Journal of Engineering and Applied Sciences 13 (Special Issue 1): 2322-2329, 2018
ISSN: 1816-949X
© Medwell Journals, 2018

Generating test Cases for Model-Based Testing and Detecting Deadlocks Using Tarjan's Algorithm

Yasir Dawood Salman, Nor Laily Hashim, Mawarny Md Rejab,
Rohaida Romli and Haslina Mohd
Human Centered Computing Lab., Universiti Utara Malaysia, Kedah, Malaysia

Abstract: Test case generation is a task that greatly affects software testing. Model-Based Testing (MBT) has gained a significant role in generating test cases in recent years. Recent studies have also begun to generate executable test cases from Unified Modelling Language (UML). As a major issue in system execution, a model must recognize and identify deadlocks in the early stage of system testing. The current studies did not take into consideration deadlock detection also did not fulfil adequate coverage criteria. This study proposes an automated method for generating test cases from UML state chart diagrams that can help detect deadlocks during the design phase. These test cases are generated following specified coverage criteria. This method begins by converting the UML state chart diagram into an intermediate table and graph where the deadlocks are marked. The test path and cases are generated afterward. The generated test cases are deemed suitable for identifying faults and deadlocks in the early phase of software development.

Key words: UML statechart diagram, deadlock detection, software testing, model-based testing, recognize, generation

INTRODUCTION

Software testing is an important phase of software development that evaluates the functionality or capability of software systems and compares their actual and desired outcomes or performance (Bihani and Badyal, 2014). This task involves a variety of testing activities to verify whether a software satisfies the eligibility criteria specified by the customer (Kamalraj and Rajivkannan, 2016).

The automated generation of test cases has been proposed to reduce the challenges in test case generation (Korel, 1990). Given that the quality of manual testing depends on the experience and software design knowledge of the tester, an automated test case generation can provide effective test cases with the appropriate software design. This problem can also address those problems resulting from human errors and lack of testing experience. Clarke showed that using automated test case generation instead of a manual test case would reduce the test case generation time by 88%.

Given that the Unified Modelling Language (UML) diagrams created in the system follow certain specifications and designs, these diagrams can produce test cases earlier in the development lifecycle and test the system before the coding phase. Such early generation of test cases will enable software developers to find uncertainties and inconsistencies in the system specification and design (Jain and Sheikh, 2014).

Test cases are widely generated from source codes, but this practice presents a disadvantage. Specifically, software developers must finish the program before generating the test cases and generating test cases based on the system specification and design can help address this restraint (Doungsa-ard *et al*., 2008). Furthermore, generating test cases before the coding phase can help software developers develop a system for achieving the desired software design (Beck, 2003).

Software design and testing are both important in the lifecycle of software. Faultless software design helps software developers in developing a system and an excellent software design can help developers adjust to various software requirements during the software development process.

The early detection of errors has become a serious issue. As shown in Fig. 1, an error that is detected in the later phases is very costly to repair. Moreover, detecting a fault during the system testing is 10 times more costly than detecting the same fault during the system design. This same fault is up to 30 times more costly if detected during the system production. Therefore, test cases for the software system must be generated during the system design phase.

MBT has attracted many researchers for its ability to detect separate test cases for the software by using UML diagrams such as state chart, activity and sequence diagrams (Singh, 2014).

Corresponding Author: Yasir Dawood Salman, Human Centered Computing Lab., Universiti Utara Malaysia, Kedah, Malaysia

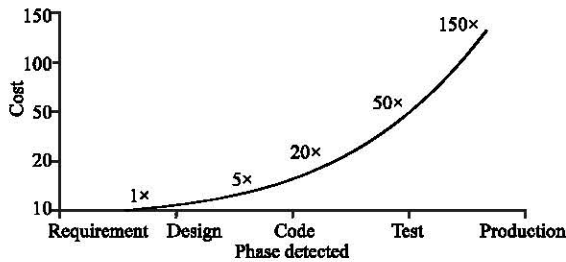

Fig. 1: Relative cost of software repair across each stage of the software lifecycle

UML diagrams have been used to characterize different issues in software design. However, generating test cases from software design than from program codesfaces several challenges (Samuel *et al*., 2008)that may be overcome by using the proper software system (Berardi *et al*., 2005). The UML state chart diagram can be used to represent the parallel activities and hierarchy relationships that are usually present in modern complex software (Silberschatz *et al*., 2013).

