iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 7 **Issue:** VI **Month of publication:** June 2019

DOI: http://doi.org/10.22214/ijraset.2019.6240

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.177
Volume 7 Issue VI, June 2019- Available at www.ijraset.com

Model Driven Testing-Functional Test Case Generation with Redundancy Check and Model Paradigm Approach

Runal G.[1], Prof. Pramod Jadhav[2]
[1, 2]*Department of Information Technology, Bharati Vidyapeeth Deemed University*

Abstract: ***Software testing is that the primary activity to provide reliable software system. Irresponsibleness of a software system is incredibly abundant passionate about the means of testing performed. Software system testing, which is sometimes last activity of the software system development cycle is performed below the pressure. Quality and irresponsibleness of software system area unit abundant passionate about take a look at ways that area unit dead by take a look at cases. Generation of optimized take a look at ways may be a difficult a part of the software system testing method. During this paper, a very important effort is formed to propose a brand new technique to get the optimized take a look at ways from UML sequence diagram. Results indicated that optimized ways from sequence diagram haven't any redundancy and made the higher results. The proposed work focuses on reducing test cases by detecting the lazy class code smells based on the cohesion and dependency of the code and applying the inline class refactoring practices before test case generation thereby significantly avoiding redundant test cases from being generated. Objective: Test cases tend to be large in number as redundant test cases are generated due to the presence of code smells, hence the need to reduce these smells. Refactoring is the process of altering an application's source code without changing its external behaviour. The purpose of code refactoring is to improve some of the non-functional properties of the code, such as readability, complexity, maintainability and extensibility. Applications/Improvements: From the results, refactoring is an effective technique that can reduce redundant test cases. While the focus is on test case reduction, it also improves the quality of generated test cases in terms of its branch coverage.***

Keywords: UML, Sequence Diagram, depth first search algorithm, software testing, test cases generator, Refactoring, Test case redundancy.

I. INTRODUCTION

Software testing is a crucial method that often won't to validate the standard of the software system. The proper testing will increase software package quality. With the rising demand for reliable software system, software system testing will add up to five hundredth of the whole software system value. Software system testing has not solely evolved for look errors or bugs within the software system however it becomes a discipline for evaluating the standard software system [1]. In line with IEEE testing is, "The method of elbow grease or evaluating system or system elements by manual or machine-driven suggests that to verify that it satisfies fixed requirements" [2]. It may be performed manually or mechanically. Machine-driven software system testing is found to be higher than manual testing. Software system testing method wants additional effort with an individual's interface. During this analysis paper author principally generating prioritization and optimization based mostly take a look at method from UML sequence diagram victimization Firefly algorithmic rule. Software system testing usually used 2 ways that area unit recording machine testing and white box testing. White box testing is understood as structural testing) is to check consistently the internals of the actual program module, recording machine testing focuses solely the output of the software system testing is understood as useful testing that victimization useful criteria [3, 4]. Model-based checking (MBT) [5] consists in employing a model to get check cases and work out the test finding, in terms of expected behaviour of the system underneath check (SUT). Models are designed supported the informal needs of the system and exploited by model coverage criteria to work out check cases. Additionally, the models create it doable to work out the check oracle, particularly the expected results of the check. When a concretization step, the abstract checks will then be dead on the SUT and also the test finding is often mechanically assigned. MBT is so a convenient manner to Automate check generation and, to some extent, checks execution. numerous approaches for MBT exist [6], supported totally different formalisms victimization, as an example, automata (mealy machines, IOLTS, IOSTS), or pre-post conditions notations (B, VDM, JML, UML/OCL), etc. related

