

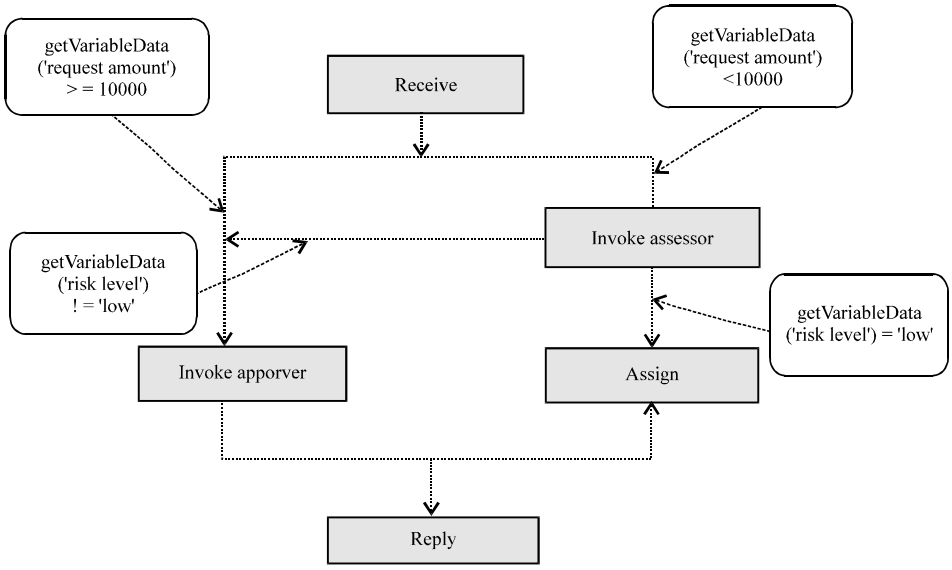

Fig. 6: Business scenario of the loan approval process

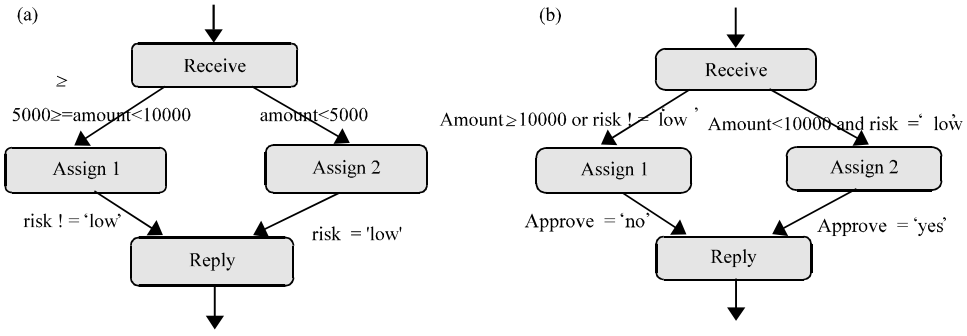

Fig. 7: Business scenario of the assessment and approval services: a) Assessment service and b) Approval service

