Part II

Studies on software reliability
Research methodology

This chapter describes the research methodology followed in the present study. The rationale behind the software reliability definition, choice of case studies, the methods used to estimate software reliability, and experimental details presented.

4.1 Software reliability definition

As mentioned in Section 1.3 on page 2, the definition of software reliability wrt. time is arguable. And in general, reliability in safety systems is quantified in terms of number of failures per demand in case of protection systems, and in number of failures per hour in continuous systems. To cater to both kinds of systems, the present study considers the software reliability definition as [16]: "The reliability of a program P is the probability of its successful execution on a randomly selected element from its input domain". In protection systems, to convert the estimated reliability in to PFD, it is multiplied by demand per hour/year. Whereas, in continuous systems, the estimated reliability is multiplied by the operational profile to get the reliability in terms of failures/hour.

In general, software tends to be slower and unreliable wrt. time due to software aging [183]. The major reasons for software aging include: (i) memory leaks, (ii) floating point error accumulation, (iii) increase in the amount of data to be processed wrt. time, (iv) infection by malware, etc. As safety-critical software tends to smaller, focused, and written in safe subset of programming languages; the above problems can be pro-actively monitored and controlled. Also, as software is fused in to Read Only Memory (ROM), the software cannot be modified by malware. Hence, in the present
study, the software reliability is assumed to remain constant wrt. time as long as the environment remains the same.

### 4.2 Choice of case-studies

![Figure 4.1: Various states of a nuclear reactor](image)

<table>
<thead>
<tr>
<th>System</th>
<th>Abbreviation</th>
<th>Active in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fresh Subassembly Handling System</td>
<td>FSHS</td>
<td>Fuel handling state</td>
</tr>
<tr>
<td>Reactor Startup System</td>
<td>RSU</td>
<td>Reactor startup state</td>
</tr>
<tr>
<td>Steam Generator Tube Leak Detection system</td>
<td>SGTLD</td>
<td>All states</td>
</tr>
<tr>
<td>Core Temperature Monitoring System</td>
<td>CTMS</td>
<td>Reactor in operation state</td>
</tr>
<tr>
<td>Radioactive Gaseous Effluent System</td>
<td>GES</td>
<td>All states</td>
</tr>
<tr>
<td>Safety Grade Decay Heat Removal system</td>
<td>SGDHR</td>
<td>Reactor in shutdown state</td>
</tr>
</tbody>
</table>

Table 4.1: Case studies chosen in the present study

A nuclear reactor can be in any one of the following states (Figure 4.1): (i) Reactor start-up, (ii) Reactor in operation, (iii) Fuel handling start-up, (iv) Fuel handling, and (v) Reactor shutdown. To cover all the states of a nuclear reactor, six case studies have been chosen for the thesis (Table 4.1). Also, as a nuclear reactor spends most of the time in operation state, three case studies have been chosen for reactor in operation state. The RSU and Fuel handling Startup system (FSU) are similar in nature, hence among the two, the current study only presents the results of FSU.

### 4.3 Method

The present study uses the results of software testing to quantify software reliability. The method involves the following steps:
1. **Creation of a model of the software:**

A semi-formal and executable model of the software is created using a pure functional programming paradigm. The model is used as a test oracle.

2. **Generating effective test cases:**

For the given software under test, a set of test cases is generated, such that each test-case has a unique execution path. The test cases are expected to have high MC/DC, LCSAJ coverage, and mutation score.

3. **Calculation of software test adequacy:**

For each case study, the test adequacy of the software with the generated test cases is determined using conservative test coverage and mutation score. The computed test adequacy is in range \([0,1]\); where 0 indicates no testing has been carried out, and 1 indicates that the test cases are likely to detect all faults in the software.

4. **Quantification of software reliability:**

Using the test adequacy value, and based on the accuracy of the test oracle, three approaches to estimate software reliability are proposed.

### 4.4 Experimental details

#### 4.4.1 Software under test

For each case study (*Section 3.2 on page 34*), the software is modeled using the graphical Drakon editor [184], and is converted to the Erlang [185] programming language, which acts as a test oracle. Erlang was chosen due to its pure functional programming paradigm, single assignment variables and pattern matching; which makes it possible to reason with the correctness of the model. Also, erlang has been used to build highly reliable and available telecom systems [186].

The software under test is written in C programming language, following important MISRA [108] guidelines.
4.4.2 Software testing

1. On host:

Most of the testing is carried on the host machine as the target platform may not be powerful enough to perform computationally intensive tasks such as mutation based testing. As the software under test is written in portable C programming language using MISRA guidelines, it is easily portable on the target with minimal changes. The model (which acts as a test oracle) written in the erlang programming language is run on the host machine.

The results on the host machine are matched with results using the Motorola m68k instruction set simulator Musasim [187] before testing on the target hardware.

