Hybridization of Ant Colony Optimization and Decision Tree for Test Paths Generation and Optimization

Manisha Verma¹, Priya Arora²

¹M.Tech Scholar, Computer Science & Engineering Department
²Assistant Professor, Computer Science & Engineering Department
¹,²Baddi University of Emerging Sciences & Technology
Makhnumajra, Distt. Solan, Himachal Pradesh, India

Abstract — Test case generation is among the most labor-intensive tasks in software testing and also one that has a strong impact on the effectiveness and efficiency of software testing. For these reasons, it has also been one of the most active topics in the research on software testing for several decades, resulting in many different approaches and tools. This paper presents an approach for test path generation and prioritization in model based testing. In this research we proposed an algorithm which will be the combination of Ant colony optimization and Decision tree algorithm, for test paths generation and prioritization from UML Activity diagram. Prioritization is based on the test paths weights.

Keywords — Model based testing, UML, Activity Diagram, Prioritization.

I. INTRODUCTION

Software testing involves the execution of a software component or system component to evaluate one or more properties of interest. In general, these properties indicate the extent to which the component or system under test. Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. Test techniques include the process of executing a program or application with the intent of finding software bugs (errors or other defects).

A. Software Testing Approaches:

Software testing approaches are mainly divided into three types that is code based testing, specification based testing and model based testing[13]. Test cases are designed mainly from the source code of the program. The code can be generated after the analysis and design of the software. So it is difficult to test at prior stages. To avoid wasting of time consumption and cost utilized in testing in the code based system, it is desirable to generate the test cases at the design level so that reliability of the software will be increased to the optimum level. Model Based Testing is more efficient and effective than code based approach as it is the mixed approach of source code and specification requirements for testing the software. Models are the intermediate artefacts between source code and requirement specification. Models preserve the essential information from requirement specification and are base for the final implementation.

B. Test Case Generation:

Manual generation of test case is time consuming and laborious. Hence either automatic or semi-automatic generation of test case from design document is required. Test cases are generally generated from the source program. It is necessary to design the test case at the time of designing process. Generating tests during design also allows testing activities to be shifted to an earlier part of the development process, allowing for more effective planning of test cases. When the tests are generated, the test engineer will often find inconsistencies and ambiguities in the specifications and design, allowing the specifications and design to be improved before the program is written. Another advantage is that the test data is independent of any particular implementation. So, generation of test cases at early stages is an important supplement to testing process. Automating the process would result in tremendous improvement in testing process.

C. Software Testing Strategies:

Verity of software testing leads to standardization. Some major software testing process includes black box testing and white box testing. Basically software testing is a technique to validate and verify the program whether it meets the customer’s need or not. Black box testing involves looking at the specifications. This testing process does not require examining the code. Black box testing is mainly used for functional testing. Functional testing evaluates the correctness of the program by
Comparing the needs of users without any knowledge of how the software is implemented. In black box testing, testers test software through user interfaces or the application programming interfaces.

White box testing test all possible coverage paths i.e. all paths of the program must be tested through some criteria. White box testing takes into consider the program code, code structure and internal design flow. It qualifies the test suite by coverage criteria. Code coverage criteria is defined using segment coverage, branch coverage, node testing, condition coverage, basis path testing, data flow testing, path testing and loop testing. The test procedure attempts to execute every part of the source code using the test data. The more test cases are generated, the more coverage is gained.

Grey-box testing is a combination of black box-testing and white box-testing. As with black-box testing, grey box testing uses a specification for creating test cases. The specification used in grey-box testing does not specify only the requirements of a system, but it also describes the behaviour of the system. Grey-box testing is similar to white-box testing in this sense. The behavioural information embedded in a specification is also used for generating test cases. The selection of test cases is based on the implementation of the software entity. The main focus of such test cases is to cause the execution of specific spots in the software entity, such as specific statements, program branches or paths. The expected results are evaluated on a set of coverage criteria like path coverage, branch coverage, and data-flow coverage.