to them, check choice criteria create it doable to get check cases that guarantee a given level of assurance that the system has been sufficiently exercised. Code refactoring is the process of restructuring existing computer code—changing the factoring—without changing its external behaviour. Refactoring is intended to improve non-functional attributes of the software. Advantages include improved code readability and reduced complexity; these can improve source-code maintain ability and create a more expressive internal architecture or object model to improve extensibility. Typically, refactoring applies a series of standardized basic micro-refactoring, each of which is (usually) a tiny change in a computer program's source code that either preserves the behaviour of the software, or at least does not modify its conformance to functional requirements. Many development environments provide automated support for performing the mechanical aspects of these basic refactoring. If done extremely well, code refactoring may help software developers discover and fix hidden or dormant bugs or vulnerabilities in the system by simplifying the underlying logic and eliminating unnecessary levels of complexity. If done poorly it may fail the requirement that external functionality not be changed, introduce new bugs, or both. By continuously improving the design of code, we make it easier and easier to work with. This is in sharp contrast to what typically happens: little refactoring and a great deal of attention paid to expediently adding new features. If you get into the hygienic habit of refactoring continuously, you'll find that it is easier to extend and maintain code

The proposed test model and test paths generation consist of the following steps:

1) *Activity Diagram Translator:* For generating the activity diagram from WS-BPEL the translator is used. There are two methods which will be used for processes of translating the activity diagram that are XML Metadata Interchange (XMI) and Extensible Style sheet Language Transformations (XSLT). XMI is typical approach used for interchanging the UML models like activity diagrams. And another one is the XSLT is a language used for transforming the one XML file o another XML file. We are using XSLT to translate WS-BPEL into XMI and this is our first step. The second step is retrieving of the activity diagram from those XMI file.
2) *Test Paths Generator:* From the test model, we will represent various types of coverage criteria which are based on all available scenarios which will cover each activity of the services. After representing the coverage criteria, the test path will be
3) *Test case Redundancy Removal:* Refactoring is an effective technique that can reduce redundant test cases. While the focus is on test case reduction, it also improves the quality of generating test cases in terms of its branch coverage.

II. RELATED WORK

In this project, Authors [1] Aritra Bandyopadhyay, Sudipto Ghosh, Vikas Panthi, Durga Prasad Mohapatra published a unique testing approach that mixes data from UML sequence models and state machine models. Current approaches that bank entirely on sequence models don't contemplate the results of the message path underneath check on the states of the taking part objects. We have a tendency to extend their Variable Assignment Graph (VAG) based mostly approach to incorporate data from state machine models. The extended VAG (EVAG) produces multiple execution methods representing the results of the messages on the states of their target objects. We are going to additionally compare the price and effectiveness of our approach with state machine based mostly approaches.

Author proposes a method for check Sequence Generation exploitation UML Model Sequence Diagram.UML models provides a ton of knowledge that ought to not be unheeded in testing. During this paper main options extract from Sequence Diagram at the moment we are able to write the Java ASCII text file for that options consistent with Model Junit Library. Model JUnit is associate degree extended library of JUnit Library. By exploitation that ASCII text file we are able to generate action Automatic and check Coverage. This paper describes a scientific action Generation Technique performed on model primarily based testing (MBT) approaches By exploitation Sequence Diagram.

In this project, Authors [2] Md Azaharuddin Ali et.al, S. Shanmuga Priya et.al, proposed a method for generating the check cases mechanically. This projected methodology decreases time and will increase the reliableness of the software package testing part. The main criterion of software package checking is to supply test cases. This methodology contains reworking the state diagram (UML) into finite state machine (DFA / NDFA) wherever each node represents state and transactions area unit diagrammatical by arrow connecting the states. The projected methodology achieves enough check coverage while not increasing the quantity of check cases. It additionally attains a lot of vital coverage like transition coverage, transition try coverage, and provides state coverage.

They proposes a piece that represents a model based mostly take a look ating techniques from that the test methods area unit automatic generated and achieved earlier or through the method of development and then, once the code of application is offered, the take a look at cases may well be dead that aids in fixing the errors at initial part. Unified Modelling Language (UML) Sequence Diagram is taken into account for planning and a case study of Medical Consultation System is taken for

the projected work. Testing are going to be disbursed on collected demand, planning and committal to writing. Though, if testing is followed within the initial innovate SDLC several errors would be removed and can be prohibited while not commercial enterprise to succeeding part. Therefore, testing should not be isolated to one part alone in SDLC.

