International Journal of Electrical and Computer Engineering (IJECE)
Vol. 8, No. 6, December 2018, pp. 5449~5456
ISSN: 2088-8708, DOI: 10.11591/ijece.v8i6.pp5449-5456

Test Case Optimization and Redundancy Reduction Using GA and Neural Networks

Itti Hooda, R.S. Chhillar
Department of Computer Science and Applications, Maharishi Dayanand University, India

Article Info

Article history:

Received Dec 26, 2017
Revised Jun 29, 2018
Accepted Jul 22, 2018

Keyword:

Automation testing
Software testing life cycle
Test driven development
Test optimization
Verification

ABSTRACT

More than 50% of software development effort is spent in testing phase in a typical software development project. Test case design as well as execution consume a lot of time. Hence, automated generation of test cases is highly required. Here a novel testing methodology is being presented to test object-oriented software based on UML state chart diagrams. In this approach, function minimization technique is being applied and generate test cases automatically from UML state chart diagrams. Software testing forms an integral part of the software development life cycle. Since the objective of testing is to ensure the conformity of an application to its specification, a test "oracle" is needed to determine whether a given test case exposes a fault or not. An automated oracle to support the activities of human testers can reduce the actual cost of the testing process and the related maintenance costs. In this paper, a new concept is being presented using an UML state chart diagram and tables for the test case generation, artificial neural network as an optimization tool for reducing the redundancy in the test case generated using the genetic algorithm. A neural network is trained by the back-propagation algorithm on a set of test cases applied to the original version of the system.

Corresponding Author:

Itti Hooda,
Department of Computer Science and Applications,
Maharishi Dayanand University, Rohtak,
Pin Code – 124001, Haryana, India.
Email: ittihooda01@gmail.com

1. INTRODUCTION

Testing is defined as a process of evaluation that either the specific system meets its originally specified requirements or not. It is fundamentally a procedure including approval and confirmation process that whether the created framework meets the necessities characterized by client. Subsequently, this action brings about a contrast amongst real and expected outcome. Programming Testing alludes to discovering bugs, mistakes or missing necessities in the created framework or programming. Along these lines, this is an examination that furnishes the partners with the correct learning about the nature of the item.

Software Testing can also be considered as a risk-based activity. The important thing while testing process the product analyzers must comprehend that how to limit an extensive number of tests into sensible tests set and settle on insightful choices about the dangers that are imperative to test or what are not. [1] Figure 1 demonstrates the testing expense and blunders found a relationship. The Figure 1 unmistakably demonstrates that cost goes up drastically in testing the two sorts i.e. utilitarian and nonfunctional. The basic leadership for what to test or diminish tests then it can cause to miss many bugs. The viable testing objective is to do that ideal measure of tests with the goal that additional testing exertion can be limited.

According to Figure 1, Software testing is an important component of software quality assurance. The importance of testing can be considered from life-critical software (e.g., flight control) testing which can

be highly expensive because of risk regarding schedule delays, cost overruns, or outright cancellation [2], and more about this [3], [4].

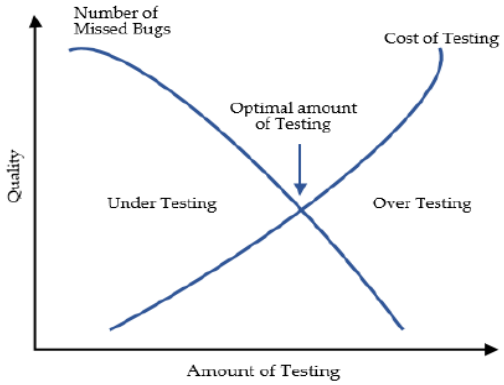

Figure 1. Every Software Project has Optimal Test Effort [1].