2. On target:

The test cases are run on the target (a real time computer) [188] by feeding the test cases through the Ethernet and are matched with results on the host machine. In the current study, the software under test runs on bare metal without an operating system. This is to avoid any uncertainty in reliability of operating system, and also due to the fact that most of the safety-critical software in nuclear reactors are simple and focused systems.

For complex safety-critical systems which require multi-tasking, multi-threading, or nested interrupt support; a trusted, safe, and certified real-time operating system must be used (e.g. INTEGRITY [189]). The reliability of such operating systems is assumed to be $\approx 1$ (i.e. using an operating system does not decrease the reliability of the application software). The current study does not report results based on trusted operating systems.

4.4.3 Parallel processing

Some of the techniques presented in the thesis are computationally expensive for large and complex applications, hence are written to support multi-core environment. The results presented in the present study were obtained by executing tasks in parallel on an Intel Xeon X7460 2.66GHz - 24 core machine using the multiprocessing module [190] in Python programming language [191].
Test adequacy in safety-critical software

This chapter proposes a metric using conservative test coverage and mutation score to determine the test adequacy in safety-critical software. The test adequacy value serves as one of the inputs to estimate the software reliability.

5.1 Introduction

Safety-critical software must adhere to stringent quality standards and is expected to be thoroughly tested. However, exhaustive testing of software is usually impractical. The two main challenges faced by a software testing team are generation of effective test cases and demonstration of testing adequacy. The goal of this chapter is to propose a method to generate a set of test cases, and to propose an intuitive and conservative approach to determine the test adequacy in safety-critical software.

Test cases are generated based on the control flow information generated by the compiler, and by using genetic algorithms. The conservative test coverage of unique execution path test cases and the results from mutation testing are combined to determine the test adequacy. Although mutation testing is a powerful technique, the difficulty in identifying equivalent mutants has limited its practical utility. To gain confidence on the computed test adequacy: (i) faults during mutation testing must be induced at all possible execution paths of the code, (ii) properties of unskilled mutants must be studied, and (iii) all equivalent mutants must be detected. To achieve the above goals; results of static, dynamic and coverage analysis of the mutants is presented, and a technique to identify the likely equivalent mutants is proposed.
5.2 Challenges

Software in safety and mission critical applications often require proof that they have been thoroughly tested. Hence, programmers and testers are expected to write good test cases [118] which can verify the behavior of the entire system. However, in real life applications, exhaustive testing is impractical as the input domain could be extremely large or infinite. Thus, the main challenge is to demonstrate the adequacy of testing effectively.

5.3 Software in the case studies

As mentioned in Section 3.2 on page 34, six safety systems in a nuclear reactor are taken up as case studies. The execution flow of the software in case studies is illustrated in Figure 5.1. And the software in case studies has the following characteristics:
1. Software is written in portable C programming language, following important MISRA [108] guidelines.

2. Unless required, signed integers are avoided.

3. Function-like macros are avoided.

4. Only fixed bounded for loops are used.

5. No dynamic memory allocations are used.

6. Cyclomatic complexity of each function is kept below 10 (with few exceptions).

7. The software passes the following static, dynamic, and security checkers:
   
   (a) No warnings with static analyzers: Clang [192], and Cppcheck [193] with 
   
   \[--enable=all\] as argument.

   (b) No warnings with Splint [194] static analyzer using -checks, -strict-lib, and 

   -realrelatecompare as arguments.

   (c) Final score = 0 using BogoSec [195] code security scanner. The scanners 

   include: FlawFinder [196], RATS [197], and Lintian [198].

   (d) No warnings or errors found with dynamic analyzers: Valgrind [199] with 

   \[--leak-check=full\] as argument and Electric-Fence [200] for the generated test 

   cases (Section 5.4.1 on the next page).

8. Assertions have been used to validate inputs and to check impossible conditions 

   during execution. Functions which do not have assertions are either very 

   simple/have error handling code/return the error code to the caller.

9. Apart from assertions in functions, system properties (as post-conditions) in the 

   form of assertions must be met (Figure 5.1 on the previous page).

10. Failure of any assertion leads the system to a safe state (Figure 5.1 on the previous 

    page).
5. Test adequacy in safety-critical software

5.4 Test generation, verification, and coverage

5.4.1 Test case generation

This section proposes an automatic test case generation technique, which can generate a set of test cases (i.e. the sample) which is a good representation of the infinite input domain (i.e. the population).

Safety-critical software is often expected to have 100% MC/DC [201] and LCSAJ coverage [120, 121]. The LCSAJ coverage criterion is considered difficult to achieve and manage, as a small change in code may decrease LCSAJ coverage; thus requiring additional test cases.