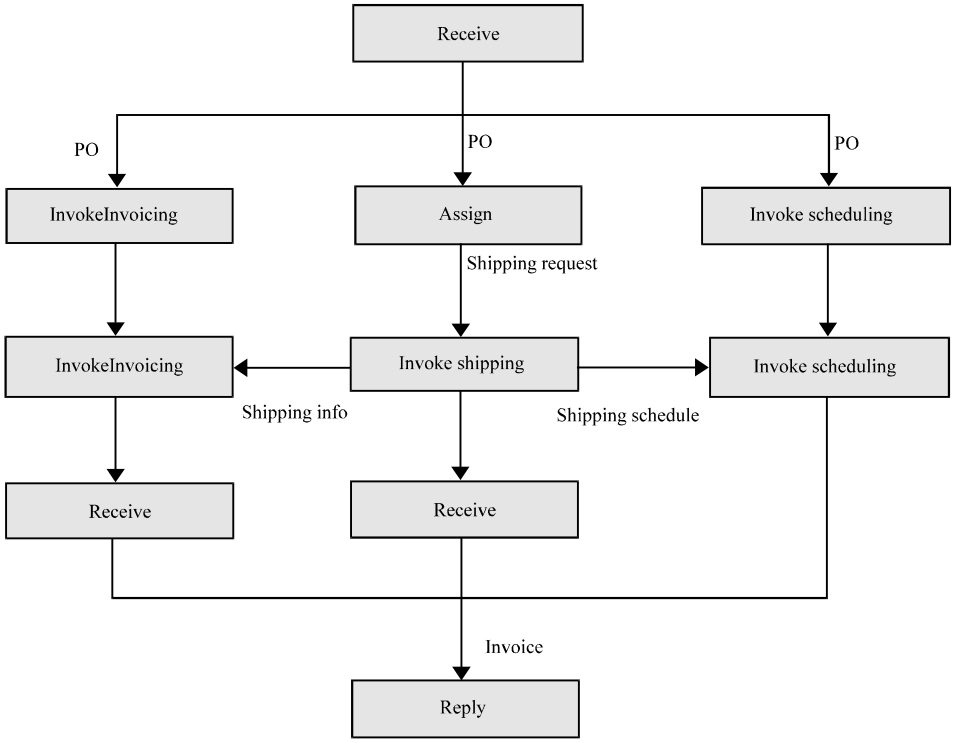

Fig. 8: Business scenario of the purchase order service

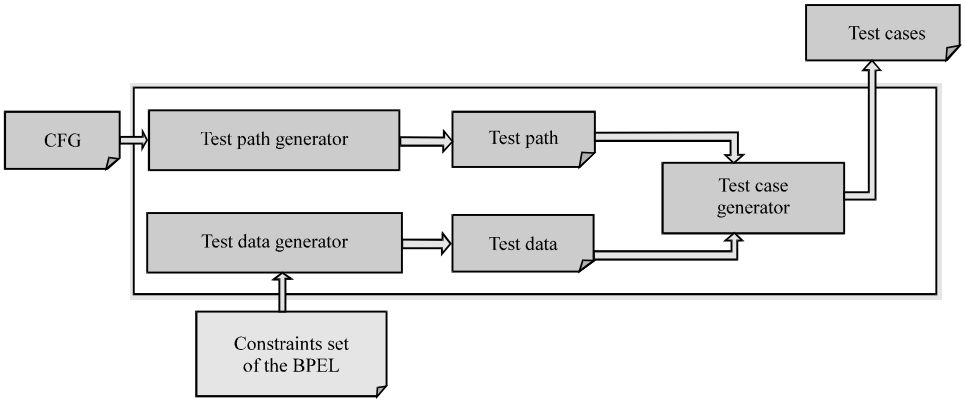

Fig. 9: Architecture of the prototype tool

Table 1: Case study summary

BPEL	Loan approval process	Purchase order process
No. of places	10	15
No. of transitions	7	11
No. of states in the reachability graph	19	36
No. of states in the CFG	7	11

The activities residing within the BPEL process, respectively, stand for different events. We transformed these events in terms of transitions when modeling into CPN. The CPN Model of the loan approval process is provided in Fig. 2. For limited space, we did not provide the CPN Model of the purchase order process. Table 1 presents the summary of the two case studies based on the structure of their CPN Model, reachability graph and CFG.

