e-ISSN: 2349-9745
p-ISSN: 2393-8161

Scientific Journal Impact Factor (SJIF): 1.711

International Journal of Modern Trends in Engineering and Research

www.ijmter.com

# Software Quality Analysis Using Mutation Testing Scheme

Ms. S. Sangeetha[1], Dr. M. Latha[2], Mr. R. Subramanian3

[1]*Research Scholar*

[2]*M.Sc., M.Phil., Ph.D, Associate Professor, Department of Computer Science.*

[1,2]Sri Sarada College for Women, Salem, Tamilnadu, India

[3]Erode Arts and Science College, Erode

**Abstract-**The software test coverage is used measure the safety measures. The safety critical analysis is carried out for the source code designed in Java language. Testing provides a primary means for assuring software in safety-critical systems. To demonstrate, particularly to a certification authority, that sufficient testing has been performed, it is necessary to achieve the test coverage levels recommended or mandated by safety standards and industry guidelines. Mutation testing provides an alternative or complementary method of measuring test sufficiency, but has not been widely adopted in the safety-critical industry. The system provides an empirical evaluation of the application of mutation testing to airborne software systems which have already satisfied the coverage requirements for certification.

The system mutation testing to safety-critical software developed using high-integrity subsets of C and Ada, identify the most effective mutant types and analyze the root causes of failures in test cases. Mutation testing could be effective where traditional structural coverage analysis and manual peer review have failed. They also show that several testing issues have origins beyond the test activity and this suggests improvements to the requirements definition and coding process. The system also examines the relationship between program characteristics and mutation survival and considers how program size can provide a means for targeting test areas most likely to have dormant faults. Industry feedback is also provided, particularly on how mutation testing can be integrated into a typical verification life cycle of airborne software. The system also covers the safety and criticality levels of Java source code.

## I. INTRODUCTION

Software Testing is the process used to help identify the correctness, completeness, security and quality of developed computer software. Testing is a process of technical investigation, performed on behalf of stakeholders, that is intended to reveal quality-related information about the product with respect to the context in which it is intended to operate. This includes, but is not limited to, the process of executing a program or application with the intent of finding errors. Quality is not an absolute; it is value to some person [1]. With that in mind, testing can never completely establish the correctness of arbitrary computer software; testing furnishes a 'criticism' or comparison that compares the state and behaviour of the product against a specification. An important point is that software testing should be distinguished from the separate discipline of Software Quality Assurance (SQA), which encompasses all business process areas, not just testing.

There are many approaches to software testing, but effective testing of complex products is essentially a process of investigation, not merely a matter of creating and following routine procedure. One definition of testing is "the process of questioning a product in order to evaluate it", where the "questions" are operations the tester attempts to execute with the product and the product answers with its behavior in reaction to the probing of the tester [2]. Although most of the intellectual processes of testing are nearly identical to that of review or inspection, the word testing is connoted to mean the dynamic analysis of the product—putting the product through its paces. Some of the common quality attributes include capability, reliability, efficiency, portability, maintainability, compatibility and usability. A good test is sometimes described as one which reveals an error; more recent thinking suggests that a good test is one which reveals information of interest to someone who matters within the project community.

Test Automation is software that automates any aspect of testing of an application system. It includes capabilities to generate test inputs and expected results, to run test suits without manual interventions and to evaluate pass/no pass. To be effective and repeatable testing must be automates. The appropriate extent of automated testing depends on the testing goals, budget, software process, kind of application under development and particulars of the development and target environment.

Safety-critical software implements functions which could, under certain conditions, lead to human injury, death, or harm to the environment. In the civil aerospace domain, DO-178B is the primary guidance for the approval of safetycritical airborne software [3]. The purpose of DO-178B is “to provide guidelines for the production of software for airborne systems and equipment that performs its intended function with a level of confidence in safety that complies with airworthiness requirements”. The DO-178B guidance distinguishes between different levels of assurance based on the safety criticality of the software, i.e., how the software can contribute to system hazards.

The safety criticality of software is determined at the system level during the system safety assessment process based on the failure conditions associated with software components [4]. These safety conditions are grouped into five categories: “Catastrophic,” “Hazardous/Severe-Major,” “Major,” “Minor,” and “No Effect.” The DO-178B guidance then defines five different assurance levels relating to the above categorization of failure conditions. Each level of software assurance is associated with a set of objectives, mostly related to the underlying life-cycle process. For example, to achieve software assurance level “C,” where faulty software behavior may contribute to a major failure condition, 57 objectives have to be satisfied. On the other hand, to achieve software level “A,” where faulty software behavior may contribute to a catastrophic failure condition, nine additional objectives have to be satisfied—some objectives achieved with independence.