The primary target of testing a software module is to decide how well an assessed application fits in with its requirements. Two normal ways to deal with programming testing are black-box and white-box testing. While the white-box approach utilizes the written code of the program under test to play out its testing, the black box approach checks the program yield against the contribution without considering its inward workings. Software testing is well explained into three phases: generation of test data, application of the data to the software being tested, and evaluation of the results. In past, programming testing was done physically by a human analyzer who picked the test cases and investigated the outcomes of the software module. Now days, because of the expansion in the number and size of the projects being tried in present days, the burden on the human analyzer is more, and alternatively, some automatic programming testing strategies are required. While automatic strategies seem to consider the control over the part of the human analyzer, the issues of reliability, quality and the capacity of the product testing techniques still should be verified. Along these lines, testing is a critical viewpoint in the outline of a software product [5].

Automated test case generation is process of generating the test data on the basis of usage of the application and real time scenario. When the process is automated then it is quite possible that the process might produce the case, which are same in some manner for which the output is same for all instances, the similar types of test cases will not affect the efficiency of the software module but will affect the efficiency of the testing process in terms of the running time. The two functions of redundancy are passive redundancy and active redundancy. Both functions prevent performance decline from exceeding specification limits without human intervention using extra capacity.

Passive redundancy uses excess capacity to reduce the impact of component failures. One common form of passive redundancy is the extra strength of cabling and struts used in bridges. This extra strength allows some structural components to fail without bridge collapse. The extra strength used in the design is called the margin of safety.

Active redundancy eliminates performance declines by monitoring the performance of individual devices, and this monitoring is used in voting logic. The voting logic is linked to switching that automatically reconfigures the components. Error detection and correction and the Global Positioning System (GPS) are two examples of active redundancy.

There are some of the challenges related to the concept, test case generation is having some of the defined challenges some of them which are considered in the work are:

C1: Defining the complete requirements clearly and complete, C2: lack of ability to identify the critical domain requirements, C3: Functional requirements definition gap, for the efficient test case generation better elaboration of the requirement is being required. In the proposed work UML diagram is being used for defining the functional and other related requirement of the system module under test. C4: Redundancy in test case generation process, in the case when the complete functional and other related parts of the system is being elaborated for better coverage then the generation process sometime generates duplicate test case for which in the proposed work ANN is being used for reducing the redundancy in the generation of the test cases.

2. RELATED WORK

UML activity diagram-based test case generation has been investigated in [6] by Lizhang et al. They have produced test cases utilizing a grey box strategy. In their approach, test situations are straightforwardly gotten from the activity charts demonstrating an operation. This technique manages the logical coverage criteria of white box strategy and discovers all the conceivable ways from the design model which explains the normal conduct of an operation. Along these lines, all the data for generation of the test case (i.e. input/output grouping parameters, the constraints conditions and expected object method strategy) is extricated from each test situations. At long last, they produce the possible values of all the information/yield parameters by applying category partition strategy. It creates test cases which can accomplish the path coverage. In any case, this technique disregards data about the condition of the items inside the framework at the time of execution.

The approach introduced by Andrews et al. [7] recognized thin strings from top-level UML activity diagram. A thin thread is a base use situation in a software framework. It signifies a total situation from the end client's perspective. That is, the framework takes input information, plays out a few calculations, and yields the outcome. They proposed a novel methodology to produce thin threads from activity diagram, which included preprocessing of the system level activity charts, changing over them into action hyper diagrams and after that getting all execution paths from the diagram. Their technique does not contain any state data for the objects of the framework. Chen Mingsong et al. [8] displayed a plan to get the decreased test suite for a usage utilizing activity graphs. They considered the arbitrary era of experiments for Java programs. Running the projects with applying the experiments, they got the traces of the program execution. At long last, a diminished test suite is gotten by contrasting the basic ways and program execution follows. Simple path coverage criterion helps to avoid the path explosion due to the presence of loops and concurrency.