D. Coverage Criteria:
Test coverage criteria are used to decide which test inputs to be used as we cannot test with all inputs. It is a set of rules that decides appropriate elements to be covered to make test case design adequate. The test developers will easily find the faults by effectively using the coverage criteria. Test coverage analysis helps to determine the "thoroughness" of testing achieved. Major test coverage criteria used today are Graph coverage which includes structural coverage, Data flow coverage, Graph coverage for source code, Graph coverage for design elements, Graph coverage for specifications, Graph coverage for use cases and logic coverage. A coverage criterion is a set of rules that impose test requirements on a test set.

E. Unified Modelling Language (UML):
The Unified Modelling Language (UML) is a collection of languages for specifying, visualizing, constructing, and documenting the artefacts of software systems. Over some years, UML model-based testing has evolved to face challenges of large and complex industrial software systems. The use of UML specification for software development has led us to a new way of development. It also changed the testing process. UML has taken root in analyzing and designing large systems. Moreover, it is a de-facto standard for both software industry as well as academics. UML analysis and design are performed in many software projects. It is a standard modelling language for visually describing the structure and understanding the behaviours of system. Unified modelling language is used to model dynamic and also static behaviour of a software. Activity, sequence and state diagram are used to represent the dynamic behaviour of the software whereas class, component and deployment diagrams are used to represent the static behaviour of the software. Activity diagram depicts the activities of the different objects of the software, so the operations can be realized in the design stage itself.

F. Ant Colony Optimization:
It is an algorithm based on the behaviour of real ants. It is basically the behaviour of ants and the pheromone trail left on the path by them whenever they move toward the food source. The trails can be sensed by other ants and they will decide which path to choose or not. ACO is a probabilistic technique that is applied to generate solutions for combinatorial optimization problems. The artificial ants in the algorithm are used to find the path in the activity diagram. The main theme of ACO is that, in real world, Ants wander to find food and lay pheromone trails on their path. If other ant find this trail, then it would stop wandering and start following the trail in search of food. However, the trail of pheromone start to evaporate after sometime, so it loses its attraction strength. The more ant take time to travel from one end to another, the more the path loses its pheromone trail. So a shorter path have more possibility to have the stronger pheromone level. If no evaporation occur at all, then the path chosen by the first ant will be followed by other ants. Thus, if one ant choses a shortest path, other ants are more likely to follow it, and the usefulness of the path is spread to other ants. So all ants will follow a single pheromone trail left the first ant and keep its level strong. The knowledge of this theorem is used to solve traversal in graph.

G. Decision tree:
Decision tree learning uses a decision tree as a predictive model which maps observations about an item to conclusions about the item's target value. It is one of the predictive modelling approaches used in statistics, data mining and machine learning. Tree models where the target variable can take a finite set of values are called classification trees. In these tree structures, leaves represent class labels and branches represent conjunctions of features that lead to those class labels. Decision trees where the target variable can take continuous values (typically real numbers)
are called regression trees. In decision analysis, a decision tree can be used to visually and explicitly represent decisions and decision making.

II. RELATED STUDY

Doungsa-ard et al. [4] proposed a framework for generating test data from software specifications. Test data is generated using control flow graph with a deterministic algorithm. State machine diagram was considered. A tool and an approach for generating test data from the software specification is used. To find the best test data heuristic technique GA was applied. When the lengths of chromosomes vary the result will certainly vary as described in their approach.

Jena et al. [12], presented test case generation by using UML activity diagram. They applied the depth first search to generate test cases. They applied the genetic algorithm to optimize best test cases. They convert activity diagram into activity flow table to generate test paths.

Sharma et al. [10], focused a novel approach for generating test case from the design diagram. They transform sequence diagram into a graph called Sequence Diagram Graph (SDG). They traversed the SDG to generate test cases. They generate a test suite to cover interaction faults and scenario faults.

Shirole et al. [7], have presented an approach for automatic generation of feasible test paths using GA. The proposed architecture produces a set of test cases from UML state machine diagram. The authors transformed the state machine diagram to extended finite state machines and subsequently to Extended Control Flow Graph. They applied GA, to generate test cases satisfying the coverage specified (all-definition cover, all-due path cover). It was observed by the authors that the feasibility of path is affected by sequence of transitions.