The UML state chart diagram is known for its ability to determine how the structure of the solution must appear at the most detailed level. This diagram can also show the implementation and interaction among different classes (Bell, 2003).

Many studies have addressed modelling problems by using UML diagrams to generate test cases. However, only few have used UML diagrams to find deadlocks in multiprogramming (Ansari, 2012). Deadlocks refer to those situationswhere a process or a set of processes is blocked by waiting for some resources that are holed in some other process (Galvin *et al*., 2013).

Accordingly, this study proposes an automated test case generation method that detects deadlocksby using an UML state chart diagram as an input from the system design documents.

Background: This study describes the basic concepts that are used in this study including model-based testing, UML statechart, deadlocks and Tarjan's algorithm which are all essential in analysing the proposed method.

Model-based testing: MBT allows software testers to update a model and quickly regenerate a new test case. This method is particularly effective in the system that is currently being tested and its specifications (Tewari and Misra, 2015).

MBT is suitable when the requirements are formally specified through graphical notations such as state charts and when the test cases are generated using the formal specifications. Using this method for software testing is generally preferred for the following reasons it can be easily understood by both business and developer communities itseparates the business rationale from the testing code it can quickly achieve automated testing it allows developers to switch testing instruments if required or utilize the same model in various stages it focuses on requirement coverage and it helps developers design more and code less.

UML state chart diagram: UML state chart diagrams present a dynamic view of the system (Aggarwal and Sabharwal, 2012) and identify those events that change the state of a system throughout its lifecycle. Well-formed control systems are known for their lack of deadlocks. Therefore, checking for this particular characteristic is an important step in the formal verification of control systems. As far as UML state chart diagrams represent hierarchical systems, the local deadlocks usually cannot block the whole system. However, detecting the local deadlocks is also important.

The quality of test casesin UML state chart diagrams is evaluated based on coverage criteria and feasibility. Several testing methods for UML state chart diagram have been proposed in recent years (Aggarwal and Sabharwal, 2012). This diagram describes the complete function of a system and all of its possible states. Therefore, UML state chart diagrams can reduce software "deadlocks" and other unexpected behaviour by forcing the software tester to consider each aspect of the software.

Deadlock: A system is deadlocked if each of its processes is waiting for an event that can only be triggered by another process in the same set (Ashfield *et al*., 2002). A deadlock may be classified into communication or resource deadlock. A communication deadlock occurs when process A tries to send a message to process B, process B tries to send a message to process C and process ctries to send a message to process A (Mallick *et al*., 2014). By contrast, resource deadlock occurs when several processes simultaneously try to have exclusive access to various resources including devices, files, locks and servers. This type of deadlock can be represented using a resource allocation graph. If the graph contains a cycle, then the system faces a deadlock (Mallick *et al*., 2014).

Deadlocks can be precisely described using a directed graph called the system resource-allocation graph (Ramesh, 2010). The set of nodes N is partitioned into two types of nodes, namely, $N = [N_1, N_2, ..., N_n]$ which comprises all active processes in the system and $R = [R_1, R_2, ..., R_m]$ which comprises all resource types in the system. The set of edges E include request and assignment edges.

Based on the definition of a resource-allocation graph, if the graph does not contain any loop, then no process in the system is deadlocked. Otherwise, a deadlock may exist.

Tarjan's algorithm: Tarjan's algorithm (Tarjan, 1972) is used to find Strongly Connected Components (SCC). This algorithm was originally introduced to identify the SCCs for a graph that is constructed from sets of Nodes N and Edges E. Each SCC represents a sub graph of the original directed graph in which a path that comprises a series of directed edges and nodes that connect each node to every other node in the sub graph. In other words, a cycle links all the nodes in the SCC unless this SCC only contains a single node. Condensing or collapsing the SCCs of a directed graph into single nodes may result in a directed acyclic graph that lacks any directed cycles. Traditional task scheduling or topological sorting algorithms are applied in this type of directed graph (Datla *et al*., 2011). However, as a result of the Depth-First Search (DFS) process that is performed by Tarjan's algorithm to traverse a directed graph, the order in which the algorithm identifies the SCCs corresponds to the reverse topological sort of the directed acyclic graph that is produced after identifying all SCCs (Tarjan, 1972). Other algorithms for identifying the SCCs of directed graphs have also been developed including the Kosaraju-Sharir algorithm (Sharir, 1981) and the path-based strong component algorithm (Gabow, 2000). However, Tarjan's algorithm is considered to be the most efficient and easiest algorithm to implement because this algorithm requires a linear search time on the order of N where N denotes the number of nodes in the directed graph (Tarjan, 1972). Comparatively, the Kosaraju-Sharir and path-based strong algorithms do not have the same scheduling or topological sorting capabilities of Tarjan's algorithm.