## II. RELATED WORK

In their recent survey of mutation testing, Jia and Harman [6] illustrated the increasing trend in mutation research over the last three decades and indicated that from 2006 onward more papers have been published based on empirical evidence and practical application than on mutation theory. Their thorough survey also indicates a movement of mutation testing from basic research toward realization in industry. While mutation testing still represents an overhead, this cost needs to be offset against the long-term gains which can be achieved. As our study shows, mutation testing does not need to be exhaustive in order to reveal improvements which can be maintained through better development standards. Concerning the application of mutation testing in safety-critical software projects, apart from the case study on the use of mutation testing in a civil nuclear software program, identified in [6], we are not aware of any studies which have examined the application of mutation testing to safety-critical software systems developed using high-integrity subsets and approved against certification guidelines. As highlighted earlier, the development processes of the two software systems used in our study prohibit the use of many mutant types published in the mutation testing literature.

Our study discusses the link between cyclomatic complexity and mutant generation and survival. The link between code complexity and testing is also examined by the work of Watson and McCabe. They offer several reasons to associate code complexity with the test process. Watson and McCabe state that when measured against a structured testing methodology, the code complexity score will provide an explicit connection with the test. To this end, the code items with a higher complexity are more likely to contain code errors. A further assumption in this relationship which is applicable to mutation testing is that if the higher complexity code is more likely to harbor errors, then so are the test cases developed against them. In our study, we were more interested in the error rates within software test cases. Demonstrating a link between code complexity and mutant survival rates would be beneficial, but complexity is also a constraining factor as complex code will take longer to test. These are conflicting considerations in sampling code items for mutation. The need to reexecute test cases for complex code items competes with the need to keep to planned project timescales. McCabe and Watson also make the association between complexity and reliability. Where reliability can be assumed based on a combination of thorough testing and good understanding, developers can potentially ensure both by applying mutation testing and maintaining a low code complexity. The use of cyclomatic complexity as a measure of code complexity rather than size has been criticized, particularly by Shepperd and Jay et al. [8]. As such, any data concerning the relationship between mutant survival rates and cyclomatic complexity should be considered in the light of these criticisms.

To comply with the DO-178B objectives, the achievement of the required levels of structural coverage represents a significant proportion of the project life cycle. Statement coverage is already acknowledged to be the weakest form of testing and the data in this study question the value of applying only statement coverage. Statement coverage does not detect errors occurring in branch and decision statements. Chilenski states that test sets which satisfy all forms of MC/DC are capable of detecting certain types of errors. But, as with all forms of nonexhaustive testing, MC/DC is not guaranteed to detect all errors. Chilenski demonstrates that, when compared with the other coverage criteria specified in DO-178B, MC/DC has a higher probability of error detection per number of tests. Chilenski also performed a number of studies using mutation testing in order to determine the capability of different forms of MC/DC. His conclusions were that despite the problems commonly associated with mutation testing, it does provide a yardstick against which the effectiveness of MC/DC can be measured. The fact that these studies demonstrated variability in MC/DC effectiveness indicates mutation testing can provide coverage which MC/DC cannot. One item of particular relevance was that MC/DC test coverage performed very well in detecting logic mutants. The data collected during our study support Chilenski's findings; once test coverage includes MC/DC, the effectiveness of logic mutants diminishes. Our study complements some of the results of an empirical evaluation of mutation testing and coverage criteria by Andrews et al.

Their evaluation is based on applying mutation testing to a C program developed by the European Space Agency and compares four coverage criteria: Block, Decision, C-Use and P-Use. The findings reported by Andrews et al. provide evidence of how mutation testing can help in evaluating the cost effectiveness of test coverage criteria and how these criteria relate to test suite size and fault detection rates. Finally, in our study, the manual mutation testing of the Ada program, through handseeding faults based on the subjective judgment of a domain expert, identified certain advantages with respect to the reduction of equivalent mutants. Andrews et al. also address hand-seeded faults where they conclude that these faults appear to be different from automatically generated faults and harder to detect in comparison with real faults. The issue of using hand-seeded faults is also considered by Do and Rothermel, although with an emphasis on regression testing, where they conclude that hand-seeded faults might pose problems which can limit the validity of empirical studies using mutation testing. They acknowledge that further studies are still needed to support that observation. Although our study does not address this issue, it would be beneficial to replicate Andrews et al.'s experiment for safety-critical software and examine whether the same observations can be made.