Prototype tool: The implemented software system (prototype tool) is a Java based desktop application developed using Java (1.7.0 J2SDK) and third party kits (Window Builder, Junit test plug-in) on the Eclipse IDE Mars (4.5.1). The architecture of the prototype tool is presented in Fig. 9. It implements our approach and contains mainly three components:

- Test path generator: generates test paths by traversing the CFG by applying the depth first search algorithm
- Test data generator: generates test data for each of the test paths by solving the constraints of the paths
- Test case generator: generates test cases by combining the generated test paths and test data

We implemented a prototype tool that has been developed in Java to generate test cases from the CFG. The input includes the CFG nodes, start node, end node, edges, transition names and the accessor functions. The transition names denote the event's names related to the edges and the accessor functions return the values of the get methods from the SUT to evaluate the expected results. The outputs of the tool contains the test paths, test data and test cases. Each of the transitions residing within a test path represents a method in the SUT. Therefore, the test cases for each of the paths can be represented as the sequence of method calls including the associated variables, preconditions, post conditions and some additional information. The generated test cases support the JUnit format and thus can be executed using the JUnit test framework. The users can enter the test input via. the user interface.

Evaluation

Experimental procedures: In order to conduct the experiment, at first we applied the proposed guidelines and manually constructed a Control Flow Graph (CFG) of the BPEL-based web service composition using CPN as a mediate. Meanwhile, the BPEL program is converted into the programming language java as the Service Under Test (SUT). The CFG is then provided as the input of the prototype tool through the interface. From these inputs, we generated the test paths, test data and test cases in Java. The tool is able to generate these three elements in three different tabs.

To evaluate the fault detection capability of the generated test cases, we created some conditions that do not match with the conditions defined in the BPEL program. We mapped those faulty conditions into implementation faults and seeded them in the SUT. To avert any preconception that could be created by having insight of the faults, the faults were generated after the generation of the test cases. Finally, we ran the generated test cases using the JUnit test framework to evaluate whether the test cases are able of finding errors in the SUT.

RESULTS AND DISCUSSION

Following the experimental procedures described in the previous section, we constructed CFG for both of the two case studies and used the CFG as the input of the tool. For the first case study of the "Loan approval process" we found three test paths.

Afterwards, based on the test data generation approach, we computed the transition sequence and the allowed data ranges for each of the paths which are as follows:

- Path 1: receive, approver, reply (amount≥10000, risk≠ "low", approve = "no")
- Path 2: receive, assessor, approver, reply. (5000≥ amount<10000, risk≠"low", approve = "no")
- Path 3: receive, assessor, sendmessage, reply. (amount<5000, risk = "low", approve = "yes")

Finally, we got three abstract test cases by combining the test paths with their specified data ranges.

For the second case study of the "purchase order process", two guidelines (1 and 2) were applicable among the four in order to construct the CFG from the reachability graph. However, since, there is no brunch in the BPEL program of the purchase order process, applying guideline 1 may cause some important activities to be disappeared and make the CFG too small. Therefore, we applied only guideline 2 that combines two parallel states together but do not reduce its length. Finally, the CFG we get after applying the guideline is a sequence of eleven states which consists of ten transitions. As a consequence, one test path is generated which is as follows: path 1: S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11. Based on the functionalities of the BPEL program, the transition sequence for the above path is as follows: path 1: receivePO, reqShip, shipping, sendShipPrice, sendShipSche, IPC, invoicing, RPC, scheduling, reply. Finally, the abstract test case we got can be represented as follows: tc_1: {(null, receivePO, "POmessage"), ("POmessage", reqShip, "POmessage"), ("POmessage" shipping, ("shippingInfo", "schedulingInfo")), ("POmessage", IPC, "POmessage"), ("shippingInfo", sendShipPrice, "shippingInfo"), (("POmessage", "shippingInfo"), invoicing, "invoice"), ("POmessage", RPC, "POmessage"), ("schedulingInfo", sendShipSche, "schedulingInfo"), (("POmessage", "shippingInfo"), scheduling, "schedule"), (("invoice", "schedule"), reply, "invoice")}.

From the summary of the two case studies we can notice that, the CFG of the purchase order process has more states than the loan approval process. However, only one test path is generated from it whereas three test paths are generated for the loan approval process; this is because the BPEL program of the purchase order service does not contain any branch. Thus, the CFG becomes a sequence of states which represents only a single test path.