Kansomkeat and Rivepiboon [9] have proposed a technique for creating test cases utilizing UML state charts diagram. They change the state chart graph into a smoothed various leveled structure of states called testing flow diagram (TFG). The TFG is then crossed from the root node to the leaf node to create test cases. From the TFG, they list conceivable event sequence which they consider as test sequence. The testing basis they used to direct the era of test sequence is the scope of the states and changes of a TFG. This strategy manages a specific state chart diagram. However, in an execution of a use case, more than one object regularly takes an interest. Such conduct would be hard to test utilizing this approach.

Kim et al. [10] proposed a technique to produce test cases for class level testing utilizing UML state graph charts. They changed state diagrams to extended finite state machine (EFSMs) to infer test cases. The various leveled and simultaneous structure of states is smoothed and communicated correspondences are dispensed with in the subsequent EFSMs. Then, data flow is identified by transforming EFSMs into flow graphs to which conventional data flow analysis techniques are applied. Hartmann et al. [11] enlarge the UML depiction with particular documentations to make a design-based testing condition. The engineers initially characterize the dynamic conduct of every framework part utilizing a state chart diagram. The associations between modules are then determined by explaining the state chart graphs, and the subsequent global FSM that relates to the incorporated framework conduct is utilized to create the tests.

Gnesi et al. [12] gave a mathematical way to deal with conformance testing and automated test case generation for UML state diagrams. They proposed a formal conformance testing connection for input-empowered transition frameworks with advances named by input/output sets (IOLTSs). Testing programming so as to set up the satisfaction of the predetermined prerequisites is known as conformance testing. A conformance connection characterizes the effectiveness of the execution as for the formal determination. IOLTSs give a reasonable semantic model to a conduct spoke to by a subset of state diagrams. They additionally give an algorithm which produces a test suite for a given state chart model.

Ali et al. [13] have proposed an approach for state-based integration testing. Their work constructs a transitional test model called SCOTEM (State Collaboration Test Model) from UML collaboration diagram and the corresponding state charts. SCOTEM models every conceivable way for object state coverage criteria that a message pattern may trigger. SCOTEM at that point creates test ways in view of different coverage criteria. Their produced test cases intend to reveal state dependent interaction faults. Their work considers the scope of every single conceivable condition of collaboration among classes in a communication. Briand et al. [14] have considered communications among objects in their work, however their attention is again on class-level testing. Their work delivers an experiment detail comprising of a feasible sequence of transition. In their work, to catch the connections among state dependent objects, an invocation sequence tree is built which is then used to derive test constraints for the transition sequences to be tested.

3. PROPOSED METHODOLOGY

Programming testing plays a critical part in software development since it can limit the development cost. Programming testing approaches are partitioned into three sections i.e. code-based testing, specification-based testing and model-based testing. In model-based testing, the testing begins at design phase. Thusly, early revelation of lacks can be refined by using this approach moreover reducing time, cost and endeavors of the programmer to an extensive degree. In this paper a procedure is proposed for making test cases using UML diagram, genetic algorithm and the ANN (Artificial Neural Network).

In the initial stage a rich and finish UML graph is imported which will be parsed to extricate fundamental meta-data and shape data to populate a statechart table which will comprise of subtle elements of the individual graph. This will additionally help us to define the test cases utilizing the approach examined under the heading Test Case Generator [15].

In the second stage we change over a model graph by extricating the data by parsing method and populating this information in a decision table. Test cases are obtained using all combinations using Genetic Algorithm and further optimized using Artificial Neural Network.

In the first stage we import a model chart of the module of software under test. Again, we will be concentrating just on the activity and statechart graphs made according to the UML 2.0 principles. We accept the chart to be finished and rich yet there is a chance for the client to import an UML graph which may not be in similarity with UML 2.0 benchmarks. In the paper [16] V. Mary Sumalatha and Dr. G. S. V. P. Raju talked about a simple approach to expel ambiguities from activity charts. This can be connected to influence the graph to free from imperfections.