## III. TEST QUALITY OF SAFETY-CRITICAL SOFTWARE

Testing is an essential activity in the verification and validation of safety-critical software. It provides a level of confidence in the end product based on the coverage of the requirements and code structures achieved by the test cases. It has been suggested that verification and validation require 60 to 70 percent of the total effort in a safety-critical software project. The tradeoff between cost and test sufficiency has been analyzed by Muessig and questions over test sufficiency have led to the reemergence of test methods which were historically deemed infeasible due to test environment limitations. Mutation testing is one such method. Mutation testing was originally proposed in the 1970s as a means to ensure robustness in testcase development [5]. By making syntactically correct substitutions within the software under test (SUT) and repeating the test execution phase against the modified code, an assessment could be made of test quality depending on whether the original test cases could detect the code modification. This method has not been widely adopted in the safety-critical industry despite its potential benefits—traditionally, it has been considered to be labor intensive and to require a high level of computing resources. Given changes in processing power and the emergence of tool support, as well as further research into streamlining the mutation process, mutation testing potentially now becomes a more feasible and attractive method for the safety-critical industry.

The aim of this study is to empirically evaluate mutation testing within a safety-critical environment using two realworld airborne software systems. These systems were developed to the required certification levels using high integrity subsets of the C and Ada languages and achieved statement coverage, decision coverage and modified condition/ decision coverage (MC/DC). Currently, there is no motivation for software engineers, particularly in the civil aerospace domain, to employ mutation testing. Most software safety guidelines do not require the application of mutation testing. Further, there is little empirical evidence on the effectiveness of mutation testing in improving the verification of safety-critical software. To this end, this study has the following objectives:

1. To define an effective subset of mutant types applicable to safety-critical software developed in SPARK Ada and MISRA C. Our objective is to examine how mutation testing can still add value through the application of this subset, despite the use of rigorous processes and software safety guidelines in which many of the mutant types published in the literature are already prohibited.
2. To identify and categorize the root causes of failures in test cases, based on the results of applying the above mutant types and to examine how the verification process can be re-enforced to prevent these issues.
3. To explore the relationship between program characteristics, mutant survival and errors in test cases. The aim is to identify which types of code structures should best be sampled in order to gain insight into the quality of testing.
4. To examine the relationship between mutation testing and peer reviews of test cases. The mutant subset is applied to peer-reviewed test cases in order to understand the value of mutation testing over manual review.

## IV. PROBLEM STATEMENT

The software safety analysis system is used to measure the criticality level for the software using the source code. The C and ADA language source codes are used in the analysis process. The Mutation testing technique is used in the testing process. The mutation testing scheme is not adapted for the Objected Oriented languages. The test coverage estimation is not optimized for the inheritance levels.

## V. SOFTWARE TESTING WITH MUTATION ANALYSIS

The safety critical analysis is carried out for the source code designed in Java language. Testing provides a primary means for assuring software in safety-critical systems. To demonstrate, particularly to a certification authority, that sufficient testing has been performed, it is necessary to achieve the test coverage levels recommended or mandated by safety standards and industry guidelines. Mutation testing provides an alternative or complementary method of measuring test sufficiency, but has not been widely adopted in the safety-critical industry. The system provides an empirical evaluation of the application of mutation testing to airborne software systems which have already satisfied the coverage requirements for certification.

The system mutation testing to safety-critical software developed using high-integrity subsets of C and Ada, identify the most effective mutant types and analyze the root causes of failures in test cases. Mutation testing could be effective where traditional structural coverage analysis and manual peer review have failed. They also show that several testing issues have origins beyond the test activity and this suggests improvements to the requirements definition and coding process. The system also examines the relationship between program characteristics and mutation survival and considers how program size can provide a means for targeting test areas most likely to have dormant faults. Industry feedback is also provided, particularly on how mutation testing can be integrated into a typical verification life cycle of airborne software. The system also covers the safety and criticality levels of Java source code.