Fault detection capability: In order to evaluate the fault detection capability of our proposed approach and the generated test cases, we create a version of the SUT of the "loan approval process" which is written in java. In the new version we have seeded an error by changing a condition of the BPEL code. To do this, we changed a condition of the incoming link of the "Approval" web service as "getVariableData('request', 'amount')≤12000" which previously was "get variable data('request', 'amount')≥10000". Due to this change, the incoming link condition ("get variable data('request', 'amount') <10000") of the "Assessment" web service becomes overlaid with the condition of the partner service "Approval" for any value which is <10000, i.e., 8000.

Finally, we execute the three generated test cases for the new modified version of the "loan approval process" using the JUnit test framework. The execution result shows that two test cases among the three are passed and the other one is failed. The reason behind the failed test case is that a test input for that test case could be 15000 based on the specification of the BPEL code. However, it does not satisfy the modified condition as a result, the test case is failed.

Effectiveness of the approach: In order to evaluate the effectiveness of our approach, we conducted experiments for our approach and two other state-of-the-art approaches and compared the results of the experiment. Since, we combined reachability graph and control flow graph together in our approach we select a reachability graph-based approach (Cai *et al*., 2011) and a control flow graph-based approach (Yuan *et al*., 2006) for the comparison. We generated test paths by applying these two approaches using state coverage and transition coverage for both of the case studies. Table 2 represents the summery of the comparison.

Table 2, we observe that the test paths generated by the other two approaches is higher than our approach. The reachability graph-based approach and CFG-based approach have generated 6 and 10 paths, respectively, for the "loan approval" process and 35 and 6 paths, respectively, for the "purchase order" process. Our approach, on the other hand, generates only 3 and 1 test paths, respectively, for the "loan approval" and "purchase order" processes. This is because the proposed guideline (Guideline 3) for CFG construction added two parallel nodes together as a result, two paths are merged into a single one. Therefore, the quantity of the paths is being minimized; thus, the time required for the test case generation will be minimized correspondingly. Furthermore, reachability graph-based approach generates a large number of paths which is time consuming.

Table 2: Summary of the comparison with existing approaches

Case studies	Approaches	Cai *et al.* (2011)	Yuan *et al.* (2006)	Our
Loan approval process	Total paths	6	10	3
	Feasible paths	6	3	3
	Infeasible paths	0	7	0
Purchase order process	Total paths	35	6	1
	Feasible paths	35	4	1
	Infeasible paths	0	2	0

We also observe that the CFG-based approach has generated some infeasible paths. Though, the researchers have provided some techniques to handles these infeasible paths, however, it requires some additional work. On the other hand, our approach does not generate any infeasible test case. Therefore, no additional work is required to handle this problem which makes our approach both time and cost effective.

CONCLUSION

In this researcher we present a CPN-based test case generation approach for web services compositions written in BPEL. The main achievement of this research work can be illustrated as the enhancement and adaptation of existing and new approaches in order to generate test cases. We combined a reachability graph and a control flow graph together to generate feasible test cases and minimize the consequences that could be caused by the state space explosion problem of the reachability graph. In addition, we proposed some guidelines to construct a CFG from the reachability graph. Finally, a prototype tool is developed that implements our approach and determines the effectiveness of our approach. The experimental validation results showed that all the test cases generated by our approach are feasible and able to expose the seeded faults within a system.

At present, we applied our approach to only two BPEL processes. In future, we would like to apply this approach to other BPEL-based web service compositions that contains more advanced and complicated BPEL features. The applications might further help us to observe both the effectiveness and lacking of the technique. We currently manually constructed the CFG from the reachability graph by applying the proposed guidelines. In future, we plan to make the prototype tool fully automatic in order to automatically construct the CFG from the RG. For now, the proposed guidelines for constructing the CFG are restricted to the BPEL programs that do not contain loops. Extending our approach and

providing solution of this issue will also be a part of our future research. Besides, in future we plan to perform a more detailed analysis of time complexity for our approach.

ACKNOWLEDGEMENT

The researchers are grateful to all the reviewers for reviewing our manuscript. The study is funded by SEEDSlab (Software Intelligence and Data Science Research Group).

