DOI: 10.15415/jotitt.2014.21001

Test Sequence Generation for Java7 Fork/Join Using Interference Dependence

VIPIN VERMA*, VINAY ARORA

Computer Science and Engineering Department, Thapar University, Patiala-147004, Punjab, India

***E-mail: vipin.vipin2008@gmail.com**

Received: April 13, 2014| Revised: April 29, 2014| Accepted: May 6, 2014

Published online: June 30, 2014

Abstract: Test sequence generation through code is mainly done by using some sort of a flow graph viz. Control Flow Graph (CFG), Concurrent Control Flow Graph (CCFG), Event Graph etc. Approaches that use UML also need flow graph as an intermediate representation for final test sequence generation. In the present approach, a Flow Graph for a new concept i.e. Java7 Fork/Join is constructed and hence, by traversing the graph, test sequences are generated on the basis of all path and all node coverage criteria considering interference dependence. Further, interference dependencies are also represented in the form of a directed graph to aid the analysis of Java7 fork/join programs.

Keywords : Test Sequence Generation, Java7 Fork/Join, JFJFG, Interference Dependence

1. INTRODUCTION

Today, in the world of multi-core processors, there are several ways to utilize their powers. One of them is to employ the new Java7 Fork/Join (2013). The package for Java7 Fork/Join is 'java.util.concurrent'. The Java7 Fork/Join works on 'Work-Stealing algorithm' i.e. whenever some threads don't have anything to do, they can steal work from other busy threads. The class 'java.util.concurrent.ForkJoinPool' uses this algorithm and can execute various 'java.util.concurrent.ForkJoinTask' processes at the same time (2013). To use the Java7 Fork/Join utility, the code to be executed in parallel must be written in the compute() method as shown in the Figure 1.

A basic block is a sequence of instructions executed one after the other having one entry and one exit point. Control Flow Graph is a directed graph in which the nodes represent the basic blocks and the edges between them show

Journal on Today's Ideas – Tomorrow's Technologies, Vol. 2, No. 1, June 2014 pp. 1–12

```
if(the work to be done is small enough)
   do the work
else
   divide the work in two pieces
   invoke the two pieces, wait for result
```

Figure 1: Principle of Java7 Fork/Join(2013)

the flow of control (1970). Java Fork/Join Flow Graph (JFJFG) is a Control Flow Graph for concurrent programs representing the flow of control and the concurrent paths of a Java7 Fork/Join program.

In a sequential program, a statement *m* is data dependent on statement *n*, if *n* defines some variable and node *m* uses this variable along a control-flow path (2004). Data Dependence can also be termed as Read-After-Write. Interference Dependence is a special type of data dependence between the instructions of a concurrent program. Say, a variable *x* of any object is written by a thread T_1 at node *n* and it is read by some other thread, say T_2 at a statement *m*. In such a case, node *m* is interference dependent on node *n*(2004). The compute() method for a Java7 Fork/Join program may be accessed by multiple threads at the same time. So, the Read-After-Write in the compute() method are Interference Dependencies.

2. RELATED WORK

Test case generation can be done by using models or code. In sections 2.1 and 2.2, work related to test case generation from code has been explained. The sections 2.3 and 2.4 describe the related work for generating test cases from UML models. Section 2.5 explains some adequacy criterion.

2.1. Test Case Generation using Event Graphs

Event Graph is a Control Flow Graph showing a unit of a concurrent program. Event InterAction Graph (EIAG) (1995) is a graph that represents the behavior of a concurrent program which has the events and their interactions as the main components. Interactions can be for synchronization, communications or wait. EIAGs depend on the source code. The co-paths (cooperated paths) on EIAG provide the test cases. T. Katayama *et al.* (1995) generated the co-paths automatically. This approach is able to detect unreachable statements and communication errors in testing. Later T. Katayama *et al.* (1999) used the Interaction Sequence Testing Criteria (ISTC) for generating the co-paths. These test cases are able to find out unreachable statements, also

some communication errors and deadlock. X. Bao *et al.* (2009) generated the test cases for concurrent programs based upon the Event Graphs. Test cases, also known as sub-event graphs, are generated by the analysis of Event Graph.