Varikutti et al. [8], presented an integrated approach for generating test cases using Genetic Algorithm and dominance relation with fitness function. They investigated early generation of test cases using use cases and scenarios. They constructed a control flow graph based on that dominance tree. Test cases are generated by applying Genetic algorithm on dominance tree with concepts of crossover and mutation.

Korel et al. [13], performed a small evaluation test to verify efficiency of both simple code based and model-based test prioritization techniques. The target of this experiment was to evaluate these methods to check performance of early fault detection in the modified system. This analysis result set shown that model based test prioritization may improve the early fault detection as compared to the code based test prioritization because the execution of the model is very fast as compared to the execution of the actual system. Therefore, execution of the model for the whole test suite is cheaper as compared with code based test case prioritization.

After performing some experiments again Korel et al.[14], prioritized the test cases by using several model-based test prioritization heuristics. It had few issues that selective model based prioritization considers only the number of executions of marked transitions which does not have a significant influence on the improvement of the early fault detection.

Prasanna et al. [5] has proposed a model based approach for generation of automated test cases. The test cases are derived by analyzing the dynamic behavior of the objects due to internal and external stimuli. A general tree is created from the object diagram. Genetic Algorithm is applied on general tree to achieve tree structure. Tree crossover has been proposed to bring out all possible test cases. The scope of the paper has been limited to the object diagrams taken from the UML model of the system. No discussion has been mentioned for system level faults and also behavioral diagram which require every activity edge of an activity diagram to be covered.

Praveen et al. [19] proposed an approach for generation of test sequences using Ant colony optimization. It consists of path generation using the behaviour of ants. They presented a simple algorithm with the help of an ant colony optimization (ACO) technique, for the optimal path identification by using the basic property and behaviour of the ants. The approach consists of certain collection of method and principles to find out all the useful or effective paths via ant ACO technique.

III. PROPOSED WORK

In this research we proposed an algorithm for test paths generation and prioritization. Proposed algorithm is combination of Ant Colony Optimization and Decision Tree algorithms. Test paths will be generate from UML Activity diagram. XMI file will be extract from UML Activity diagram. We will develop a test case generator using our proposed algorithm which can take the input as an XMI file and generate a tree from that XMI file using Decision Tree algorithm. Test paths are generated and prioritized using Ant Colony Optimization algorithm with some modifications. Prioritization is based on the lowest path weights. In the previous approach gives us the test paths according to the length of the path. Test path having higher length are given higher priority. But
the concept adopted earlier was, test path having maximum number of decision nodes and fork node are given higher priority. As the previous algorithm is a path based testing it does not depend on number of decision nodes and fork nodes. Also that does not work for the diagram containing fork nodes. So that algorithm need to be modified on the basis of weightage of nodes. We take the node weightage values as [6].

Steps of proposed algorithm:

1. First take an UML Activity diagram.

2. XML file is extracted from the UML Activity diagram and taken as an input to the Test case generator.

3. After that we start to implement the steps of Decision Tree to generate the tree from XMI file given below:
   a. First we find out the parent id and set as a root we consider the one attribute as a root only to which is topmost.

b. Then we iterate whole attributes of XMI files till of all tags are not traversed.

c. Then we find out all the attributes and their ids from the XMI representation after that we will find out the relationship of that id with other attribute id to which they are set as an either source or target and stored in linked lists.

d. After finding the source and target ids we will connect the target node ids to source node id.

4. Ant Colony Optimization algorithm is applied to generate and optimize the test paths:
   a. length = decision tree.length;

b. Traverse from the first node(i)

c. Update visited node status for current node from 0 to 1.

d. If the current node is Decision node

e. If ‘i’ is the decision node and ‘j’ and ‘k’ are nodes connected to it.
   Find probability of path based on generate of child stack. Every stack name is consider as parent node name whereas all attributes inside that stack is called an child which belongs to parent.

f. Update Pheromone stack of the selected path values get extracted from stack.

g. After that we will empty that stack.

5. Set count = count++

6. If count <= length
   set visit status(v) of all node to 0, sum = 0, set current node = “start” and go to step 2.

7. End of algorithm.

IV. IMPLEMENTATION

Here we implement proposed algorithm with java program. We take XMI representation of the activity diagram as an input to the program and generate all possible test paths. After that we optimized the test paths, optimization of these test paths is based on their weight.