In this project, Authors [3] Ching-Seh Wu and Chi-Hsin Huang, They propose a technique of Model-Based Testing (MBT) to reinforce testing of interactions among the online services. The technique combines Extended Finite State Machine (EFSM) and UML sequence diagram to get a take a look at model, referred to as EFSM-SeTM. they need additionally outlined numerous coverage criteria to get valid take a look at methods from EFSM-SeTM model for an improved take a look at coverage of all attainable situations. This approach focuses on making a take a look at model from the UML sequence diagram and EFSM. Derived the take a look at ways from the take a look at model for composite net service testing. To develop the take a look at model, EFSM-SeTM is generated from sequence diagram and EFSMs for composite net service testing. The EFSM-SeTM represents states of objects and interaction info within the take a look at model.

A. Refactoring Lowers The Price Of Enhancements

When a package is winning, there's continuously a desire to stay enhancing it, to repair issues and add new options. After all, it's known as package for a reason! However the character of a code-base makes an enormous distinction on however straightforward it's to form these changes. Usually enhancements are applied on prime of every different during a manner that creates it more and tougher to form changes. Over time new work slows to a crawl. To combat this variation, it is important to refactor code in order that additional enhancements do not cause needless quality.

B. Refactoring Is A Part Of Day-To-Day Programming

Refactoring is not a special task that might show up in a very project arrange. Done well, it is a regular a part of programming activity. after I got to add a brand new feature to a codebase, I inspect the present code and take into account whether or not it's structured in such some way to create the new modification simple. If it is not, then I refactor the present code to create this new addition simple. By refactoring initial during this manner, I typically notice it's quicker than if I hadn't distributed the refactoring initial.

In this project, Authors [4]Ke Z, Bo J, Chan WK , Papadakis M, Malevris N. and Zhang C, Groce A, Alipour MA, editors proposes the two well-known approaches presently being looked into square measure test suit prioritization and test suit step-down. In these 2 approaches, the check cases square measure generated which incorporates the redundant check cases. The two approaches square measure well established within the space of regression testing. There's less attention given to redundancy shunning in an exceedingly recently developed application. Hence, there's a requirement for a way that averts redundant check cases from being generated in an exceedingly new system. Automatic unit tests ought to be came upon before refactoring to confirm routines still behave obviously. Unit tests will bring stability to even giant refactors once performed with one atomic commit. A typical strategy to permit safe and atomic refactors spanning multiple comes is to store all comes in an exceedingly single repository, referred to as monorepo. With unit testing in situ, refactoring is then associate degree repetitious cycle of creating atiny low program transformation, testing it to confirm correctness, and creating another little transformation. If at any purpose a check fails, the last hard cash is undone and continual in an exceedingly totally different manner. Through several little steps the program moves from wherever it absolutely was to wherever you wish it to be. For this terribly repetitious method to be sensible, the tests should run terribly quickly, or the technologist would get to pay an oversized fraction of their time watching for the tests to end. Proponents of maximum programming associate degreed alternative agile code development describe this activity as an integral a part of the code development cycle.

In this project, Authors [5]. Lashari SA, Ibrahim R, Senan N. , Ahmed M, Ibrahim R, Ibrahim N, Fowler M. proposes Refactoring the ASCII text file will eliminate code smells that would generate redundant take a look at cases. Refactoring is a very important methodology of eliminating the weaknesses of a package by applying modifications to the ASCII text file while not dynamical the outward behaviour of the system. A lazy category is refactored victimization the inline methodology of refactoring. This study still, applies the inline category refactoring technique to refactor the mechanical man ASCII text file to eliminate lazy category code smells that would probably cause generating redundant take a look at cases. This is often achieved by planning the detection and refactoring rules for lazy category smell. The code smell with its detection and refactoring technique is formally specified. Manually performed refactoring is time overwhelming and takes numerous efforts. Automating the method is

indispensable. Hence, a tool named Direct Attention Thinking Tools (DATT) was developed to implement the detection and refactoring rules. DATT is evaluated victimization allot application by generating branch coverage and cyclamate quality for the first and refactored ASCII text file victimization herbaceous plant for mechanical man and also the results obtained were compared. Take a look at cases area unit generated before and once refactoring to validate DATT. Therefore, this paper is organized as follows: ensuing section discusses connected work on action minimization and refactoring, framework for this study is then presented. Implementation of the framework is mentioned next, followed by the results and analysis. Finally, the conclusion of the study is mentioned supported the findings.