2.2. Test Case Generation for Business Process Execution Language (BPEL)

Y. Yuan *et al.* (2006) created a BPEL Flow Graph (BFG), an extension of Control Flow Graph. The BFG is traversed using a constraint solving method and test paths are combined for generating the test cases. J. Yan *et al.* (2006) created an Extended Control Flow Graph (ECFG) from the language BPEL. Then all the sequential test paths are generated. On combining the sequential test paths, the concurrent test paths are generated. Y. Zheng *et al.* (2007) used SPIN (Simple PROMELA (PROcess MEta LAnguage)) model checker as test generation engine. For Control Flow testing, state and transition coverage are used and for data flow testing, all-du (def-use) path coverage is used. The generated test cases are then executed on JUnit test execution engine.

2.3. Test Case Generation from Activity Diagram

C. Mingsong *et al.*(2006) presented first technique for automatic test case generation by a tool AGTCG (Activity Graph Test Case Generator). Test cases are generated at random and the execution traces are compared with the Activity Diagram to get a reduced set of test cases. H. Kim *et al.* (2007) converted the Activity Diagram into Input Output explicit Activity Diagram (IOAD) in which the inputs and outputs are taken under consideration. This intermediate form IOAD is then transformed into a directed graph from which the test cases are derived. D. Kundu *et al.*(2009) converted the Activity Diagram into another intermediate representation, Activity Graph and the test cases are then generated on the basis of path coverage criteria. C. Sun (2008) converted the Activity Diagram into BET (Binary Extended AND_OR Tree), which is traversed using Depth-First Traversal to generate the test scenarios. He also presented a tool 'TCaseUML'. B. Lei *et al.*(2008) also presented a tool named as 'tof4j' (Testing of concurrency for java program) in which Activity Diagram is extended and this extended Activity Diagram is traversed on the basis of path analysis technique. M. Khandai *et al.* (2011) presented a survey on test case generation from UML Models and stated two approaches for the same. First, Activity Diagram is converted to Activity Graph and by traversing that test cases are generated. Second, Activity Diagram is converted to some intermediate form using some transformation rules and then test cases are generated.

2.4. Test Case Generation from Sequence Diagram

M. Shirole *et al.*(2012) presented an approach in which the Sequence Diagram is first converted to Activity Diagram using some rules. An algorithm named as Concurrent Queue Search (CQS) is also presented to traverse the Activity Diagram generating the test sequences. This algorithm is better than Depth First Search (DFS) and Breadth First Search (BFS). M. Khandai *et al.* (2011) showed a technique to convert the Sequence Diagram into Concurrent Composite Graph (CCG), an intermediate representation which is traversed to generate the test cases. The problem of test case explosion is avoided and issues like deadlock and synchronization are also handled.

2.5. Test Adequacy Criteria

A test case T is adequate according to statement (all node) coverage criteria, if it covers all the reachable nodes (1985). A test case T is adequate according to all def-use (du) path coverage if all the du paths are covered by it. A def-use path, say (n_1,n_2…. n_k)is the path in CFG (Control Flow Graph) on which any variable is defined on $n_1$and then used on n_k(1985).

3. METHODOLOGY

The methodology used to generate the test sequences for Java7 Fork/Join programs is shown in the Figure 2. Java7 program, for adding the elements of an array utilizing the Java7 Fork/Join capability, is taken as input. The example program taken as input is shown in the Figure 3. The value of *SEQUENTIAL_THRESHOLD* variable is set to be 5000. If the number of elements to be added are lesser than or equal to 5000, the work is carried out sequentially otherwise the work is divided using fork().