The graph imported will be parsed to obtain meta-data. In the event of an activity graph we recover the swim-lane actors, swim-lane specific actions, decision node, fork node, join node, connectors, dependencies and also its relationship with the previous and next shape. Transformation of any graph to test cases includes parsing as its first phase. Alteration of statechart graph to statechart table is straightforward. As for our first approach, we found that changing over any graph to a statechart graph would profit us instead of framing a different procedure for each kind of graphs.

Presently concerning the state chart graph, we extricate data, for example, the states, composite states, submachine states, decision nodes, connectors, conditions and furthermore its association with the past and next shape. A statechart table is an optional method for describing sequential modal logic. Rather than drawing states and transitions graphically in a statechart graph, the modal logic is communicated in a tabular organization. We will be changing over statechart graph into statechart table since it is a brisk, fresh configuration for a statechart graph. They additionally lessen the upkeep of graphical items. Not at all like statechart graphs, will expansion and erasure of states into a statechart table discard the over-head of modifying states, advances and intersections.

After the activity graph is parsed and significant shape data is extricated our framework will isolate activities in light of on-screen characters in two classes, client activities and framework produced activities. Client activities in the swim-lane diagrams exceptionally well portray the info conditions to be incorporated into the decision table. A decision hub likewise depicts the conditions on which the application under test will depend. In our approach client activities and decision hubs will act like conditions to the decision table.

The framework created activities represents output activities to be completed in the decision table. Condition options or generally called combinations are created utilizing Cartesian product. This is likewise called as exhaustive testing. These condition options will be in True/false shape. To get the estimation of the normal output we follow the UML Diagram considering it as a tree with the beginning node going about as the root. Beginning with the root node we go through the tree to discover the output activity whose normal result is to be computed. Based on the combination column values we decide the direction of the traversal. At whatever point there is an activity and a true output is gotten from the groups for the individual activity we move forward. In the event that a false output is gotten we stop and restore a false value to the normal outcome segment. At whatever point there is a decision hub in the way to the output activity the choice of which child to move relies upon the mix values for that choice. At whatever point there is a fork hub in the way to the output activity all the offspring of the fork hub are navigated one by one to search for the output state.

The statechart table as of now contains all the data important to make a test case. While making test cases from statechart graph care ought to be taken that every one of the transitions are practiced in any event once. This strategy for testing guarantees ideal scope without creating expansive number tests [17].

Till the time, output of the methodology is the set of certain test cases and hence so as to provide the maximum code coverage, different combinations of the test cases are generated using the Genetic algorithm. Genetic algorithms use the following three operations on its population.

a. Selection:
   A selection process is applied to determine a way in which individuals are chosen for mating from a population based on their fitness.
b. Crossover:
   After the selection process, the crossover operation is applied to the chromosomes selected from the population.
c. Mutation:
   After the crossover process, the mutate operation is applied to a randomly select subset of the population [18].

After applying the Genetic Algorithm there are different test cases obtained by different combinations of existing test cases that means the redundancy of the test cases increases so optimization of the test cases is being done using the Artificial Neural Network. ANN is being used like if the newly generated test case is matching with the already existing test case in the list. If yes, then it will just remove the generated test case and else the generated test case is added to the test pool.

Artificial neural networks (ANNs) are made out of straightforward components which work simultaneously. A NN system can be prepared to play out a specific part by changing the weights between components. System work is controlled by the associations between components. There is activation function used to create applicable outcome. Input process [19] with NN organize that including weights delivered outcome. The output is contrasted with the expected outcome, if the delivered outcome perfectly matches with output then the information is right else make an adjustment in the weight. NN system essentially worked with weights. The back propagation neural network used in this experiment is capable of generalizing the training data; however, the network cannot be applied to the raw data, as the attributes do not have uniform presentation. Some input attributes are not numeric, and in order for the neural network to be trained on the test data, the raw data have to be preprocessed, and, as the values and types differ for each attribute, it becomes necessary that the input data are normalized. The values of a continuous attribute vary over a range, while there are only two possible values for a binary attribute (0 or 1). Thus, for the network to be able to process the data uniformly, the continuous input attributes have to be normalized to a number between 0 and 1. The output attributes were processed in a different manner. If the output attribute was binary, the two possible network outputs are 0 and 1. On the other hand, continuous outputs cannot be treated in the same way, since they can take an unlimited number of values [20].