III. SYSTEM DESCRIPTION

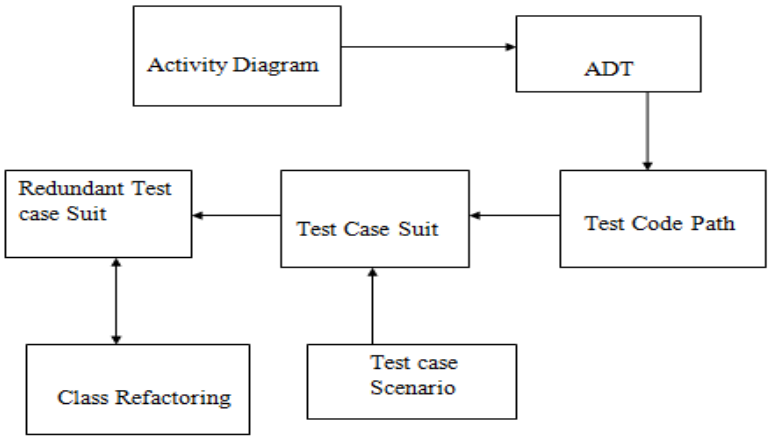


Fig 1: System architecture

A. *Methodology*

To achieve all our objectives we are going to use following methodology:

1) Input Activity Diagram:
2) ADT Generation
a) XML Code generation
3) Test Suit Generation
4) Redundant test case suit

B. *Algorithm*

1) Procedure Refactoring of Lazy Class
2) Identify the movable class
3) If method in movable class not null
4) Move method content to a host class of shortest distance If attribute not null, Push Field to the host class
5) Remove all methods from movable class until method = null Remove all attributes from movable class until attributes = null Update method in the host class
6) Update field in the host class
7) Redirect reference calls from movable class to the host class Delete the lazy class
8) end procedure

The system takes two inputs as activity diagram and generate test coverage path. Compare both the results and select most appropriate path for test case generation. This considers the various coverage criteria to generate test path that covers better test coverage of all scenario. The proposed work takes an activity diagram as input. Each activity diagram is well-known about making its ADT. This means to hold all needed details which are able to modify the model to look at capabilities and functionalities of all activity diagrams. The ADT will then produce the ADG automatically. ADG are access by utilizing DFS for all necessary obtainable test strategies. To design the final test case path utilized generated from ADT that gives the detail information. Each activity diagram ought to be undergo every of the four modules for creating high collection of very economical test cases. The outlines of each module are given as follows:

a) *Generation of ADT:* Activity Dependency Table defines the loops, synchronization and strategies presenting the actions of the task are created technically utilizing every activity diagram. This decides to indicate the activities can move into the totally different entities. This will be very supportive for system, integration and regression testing. Additionally, it holds the input with the specified value of output for each activity of the system. Activity Dependency Table shows each activity dependency on one another very clearly. Each activity has its special symbol for simply mentioning it among determinant dependencies additionally using it among varied involved units of the system. To reduce the searches of the created ADG, static activity that are permanent, distributed among one image completely instead of using several symbols together for each connected activity.

b) *Generation of ADG:* ADG are mechanically generated from the activity dependency table that is ADT. Names are given to every node by utilizing the symbols of the every task among the ADG. Here each node will display component or functionality within the activity diagram. As repetitive functionalities are specifies a consistent image among the ADT, only one node is made for them however what proportion times they're used among the activity diagram. It'll decreases the time needed for search operation within ADG. Edges present in activity dependency graph that is ADG, displays the transaction between current activities to another activity. The presence of a transaction between one node and another one are going to be set through testing the column of dependency in the ADT for the image of this node. Significantly, if it holds the earlier image of the node then a position from earlier node toward the present one is generated within the ADG. Alternatively we've a tended to return to the ADG, still searching node image that one declared among this dependency column of node and make a position from this column. This will present node continuously until all the rows among the ADT are achieved. Loop synchronization and Selection of particular loop are done utilizing the similarity of edges.