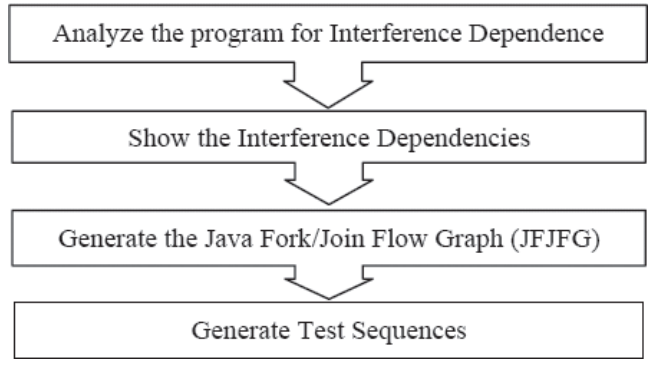

Figure 2: Methodology used in the approach

```
1.  import java.util.concurrent.ForkJoinPool;
2.  import java.util.concurrent.RecursiveTask;
3.
4.  class Globals
5.  {
6.  static ForkJoinPool fjPool = new ForkJoinPool();
7.  }
8.
9.  class Sum extends RecursiveTask<Long>
10. {
11. static final int SEQUENTIAL_THRESHOLD = 5000;
12.
13. int low;
14. int high;
15. int[] array;
16.
17. Sum(int[] arr, int lo, int hi)
18. {
19. array = arr;
20. low  = lo;
21. high  = hi;
22. }
23. protected Long compute()
24. {
25.  if(high-low<=SEQUENTIAL_THRESHOLD)
26.  { // if the task to be done is small: do the work now
27.   long sum=0;
28.   for(int i=low;i<high;i++)
29.     sum=sum+array[i];
30.   return sum;
31.   }
32.  else
33.   { //the task to be done is too big: divide the work
34.   int mid=low+(high-low)/2;
35.   Sum left=new Sum(array, low, mid);
36.   Sum right=new Sum(array, mid, high);
37.   left.fork();
38.   long rightAns=right.compute();
39.   System.out.println("This is the sample program");
40.   long leftAns=left.join();
41.   return leftAns+rightAns;
42.   }}
43. static long sumArray(int[] array)
44. {
45. return Globals.fjPool.invoke(new Sum(array,0,array.length));
46. }}
```

Figure 3: Input file Sum.java (2014)

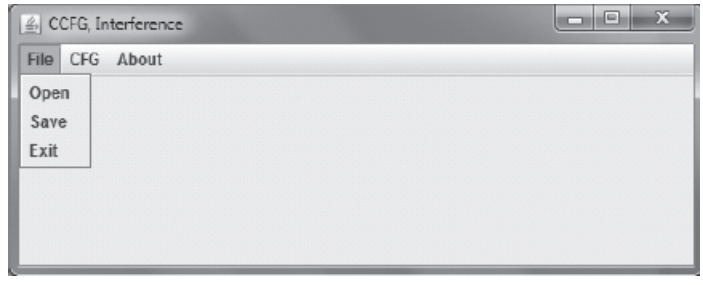

Figure 4: User Interface (UI) of the application

The snapshot showing user interface of implemented prototype tool is shown in the Figure 4. The implementation is done in '*jdk1.7.0_45*'. It contains 3 menus, out of which the 2 menus File and CFG do the main task. File menu lets the user choose the Java7 fork/join file to be given as input, also to save the generated directed graph for the given input program. The menu CFG lets the user to draw the flow graph of the file chosen.

The methodology of the approach presented in the paper is as follows:

3.1. Identifying Interference Dependence

The definitions and then uses of the variables inside the compute() method i.e. for simultaneously executable sections, are treated as interference dependence. The steps for finding the interference dependence are given in the Algorithm 1.

Algorithm 1: Identifying Interference Dependence

/* *interference* is the output adjacency matrix having interference dependencies*/

Input: Java7 Fork/Join Program

Output: Interference Dependence Matrix

1. Initialize each cell of the matrix *interference*[][] to 'false'.
2. Provide numbering to all statements of the program.
3. Traverse compute() method statement by statement.
   //because compute() method has Fork/Join section which
   //makes parallel executions inside the program.
4. If a variable v is defined at statement L_1 and used at statement L_2, Then *interference*[L_2][L_1] = true. //statement L_2 is dependent on statement L_1.

3.2. Visualizing Interference Dependence

After identifying interference dependence among the various statements, they are shown in the form of a directed graph for better understanding of the concepts. The algorithm for drawing the directed graph for showing the interference dependence among the statements of the program is given in the Algorithm 2.

Algorithm 2: Visualizing interference dependence

/* *Visited* is the list of nodes already drawn, *interference* is the adjacency matrix for interference dependence */

Input: Adjacency Matrix for interference dependence.

Output: Directed graph

1. *Visited* = Φ.
2. Traverse the interference dependence matrix i.e. *interference*[][] for each cell.
3. Repeat the step 4 until all the nodes are visited.
4. If *interference*[*i*][*j*] = true, Then
   a. If *i* OR *j* OR both nodes ∉*Visited*, Then
      Draw the corresponding node(s).

Add *i* OR *j* OR both to *Visited*.

b. Draw directed line form node *i* to node *j*, showing node *i* is dependent on node *j*.

The interference dependencies for the example Java7 fork/join program are shown in Figure 5. Statement number 28 and 29 are dependent on themselves. Statement number 35 and 36 are dependent on statement number 34 and similarly other statements are dependent.