To solve the above problem for the case studies, basic functional, safety and boundary tests are written manually, but majority of the test cases are generated through pseudo and true random number generation [202]. A large number of random test cases are generated, out of which unique execution path test cases identified by Message
5. Test adequacy in safety-critical software

For each test case in the list of test cases

Run the test case and generate the coverage information (the .gcov file) using gcov -abcfu

Ignore this test case

Delete the .gcda file

Add this test case to the test suite

Calculate the MD5 hash of the .gcov file using md5sum

Did any of the earlier test cases generate this hash value?
Yes
No

Figure 5.3: Technique to select unique execution path test cases using gcc, gcov and md5sum. (The -abcfu arguments to gcc implies to display coverage information of: all blocks, branch probabilities, branch counts, function summaries, and unconditional branches. The .gcov file consist of the coverage information in text format, where as the .gcda file consist of the arc transition counts and other information in binary format)

Digest 5 (MD5) hash of the coverage information are selected and added to the test suite (Figures 5.2 to 5.3 on pages 48–49).

However, to get good coverage for complex software, simple random numbers are not sufficient. Hence, genetic algorithms \[128\] are used to generate test cases (Figure 5.4 on the next page). Genetic algorithms are evolutionary algorithms, which attempts to generate solutions for search and optimization problems using techniques which mimick natural evolution, such as: inheritance, mutation, selection, and crossover. The algorithm usually starts with a set of randomly generated population and some known solutions. The algorithm is an iterative process where the population
in each iteration is called a generation. The algorithm attempts to generate new and better individuals from the population for the next generation, based on a predefined fitness function.

In the current study, the initial population for genetic algorithm contains randomly generated test cases and black box test cases. From the initial population, new test cases are generated using generic operators (Figure 5.4), out of which unique execution path test cases are selected. Here, the selection of unique execution path test cases serves as the fitness function. This cycle (Figures 5.2 to 5.3 on pages 48–49) is repeated till the required code coverage is achieved (i.e. 100% MC/DC and LCSAJ). Thus, at the end of $n$ iterations, large number of test cases are generated, where each test case has a unique execution path. The goal of generating large number of test cases is to generate as many different execution path test cases as possible, and to ensure that none of them can lead to an unsafe state. For all the test cases generated, it is ensured that the software under test satisfies all the assertions and post-conditions.

![Figure 5.4: Genetic algorithms - inspired by the genetic evolution: crossovers and mutations](image)

**Figure 5.4:** Genetic algorithms - inspired by the genetic evolution: crossovers and mutations

### 5.4.2 Verification of test cases

The generated test cases are verified using a model written using the Drakon editor [184]. The Drakon notations were developed for the Buran space project in Russia [203–205] to provide simple and clean graphical notations for program writing. The
Drakon notations can also be used for requirements modeling, and the resultant model is a semi-formal specification of the software. An example of semi-formal specification in drakon for FSHS is shown in Appendix – A on page 101.

The Drakon editor can automatically convert the diagrams into the Erlang \[185\] programming language; the Drakon-Erlang combination is used to model requirements in visual functional programming paradigm \[206\] using the Drakon editor \[207\]. The generated erlang program is the executable specification of the software, and is used as a test oracle. Erlang was chosen primarily due to its pure functional programming paradigm, single assignment variables, and pattern matching; which makes it possible to reason with the correctness of the model.

After the requirements modeling, the semi-formal specification undergoes basic checks by the Drakon editor, and Erlang specific checks by Dialyzer (Discrepancy Analyzer for Erlang programs) \[208,209\]. Dialyzer checks are performed by enabling all warnings (i.e. -Wunmatched_returns -Wererror_handling -Wrace_conditions -Wunderspecs) \[210\]. Also, the model written in the erlang programming language must always have 100% MC/DC and statement coverage with the generated test cases (Section 5.4.1 on page 48).

5.4.3 Conservative test coverage

The final set of test cases must result in high MC/DC and LCSAJ coverage in the implementation code; else additional test cases should be added manually. As mentioned in Section 2.4 on page 22, use of single control coverage criterion alone could be misleading; hence, we define a conservative coverage metric defined as: the minimum of LCSAJ coverage, MC/DC, branch, and statement coverage. As the branch coverage is always $\leq$ LCSAJ coverage, the conservative test coverage of a function in a program is defined as:

$$\min (\text{LCSAJ coverage, MC/DC, Statement coverage}) \quad (5.1)$$

It must be noted that the above metric (Equation (5.1)) indicates the test coverage
achieved during system testing, and not during unit testing of a function.

5.5 Mutation testing

An effective set of test case must have both good coverage and good fault catching capability. Hence, apart from calculating conservative test coverage, the program under test is subjected to mutation testing. Prior to carrying out mutation testing, the source code is preprocessed by removing all comments; and is formatted/indented to make the syntax consistent for parsing. While compiling mutants, assertions are enabled to kill mutants as quickly as possible. Also, `assert` statements are not mutated, as they represent the conditions which cannot occur during execution.

The effectiveness of mutation testing may be judged by the quality and number of mutation operators used (Tables B.1 to B.2 on pages 109–110). And, to gain confidence on mutation testing, faults must be induced at all possible execution paths of a program.

All execution paths of a program can be visualized by concatenating all the LCSAJs. The Figures 5.5 to 5.10 on pages 53–58 shows all the paths (including the unfeasible paths) in the case studies. It also shows the LCSAJ jump points where faults have been induced and killed. The results indicate that: there exists no path (from program entry to exit) where faults have not been induced and caught, hence giving confidence on the effectiveness of mutation testing.

A mutant program while under execution is polled at regular intervals, and if it does not finish its execution within a specified time period, it is considered to be in infinite loop, and is terminated.

5.5.1 Mutant properties

To gain confidence on the test cases, it is also necessary to understand the characteristics of the mutants which could not be killed, and how they differ from the killed mutants. In this regard static, dynamic and coverage analysis of mutants is performed.

The results (Tables C.1 to C.12 on pages 111–120) indicate that the static analysis of mutants (using Splint [194], Clang [192], and Cppcheck [193] ) alone could not clearly
5. Test adequacy in safety-critical software

Figure 5.5: Concatenated LCSAJs for the FSHS. (The green colored nodes indicate the LCSAJ points where faults have been induced and caught; the red colored nodes indicate otherwise)
Figure 5.6: Concatenated LCSAJs for the RSU. (The green colored nodes indicate the LCSAJ points where faults have been induced and caught; the red colored nodes indicate otherwise)
Figure 5.7: Concatenated LCSAJs for the SGTLD. (The green colored nodes indicate the LCSAJ points where faults have been induced and caught; the red colored nodes indicate otherwise.)
Figure 5.8: Concatenated LCSAJs for the CTMS. (The green colored nodes indicate the LCSAJ points where faults have been induced and caught; the red colored nodes indicate otherwise)
Figure 5.9: Concatenated LCSAJs for the GES. (The green colored nodes indicate the LCSAJ points where faults have been induced and caught; the red colored nodes indicate otherwise)
Figure 5.10: Concatenated LCSAJs for the SGDHR. (The green colored nodes indicate the LCSAJ points where faults have been induced and caught; the red colored nodes indicate otherwise)
differentiate between killed and unkill mutants. Whereas, the dynamic analysis (using Valgrind [199] and Electric-Fence [200]) indicate that the unkill mutants are not likely to have any memory corruptions or leaks. Also, the coverage impact (calculated as the average change in the number of times a statement/branch/jump/function-call was executed in a mutant program with respect to the original program) suggests that the majority of unkill mutants have little or no change in their code coverage.

From the obtained results (e.g. for CTMS — Figures 5.11 to 5.12 on pages 60–61), it is difficult to understand the characteristics of mutants by plotting results of static and dynamic analysis alone. Hence, Principal Component Analysis (PCA) [211] of static, dynamic, and coverage analysis results of mutants is performed. The PCA plot results (Figures 5.13 to 5.15 on pages 62–64) indicate that the characteristics of the unkill mutants have little variance (i.e. they have similar static, dynamic, and coverage properties); when compared to the killed mutants. This result provides little confidence that the majority of the un-killed mutants are likely to be equivalent.

The result also indicates which of the unkill mutants are far away from the original program on the PCA plot (i.e. the mutants which are very much different from the original program). This result helps in prioritizing the unkill mutants (i.e. the farthest unkill mutant from the original program on the PCA plot must be attempted to be killed first). It has also been observed that: similar mutants are nearer to each other on the PCA plot (Figure 5.16 on page 65). Thus, similar unkill mutants could be killed by a adding a new test case.

5.5.2 Calculating mutant score

As mentioned in Equation (2.1) on page 23, the result of mutation testing is the mutation score, defined as:

\[
\text{Mutation score} = \frac{K}{G - E}
\]

where: \(K\) is the number of mutants killed, \(G\) is the number of mutants generated, and \(E\) is the number of equivalent mutants. And unless all the equivalent mutants are detected, the mutation score will always be \(< 1\).
Figure 5.11: Dynamic analysis of CTMS mutants using: Valgrind and Change in coverage
5. Test adequacy in safety-critical software

Figure 5.12: Static analysis of CTMS mutants using: Splint, Clang, and Cppcheck
5. Test adequacy in safety-critical software

Figure 5.13: Principal component analysis (PCA) of static, dynamic, and coverage analysis of mutants for: FSHS and RSU
Figure 5.14: Principal component analysis (PCA) of static, dynamic, and coverage analysis of mutants for: SGTLD and CTMS
Figure 5.15: Principal component analysis (PCA) of static, dynamic, and coverage analysis of mutants for: GES and SGDHR
5. Test adequacy in safety-critical software

```c
unsigned int Sum(unsigned int array[], size_t length) {
    unsigned int i = 0, sum = 0;
    assert (length > 0);
    for (i = 0; i < length; ++i) {
        sum += array[i];
    }
    return sum;
}
```

(a)

```c
unsigned int Sum(unsigned int array[], size_t length) {
    unsigned int i = 0, sum = 0;
    assert (length > 0);
    for (i == 0; i < length; ++i) {
        sum += array[i];
    }
    return sum;
}
```

(b)

```c
unsigned int Sum(unsigned int array[], size_t length) {
    unsigned int i = 0, sum = 0;
    assert (length > 0);
    for (i <= 0; i < length; ++i) {
        sum += array[i];
    }
    return sum;
}
```

(c)

Figure 5.16: Example of mutants with similar static, dynamic, and coverage properties: (a) The original program, (b) Mutant-1, and (c) Mutant-2 (the induced faults are indicated by red color). Both Mutant-1 and Mutant-2 share the same coordinates on the PCA plot.
Hence, to detect likely equivalent mutants, a technique is proposed, which is based on the principle that: if $P$ is a program, and $M$ is its equivalent mutant created by injecting a fault $F$ in the statement $S$, then $P'$ (mutant of $P$) and $M'$ (mutant of $M$) created by injecting fault(s) $F'$ in statement(s) succeeding $S$, must also be equivalent (Figures 5.17 to 5.18 on pages 66–67). Assuming an effective set of test cases, if several such equivalent $P'$ and $M'$ are generated; then $P$ and $M$ are likely to be equivalent.

As the above detection algorithm requires creating large number of mutants, it is computationally intensive. To improve the speed of detection, higher order mutation [212] is used. Also, as each mutant can be executed in parallel, the algorithm is run on Intel Xeon X7460 2.66GHz - 24 core machine using the multiprocessing module [190] in Python [191].
Figure 5.18: Example of equivalent mutant detection: (a) $P$, the original program; (b) $M$, the equivalent mutant of $P$; (c) $P'$, the mutant of $P$; and, (d) $M'$, the mutant of $M$. (The induced faults are indicated by red color) and keywords by blue color)
For every unkill mutant, 10 higher order mutants (each with 10 faults) are generated and are checked for equivalence. The equivalent mutant detection algorithm has detected several non-equivalent mutants and few false positives (i.e. equivalent mutant detected as non-equivalent) (Table 5.1). The false positives are identified manually, and further test cases are added to kill the identified non-equivalent mutants.

5.5.3 Threat to validity

The following situations are the main threats to the validity of the approach:

1. Two equivalent mutants reading data from uninitialized memory locations produce different results, thus the algorithm may identify them incorrectly as non-equivalent.

2. As mentioned in Section 5.5.2 on page 59, the faults are induced in $P$ and $M$ in statement(s) succeeding $S$, to produce $P'$ and $M'$. If the induced fault(s) changes the outcome of the statement $S$ itself (e.g. in loops), then two equivalent mutants may be incorrectly identified as non-equivalent.

3. If the number of equivalent $P'$ and $M'$ generated are very low, then the algorithm may incorrectly identify a non-equivalent mutant as equivalent.

Assuming that the above mentioned uncertainties in mutation score calculation are low, the mutation score is $\approx 1$. An interesting by-product of mutation testing is the identification of safety-critical functions in a program. That is: if a mutant for any of

<table>
<thead>
<tr>
<th>System under test</th>
<th>Number of mutants</th>
<th>No. mutants unkill</th>
<th>No. non-equivalent mutants detected</th>
<th>False positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSHS</td>
<td>1083</td>
<td>309</td>
<td>19</td>
<td>1</td>
</tr>
<tr>
<td>RSUP</td>
<td>343</td>
<td>97</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>SGTLD</td>
<td>4870</td>
<td>1195</td>
<td>58</td>
<td>5</td>
</tr>
<tr>
<td>CTMS</td>
<td>5946</td>
<td>1623</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>GES</td>
<td>334</td>
<td>125</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>SGDHR</td>
<td>877</td>
<td>178</td>
<td>17</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 5.1: Results of the equivalent mutant detection algorithm
the test cases, fails in an unsafe state, then the function in which fault was induced is safety-critical in nature. Such functions must have high test coverage.

5.6 Assurance of rigorous testing through test adequacy

To give weightage to large, complex, frequently called, and safety-critical functions; the test coverage is calculated as a weighted average, given as:

\[
\text{Test coverage} = \frac{\sum t_i w_i}{\sum w_i}
\]  

(5.2)

where, \(t_i\) is the conservative test coverage (Section 5.4.3 on page 51) achieved in a function during the system testing; and \(w_i\) is the weight assigned to each function as:

\[
w_i = \text{No. of statements} \times \text{Cyclomatic complexity} \times \text{Frequency of the function call} \times \text{Safety-critical nature}
\]  

(5.3)

As an effective set of test cases must have good fault catching capability as well as good test coverage, the rigor in software testing expressed as the adequacy of testing, is estimated using both test coverage (Equation (5.2)) and mutation score (Equation (2.1) on page 23) as:

\[
\text{Test adequacy} = \text{Test coverage} \times \text{Mutation score}
\]  

(5.4)

5.7 Results

In the test adequacy results (Table 5.2 on the next page), using the LDRA Testbed [213], it was found that few of the MC/DCs and some of the LCSAJs could not be covered as they are not feasible. Hence, they are ignored. The unfeasible MC/DCs and LCSAJs have been identified and manually checked. In all the case studies, all the feasible MC/DCs and LCSAJs have been covered; hence, the test adequacy in the all case studies is \(\approx 1\).
5. Test adequacy in safety-critical software

System No. of unique No. of unique execution Test
under test test cases path test cases adequacy

<table>
<thead>
<tr>
<th>System under test</th>
<th>No. of unique test cases</th>
<th>No. of unique execution path test cases</th>
<th>Test adequacy</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSHS</td>
<td>$&gt; 3 \times 10^5$</td>
<td>302174</td>
<td>0.9001</td>
</tr>
<tr>
<td>RSUP</td>
<td>$&gt; 3 \times 10^5$</td>
<td>10772</td>
<td>0.8824</td>
</tr>
<tr>
<td>SGTLD</td>
<td>$&gt; 3 \times 10^5$</td>
<td>378554</td>
<td>0.9806</td>
</tr>
<tr>
<td>CTMS</td>
<td>$&gt; 3 \times 10^5$</td>
<td>311002</td>
<td>0.9922</td>
</tr>
<tr>
<td>GES</td>
<td>$&gt; 3 \times 10^5$</td>
<td>95</td>
<td>0.8600</td>
</tr>
<tr>
<td>SGDHR</td>
<td>$&gt; 3 \times 10^5$</td>
<td>98118</td>
<td>0.9655</td>
</tr>
</tbody>
</table>

Table 5.2: Test adequacy achieved in case studies

5.8 Summary of results

To inspire confidence on safety-critical software, proof of adequate testing is a must. This chapter has demonstrated an approach to determine the test adequacy through safety-critical case studies in a nuclear reactor. The conservative test coverage combined with the mutation score is used as a measure of test adequacy. Also, to gain confidence on the computed test adequacy, three main issues are addressed:

1. To ensure that faults during mutation testing are induced at all execution paths of the software, all the LCSAJ triplets of the program under test are concatenated to form a graph. And it is ensured that faults are induced at all possible execution paths of the above graph.

2. To study the characteristics of unkill mutants — PCA of static, dynamic, and coverage analysis of mutants is performed. The PCA results of the case studies indicate that the majority of the unkill mutants have similar static, dynamic, and coverage properties as the original program. Also, the unkill mutants have been found to have lesser variation in their characteristics when compared to the killed mutants. These results give some confidence that: the majority of unkill mutants in the case studies are likely to be equivalent mutants.

3. To detect equivalent mutants — a technique to identify equivalent mutants has been demonstrated. The proposed technique when applied to the case study has resulted in mutation score $\approx 1$. 
Using the proposed method, high test adequacy in the case studies has been achieved. The computed test adequacy (ignoring the unfeasible MC/DCs and LCSAJs), indicate the rigor in software testing carried out. The regulators in safety-critical industries may require the software reliability estimate before permitting the software to be used in the field. The test adequacy value serves as one of the inputs for the software reliability estimate.
Quantification of software reliability

This chapter proposes an approach combining software verification and mutation testing to quantify the software reliability in safety systems. Some theoretical results on factors that may affect software reliability are also presented.

6.1 Prerequisites for the approach

As the proposed approaches are based on software verification and mutation testing, the prerequisites for this approach are:

6.1.1 Set of test cases

As mentioned in Section 5.4.1 on page 48, software in safety applications often requires 100% MC/DC [201] and LCSAJ coverage [120, 121]. Achieving the above criteria may require hundreds (in some cases, thousands) of test cases.

Also, the generated test suite must be reduced by removing redundant test cases which follow the same path of execution (Figure 5.2 on page 48 and Figure 5.3 on page 49).

6.1.2 Set of mutants

The proposed approach requires a set of single fault (first order) mutants. The number of mutants that can be generated depends on the number of mutant operators and the size of the code. As the approach is statistical in nature, the number of generated mutants should be as large as feasible to achieve the required accuracy.
6.1.3 A test oracle

The generated test cases are verified by checking against functional specification, invariants, post-conditions, and safety properties. The test cases which satisfy these conditions are termed as verified test cases. It may not be always feasible to write complete functional specification, safety properties, invariants, and post-conditions to verify all the test cases. In such cases, the test suite is partially verified. If a path in the program is proven or verified, then the reliability of the path is assumed to be $\approx 1$.

6.1.4 Test adequacy computation

Safety-critical software undergoes rigorous testing, but it is impractical to expect that all possible execution paths in a program can be tested. Hence, the rigor in software testing may be expressed as the adequacy of testing, and is estimated using Equation (5.4) on page 69:

$$\text{Test adequacy} = \text{Test Coverage} \times \text{Mutation score}$$

The computed test adequacy is in the range [0,1]; and is useful in achieving a realistic estimate of the reliability based on software testing approaches described in Sections 6.2.1 to 6.2.3 on pages 74–76.

6.1.5 Compiler correctness

Before starting software testing, it must be ensured that the probability that the compiler being used produces correct machine code is $\approx 1$. A verifying compiler is a grand challenge and is an ongoing area of research [214]. And no existing methods can guarantee that a compiler will always produce 100% correct machine code. However, instead of proving the entire compiler correct, the present study attempts to prove that:

"Even if a compiler has faults, they are not likely to be triggered by the program under test".

To achieve the above, three compilers are used to match results of test cases covering all MC/DCs, all LCSAJs of the program under test, and with mutation score $\approx 1$. And, if
all the three compilers (in the present study: gcc [215], llvm [192], and pcc [216, 217]) agree to the outputs of all the test cases, then: the likelihood of incorrect code produced by any of the three compilers is $\approx 0$. Also, the redundant architectures using multiple compilers (Figure 6.1) may provide little diversity similar to the N-version programming [41].

### 6.2 Software reliability estimation

The computed test adequacy Equation (5.4) on page 69 is used as one of the inputs to determine the software reliability. If the test adequacy is $\approx 1$, then the test cases (i.e. the sample) is a good representation of the infinite input domain (i.e. the population). Using the test adequacy value, three approaches to estimate software reliability are proposed:

#### 6.2.1 Approach – 1

If the test oracle or model is an exact representation of the system under test, then: the reliability of the software can be estimated by:

$$\text{Test adequacy} \times \frac{\text{No. of test cases verified}}{\text{Total no. of test cases}}$$  \hspace{1cm} (6.1)
The above approach requires true oracle or pseudo oracle [142]. However, in most of the real life applications, the software under test is large and complex. Hence, true/pseudo oracle may not be available. For such systems: Approach-2 and 3 are appropriate (Sections 6.2.2 to 6.2.3 on pages 75–76).

6.2.2 Approach – 2

The approach is similar to the Monte-Carlo method [218] of calculating the value of \( \pi \). In which, random darts are thrown at a square in which a circle is inscribed (Figure 6.2).

![Figure 6.2: Monte-Carlo method of determining the value of \( \pi \)](image)

\[
\frac{\text{Area of circle}}{\text{Area of square}} = \frac{\pi \times r^2}{(2r)^2}
\]

\[
\frac{\text{Area of circle}}{\text{Area of square}} = \frac{\pi}{4}
\]

\[
\pi = 4 \times \frac{\text{Area of circle}}{\text{Area of square}}
\]

\[
\pi \approx 4 \times \frac{\text{No. of darts inside the circle}}{\text{Total No. of darts thrown}}
\]

Similarly, a program under test may be visualized as a graph (Figure 6.3) consisting of verified (indicated by the symbol \( \Rightarrow \)) and un-verified (indicated by the symbol \( \rightarrow \)) paths; and randomly induced fault is the dart thrown at it.

![Figure 6.3: Example of paths in a program, where reliability of the path \( p_2 \) is known (indicated by \( \Rightarrow \))](image)
In the method of $\pi$ value calculation, a random dart may either fall inside the circle or outside it; similarly, an induced fault may have three possible outcomes (i.e. result of the mutant execution): (i) it fails at least one of the verified test cases, indicating that the fault has been induced in a verified path; (ii) passes all verified test case but fails at least one of the un-verified test cases, indicating that fault has been induced in an un-verified path; (iii) does not fail any of the test cases (the un killed mutant), indicating that the induced fault may not have any effect on the program.

By generating such large number of mutants, and ignoring all the unkilled mutants, the reliability is estimated as:

$$\text{Test adequacy} \times \frac{\text{No. of times at least one of the verified test cases failed}}{\text{No. of mutants killed}}$$

(6.2)

The advantage of this approach is its simplicity, but its results could be biased when estimating reliability for a highly verified software (i.e. if the mutation testing is not effective enough, then large number of verified test cases may incorrectly lead to a higher reliability estimate). Also, it is difficult to integrate operational profile into the approach. This approach is more suitable for non-nuclear safety applications, but may also be used for systems important to safety to get an initial/quick approximate reliability estimate.

6.2.3 Approach — 3

The approach is similar to the Approach - 2 (Section 6.2.2 on the previous page); and is based on the principle that, if in a given program, reliability of an execution path $p$ is known, then other paths in the program sharing code with the path $p$ also share the reliability of $p$.

6.2.3.1 Estimating fraction of shared code

To estimate the fraction of code shared between paths, mutation based testing is performed. For example: in Figure 6.4 on the next page, a program has four paths $p_1, p_2, p_3$ and $p_4$; and the paths $p_3, p_4$ share reliability of $p_2$. If $R_2$, the reliability of path...
6. Quantification of software reliability

Figure 6.4: Faults induced in path $p_3$. (The symbol $\Rightarrow$ indicates a path whose reliability is known, and $\star$ indicates an induced fault.)