c) *Test cases Generation:* In the module 2 above we have generated the ADG i.e. Activity Dependency Graph. Now here we are going to apply Depth First Search strategy for obtaining all the test paths consider for testing. The test path is designed from steps presenting the consecutive nodes. These steps will form complete path which start from first node to the end node in the ADG. These nodes are separated through arrows. All final test cases are collected from the details gathered from the activity dependency table. Every node is presented with input and desired result within the test case. Similarly, all the test cases are come with their inputs and final predicted outputs.

d) *Redundancy Controller:* One way to provide multiple "controllers" is to implement a "mirroring backup" so that another system also collects all data. If the system is controlling in real-time, having more than one *CPU* in the system is ideal, creating a *bump less* control system. This is a common requirement for redundancy. For example, if the CPU fails in a system controlling a furnace, waiting for a replacement might destroy the product. Continual process and CPU data exchange creates redundancy to ensure system reliability. Taking advantage Microsoft Windows NT robustness and including features for redundancy in the software are key challenges for PC-based control in factory-floor automation. To provide redundancy, PC-based control software acknowledges more than one CPU on the system, whether it is distributed or centralized control.

e) *Class Refactoring:* Much of refactoring is devoted to correctly composing methods. In most cases, excessively long methods are the root of all evil. The vagaries of code inside these methods conceal the execution logic and make the method extremely hard to understand and even harder to change. The refactoring techniques in this group streamline methods, remove code duplication, and pave the way for future improvements. Even if you have distributed functionality among different classes in a less-than-perfect way, there is still hope. These refactoring techniques show how to safely move functionality between classes, create new classes, and hide implementation details from public access. These refactoring techniques help with data handling, replacing primitives with rich class functionality. Another important result is untangling of class associations, which makes classes more portable and reusable.

IV. RESULT ANALYSIS

As shown in the Table 1 shows the evaluation of lazy class refactoring based on the cyclamate complexity and branch coverage in comparison with the original allocate source code.

A test case minimization approach is incomplete if the quality of the test case is not ensured. One of the ways to do this is calculating the branch coverage of the before and after refactoring code and comparing the result. Hence, the branch coverage is calculated for each of the affected classes in the source code using the formula in

A. Branch Coverage

Branch Coverage = $x = \frac{\text{No. of lines of code covered}}{\text{Total no. of Lines of code (20)}} * 100$

Table 1: Before Class Refactoring Branch Coverage

Sr. No	Class Name	No. of Methods	Branch Coverage
1	Account	3	45%
2	Saving Account	1	15%
3	Current Account	1	15%

After Class Refactoring Branch Coverage

Sr. No	Class Name	No. of Methods	Branch Coverage
1	Account	3	45%
2	Lazy Class (Merged both class into host class)	1	15%

Table 2 shows the test case generated from the original source code prior to refactoring while Table 3 shows the test cases generated after refactoring.

Table 2: Test Cases Generated Before Class Refactoring TC generated

Sr. No.	Test Case	Scenario	Input	Expected Output	Actual Output
1	Saving Account - TC1	Calculate Interest	Interest Rate	Get result with float value	Get result with float value
2	Current Account - TC1	Calculate Interest	Interest Rate	Get result with float value	Get result with float value

Table 3 : After Class Refactoring TC generated

Sr. No.	Test Case	Scenario	Input	Expected Output	Actual Output
1	Saving Account and Current Account - TC1*	Calculate Interest	Interest Rate	Get result with float value	Get result with float value

V. CONCLUSIONS

Here, we are presented several criteria of covering test path generation and algorithm for generating the test path automatically. In this paper, we have shown how to develop test model and how to define coverage criteria. Also focus on to develop test path generating algorithm for Activity diagram given path. The outcome shows that refactoring the source code prior to test case generation has reduced the test cases by 33.3% and increased the branch coverage up to 9.2%. Hence, the approach is a prospective test case reduction technique. This implies that the cost and effort in testing can be reduced by eliminating the code smells prior to test case generation.