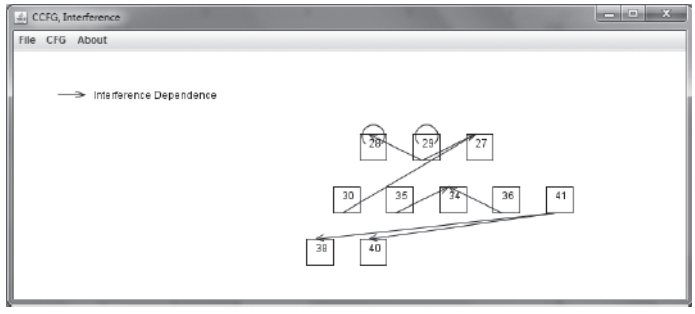

Figure 5: Directed graph showing interference dependencies

3.3. Generating Java Fork/Join Flow Graph (JFJFG)

For the program taken as input, the JFJFG is drawn for the compute() method. Call to the fork() method is shown as the call to the parallel tasks which invokes the compute() method for that variable. And call to the compute() after the fork(), invokes the other parallel activity. Whereas call to the join() function returns the value of the thread on which fork() was called. The execution is just like sequential methods up to the call of fork() method and after the call to join() method. The steps for drawing the JFJFG are presented in the Algorithm 3. The output of Algorithm 3 is shown in Figure 6 presented in the results section.

Algorithm 3: Drawing JFJFG (Java7 Fork/Join Flow Graph)

/* array '*fork_join*' is the array to store the location of call to fork() and join() */

Input: Java7 Fork/Join Program

Output: Java7 Fork/Join Flow Graph (JFJFG)

1. Initialize array *fork_join* = Φ.
2. Search for compute() method. In this fork() and join() calls are considered.

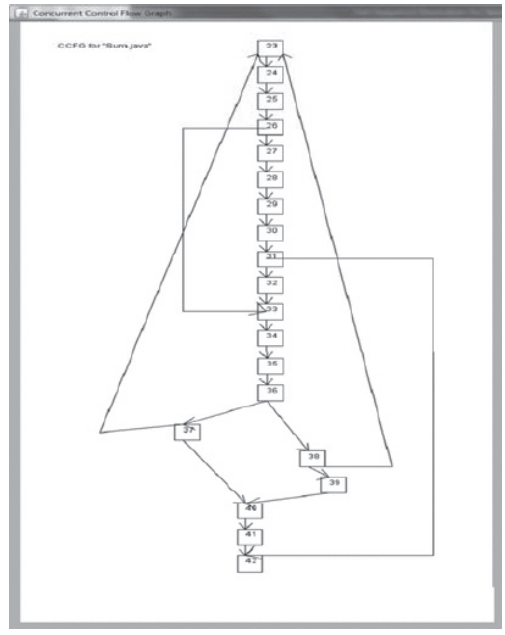

Figure 6: Java Fork/Join Flow Graph (JFJFG)

3. Repeat step 4 for each fork/join call.
4. Note statement number of fork(). Say it is at statement L_1 and corresponding join() is at statement L_2, for object v.

   $fork\_join_v[0] = L_1$ and $fork\_join_v[1] = L_2$.
5. Repeat step 6 for each *fork_join* variable entry in *fork_join* array.
6. Generate flow graph using *fork_join* array by using the following steps:

   Show all the statements in sequential order up to $fork\_join_v[0]$ statement.

   Show the statements between $fork\_join_v[0]$ and $fork\_join_v[1]$ statements in parallel in flow graph. //because these statements can execute in parallel.

   Show the flow from $fork\_join_v[0]$ to the statement in which compute() method is called by directed line.

   Show all the remaining statements in compute() method in sequential manner after $fork\_join_v[1]$ statement.

3.4 Generating Test Sequences

After the Java Fork/Join Flow Graph (JFJFG) has been generated, it is traversed on the basis of all node and all path coverage criteria considering the

interference dependence in order to find out the Test Sequences. The algorithm for generating the test sequences is shown in the Algorithm 4.