Tarjan's algorithm performs a DFS on the graph and then uses a stack to store the nodes that have been identified in the search yet are not placed into an SCC (Lowe, 2016). This algorithm also detects completed SCCs starting from the "deepest" level but only performs a single DFS and can readily detect those states that belong to the same SCC "on its way down". Therefore, this algorithm must "remember" the states by placing them on a second stack (Geldenhuys and Valmari, 2005).

Literature review: This study presents the survey-related work on MBT, proposes a test case generation technique that uses different UML diagrams and provides an overview of the deadlock detection technique. Under various circumstances, several studies have attempted to generate test cases through MBT based on the system specifications or design. MBT deals with various types of UML diagrams including activity, state chart and sequence diagrams.

Karatkevich (2003) proposed a method for detecting deadlocks in a system using UML state chart diagrams. The inputted UML state chart diagram was transformed into petri nets and the proposed method was used to detect the deadlocks in these nets. This method involves two steps, namely, static analysis, wherein the possibility of detecting deadlocks in a system is evaluated and dynamic analysis wherein the reachability of deadlocks from the initial state is measured. Dynamic analysis is performed by constructing the sub graphs of reachability graphs to avoid interleaving.

Ansari (2012) manually detected deadlocks in the early stage using UML diagrams and found that the UML diagram is a powerful modelling language to represent the scientific research problem. Ansari also proposed a modelling method for detecting deadlock situations in a system and showed that the UML model is a very useful and efficient tool that can help software developers avoid deadlock situations. UML diagrams are also flexible and can be easily extended.

Mallick *et al*. (2014) proposed a method for generating test cases from a UML sequence diagram and for detecting deadlocks using a loop detection algorithm. They converted the UML sequence diagram into an intermediate graph where deadlock points are marked and traversed to generate test cases. However, this method was not tested in real scenarios and failed to satisfy any coverage criteria.

Ali *et al*. (2014) proposed a test-case-based technique using the UML state diagram. They transferred the UML state chart diagram into an intermediate graph called the Finite State Machine (FSM). Each node in this graph stores the necessary information for the test case to be generated later. Ali *et al*., 2014 also used additional parameters for test case generation including pre- and post-conditions and object constrained language. Using FSM as input to breadth-first search, Ali *et al*. (2014) generatedand transformed all basic paths to obtain a suitable test case using the test set generation algorithm. Their generated test cases satisfied the transition, transition pair and state coverage criteria. However, apart from ignoring deadlocks and loops, these cases required additional inputs to satisfy the coverage criteria.

Jena *et al*. (2014) proposed a test case generation approach using the UML activity diagram. They converted the UML activity diagram into an activity flow table which in turn was converted intoan Activity Flow Graph (AFG). They used DFS to traverse the AFG and obtain all possible test paths. Afterward, they applied the genetic algorithm to generate the test cases. This algorithmwas then called the "Activity Test Case Generation using Simple Genetic Algorithm". However, their test cases only achieved activity node coverage and could not detect deadlocks.

MATERIALS AND METHODS

Proposed model for generating test cases: The proposed model transforms the inputted UML state chart diagram into an intermediate table and graph. Each node in the intermediate mode stores the necessary information for test case generation. The intermediate graphis traced to generate the basic paths. The suitable test case is generated and the possible deadlock is identified.

Figure 2 shows the model diagram of the proposed approach model which begins by using the UML state chart diagram as an input and followed by several steps to generate the test cases. The first step after getting the input is to convert it to intermediate table and intermediate graph the proposed algorithm will detect the possible deadlock from them. Then, the test paths will be generated to generate the final output the test case using test case generation algorithm. This study aims to generate test cases that can detect deadlocks. The UML state chart diagram goes through six processes to generate the test cases with high coverage and deadlock notification rate.