The simplest function that does this is the step function.

The step function is defined as:

$$H(x) = \begin{cases} 1 \ if \ x \geq 0 \\ 0 \ if \ x < 0 \end{cases} \quad (1)$$

Thus, calculating the output of our neuron model is comprised of two steps:

1) Calculate theintegration. The integration, as defined above, is the sum $\overrightarrow{w}.\overrightarrow{x}+b$ for vectors,w, x and b scalar.
2) Calculate theoutput. The output is the activation function applied to the result of step 1. Since the activation function in our model is the step function, the output of the neuron is $H(\overrightarrow{w}.\overrightarrow{x}+b)$, which is1when $\overrightarrow{w}.\overrightarrow{x}+b$>0 and0otherwise.

There is a continuous approximation of the step function called the logistic curve, orsigmoid function, denoted as. This function's output ranges over all values between0 and1and makes a transition from values near0to values near1at at x=0, similar to the step function H(x).

Sigmoidal Function is represented as:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (2)$$

Step by step execution of the proposed methodology

Step 1: UML diagram of the system module under is imported (state and activity diagrams are considered),

Step 2: UML diagram is converted into the decision table and state chart table using parser for information extraction,

Step 3: Test case generation,

Step 4: Now the test case generated using the decision table are regenerated using Genetic algorithm for better code coverage using crossover and mutation,

Step 5: The redundancy of test cases is increased because of the regeneration of the test cases using the Genetic Algorithm, hence Optimization is done,

Step 6: Optimization of the test cases using Artificial Neural Network.

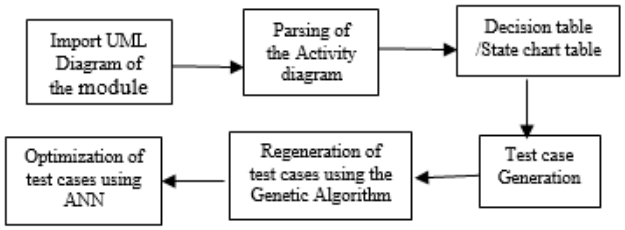

Figure 2. Block diagram of the proposed methodology

Example illustrating the methodology:

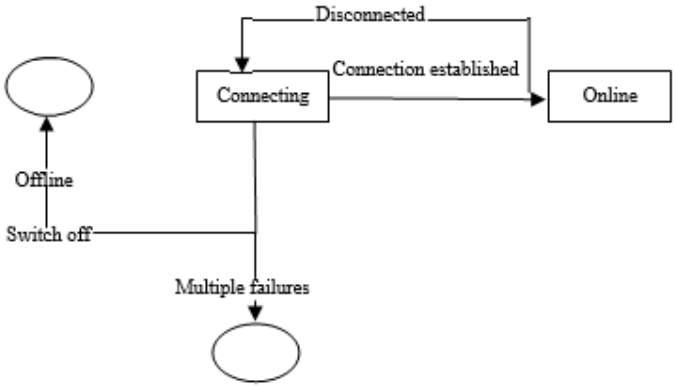

Figure 3. Activity diagram of the Connection making process.

Considering the example of the connection making process of any device with different possibilities, the UML diagram of the module showing the connection making process:

Table 1. State chart table corresponding to activity diagram in figure 3.