To achieve compliance with the DO-178B testing objectives, test cases are often reviewed for adequacy through manual analysis. The problem with manual analysis is that the quality of the review is hard to measure. Mutation testing provides a repeatable process for measuring the effectiveness of test cases and identifying disparities in the test set. Mutation testing involves the substitution of simple code constructs and operands with syntactically legal constructs to simulate fault scenarios. The mutated program, i.e., the mutant, can then be reexecuted against the original test cases to determine if a test case which can kill the mutant exists. If the mutant is not killed by the test cases, this might indicate that these test cases are insufficient and should be enhanced. This process of reexecuting tests can continue until all of the generated mutants are captured by the test cases. Whereas structural coverage analysis, against coverage criteria such as statement coverage or MC/DC, considers the extent to which the code structure was exercised by the test cases, mutation testing considers the effectiveness of these test cases in identifying different categories of coding errors. For instance, a set of test cases might achieve the required coverage criterion yet can fail to detect certain types of coding errors. As such, the role of both structural coverage analysis and mutation testing can be seen to be complementary.

Mutation testing on the basis of two main assumptions about the types of errors which typically occur in software. The first is the “Competent Programmer Hypothesis,” which essentially states that programmers write programs that are reasonably close to the desired program. Second, the “Coupling Effect” postulates that all complex faults are the product of one, or more, simple fault occurring within the software. Both of these assumptions are essential in demonstrating that, by making simple substitutions in the SUT, mutation replicates the types of errors typically made by developers. Also, reinforcing test cases to kill mutants that include these types of errors prevents the occurrence of more complex faults occurring in the software. Normally, the mutation process consists of applying a mutation operator that might create a set of mutants. This is referred to as the First Order Mutation. Higher Order Mutation, on the other hand, involves the application of more than one mutant operator in each mutant [7].

Research into reducing the effort of mutation testing while ensuring its effectiveness is often classified into the following groups, identified by Offutt and Untch: “Do Fewer”, “Do Faster” and “Do Smarter”. The difference lies in where in the mutation process the reduction is achieved, i.e., whether in the mutant generation phase, in test-case execution, or in the analysis of the results.

## VI. MUTATION TESTING SCHEME FOR SOFTWARE QUALITY ANALYSIS

The software test coverage and safety analysis tool is developed as a graphical user interface based system. The system is designed to analyze the Java based source code only. This analysis is called as static analysis. The system implementation is carried out by using the Java language and Microsoft Access back end tool. The system is designed to analyze any third party and Sun Microsystems open source code. The knowledge model is updated for each test cycle. Future test cycle uses the knowledge model details.

Three major modules are used in the system implementation. They are the code analysis, knowledge model preparation and the test coverage and safety criticality analysis. The code analysis module is developed to extract details from the source code files. The knowledge base preparation module is developed to update the knowledge base in an organized manner. The test coverage and safety analysis is done on the source code details that are maintained in the database. The system uses the product path as the input.

### 6.1. Code Analysis

The code analysis is the initial module for the system. The product path is given as the input for the system. The Java source code files are identified first. Then each file content is fetched from the file. The noize filtering is performed after the code fetching process. The documentation comments and general comments from the source are removed. These comments are called as noise in the source code. The filtered code and class details are extracted. The package details and class details are updated into the database.

### 6.2. Knowledge Base Preparation

The knowledge base is a collection of source code elements for all Java programs in the product source code. The packages are the top level elements in the knowledge base. Each class details like method and attribute details are collected and updated into the database. The class relationship with other classes is also maintained separately. The interface for the class details are also collected and maintained in the database. The attribute details include the name of the attribute, type of the attribute, modifier details. The method details also include the method name, argument details and return type values.

### 6.3. Test Coverage and Safety Analysis

The test coverage and safety analysis module is designed to detect the hidden errors in the source code. Incorrect Initialization, inadvertent bindings, missing override, naked access, naughty children and spaghetti inheritance and Fat Interface bugs are detected by the system. The bugs are related to the inheritance concepts. Each type of bug is detected for the source code and listed in a separate form. The mutation testing model is used to evaluate the software errors. The data safety levels are analyzed with reference to the inheritance information. The standard bug model is used with mutation test scheme.

## VII. PERFORMANCE ANALYSIS

The Mutation Testing (MT) Scheme is used to measure the test coverage and safety levels for the source code. The Java source code is analyzed with Mutation Technique and Enhanced Mutation Technique. The test coverage accuracy is used measure the performance of the system. The accuracy rate analysis is shown in figure 7.1. and table 7.1. The analysis results show that the Enhanced Mutation Technique produces the accuracy rate 30% than the Mutation Technique.

| S. No | Source Files | MT | EMT |
|---|---|---|---|
| 1 | 20 | 61.73 | 82.47 |
| 2 | 40 | 64.05 | 85.14 |
| 3 | 60 | 66.75 | 87.98 |
| 4 | 80 | 69.23 | 91.23 |
| 5 | 100 | 71.68 | 94.56 |