UML state chart diagram: UML state chart diagram is used to generate the test cases automatically. This diagram determines the behaviour of the system by analysing how its state changes in response to input data, to the model lifetime of a reactive system and to the different states of an object (Swain *et al*., 2012b). However, this process needs to go through few steps before generating the test cases. Figure 3 shows the UML state chart diagram applied in this research. The UML state chart will be transferred later into graph G which is expressed as follows:

$$G = (N, E) \quad (1)$$

The set of N consists of non-empty nodes while the set of E consists of several edges (Diestel, 2012) with each edge comprising a pair of nodes. For instance N = (N_1, N_2, N_3,) and E {($N_1 \rightarrow N_2$), ($N_2 \rightarrow N_3$), ...}. As shown in Fig. 4, graphs have natural visual illustrations in which each node is represented by a point and each edge is represented by a line that connects two points.

The UML state chart can be formally described as a quadruple S_c = (S_s, T, V_a, S_0) where S_s is a set of simple

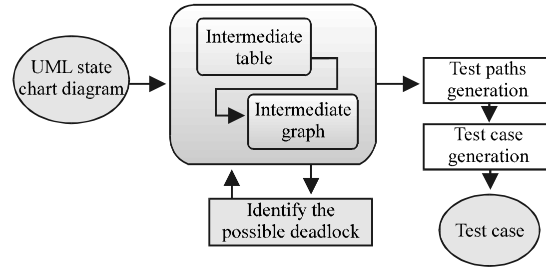

Fig. 2: Proposed approach model

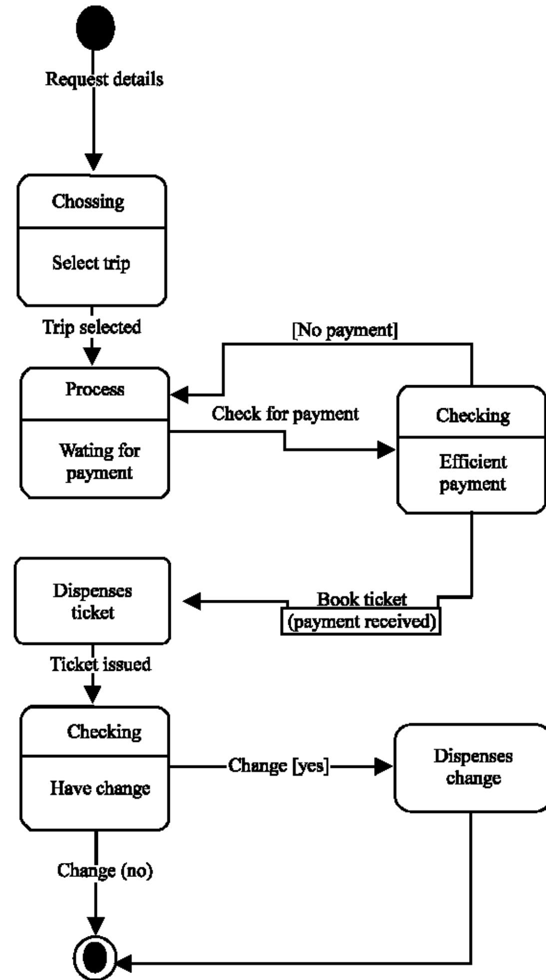

Fig. 3: UML state chart diagram for a ticket vending machine (Swain *et al*., 2012a)

Table 1: Intermediate table

N_i	N_j	Nods	Edge
S_0	1	Initial state	Request details
1	2	Select trip	Trip selected
2	3	Waiting for payment	Check for payment
3	2	Efficient payment	(No payment)
3	4	Efficient payment	Book ticket (payment received)
4	5	Dispenses ticket	Ticket issued
5	6	Have change	Change (yes)
5	d	Have change	Change (no)
6	d	Dispenses change	

nodes, T is a set of edges V_a is a set of variables used in the state chart and S_0 is the initial state of the state chart (Kot, 2003).

To explain the algorithm and its phases, a ticket vending machine was used as an example as shown in Fig. 3. This machine dispenses tickets to the customer after receiving monetary payment. The machine display lists all of the available trips. After the user selects a trip on the screen, the machine entersits waiting for payment state. The user then pays for the trip and the machine checks the payment. When the payment is greater than or equal the indicated price, a ticket will be dispensed. If needed, the machine dispenses change to the user. Otherwise, the machine terminates the process.