Algorithm 4: Generate Test Sequences

/* V_p is the present node being explored and V_{end} is the end node of compute() method, *visited* is the array that stores the status of the nodes whether they are visited or not*/

Input: A Java7 Fork/Join Flow Graph (JFJFG) G (V, E)

Output: Test Sequences

1. Start from the beginning of compute() method.
2. Repeat until V_p != V_{end}.
3. If V_p is a call to fork() method, use algorithm Breadth First Search (BFS):
   a. Mark all the nodes as unvisited.
      $\forall V_i \in V$, set *visited*[V_i] = false.
   b. Enqueue the present node V_p.
   c. Dequeue from the front of queue. Mark it as V_p. Set *visited*[V_p]= true.
   d. Enqueue all the nodes adjacent to V_p.
   e. Repeat the steps 3.b to 3.d until the queue is empty.
   f. Exit when the node join() is found.
4. If V_p is any other statement, use algorithm Depth First Search (DFS):
   a. Mark all the nodes as unvisited.
      $\forall V_i \in V$, set *visited*[V_i] = false.
   b. Push the present node V_p on the stack.
   c. Pop from the top of stack. Mark it as V_p. Set *visited*[V_p] = true.
   d. Push all the nodes adjacent to V_p on the stack.
   e. Repeat the steps 4.b to 4.d until the queue is empty.
   f. Exit.
5. End of repeat.

4. RESULTS

The outcome of Algorithm 3 is a flow graph which we call as Java7 Fork/Join Flow Graph (JFJFG). There is a call to fork() method in statement number 37 of the program. And call to join() method is at statement number 40 of the program. Figure 6 shows the JFJFG.

Table 1 shows the description of the nodes that are present in JFJFG. The method compute() starts at statement number 23 and ends at statement number 42. Inside the compute() method, 'if' block is from statement number 26 to 31. The 'else' block is from statement 32 to 42. The call to fork() and join() methods are at statement number 37 and 40.

Table 1: Description of Nodes in JFJFG

Node No.	Description
23	Start of compute() method
26	Start of if() block
31	End of if() block
32	Start of else block
37	Call to fork()
40	Call to join
42	End of else block and compute() method

The Algorithm 4 given in the paper, generates the test sequences for a Java7 Fork/Join Flow Graph given as input. Whenever there is a call to fork() method, there are concurrent paths present in the structure of the program. Therefore, to traverse those paths, algorithm BFS is applied so as to cover those concurrent paths at the same time. Otherwise, the algorithm DFS is used for traversing the graph and hence finding the test sequences. The test sequences generated by the algorithm are as follows in the form of node numbers i.e. statement numbers:

23, 24 → 25... 31 → 42
23, 24 → 25 → 32... 36 → 37→ 38 → 39 → 40 → 41 → 42
23, 24 → 25 → 32... 36 → 38 → 39 → 37 → 40 → 41 → 42

Where *x... y* means nodes traversed from node *x* to node *y* in serial order.

Test Sequence 1:

Start of compute() method.
The threshold value is > difference of high and low, so 'if' part gets executed.
End of compute method.

Test Sequence 2:

Start of compute() method.

The threshold value is < difference of high and low, causing the 'else' block to execute.
If the algorithm finishes the work of fork() first, the order of execution would be like this test sequence. Or the algorithm BFS takes the left child into first consideration.

Test Sequence 3:

Start of compute() method.
The threshold value is < difference of high and low, causing the 'else' block to execute.
If the algorithm finishes the work of compute() first, the order of execution would be like this test sequence. Or the algorithm BFS takes the right child into first consideration.

5. CONCLUSION

Test Sequences for Java7 Fork/Join program have been generated on the basis of all node and all path coverage criteria considering the interference dependence, in which all the nodes, including the call to fork() and join() have been covered. The problem of test case explosion is avoided in this approach. In future, this work is to be extended for more coverage criteria. Also, design phase can be introduced in future for test sequence generation.

REFERENCES

[1] Allen, F. E. (July 1970). Control Flow Analysis. SIGPLANNot. 5(7) (July 1970), 1-19. DOI= http://doi.acm.org/10.1145/390013.808479.

[2] Bao, X., Zhang, N., & Ding, Z. (2009). Test Case Generation of Concurrent Programs Based On Event Graph. In Proceedings of Fifth International Joint Conference on INC, IMS and IDC, NCM, 143-149. DOI= http://dx.doi.org/10.1109/NCM.2009.39.

[3] Fork/Join, October 15, 2013.Accessed January 2, 2014: http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html.

[4] Grossman, D. Beginner's Introduction to Java's ForkJoin Framework. Accessed January 10, 2014: h ttp://homes.cs.washington.edu/~djg/teachingMaterials/grossmanSPAC_forkJoinFramework.html.