Step.1 Draw Activity Diagram
Activity Diagram is designed using the My Eclipse software and this software has the option of exporting the UML diagram to XMI file.

Step.2 Extract XMI Representation
XMI representation of this activity diagram is extracted using MyEclipse tool and stored in a file. From there, nodes and edges are extracted and stored in an array or linked list. Using this information we can draw tree out of it using Decision Tree algorithm. Using Ant Colony Optimization algorithm we can generate all possible test paths from it and optimized these.

Step.3 Apply Decision Tree to extract the detailed information of the .xmi file
Using My Eclipse, nodes and edges from the corresponding XMI representation is extracted and saved in linked lists. Source and target of each edges are extracted and saved in linked list. Two linked lists are made, Node_name and Node_id. Another two linked list are made. Source of an edge and destination of an edge. Initially source and destination of each edges contained Node_id of nodes as from the XMI representation. This is a table of node name and node id extracted from the XMI representation code. Each node has a unique node_id, so they can be distinguished from each other by their node_id. Similarly, each edge has a unique node_id for source and target section.

Generation and optimization of test paths are done using proposed and existing algorithms. UML activity diagram is taken into example and generated test paths are optimized according to their weight calculated using algorithms. Test paths having lower weight is the most suitable one for testing. We take some modules for consideration, here we discuss activity diagram of ATM.
Step.4 Weights Assignment
These are the activity nodes which are used to draw UML Activity diagrams of the projects[12]. In previous research, test paths having higher weight is the most suitable one for testing and ACO algorithm does not consider fork nodes. But in our proposed approach we consider the fork nodes, as well as other nodes and assign weight to the edges. If the control flow goes to decision node we assign weight to it is four. If the control flow goes to Join node and/or Fork node we assign weight to it is two. If the control flow goes to Merge node we assign the weight to it is three.

Step.5 Apply Ant Colony Optimization algorithm to generate optimized test paths
Optimized test paths are generated using Ant Colony Optimization algorithm. Optimization of test paths are done using proposed and existing algorithms. UML activity diagram is taken into example and generated test paths are optimized according to their weight calculated using algorithms. Priority is given to lowest weight first.

Modified Metric for APFD
Test paths optimization achieves many possible goals. Out of these goals, we restrict our attention to devise techniques that will improve the efficiency by early detection of faults and provide confidence in reliability with good testing coverage. To formally illustrate how rapidly a prioritized test suite detects faults, Rothermel et al. [21] introduced a metric called Average Percentage of Faults Detected (APFD) to measure the weighted average of the percentage of faults detected during the execution of the test suite. The APFD values range from 0 to 100; the higher the number, the faster (better) is the fault detection rates.

The APFD for test suite \( T \) is given by the equation:
\[
APFD = 1 - \frac{1}{\frac{\sum_{i=1}^{m} \frac{T_{Fi}}{n}}{2n}}
\]
But there is a disadvantage; prior knowledge of faults are needed. As the formula for APFD shows that calculating APFD is only possible when prior knowledge of faults is available. Actual knowledge of faults is only after the coding phase possible, so we modified this metric.
\[
APFD = 1 - \frac{\sum_{i=1}^{m} \frac{T_{Fi}}{n}}{2n} + \frac{1}{2n}
\]
Here \( n \) - Total number of test cases and \( m \) - total number of faults covered by test suite.

Generated test paths:
path1: InitialNode1>>>b=balance>>>Account Withdraw>>>check withdraw limit >>> Insufficient Balance >>> return true >>> CallBehaviorAction1 >>> ActivityFinalNode1 >>>
path2: InitialNode1>>>b=balance>>>Account Withdraw >>> check withdraw limit >>> Cross Limit>>> return false >>>CallBehaviorAction1>>>ActivityFinalNode1>>>
path3: InitialNode1>>>b=balance>>>Account Withdraw >>> sufficient balance >>>check withdraw limit >>> Within Limit>>>return true >>> CallBehaviorAction1 >>> ActivityFinalNode1 >>>
path4: InitialNode1>>>b=balance>>>Account Withdraw >>> Insufficient Balance >>>check withdraw limit >>> Cross limit>>>return false >>> CallBehaviorAction1 >>> ActivityFinalNode1 >>>
path5: InitialNode1>>>b=balance>>>Account Withdraw>>>Insufficient Balance >>> return false >>> CallBehaviorAction1 >>> ActivityFinalNode1 >>>