State	Offline	online	Connecting	Error
Offline	-	-	Switch ON	-
Online	-	-	-	-
Connecting	-	-	-	Multiple failure
Error	Switch off	Connection established	-	-

Test cases are considered from the information obtained from the merged graphs of the software module under test, considering two test cases for an example as 11 and 18, know obtain the binary equivalent of the test cases,

Binary equivalent of 11 is 00001011

Binary equivalent of 18 is 00010010

Apply crossover (Genetic Algorithm) at the 4 bits of the obtained binary string of the test cases,

New binary strings after crossover are 00000010 and 00011011, compute the decimal equivalent of the obtained binary strings as:

00000010=2, 00011011=27

2 and 27 are two expected test cases for the system module under test hence know the test suite T is written as:

T= {11, 18, 2,27}, similarly we can go more crossover by choosing other binary bit position onto the binary bits and also the genetic algorithm can also be applied between the newly obtained test cases and already existing test cases. Now after obtaining the expected test cases the ANN checks whether we have the same test case already in the pool of the test cases written as T. For that the ANN assigns weights to the input (new test cases) and forward the same towards the hidden layers of the network and after processing from the hidden layer to output layer as 0 or 1 where 0 signifies not present and 1 signifies presence of the test case in the test pool. The ultimate goal of the ANN is to reduce the redundancy of the test cases generated after applying genetic algorithm for generating the expected test cases for better testing of the system module.

4. RESULT ANALYSIS

The proposed methodology is qualitatively analyzed on the basis certain factor as efficiency means the ability of the methodology to detect the error in the available code, number of test cases generated which defines like when the test cases generated are more then there is more chance of testing the module which is under test, code coverage. Redundancy of the test cases that means to ensure that whether the generated test cases are repeated in the final list, code coverage means level to which the test cases covering the code or testing the code:

Table 1 Qualitative Analysis of the related work and the proposed methodology.

Parameters Methodologies	Efficiency	No. of Test cases generated	Redun dancy	Code Coverage
1.	Medium	Medium	High	Moderate
2.	Low	Medium	High	Moderate
3.	Medium	Medium	Moderate	Moderate
4.	Medium	Medium	High	Low
5.	Medium	Medium	High	Low
6.	High	Increased	Low	High

Table 2 Factors and Impact Ratio.

Factors	Impact in Lizhang et al [6]	Impact in Andrews et al. [7]	Impacts in Proposed Methodology
Regulated Deoiction	55%	50%	60%
Valid and Invalid inputs	38%	35%	45%
Good detaillevel	44%	48%	50%
Variation in test cases	75%	70%	80%
Understand System framework	50%	55%	60%
Clear beginning position	60%	55%	65%
Test design techniques	80%	70%	84%
Accurate suppositions	70%	78%	80%
Test caes assessment	50%	45%	55%
Tidy up after execution	78%	70%	85%

These factors are playing very important role in test cases. By analyzing it finds that elaborated factors having different level of influence on test cases for different techniques discussed in the literature review. All these factors have different impact ratio and these entire ratios described in table 2. Here are four sets of test cases, in which maximum impact of regulated depiction is 60% for the proposed methodology and is better than all discussed in the review. For different techniques considered, maximum impact of input analysis is 38% is for the proposed methodology. This study indicates that 29% of the subjects did not read the direction of conveyance of experiment and 16% did not finish the whole layout, so it's impact 45% for the proposed methodology.

5. CONCLUSION

Testing is the most critical part of the Software Development Lifecycle, as it is something upon which the final delivery of the product is dependent. It is time consuming and an intensive process, therefore, enhanced techniques and innovative methodologies are requisite. For which the efficient testing methodology is required which can generate test cases as many as possible with considering the factor like redundancy and also should provide maximum code coverage. In the proposed methodology the UML algorithm, Genetic algorithm and lastly the trained ANN is being used for reducing the redundancy of the generated test cases. The UML algorithm in the form of the Sequence diagram and the state diagram is used for the better representation of the software module working so that the test case gives the maximum code coverage. Genetic algorithm is being used for generating all possible test cases for the module under test. As per the results the algorithm provides the maximum efficiency and code coverage along with dealing with the redundancy of the generated test cases.