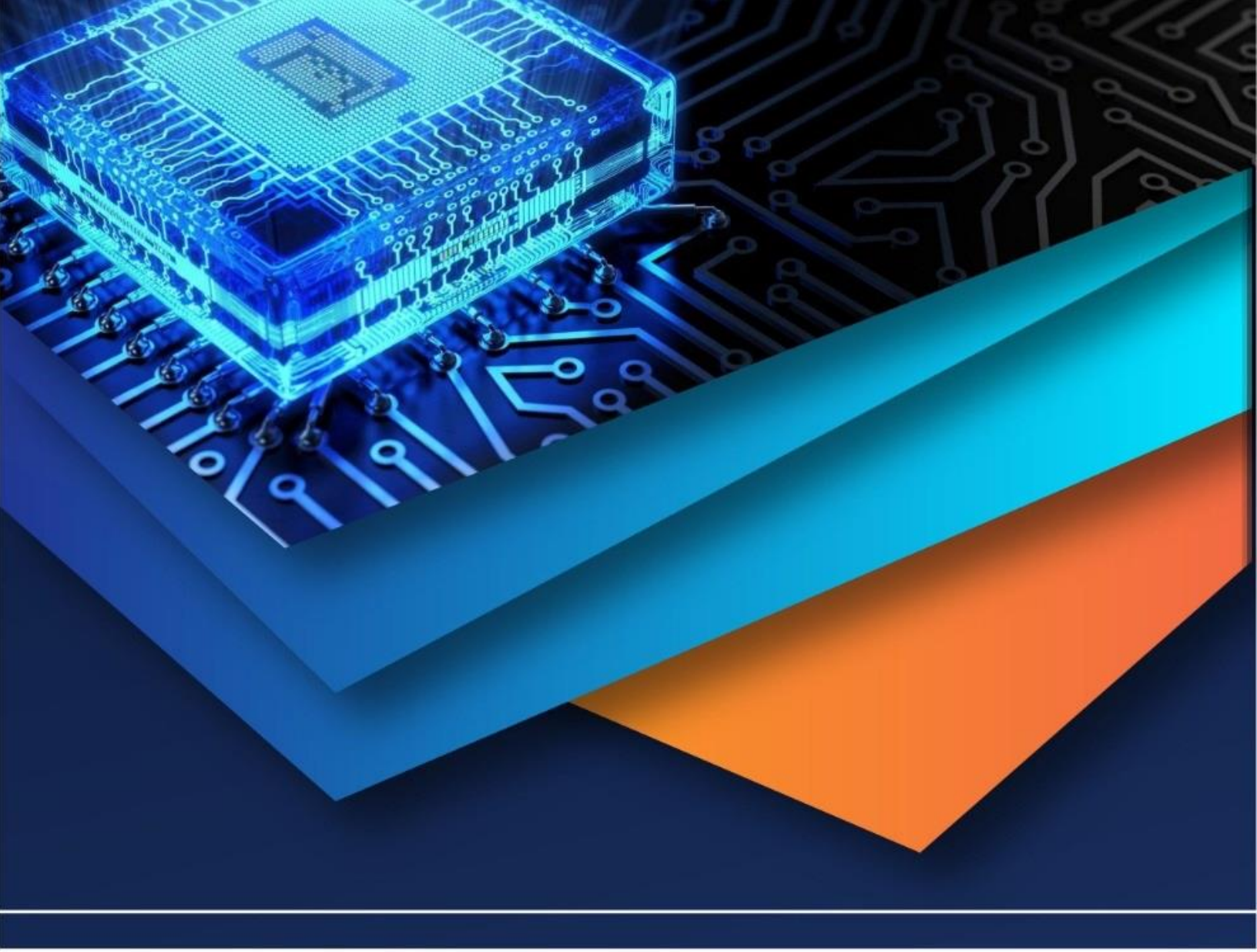
VI. ACKNOWLEDGMENT

To prepare proposed methodology paper on “Model Driven Testing- Functional Test Case Generation with Redundancy Check and Model Paradigm Approach” has been prepared by Miss. Runal G. I would like to thank my faculty as well as my whole department, parents, friends for their support.