[5] Katayama, T., Furukawa, Z., & Ushijima, K. (1995). Event Interactions Graph for Test-Case Generation of Concurrent Programs. In Proceedings of Asia Pacific Software Engineering Conference, 29-37. DOI= http://dx.doi.org/10.1109/APSEC.1995.496951.

[6] Katayama, T., Itoh, E., Furukawa, Z., & Ushijima, K. (1999). Test-Case Generation for Concurrent Programs with the Testing Criteria Using Interaction Sequences. In Proceedings of Sixth Asia Pacific Software Engineering Conference, 590-597. DOI= http://dx.doi.org/10.1109/APSEC.1999.809654.

[7] Khandai, M., Acharya, A. A., & Mohapatra, D. P. (2011). A Novel Approach of Test Case Generation for Concurrent Systems Using UML Sequence Diagram. InProceedings of 3rd

International Conference on Electronics Computer Technology (ICECT), 1, 157-161. DOI= http://dx.doi.org/10.1109/ICECTECH.2011.5941581.

[8] Khandai, M., Acharya, A. A., & Mohapatra, D. P. (2011). A Survey on Test Case Generation from UML Model. International Journal of Computer Science and Information Technologies 2(3), 1164-1171.

[9] Kim, H., Kang, S., Baik, J., & Ko I. (2007). Test Cases Generation from UML Activity Diagrams. In Proceedings of Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD) 3, 556-561. DOI= http://dx.doi.org/10.1109/SNPD.2007.189.

[10] Kundu, D., & Samanta, D. (2009). A Novel Approach to Generate Test Cases from UML Activity Diagrams. Journal of Object Technology 8(3), 65-83. DOI= http://dx.doi.org/10.5381/jot.2009.8.3.a1.

[11] Lei, B., Wang, L. & Li, X. (2008). UML Activity Diagram Based Testing of Java Concurrent Programs for Data Race and Inconsistency. In Proceedings of 1st International Conference on Software Testing, Verification and Validation, 200-209. DOI= http://dx.doi.org/10.1109/ICST.2008.64.

[12] Mingsong, C., Xiaokang, Q., & Xuandong, L. (2006). Automatic Test Case Generation for UML Activity Diagrams. In Proceedings of the 2006 International Workshop on Automation of Software Test (AST '06). ACM, New York, NY, USA, 2-8. DOI= http://doi.acm.org/10.1145/1138929.1138931.

[13] Ranganath, V. P., & Hatcliff, J. (2004). Pruning Interference and Ready Dependence for Slicing Concurrent Java Programs. Compiler Construction, 39-56. DOI= http://dx.doi.org/10.1007/978-3-540-24723-4_4.

[14] Rapps, S., & Weyuker, E. J. (1985). Selecting Software Test Data Using Data Flow Information. IEEE Transactions on Software Engineering SE-11, 4, 367-375. DOI= http://dx.doi.org/10.1109/TSE.1985.232226.

[15] Shirole, M., & Kumar, R. (2012). Testing for concurrency in UML diagrams. SIGSOFT Software Engineering Notes 37(5), 1-8. DOI= http://doi.acm.org/10.1145/2347696.2347712.

[16] Sun C. (2008). A Transformation-Based Approach to Generating Scenario-Oriented Test Cases from UML Activity Diagrams for Concurrent Applications. In Proceedings of 32nd Annual IEEE International Computer Software and Applications COMPSAC, 160-167. DOI=http://dx.doi.org/10.1109/COMPSAC.2008.74.

[17] Yan, J., Li, Z., Yuan, Y., Sun, W., & Zhang, J. (2006). BPEL4WS Unit Testing: Test Case Generation Using a Concurrent Path Analysis Approach. In Proceedings of 17th International Symposium on Software Reliability Engineering ISSRE, 75-84. DOI= http://dx.doi.org/10.1109/ISSRE.2006.16.

[18] Yuan, Y., Li, Z., & Sun, W. (2006). A Graph-Search Based Approach to BPEL4WS Test Generation. In Proceedings of the International Conference on Software Engineering Advances(ICSEA '06). IEEE Computer Society, Washington, DC, USA, 14-. DOI= http://dx.doi.org/10.1109/ICSEA.2006.6.

[19] Zheng, Y., Zhou, J., & Krause, P. (2007). A Model Checking based Test Case Generation Framework for Web Services. In Proceedings of Fourth International Conference on Information Technology ITNG, 715-722. DOI= http://dx.doi.org/10.1109/ITNG.2007.8.