Table No. 7.1. Test Coverage Accuracy Analysis between Mutation Technique (MT) and Enhanced Mutation Technique (EMT)

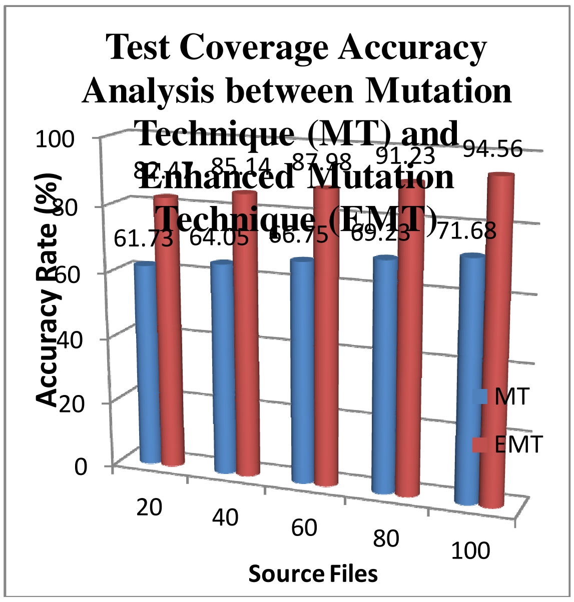

Figure No: 7.1. Test Coverage Accuracy Analysis between Mutation Technique (MT) and Enhanced Mutation Technique (EMT)

## VIII. CONCLUSION AND FUTURE WORK

The system is designed to test the software safety using test coverage with Enhanced Mutation Technique. Mutation testing in the system has identified shortfalls in test cases which are too obscure to be detected by manual review. The process of mutation adds value. Perhaps more importantly, it is necessary to understand the underlying factors which contributed to these shortfalls in the first place. Mutation testing also offers a consistent measure of test quality which peer review cannot demonstrate. The system has produced evidence that existing coverage criteria are insufficient to identify the above test issues. Potentially, the results imply that test engineers are too focused on satisfying coverage goals and less focused on producing well designed test cases. The Enhanced Mutation Testing scheme produces better accuracy level than the Mutation Testing scheme.

The mutation testing scheme is used to test the C and ADA language source code. Test coverage factors are used to estimate the software safety levels. The Java language based mutation testing process is designed to analyze the inheritance relationship based test coverage values. The system can be enhanced with the following features.

- The system can be enhanced to measure the test coverage with the reverse engineering technique.
- The system can be improved to analyze the web applications.
- The system can be applied to measure the safety risk levels in relational database environments.

## REFERENCES

[1] Junpeng Lv, Bei-Bei Yin and Kai-Yuan Cai, “On the Asymptotic Behavior of Adaptive Testing Strategy for Software Reliability Assessment”, IEEE Transactions On Software Engineering, Vol. 40, No. 4, April 2014.
[2] Alex Groce, Todd Kulesza, Chaoqiang Zhang, Shalini Shamasunder, Margaret Burnett and Kevin McIntosh, “You Are the Only Possible Oracle: Effective Test Selection for End Users of Interactive Machine Learning Systems”, IEEE Transactions On Software Engineering, Vol. 40, No. 3, March 2014.
[3] Roberto Latorre, “Effects of Developer Experience on Learning and Applying Unit Test-Driven Development”, IEEE Transactions On Software Engineering, Vol. 40, No. 4, April 2014.
[4] Narayan Ramasubbu, “Governing Software Process Improvements in Globally Distributed Product Development”, IEEE Transactions On Software Engineering, Vol. 40, No. 3, March 2014.
[5] Richard Baker and Ibrahim Habli, “An Empirical Evaluation of Mutation Testing for Improving the Test Quality of Safety-Critical Software”, IEEE Transactions On Software Engineering, Vol. 39, No. 6, June 2013.
[6] Y. Jia and M. Harman, “An Analysis and Survey of the Development of Mutation Testing, Software Engineering,” IEEE Trans. Software Eng., vol. 37, no. 5, pp. 649-678, Sept./Oct. 2011.
[7] Y. Jia and M. Harman, “Higher Order Mutation Testing,” Information and Software Technology, vol. 51, no. 10, pp. 1379-1393, 2009.
[8] G. Jay, J.E. Hale, R.K. Smith, D.P. Hale, N.A. Kraft and C. Ward, “Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship,” J. Software Eng. and Applications, vol. 3, no. 2, pp. 137-143, 2009.