REFERENCES

Aalst, V.D. and C. Stahl, 1999. Modeling Business Processes-A Petri Net-Oriented Approach. MIT Press, Cambridge, Massachusetts, USA.,.

Ammann, P. and J. Offutt, 2008. Introduction to Software Testing. Cambridge University Press, Cambridge, UK., ISBN:978-0-521-88038-1, Pages: 322.

Anonymous, 2008. Web services business process execution language version 2.0. Oasis Publishing, Clovis, California. http://docs.v oasis-open.org/ wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf

Bernot, G., M.C. Gaudel and B. Marre, 1991. Software testing based on formal specifications: A theory and a tool. Software Eng. J., 6: 387-405.

Cai, L., J. Zhang and Z. Liu, 2011. A CPN-based software testing approach. J. Software, 6: 468-474.

Dong, W.L., H. Yu and Y.B. Zhang, 2006. Testing BPEL-based web service composition using high-level petri nets. Proceedings of the 2006 10th IEEE International Conference on Enterprise Distributed Object Computing (EDOC'06), October 16-20, 2006, IEEE, Hong Kong, China, ISBN:0-7695-2558-X, pp: 441-444.

Farooq, U., C.P. Lam and H. Li, 2008. Towards automated test sequence generation. Proceedings of the 19th Australian Conference on Software Engineering, March 26-28, IEEE Computer Society, Washington, USA., pp: 441-450.

Hierons, R.M., K. Bogdanov, J.P. Bowen, R. Cleaveland and J. Derrick *et al*., 2009. Using formal specifications to support testing. ACM. Comput. Surv., 41: 1-76.

Jahan, H., S. Rao and D. Liu, 2016. Test case generation for BPEL-based web service composition using Colored Petri Nets. Proceedings of the 2016 International Conference on Progress in Informatics and Computing (PIC), December 23-25, 2016, IEEE, Shanghai, China, ISBN:978-1-5090-3485-7, pp: 623-628.

Jensen, K. and L.M. Kristensen, 2009. Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Springer, Berlin, Germany, ISBN: 978-3-642-00283-0, Pages: 384.

Jensen, K., 1996. Coloured Petri Nets Basic Concepts, Analysis Methods and Practical Use. Springer, Berlin, Germany,.

Kang, H., X. Yang and S. Yuan, 2007. Modeling and verification of web services composition based on CPN. Proceedings of the 2007 IFIP International Conference on Network and Parallel Computing Workshops (NPC 2007), September 18-21, 2007, IEEE, Liaoning, China, ISBN:0-7695-2943-7, pp: 613-617.

Liu, aJ., X. Ye and J. Li, 2011. Colored Petri nets model based conformance test generation. Proceedings of the IEEE 2011 International Symposium on Computers and Communications (ISCC), June 28-July 1, 2011, IEEE Kerkyra, Greece, ISBN:978-1-4577-0680-6, pp: 967-970.

Liu, S.Y. and S. Nakajima, 2011. A framework for automatic functional testing based on formal specifications. Proceedings of the 6th International Workshop on Automation of Software Test, May 21-28, 2011, Honolulu, HI, USA., pp: 107-108.

Miller, aK.W., aL.J. aMorell, aR.E. aNoonan, aS.K. aPark, D.M. Nikol, B.W. Murrill and J.M. Voas, 1992. Estimating the probability of failure when testing reveals no failures. IEEE Trans. Software Eng., 18: 33-43.

Murata, T., 1989. Petri nets: Properties, analysis and applications. Proc. IEEE., 77: 541-580.

Myers, G.J., 2004. The Art of Software Testing. 2nd Edn., John Wiley and Sons, ?Hoboken, New Jersey, USA., ISBN:0-471-46912-2, Pages: 234.

Ni, Y., S.S. Hou, L. Zhang, J. Zhu and Z.J. Li *et al*., 2013. Effective message-sequence generation for testing BPEL programs. IEEE. Trans. Serv. Comput., 6: 7-19.