Intermediate table: The state table simplifies large systems in a comprehensive manner. The convenient tabular form specifies the states, inputs, transitions and output (Tewari and Misra, 2015). Each nodein the graph representsa state while each edge represents the transitions between two states (Diestel, 2012).

Given that this research uses E and N, the other elements such as d, represent the maximum number of nodes in a single graph. Given a node set (N_1, N_2, ..., N_i connected by as their relationships are presented as follows (Table 1).

Intermediate graph: The intermediate table was converted into an intermediate graph using the information storedinthe table. An intermediate graph is a directed graph that constructsa set of each node including initial, join, decision, guard condition and fork nodes, that represents each state. Each border of the intermediate graph symbolizes the stream in the UML state chart diagram.

The intermediate graph is expressed as. This research assumes that each graph starts with a unique node that corresponds to the initial state and ends in one node that represents the final states. The initial state

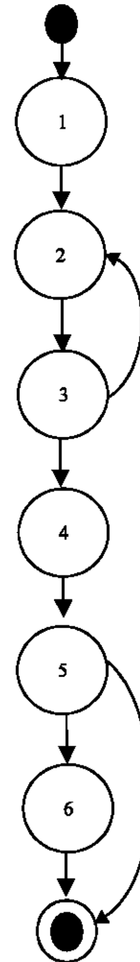

Fig. 4: State graph

is represented as the root of the tree. The nodes in the graph are then connected using the edges as shown in Fig. 4.

Detect the possible deadlock: This reserch uses the modified Tarjan's algorithm to find all deadlock paths in the system from the intermediate graph. The nodes are indexed as the algorithm traverses them. While returning from the recursion, each node N is assigned a node N as a representative. N is a node with the least index that can be reached from N. Those nodes with the same representatives are located in the same SCC Table 2.

In Fig. 4, an edge from "2"-"3" indicates that "3" is waiting for an event to be completed by process "2". Therefore, the system contains a deadlock because a cycle is presented. The algorithm in Alghorithm 1 detects the cycle and its output is 3→2.

Table 2: Test cases for detecting deadlocks

TC No.	Input	State	Expected output	Post condition
1	Request details, trip selected, check for payment, book ticket payment received], ticket issued, changed (yes)	Select trip, waiting for payment, dispenses payment	Dispenses change	Non-deadlocked path
2	Request details (trip selected, check for payment book ticket (payment received), ticket issued, change (no)	Select trip waiting for payment, dispenses payment	Dispenses ticket path	Non-deadlocked path
3	Check for payment (no payment)	Waiting for payment efficient payment		Deadlocked path

Alghorithm 1; Modified Tarjan's algorithm:

```
Tarjan's alghorithm
Input: graph G = (N, E)
Output: deadlock nodes (set of nodes)
1.   indez = 0
2.   Stacks s = empty
3.   For each node in N do
4.        If (node.index is undefined) then
5.          Strongconnect (node)
6.        end if
7.   end for
8.   function strongconnect (node)
9.        node. Index = index
10.       node.lowink = index
11.       index = index 1
12.       S.push (node)
13.       node.on Stock = true
14.       For each (node, N') in E do
15.         If (N'.index is undefined) then
16.              Strongconnect (N')
17.              node.lowlink = min (node.lowlink, N'lowlink)
18.         else if (N'.on Stack) then
19.              node.lowlink = min (node.lowlink, N'.index)
20.         end if
21.       end for
22.       If (node.lowlink = node.index) then
23.         while (N' = node) do
24.              N'.onStack = false
25.              add N' to deadlock nodes
26.         End do
27.         output deadlock nodes
28.         end if
29.  end function
```

RESULTS AND DISCUSSION

Generate the test paths: After converting the UML state chart diagram into the intermediate graph, the DFS traverses the graph to generate the basic paths. The initial path can be easily monitored using DFS. This algorithm also helps achieve all-states, all-transition, all-transition-pair and all-on-loopcoverage. Given that DFS cannot handle loops, all loop states are treated as simple states during the DFS traversal. For example, within a loop state, the traversal begins with the default initial state or at an entry point. During the traversal, the conditional predicates on each transition are examined.