Optimized Test paths:
path5: InitialNode1>>>b=balance>>>Account Withdraw >>> Insufficient Balance >>> return false >>> CallBehaviorAction1 >>> ActivityFinalNode1 >>>=weight=7
path4: InitialNode1>>>b=balance>>>Account Withdraw >>> Insufficient Balance >>> check withdraw limit >>> Cross limit >>> return false >>>

V. RESULTS ANALYSIS
We take some projects to generate and optimize the test cases. Here we consider the example of ATM system. We make the UML Activity diagram in My Eclipse tool and extract the XMI file from the Activity diagram. We take XMI file as an input to the Test Case Generator, a Tree is generated from the XMI file. All the possible test paths are generated, after that test paths are prioritized according to the test path weight.

\[
APFD = 1 - \frac{14}{5^2} + \frac{1}{2 \times 5} = 0.54
\]
For prioritized test paths:
\[
APFD = 1 - \frac{7}{5^2} + \frac{1}{2 \times 5} = 0.82
\]
VI. COMPARISON WORK

A. Comparison of Fault detection in Prioritized and Non-Prioritized Test Paths

We compare APFD value of prioritized test paths with the non prioritized test paths of some projects. APFD value of prioritized test paths are greater than the APFD value of non prioritized test paths.

**Fig.1 Comparison of fault detection**

B. Comparison of Path Coverage

We compare path coverage percentage of some modules of proposed algorithm with Genetic algorithm. From the screenshots it is clear proposed algorithm covers maximum path coverage as compared to Genetic algorithm.

**Fig.2 Comparison of Path Coverage**

C. Comparison of Execution Time

We compare execution time in seconds of some modules of proposed algorithm (PA) with GA, ABC and ACO algorithms [20]. From the figure.3 it is clear execution time is reduced by applying proposed algorithm as compared to other algorithms.

**Fig.3 Comparison of Execution Time**

D. Comparison of Number of Iterations

We compare number of iterations of some modules of proposed algorithm with GA, ABC and ACO algorithms [20]. From the figure.4, performance of proposed algorithm is superior compared to other algorithms.

In this research we extract XMI representation from the Activity diagram using MyEclipse tool. We have optimized the test paths. To measure the effectiveness of our prioritized test paths, we introduce a metric, Average Percentage of Faults Detected (APFD), which measures the weighted average of the percentage of faults detected. APFD values range from 0 to 100; higher numbers imply faster or better fault detection rates. We calculate APFD value and execution time of the test paths to generation. APFD value of proposed approach is 82% of ATM example.
In this research work we proposed an algorithm which is used to generate and prioritised test paths. Optimization is based on the weights of the test paths. We have developed a Test Paths Generator with the help of this proposed algorithm. From the implementation of proposed method it is seen that Test Paths Generator is an efficient tool for test paths generation and optimization. It can generate the test paths and then optimizes the test paths based on lowest weight come on top. This tool help for the developer to do white box testing for their module. This approach of test paths generation and optimization uses fault detection and execution time allowing tester to prioritize the test paths. We have modified the existing metric for calculation of APFD and as we can see screen shots and comparison graph, it is very clear that proposed algorithm gives better results and more effective than previous techniques.

In future more work can be done on other applications of Ant colony optimization. Moreover, further modifications can be done in existing techniques so as to get better results. The scope of future work will also include merging other UML diagrams related to use cases for prioritization. Different UML diagrams will be taken as input to the program and its test paths will be generated. Different prioritization methods will be implemented and reviewed.

**ACKNOWLEDGMENT**

I would like to thank Ms. Priya Arora Assistant professor, Baddi University of Emerging Sciences and Technology for her valuable suggestion and support. Also thank my parents for supporting throughout my career.

**REFERENCES**