REFERENCES

[1] R. A. Khan and R.K Choudhary, “Software Testing Process: A Perspective Framework” ACM SIGSOFT Software Engineering Notes” Volume 36, Number 4, pp 1-5, July 2011.
[2] Rajvir Singh, “Test Case Generation for Object-Oriented Systems: A Review” IEEE, Fourth International Conference on Communication Systems and Network Technologies, 2014
[3] R. S. Pressman, Software Engineering: A Practitioner’s Approach, 7th Edition, McGraw-Hill, 2010.
[4] Soma Sekhara Babu Lam et al. “Automated Generation of Independent Paths and Test Suite Optimization Using Artificial Bee Colony” Procedia Engineering, Elsevier pp. 191-200, 2012.
[5] Beizer, B.: Black-Box Testing: Techniques for Functional Testing of Software and Systems. Wiley, New York (1995)

[6] Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model based testing approaches. Softw.Test.Verif. Reliab. **22**(5), 297–312 (2012)
[7] Warmer, J., Kleppe, A.: The Object Constraint Language Second Edition: Getting Your Models Ready for MDA. Addison-Wesley, Reading (2003)
[8] Masson, P.-A., Potet, M.-L., Julliand, J., Tissot, R., Debois, G., Legeard, B., Chetali, B., Bouquet, F., Jaffuel, E., Van Aertrick, L., Andronick, J., Haddad, A.: An access control model based testing approach for smart card applications: results of the POSÉ project. JIAS **5**(1), 335–351 (2010)
[9] Tripathy A, Mitra A. Test case generation using activity diagram and sequence diagram. In: Proceedings of International Conference on Advances in Computing. Springer; 2013. p. 121–9.
[10] Lashari SA, Ibrahim R, Senan N. Fuzzy Soft Set based Classification for Mammogram Images. International Journal of Computer Information Systems and Industrial Management Applications. 2015; 7:66–73.
[11] Ahmed M, Ibrahim R, Ibrahim N. An Adaptation Model for Android Application Testing with Refactoring. Growth. 2015; 9(10):65–74.
[12] Fowler M. Refactoring: Improving the Design of Existing Code. 1997. Available from: http://www.martinfowler.com/ books/refactoring.html.
[13] Aritra Bandyopadhyay, Sudipto Ghosh, "Test Input Generation using UML Sequence and State Machines Models"
[14] Vikas Panthi, Durga Prasad Mohapatra, "Automatic Test Case Generation using Sequence Diagram", International Journal of Applied Information Systems (IJAIS) – ISSN : 2249-0868 Foundation of Computer Science FCS, New York, USA Volume 2– No.4, May 2012 – www.ijais.org
[15] Md Azaharuddin Ali et.al. "Test Case Generation using UML State Diagram and OCL Expression", International Journal of Computer Applications (0975 – 8887) Volume 95– No. 12, June 2014
[16] S. Shanmuga Priya et.al, " Test Path Generation Using UML Sequence Diagram", Volume 3, Issue 4, April 2013 ISSN: 2277 128X International Journal of Advanced Research in Computer Science and Software Engineering
[17] Ching-Seh Wu, Chi-Hsin Huang," The Web Services Composition Testing Based on Extended Finite State Machine and UML Model", 2013 Fifth International Conference on Service Science and Innovation
[18] Ke Z, Bo J, Chan WK. Prioritizing Test Cases for Regression Testing of Location-Based Services: Metrics, Techniques, and Case Study. IEEE Transactions on Services Computing.. 2014; 7(1):54–67.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY