

Achieving High-Integrity Software Quality and Security through Automated Code Analysis, Testing, and Advanced Code Coverage with the LDRA Tool Suite

Nakod Niharika¹

¹Advanced Systems Laboratory DRDO

Publication Date: 2025/07/15

Abstract: In areas where the failure of software entails disastrous consequences, like in the aerospace sector, automotive industry, medical sector and industrial control systems- software production is of primary importance.(10) This paper focuses on the importance of LDRA Tool Suite to attain high degree of software quality and strong security as well as demonstrate how, at a comparatively early stage of the development lifecycle, the entire potential of the LDRA suite may be utilized to uncover hidden flaws, coding standards, and possible security vulnerabilities in the code analysis with regard to the comprehensive testing of software by validation of its unit, integration and system level integration,(11) as well as the high level code coverage (e.g., MC/DC, statement and branch) which is of high importance when certification objectives are to be achieved.(12) This paper illustrates how LDRA tool Suite is useful in designing traceable and verifiable tracks to provide high-quality software reliability to face security risk, meet industry-specific safety or security standards and finally lead to predictable actions within any critical embedded systems.(13)

Keywords: Coding Standards, Security Vulnerabilities, Quality Metrics, Cyclomatic Complexity, LCSAJ Metrics, Equivalence Class Testing, Boundary Value Testing

How to Cite: Nakod Niharika (2025) Achieving High-Integrity Software Quality and Security through Automated Code Analysis, Testing, and Advanced Code Coverage with the LDRA Tool Suite. *International Journal of Innovative Science and Research Technology*, 10(7), 732-739. <https://doi.org/10.38124/ijisrt/25jul468>

I. INTRODUCTION

High-integrity software is essential in such fields as aerospace, automotive, and healthcare: otherwise, a software failure can be disastrous (6). Since more and more software is becoming complex and the needs of the cyber security is becoming threats to the secure creation of the software, it is a challenge in itself to assure that the quality and the safety are high levelled.

The old time manual checks of verification can no longer hold. This requires the usage of code analysis, test, and complete code coverage tools that are automated. LDRA Tool Suite is considered to be one of the leading integrated devices that can ensure these strict requirements. The paper discusses the use of LDRA Tool Suite in successfully meeting high-integrity software quality and security levels that are based on its feature of automated code analysis, comprehensive software testing, and technical ability of code coverage, which are imperative in high standards of certification.

II. LITERATURE REVIEW

Modern quality and security of software used in critical systems is developed on decades of study focusing on such qualities as reliability, maintainability and robustness (3) (Pressman & Maxim, 2020). The integrated security is necessary at the foundation of modern security concerns, which are caused and driven by the combination of systems and threats becoming more and more sophisticated (4) (Schneier, 2015). Code analysis, done automatically, is crucial to the detection of early defects, and identification of vulnerability. Static analysis finds the problems without executing any code and results in implementing standards such as MISRA C/C++ and CERT C/C++ (5) (Seacord et al., 2013). Dynamic testing and analysis ensure that behaviour is checked at the execution point, and the extent to which code is covered by testing (i.e., through code coverage such as MC/DC) is required by standards (1) (Hatton, 1994). Continuous integration/delivery (CI/CD) is currently the popular trend to conduct ongoing quality assurance (2) (Janssen et al., 2022). These verification activities are required in industry standards, typically having demanding verification requirements, such as DO-178C (avionics), ISO

26262 (automotive), IEC 61508 (industrial). To simplify the verification process, integrated tool suites have appeared such as LDRA Tool suite. LDRA is a complete solution with state of the art features that offer static and dynamic analysis (LDRA Testbed), enforcing coding standards (LDRArules), automatic testing (LDRAunit) and traceability (TBmanager). Although the use of LDRA on certification projects has been documented (LDRA product documentation), the proposed research will be used to describe a unified and up-to-date view of its synergistic results on automated analysis, testing and advanced coverage of quality factor combined with that of security and reducing software complexity.

III. PROPOSED METHODOLOGY

The mechanism of creating high-integrity software defined in the proposed methodology is shown in the figure.

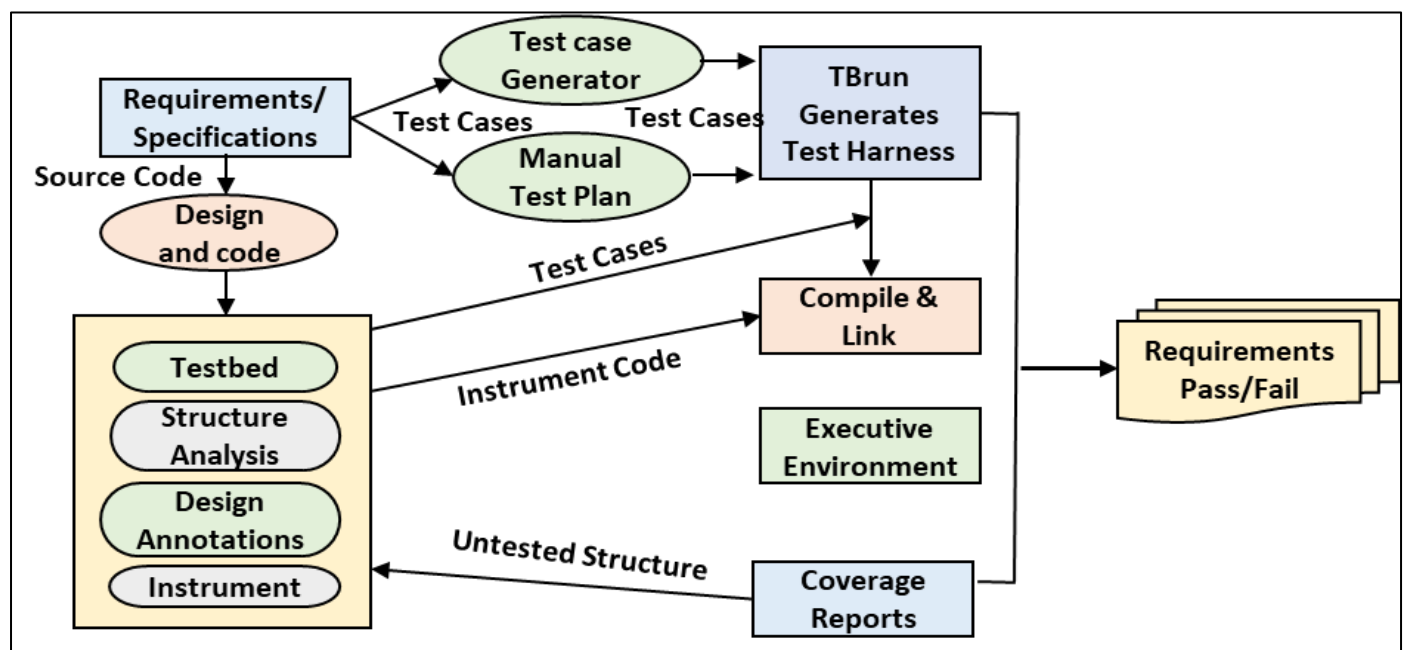


Fig 1 LDRA Flow

A. Software Quality Metrics

A metric is a measurement procedure, and a software quality metric is a procedure in which the inputs are software data, and the output is a unique numerical value that can be explained as the measure of a predetermined quality attribute of the software.

Software metrics are aimed at making observations during the software life cycle in order to determine whether the software quality requirements are being achieved. Software metrics reduce the subjectivity during the assessment and management of software quality because they

provide a mathematical foundation for arriving at a decision regarding the quality of the software.

➤ Cyclomatic Complexity

Cyclomatic Complexity of code section is the quantitative measure of number of linearly independent paths in it and it is a software metric used to indicate the complexity of the program and can be computed by using Control Flow Graph. (7) It is a useful metric for predicting those modules that are probably prone to errors. It can be used for test planning as well as test case design.

Table 1 Range of Complexity

| Complexity | Type of Module | Risk Level |
|------------|----------------|------------|
| 1 - 4 | Simple | Low |
| 5 - 10 | Marginal | Low |
| 11 - 20 | Complex | Moderate |
| 21 - 50 | Complex | High |
| > 50 | Unstable | Very High |

• *Sample Program of Prime Numbers between Two Intervals.*

```
#include<stdio.h>
int main()
{
    int low, high, i, flag;
    printf("enter 2 numbers :");
    scanf("%d %d", &low, &high);
    printf("prime numbers between %d and %d are :", low, high);
    while (low < high)
    {
        flag = 0;
        if (low <= 1)
        {
            ++low;
            continue;
        }
        for (i = 2; i <= low / 2; i++)
        {
            if (low % i == 0){
                flag = 1;
                break;
            }
        }
        if (flag == 0)
            printf("%d\n", low);
        ++low;
    }
    return 0;
}
```

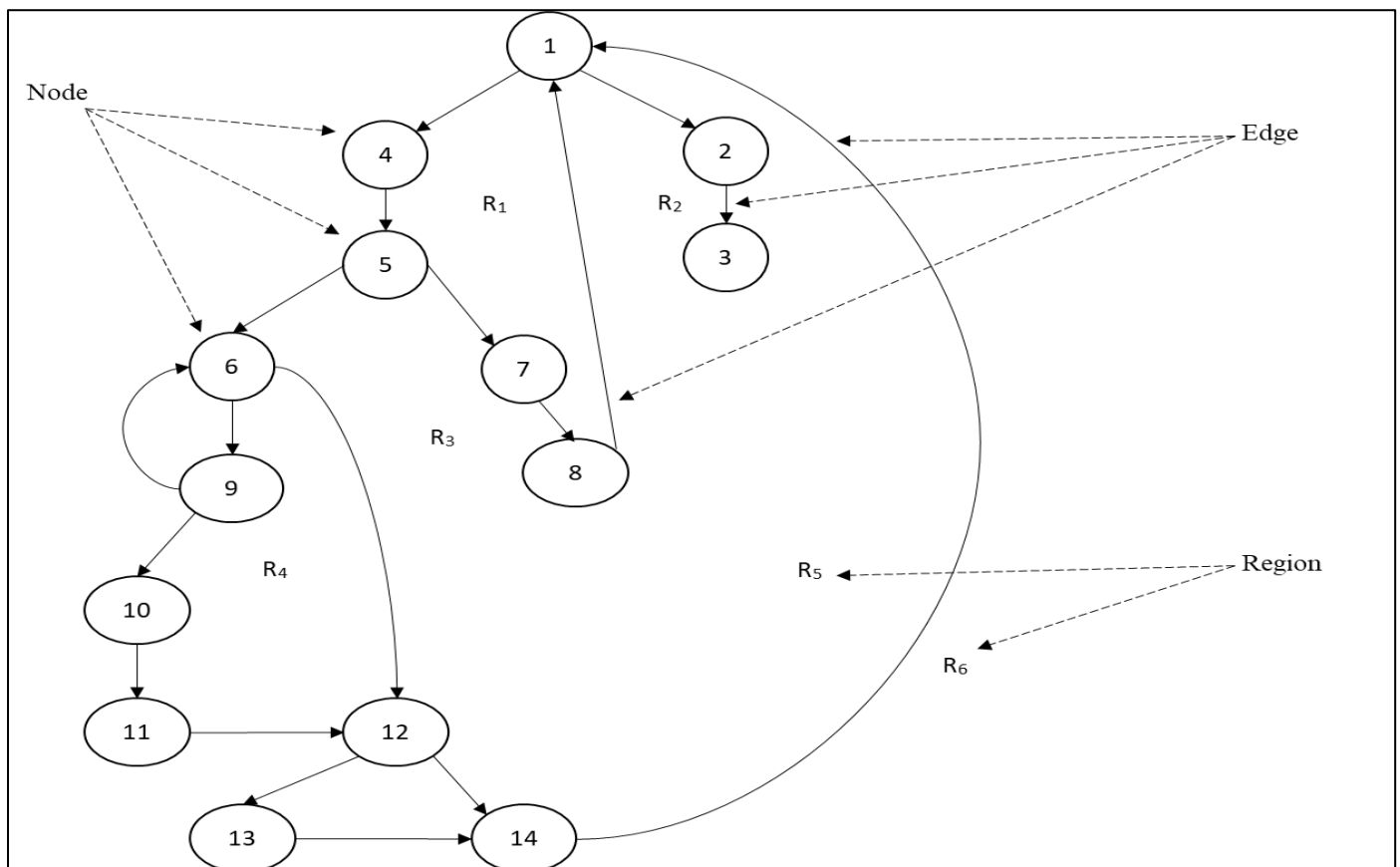


Fig 2 Control Flow

According to the above flow graph, the cyclomatic complexity can be computed manually using each of the algorithms.

- The flow graph has six regions.
- $V(G) = 18 \text{ edges} - 14 \text{ nodes} + 2 = 6$.
- $V(G) = 5 \text{ predicated nodes} + 1 = 6$.

Therefore, the cyclomatic complexity of the above flow graph is 6.

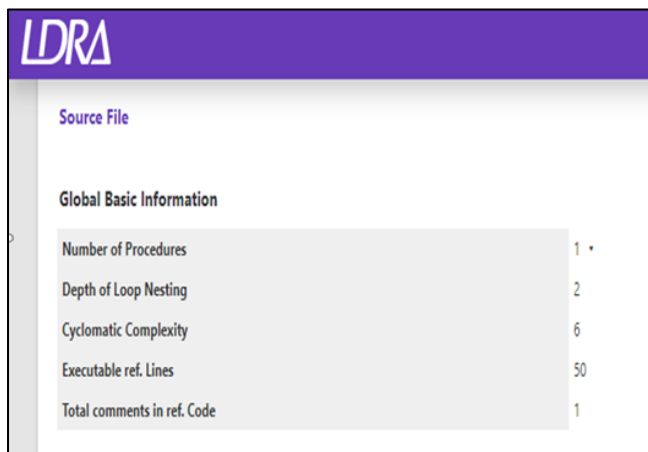


Fig 3 Cyclomatic Complexity Calculated by LDRA Tool.

➤ MCC & MC/DC

Table 2 MCC Vs MC/DC

| Multiple Condition Coverage | Modified Condition /Decision Coverage |
|--|---|
| MCC is a software technique that requires every possible combination of conditions within a decision to be tested. | MC/DC is a software technique that requires demonstrating that each condition within a decision statement independently affects the outcome of that decision. |
| For n conditions 2^n test cases are performed to complete the test | For n conditions n+1 test cases are sufficient to complete the test |

- *Sample Logic*

if(((A) && (B) && ((C))||((D)) && ((E)))

Table 3 Truth Table

| S. No | A | B | C | D | E | Output |
|-------|---|---|---|---|---|--------|
| 1. | T | T | T | F | F | T |
| 2. | F | T | T | F | F | F |
| 3. | T | F | T | F | F | F |
| 4. | T | T | F | F | T | F |
| 5. | T | T | F | T | T | T |
| 6. | T | T | F | F | T | F |

MCC requires 32 Test cases whereas MC/DC requires 6 Test cases to get 100% coverage.

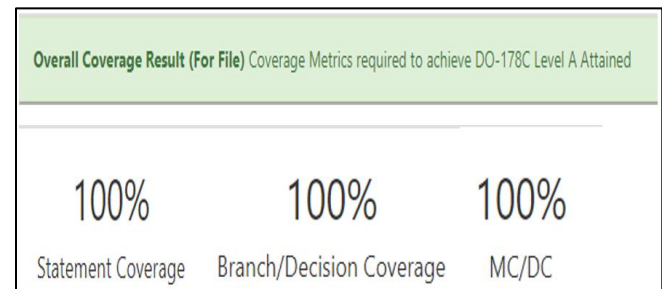


Fig 4 Coverage Report

➤ LCSAJ Metrics

LCSAJ is an acronym that **Linear Code Sequence and Jump**. It is a white-box testing approach to code what part of the code is covered by the test cases in-hand. It assists in establishing new test cases which may expand the coverage of the code under test. When the code coverage reach to a certain extent we can halt the testing. (9) This methodology is actually useful in finding the stopping point of software testing.

- *A Single LCSAJ has the Following Three Components:*

- ✓ Start of the segment, which can be a branch or the start of the program
- ✓ End of the segment, which can be the end of a branch or the end of the program
- ✓ A specific target line

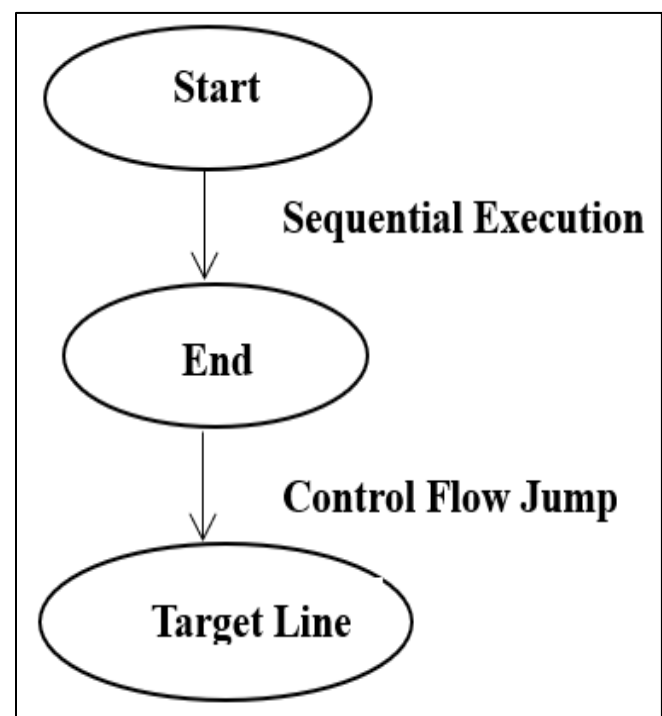


Fig 5 Flow Chart

The code is being executed starting the beginning of the segment until the last line in the segment, and then the control flow discontinues the linear execution and jumps to the line to be executed.

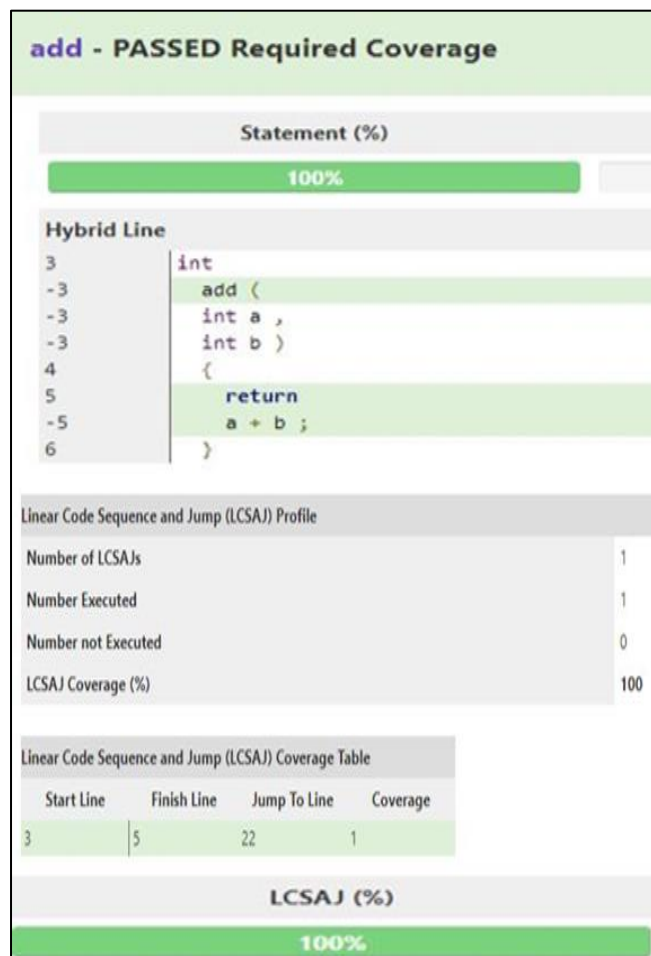
➤ Sample Program to Demonstrate LCSAJ Metrics

```

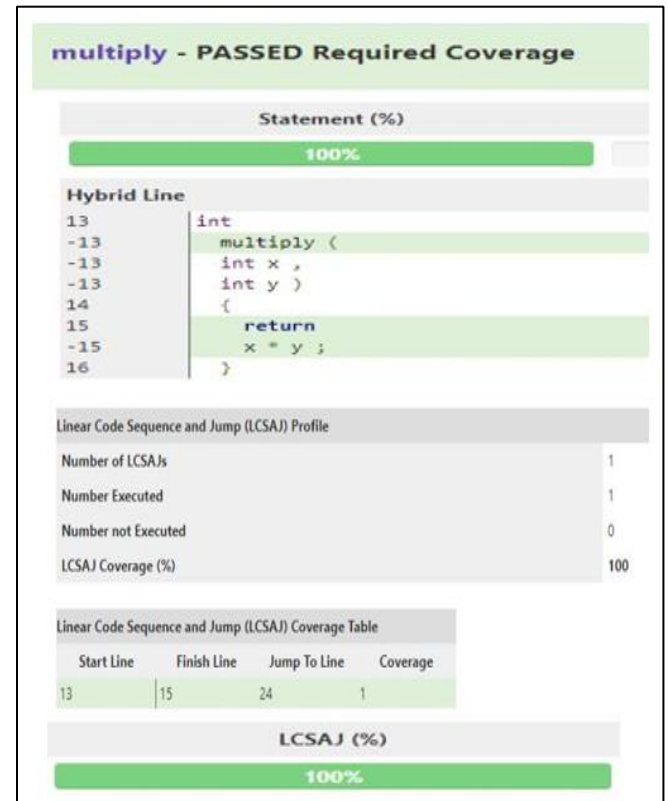
1. #include<stdio.h>
2.
3. int add(int a, int b)           //start line for add()
4. {
5.     return a+b;               //end line for add()
6. }
7.
8. void printResults(int result)   //start line for printResults()
9. {
10.    printf("Result:%d\n",result);
11. }                               //end line for printResults()
12.
13. int multiply (int x,int y)      //start line for multiply()
14. {
15.     return x*y;               //end line for multiply()
16. }
17.
18. int main()
19. {
20.     int num1=5,num2=3;
21.     int sum,Product;
22.     sum=add(num1,num2);        //Jumps to this line for add()
23.     printResults(sum);         // Jumps to this line for printResults()
24.     Product=multiply(num1,num2); // Jumps to this line for multiply()
25.     printResults(Product);
26.     return 0;
27. }

```

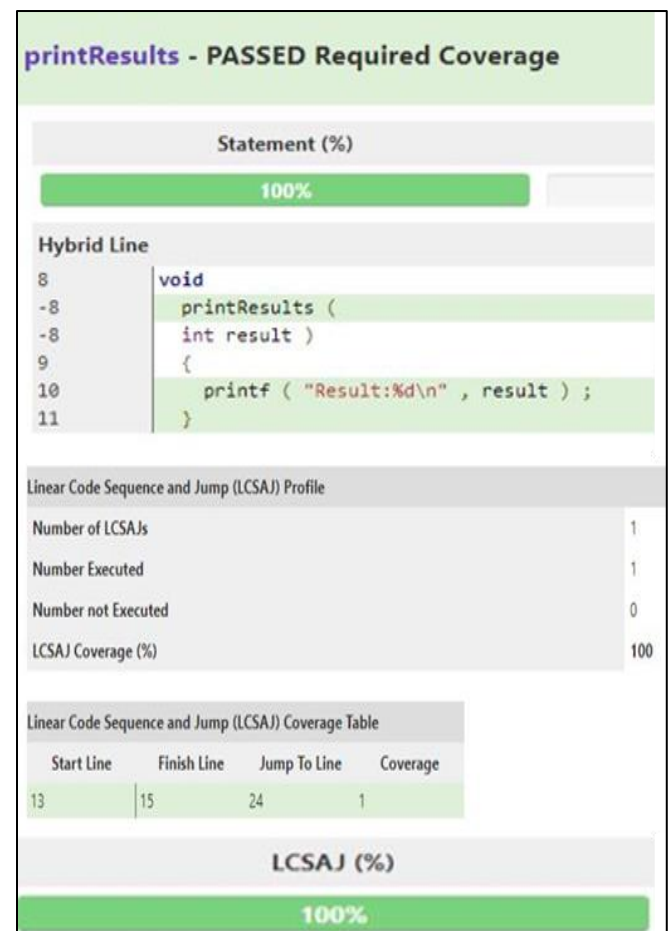
➤ LCSAJ Metric by LDRA Tool Coverage Report for Add () Function



➤ Coverage Report for Multiply () Function



➤ Coverage Report for Print Results () Function



B. Maximizing Coverage by Minimizing Tests

LDRAunit (TBrun) and TBextreme, has strong features to define the test inputs and scenarios in the minimum manual interference.

➤ Code Analysis:

LDRA consists of a fundamental analysis engine (LDRA Testbed); which is a profound static examination of source-code (C, C++, Ada, Java, and so on). It carefully examines the both control flow (the way statements, branches, loops execute in the program) and data flow (the way data moves and changes throughout the program). Such analysis assists interpretation of how the under-test code works (8) (e.g., a module, a function).

➤ Test Driver and Stub Generation:

LDRAunit can automatically generate test harness or driver based upon the interface of the code (parameters, pass output, global variables, functions called). This driver is the framework which invokes the code under test and feeds the inputs. More importantly, it is also able to automatically create what are called stubs of any functions that those functions under test call but are not actually within the unit test being written. These stubs act as the emulation of external functions so that the unit could be tested without them. This helps a lot in manual work of establishing the test environment.

➤ Test Vector (Input) Generation (via TBextreme):

This is the strong point of the "automatic test case generation" using the fine-grained code analysis, TBextreme uses the test input generation (test vectors) to reach desired test objectives, (8) e.g.:

➤ Code Coverage:

It attempts to produce inputs which can exercise all branches/ decisions/ statements and even coverage complex concepts such as Modified Condition/ Decision Coverage (MC/DC) and Linear Code Sequence and Jump (LCSAJ). (8) It does that by intelligently walking through the code.

➤ Boundary Value Testing (BVT):

This has the ability to detect a range of input variables automatically and creates test cases as near or at defined boundaries (minimum, maximum, just inside, just outside) to check common error of going by one wrong.

➤ Equivalence Class Testing (ECT):

It is able to put sets of valid and invalid inputs in equivalence classes and produce test cases that cover each of these classes in order to avoid repeated input.

➤ Robustness Testing/Fuzzing:

It can create malformed or unexpected inputs in an attempt to crash or reveal vulnerabilities in the code, particularly of cyber security interest.

➤ Execution and Reporting:

The test cases generated are then run (on host, target, or simulator) and LDRA takes the results including the code coverage obtained, the pass/fail status, and reports. The

research in this area is aimed at improving, testing and extending LDRA automated functionality in the generation of test cases.

C. Standard Coding Guidelines

The importance of the compliance to the coding standards and utilization of the static analysis tools during the software development lifecycle has been depicted in the paper with the Rule 15.6: Unreachable code shall not be present. (MISRA-C++:2008)

➤ Sample Program with Violation

```
#include <stdio.h>
int main() {
    int x;
    if (1) {printf("This is an example.\n");
    }

    if (x = 0) {
        printf("x is zero.\n");
    }
    // Unreachable code - Violates Rule 15.6
    // (Unreachable code shall not be present)
    return 0;
    printf("This line is unreachable.\n");
}
```

➤ LDRA Report with Violation

LDRA
LDRA tool suite Code Review Report
File: C:\Users\OPR\Desktop\JRF\paper_publications\guide\guide\guide.cpp

Overall Result FAIL
Programming Standards Model MISRA-C++:2008

| Date of Analysis | Report Produced on | LDRA Version | Reporting Scope |
|--------------------------|--------------------------|--------------|-----------------------------------|
| Mon Jul 07 2025 17:00:23 | Tue Jul 08 2025 09:29:18 | 10.0.3 | Source File and Associated Header |

Overall Code Review Summary

| Number of Violations | LDRA Code | Required Standards | MISRA-C++:2008 Code |
|----------------------|-----------|--|-----------------------------|
| 0 | 72 X | Parameter indexing array too small at call. | MISRA-C++:2008 5-0-16 |
| 1 | 1 J | Unreachable Code found. | MISRA-C++:2008 0-1-1,6-6-1 |
| 0 | 3 J | All internal linkage calls unreachable. | MISRA-C++:2008 0-1-1 |
| 0 | 1 Q | Call has execution order dependant side effects. | MISRA-C++:2008 5-0-1,5-14-1 |

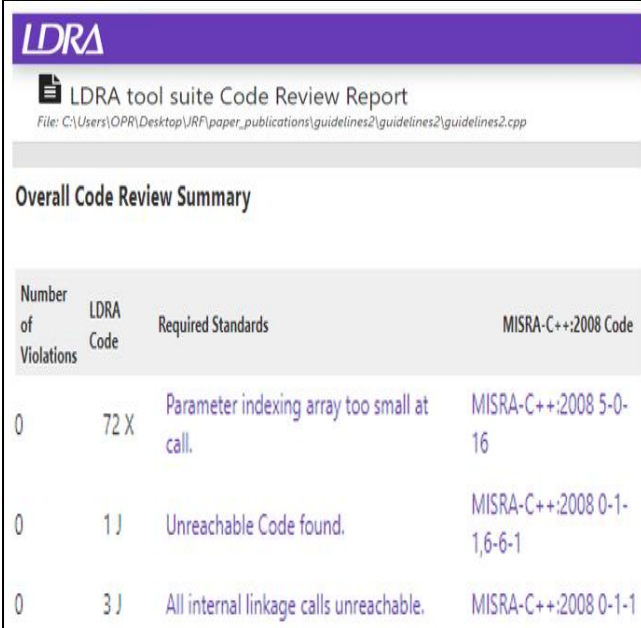
Fig 6 Code Review Report

➤ *Sample Program without Violation*

```
#include <stdio.h>
static void print_message(void);

int main(void) {
    int result = 0;
    print_message();
    result = 1;
    if (result == 1) {
        printf("Result is one.\n");
    }
    else {
    }
    // Rule 15.6: Unreachable code shall not be present
    (all code path reachable)
    return 0;
}

static void print_message(void) {
    (void)printf("This message follows guidelines.\n");
}
```

➤ *LDRA Report without Violation*


| Number of Violations | LDRA Code | Required Standards | MISRA-C++:2008 Code |
|----------------------|-----------|---|----------------------------|
| 0 | 72 X | Parameter indexing array too small at call. | MISRA-C++:2008 5-0-16 |
| 0 | 1 J | Unreachable Code found. | MISRA-C++:2008 0-1-1,6-6-1 |
| 0 | 3 J | All internal linkage calls unreachable. | MISRA-C++:2008 0-1-1 |

Fig 7 Code Review Report

In the first sample program where violation occurs, there is an actual violation of this rule that manifests itself in unreachable code. This violation is subsequently confirmed by an LDRA tool suite Code Review Report that reports a FAIL on the overall result against the MISRA-C++:2008 programming standard. On the other hand, the Sample program without violation shows the adjusted version of the code that all paths of the code can be reached without violation in consideration to the guideline. It forcefully demonstrates the need to establish and enforce rules about coding so that the quality of the code in the way it can be read and maintained, and therefore make it stronger, maintainable and compliant with the rules, will be detected and fixed with

the help of the automated tools (in this case, LDRA). Failure to follow general coding regulations reduces the clarity of such code, which makes it hard to follow and get updated. This increases chances of bugs and mistakes since there is difference in practice that can create small misinterpretation. The standardized coding becomes difficult as a result of diversified styles. There is a total increment in the cost of the project due to the time wasted in the interpretation of the messy code and removing the issues which can be easily solved. Efficiency of tools involved in static analysis e.g., LDRA reduces when they report the violation with respect to the standards, such as MISRA-C++:2008. Also, the failure to adhere to the guidelines can lead to inaccessible code, wastage of resources and complexity.

IV. CONCLUSION

LDRA Tool Suite provides an efficient and practical combination of automation code analysis, powerful testing, and code coverage features to produce high-integrity software quality and security purposes. By means of static and dynamic analysis, the suite anticipatorily finds coding standard contravention, software defects, and security vulnerability guaranteeing compliance with industry-important norms such as DO-178C, ISO 26262, and IEC 61508. Automation of unit and integration test and advanced code coverage analysis by LDRA Tool Suite simplifies not only the verification and validation process but also gives full traceability between the requirements and the code which helps the organization to develop highly reliable, secure and certifiable software that is not only cost effective but done in an efficient way as well.

REFERENCES

- [1]. Hatton, L. (1994). Safer C: Developing Software for Safety-Critical Systems. McGraw-Hill.
- [2]. Janssen, M., Von Knethen, A., & Von Knethen, B. (2022). Continuous Integration and Delivery: A Practical Guide for Software Teams. Springer.
- [3]. Pressman, R. S., & Maxim, B. R. (2020). Software Engineering: A Practitioner's Approach. McGraw-Hill Education.
- [4]. Schneier, B. (2015). Data and Goliath: The Hidden Battles to Collect Your Data and Control Your World. W. W. Norton & Company.
- [5]. Seacord, R. C., Long, M., & Mogul, J. C. (2013). The CERT C Coding Standard: 98 Rules for Developing Safe, Reliable, and Secure Systems. Addison-Wesley Professional.
- [6]. Leveson, N. (2011). Engineering a Safer World: Systems Thinking Applied to Safety. MIT Press
- [7]. McCabe, T. J. (1976). A complexity measure. IEEE Transactions on Software Engineering, SE-2(4),308–320. <https://doi.org/10.1109/TSE.1976.233837>
- [8]. LDRA.(2023).LDRA Test Suite and TBextreme Overview. <https://www.ldra.com>
- [9]. Khurana, S., & Sharma, R. (2012). Software Testing Techniques and Tools. International Journal of Computer Science and Mobile Computing, 1(1), 47-54.

- [10]. ISO 26262: Road Vehicles – Functional Safety. International Organization for Standardization, 2018.
- [11]. DO-178C: Software Considerations in Airborne Systems and Equipment Certification. RTCA Inc., 2011.
- [12]. MISRA C:2012 Guidelines for the Use of the C Language in Critical Systems. Motor Industry Software Reliability Association, 2012.
- [13]. LDRA. (2020). LDRA Tool Suite for Safety-Critical Software Development. <https://www.ldra.com>