Offutt, J., 2003. Generating test data from state-based specifications. Software Test. Verificat. Reliabil., 13: 25-53.

Rao, S., H. Jahan and D. Liu, 2016. A search-based approach for test suite generation from extended finite state machines. Proceedings of the 2016 International Conference on Progress in Informatics and Computing (PIC), December 23-25, 2016, IEEE, Shanghai, China, ISBN:978-1-5090-3485-7, pp: 82-87.

Ratzer, aA.V., aL. aWells, aH.M. aLassen, aM. aLaursen and J.F. Qvortrup *et al*., 2003. CPN tools for editing, simulating and analysing coloured Petri nets. Proceedings of the International Conference on Applications and Theory of Petri Nets and Concurrency, June 23-27, 2003, Springer, Berlin, Heidelberg, Germany, ISBN:978-3-540-40334-0, pp: 450-462.

Reza, H. and S.D. Kerlin, 2011. A model-based testing using scenarios and constraints-based modular petri nets. Proceedings of the 2011 8th International Conference on Information Technology: New Generations, April 11-13, 2011, IEEE, Las Vegas, Nevada, ISBN:978-1-61284-427-5, pp: 568-573.

Schmidt, K., 2000. Lola a low level analyser. Proceedings of the International Conference on Application and Theory of Petri Nets, June 26-30, 2000, Springer, Berlin, Heidelberg, Germany, ISBN:978-3-540-67693-5, pp: 465-474.

Stocks, P. and D. Carrington, 1996. A framework for specification-based testing. IEEE. Trans. Software Eng., 22: 777-793.

Wang, Y. and N. Yang, 2014. Test case generation of web service composition based on CP-nets. J. Software, 9: 589-596.

Watanabe, H. and T. Kudoh, 1995. Test suite generation methods for concurrent systems based on coloured Petri nets. Proceedings of the 1995 Asia Pacific Conference on Software Engineering, December 6-9, 1995, IEEE, Brisbane, Australia, ISBN:0-8186-7171-8, pp: 242-251.

Xu, D., W. Xu, M. Kent, L. Thomas and L. Wang, 2015. An automated test generation technique for software quality assurance. IEEE. Trans. Reliab., 64: 247-268.

Yan, J., Z. Li, Y. Yuan, W. Sun and J. Zhang, 2006. BPEL4WS unit testing: Test case generation using a concurrent path analysis approach. Proceedings of the 2006 17th International Symposium on Software Reliability Engineering, November 7-10, 2006, IEEE, Raleigh, North Carolina, USA., pp: 75-84.

Yang, Y., Q. Tan and Y. Xiao, 2005b. Verifying web services composition based on hierarchical colored petri nets. Proceedings of the 1st International Workshop on Interoperability of Heterogeneous Information Systems (IHIS '05), November 04, 2005, ACM, New York, USA., pp: 47-54.

Yang, Y., Q. Tan, J. Yu and F. Liu, 2005a. Transformation BPEL to CP-nets for verifying web services composition. Proceedings of the International Conference on Next Generation Web Services Practices (NWeSP'05), August 22-26, 2005, IEEE, Seoul, South Korea, ISBN:0-7695-2452-4, pp: 6-6.

Yi, X. and K.J. Kochut, 2004. A CP-nets-based design and verification framework for web services composition. Proceedings of the IEEE International Conference on Web Services, July 9, 2004, IEEE, San Diego, California, ISBN:0-7695-2167-3, pp: 756-760.

Yuan, Y., Z. Li and W. Sun, 2006. A graph-search based approach to BPEL4WS test generation. Proceedings of athe a2006 aInternational aConference aon Software Engineering Advances (ICSEA'06), October 29-November 3, 2006, IEEE, Tahiti, French Polynesia, ISBN:0-7695-2703-5, pp: 14-14.

Zhu, H. and X. He, 2002. A methodology of testing high-level Petri nets. Inf. Software Technol., 44: 473-489.