The number of basic paths is calculatedas follows using the modified McCabe's formula (Hakansson and Badran, 2016):

$$BP = E - N + 2 \quad (2)$$

Where:

PB = Basic path cyclomatic complexity
E = Number of edges of the graph
N = Number of nodes of the graph

Therefore, the basic path of this graph is computed as 8-8+2 = 2:

Possible test paths:

Tp 1: [S→1→2→3→4→5→6→E]

TP: [S→1→2→3→4→5→E]

The possible unique paths generated from the state chart graph are presented in Fig. 6.

Test cases that detect deadlocks: The test cases are generated from the paths that have been identified from the intermediate graph. The generated test cases can be created based on the information stored in the intermediate table and all paths from the start node to the end node are counted. Each path is considered a test case in addition to any loop path that may exist. Three test cases are generated from the example presented in the rows of 2. Column 1 presents the test case number, Column 2 shows the test case input, Column 3 presents the state, Column 4 presents the expected output and Column 5 presents the post condition for the test case deadlock condition.

As shown in Table 2, the proposed method detect the deadlock path in the third state, what allow the software tester to correct their design for this error not to be appear in the final coding of the system.

CONCLUSION

This study proposed a method for generating test casesfrom UML state chart diagrams that can detect deadlocks. This approach can help software developers and testers complete the testing process quickly in the software development lifecycle. The test cases were generated by transforming the UML statechart into an

intermediate graph. The proposed algorithms generated all possible path and loop nodes for generating those test cases that will be used for detecting deadlocks. The proposed method achieves all-states, all-transition, all-transition-pair and all-on-loop coverage. An example was also presented to show that the proposed method could produce all paths as well as detect and highlight the possibility of deadlocks.

RECOMMENDATIONS

Future studies may build a fully automatic tool using this method and further research must be done to combine the other UML diagrams included in this method.

REFERENCES

Aggarwal, M. and S. Sabharwal, 2012. Test case generation from UML state machine diagram: A survey. Proceedings of the 3rd International Conference on Computer and Communication Technology (ICCCT'12), November 23-25, 2012, IEEE, Allahabad, India, ISBN:978-1-4673-3149-4, pp: 133-140.

Ali, M.A., K. Shaik and S. Kumar, 2014. Test case generation using UML state diagram and OCL expression. Intl. J. Comput. Appl., 95: 7-11.

Ansari, G.A., 2012. A modeling and detection of dead lock in early stage of system using UML. Int. J. Comput. Appli., 39: 16-20.

Ashfield, B., D. Deugo, F. Oppacher and T. White, 2002. Distributed deadlock detection in mobile agent systems. Proceedings of the 15th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems: Developments in Applied Artificial Intelligence (IEA-AIE'02), June 17-20, 2002, ACM, Cairns, Queensland, Australia, ISBN:3-540-43781-9, pp: 146-156.

Beck, K., 2003. Test-Driven Development: By Example. Addison-Wesley, Boston, Massachusetts, ISBN:0-321-14653-0, Pages: 219.

Bell, D., 2003. UML basics part II: The activity diagram. Rational Software, San Jose, California, USA. http://isabelle.vincentvanrooijen.com/container%5Cprocess%5CRisk%20reduction%20with%20the%20RUP%20Phase%20Plan.pdf.

Berardi, D., D. Calvanese amd G. de Giacomo, 2005. Reasoning on UML class diagrams. Artif. Intell., 168: 70-118.

Bihani, A. and S. Badyal, 2014. Optimizing software testing and test case generation by using the concept of Hamiltonian paths. Intl. J. Eng. Trends Technol., 10: 318-322.

Datla, D., H.I. Volos, S.M. Hasan, J.H. Reed and T. Bose, 2011. Task allocation and scheduling in wireless distributed computing networks. Analog Integr. Circuits Signal Process., 69: 341-341.

Diestel, R., 2012. Graph Theory. Springer, Berlin, Germany, Pages: 437.

Doungsa-ard, C., K. Dahal, A. Hossain and T. Suwannasart, 2008. GA-Based Automatic Test Data Generation for UML State Diagrams with Parallel Paths. In: Advanced Design and Manufacture to Gain a Competitive

Edge, Yan, X.T., C. Jiang and B. Eynard (Eds.). Springer, Berlin, Germany, ISBN: 978-1-84800-240-1, pp: 147-156.

Gabow, H.N., 2000. Path-based depth-first search for strong and biconnected components. Inf. Process. Lett., 74: 107-114.