REFERENCES

[1] P. Ron, “Software testing,” *Indianapolis: Sam’s*,vol. 2, 2001.
[2] S. Amland, "Risk-based testing," *Journal of Systems and Software*, vol. 53(3), pp. 287–295, Sep. 2000.
[3] Redmill and Felix, “Theory and Practice of Risk-based Testing,*” Software Testing, Verification and Reliability*, vol. 15(1), Mar 2005.
[4] B. Agarwal, et al., “Software engineering and testing,” *Jones & Bartlett Learning*, 2010.
[5] K. Bogdan, “Automated software test data generation,” *Software Engineering, IEEE Transactions*, pp. 870-879, vol. 16(8), 1990.

[6] Jacobson, et al., "The unified software development process," *Reading: Addison-Wesley*, vol. 1, 1999.
[7] Everett, et al., "Software testing: testing across the entire software development life cycle," *John Wiley & Sons*, 2007.
[8] Verma, "Automated Test case generation using UML diagrams based on behavior," *International Journal of Innovations in Engineering and Technology (IJIET)*, vol. 4, issue 1, pp. 31-39, Jun 2014.
[9] M. Vanmali, M. Last, A. Kandel, "Using a Neural Network in the Software Testing Process," *International Journal Of Intelligent Systems*, vol. 17, pp. 45-62, 2002.
[10] J. John, "A Performance Based Study of Software Testing using Artificial Neural Network," *International Journal of Logic Based Intelligent Systems*, vol. 1(1), pp. 45-60, 2011.
[11] Sharma, R. Patani and A. Aggarwal, "Software Testing Using Genetic Algorithms," *International Journal of Computer Science & Engineering Survey (IJCSES)*, vol.7(2), pp. 21-33, Apr 2016.
[12] S. Keshavarz and R. Javidan, "Software Quality Control Based on Genetic Algorithm," *International Journal of Computer Theory and Engineering*, vol. 3, Issue 4, 2011.
[13] Soniya Malik, "Software Testing Using Genetic Algorithm," *International Conference on Technologies for Sustainability- Engineering, Information Technology, Management*, pp. 1011-1016, Nov 2015.
[14] M. Last, S. Eyal, and A. Kandel, "Effective Black-Box Testing with Genetic Algorithms," *Department of Computer Science and Engineering*, Ben-Gurion University of the Negev, Beer-Sheva, Israel,2005
[15] Ranjita Swain, Vikas Panthi, "Automatic Test case Generation from UML State Chart Diagram", *International Journal of Computer Applications (0975 – 8887)*, Vol. 42– No.7, pp.26-36, March 2012.
[16] V.MarySumalatha et al., "UML based Automated Test Case Generation technique using Activity-Sequence diagram", *The International Journal of Computer Science & Applications (TIJCSA)*, Vol. 1, No. 9, pp. 58-71, November 2012.
[17] M. Philips, N. Pawar, N. Joshi, S.Khandebharad, S. Deshmukh, K.Tambe, "Automated Test Case Generation Using Multiple Modelling Techniques", *International Journal of Science and Research (IJSR)*, Vol. 3 Issue 3, pp. 722-725, March 2014.
[18] P.R. Srivastavaand Tai-hoon Kim, "Application of Genetic Algorithm in Software Testing", *International Journal of Software Engineering and Its Applications*, Vol. 3, No.4, pp. 87-96, October 2009.
[19] Meenakshi Vanmali, Mark Last, Abraham Kandel, "Using a Neural Network in the Software Testing Process", *International Journal of Intelligent Systems*, pp. 45-62, January 2002.
[20] Gallant, "S.I. Neural Network Learning and Expert Systems", *Massachusetts Institute of Technology*, USA, 1994.