Geldenhuys, J. and A. Valmari, 2005. More efficient on-the-fly LTL verification with Tarjan's algorithm. Theor. Comput. Sci., 345: 60-82.

Hakansson, J. and S. Badran, 2016. Evaluating cyclomatic complexity on functional JavaScript. BA Thesis, Linnaeus University, Sweden.

Jain, E.S. and E.M. Sheikh, 2014. A novel test case generation method through metamorphic priority for 2-way testing method UMBCA implementation criteria. Intl. J. Eng. Manage. Res., 4: 157-163.

Jena, A.K., S.K. Swain and D.P. Mohapatra, 2014. A novel approach for test case generation from UML activity diagram. Proceedings of the 2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT'14), February 7-8, 2014, IEEE, Ghaziabad, India, ISBN:978-1-4799-2900-9, pp: 621-629.

Kamalraj, R. and A. Rajivkannan, 2016. Test case classification using tuned fuzzy logic with test case reusability for test suite reduction. Asian J. Inf. Technol., 15: 1437-1442.

Karatkevich, A., 2003. Deadlock analysis in statecharts. Proceedings of the 2003 Conference Forum on Specification and Design Languages (FDL'03), September 23-26, 2003, Goethe University Frankfurt, Frankfurt, Germany, pp: 414-425.

Korel, B., 1990. Automated software test data generation. IEEE. Trans. Software Eng., 16: 870-879.

Kot, M., 2003. The state explosion problem. MCS Thesis, Faculty of Electrical Engineering and Computer Science, Technical University of Ostrava, Ostrava, Czech Republic.

Lowe, G., 2016. Concurrent depth-first search algorithms based on Tarjan's algorithm. Intl. J. Software Tools Technol. Transfer, 18: 129-147.

Mallick, A., N. Panda and A.A. Acharya, 2014. Generation of test cases from UML sequence diagram and detecting deadlocks using loop detection algorithm. Intl. J. Comput. Sci. Eng., 2: 199-203.

Ramesh, S.V., 2010. Principles of Operating Systems. Laxmi Publications Pvt. Ltd., Ghaziabad, India, ISBN:978-93-80386-17-1, Pages: 189.

Samuel, P., R. Mall and A.K. Bothra, 2008. Automatic test case generation using Unified Modeling Language (UML) state diagrams. IET. Software, 2: 79-93.

Santiago, V., N.L. Vijaykumar, D. Guimaraes, A.S. Amaral and E. Ferreira, 2008. An environment for automated test case generation from statechart-based and finite state machine-based behavioral models. Proceedings of the IEEE International Conference on Software Testing Verification and Validation Workshop ICSTW'08, April 9-11, 2008, IEEE, Sao Jose Dos Campos, Brazil, ISBN: 978-0-7695-3388-9, pp: 63-72.

Sharir, M., 1981. A strong-connectivity algorithm and its applications in data flow analysis. Comput. Math. Appl., 7: 67-72.

Silberschatz, A., P.B. Galvin and G. Gagne, 2013. Operating System Concepts Essentials. 2nd Edn., John Wiley & Sons, Hoboken, New Jersey, USA., ISBN:9781118844007, Pages: 784.

Singh, R., 2014. Test case generation for object-oriented systems: A review. Proceedings of the 4th International Conference on Communication Systems and Network Technologies (CSNT'14), April 7-9, 2014, IEEE, Bhopal, India, ISBN:978-1-4799-3069-2, pp: 981-989.

Straunstrup, J., H.R. Andersen, H. Hulgaard, J. Lind-Nielsen and G. Behrmann *et al*., 2000a. Practical verification of embedded software. Comput., 33: 68-75.

Swain, R.K., P.K. Behera and D.P. Mohapatra, 2012. Generation and optimization of test cases for object-oriented software using state chart diagram. Software Eng., Vol. 1,

Swain, R.K., P.K. Behera and D.P. Mohapatra, 2012b. Minimal test-case generation for object-oriented software with state charts. Intl. J. Software Eng. Appl., 3: 1-21.

Tarjan, R., 1972. Depth-first search and linear graph algorithms. SIAM J. Comput., 1: 146-160.

Tewari, A. and A.K. Misra, 2015. An approach to model based test case generation for student admission process. Intl. J. Innovative Sci. Eng. Technol., 2: 818-825.