$p_2$, is known; then the $R_i$, the reliability of a path $p_i$ can be estimated by:

$$R_i = R_2 \times \text{(Fraction of code shared between } p_i \text{ and } p_2)$$

The fraction of code shared between paths is estimated statistically by injecting faults in paths for which reliability is unknown (e.g. path $p_3$). For example: in Figure 6.4, the first injected fault causes the test cases running through paths $p_2$, $p_3$, and $p_4$ to fail; whereas the second injected fault fails test case running through path $p_3$. If several such single fault (first order) mutants are generated, and are tested against the test cases, then the fraction of code shared between paths $p_1$ and $p_2$ may be estimated by:

$$\text{Fraction of code shared between } p_1 \text{ and } p_2 = \frac{F_{12}}{F_{22}}$$

where, $F_{12}$ is number of times test cases running through path $p_1$ has failed, given that a fault was induced in path $p_2$; and $F_{22}$ is number of times test cases running through path $p_2$ has failed, given that a fault was induced in path $p_2$.

In real life applications though, an un-verified path may share code with several other verified paths, and may even form cycles. To address such issues, a systematic way to estimate the fraction of code shared among paths and the software reliability is described through a pseudocode (Section 6.2.3.2 on the next page)
6. Quantification of software reliability

6.2.3.2 Pseudocode of the approach

1. let $T = \{t_1, t_2, \cdots t_N\}$ be the set of $N$ generated test cases, where $t_i$ represents an unique path $p_i$ in the program. And let adequacy$(T)$ represent the adequacy of the test cases $T$ calculated using Equation (5.4) on page 69.

2. let $V_i$ represent the number of times an un-verified test case $t_i$ in $T$ kills a mutant, given that a fault is induced in the path $p_i$.

3. let $U_i$ represent the number of times an un-verified test case $t_i$ in $T$ kills mutants, when a fault is induced in the path $p_i$.

4. let $M$ be the set of mutants generated for the program.

5. let $I_m$ represent the set of un-verified test case indices in $T$, which can kill the mutant $m$.

6. let $F_i$ represent the fraction of code the path $p_i$ shares with other verified paths.

7. for each mutant $m$ in $M$:
   
   (a) $I_m = \phi$
   
   (b) for each un-verified test case $t_i$ in $T$:
       
       if $t_i$ kills the mutant $m$ then:
       
       $U_i \leftarrow U_i + 1$
       
       $I_m \leftarrow I_m \cup \{i\}$
       
   (c) if $I_m = \phi$ then
       
       ignore the mutant $m$ and continue with next mutant in step $-7$. 


else if $\exists t \in T$ such that $t$ is a verified test case and kills the mutant $m$
then
\[ \forall i \in I_m: \]
\[ V_i \leftarrow V_i + 1 \]
end if

8. $F_i = \begin{cases} 
1 & \text{if the path } p_i \text{ is verified, and meets all properties and invariants.} \\
\frac{V_i}{U_i} & \text{if the path } p_i \text{ is un-verified, but meets all properties and invariants} \\
0 & \text{if the path } p_i \text{ is un-verified and } U_i = 0 \\
0 & \text{if the path } p_i \text{ violates any of the properties/invariants.} 
\end{cases}$

9. Reliability = \begin{cases} 
\text{adequacy}(T) \times \frac{\sum_{i=1}^{N} F_i}{N} \text{ (if all paths are equally likely to be executed)} \\
\text{adequacy}(T) \times \frac{\sum_{i=1}^{N} (F_i \times O_i)}{N} \text{ (if the path } p_i \text{ has the probability } O_i \text{ of execution)} \\
i.e. \text{ the operational profile)
\end{cases}
6. Quantification of software reliability

6.3 Theoretical results

The three approaches described in Sections 6.2.1 to 6.2.3 on pages 74–76 provides a framework for assessing software failure probability to support the licensing process. When little or no information on the operation profile of the software is available (e.g. during commissioning of a new plant). The proposed approaches can be adopted for initial software reliability estimation.

Along with the reliability estimate, it is equally important to understand on what factors the estimated reliability depends on. Hence, if \( P \) represents the number of verified test cases, and assuming that all paths of the software are equally likely to be executed, then:

\[
\text{Reliability} = \text{adequacy}(T) \times \frac{\sum_{i=1}^{N} F_i}{N} \]

\[
= \text{adequacy}(T) \times \left( \frac{P + \sum_{i=1}^{N-P} F_i}{N} \right)
\]

\[
= \text{adequacy}(T) \times \left( \frac{P}{N} + \frac{\sum_{i=1}^{N-P} F_i \times (N-P)}{(N-P) \times N} \right)
\]

\[
= \text{adequacy}(T) \times \left( x + y \times \frac{(N-P)}{N} \right)
\]

\[
= \text{adequacy}(T) \times \left( x + y \times \left( 1 - \frac{P}{N} \right) \right)
\]

\[
= \text{adequacy}(T) \times (x + y \times (1-x))
\]

\[
= \text{adequacy}(T) \times (x + y - xy) \quad (6.3)
\]
6.3.1 Factors affecting the estimated reliability

The Equation (6.3) on the previous page represents the estimated software reliability, which is a function of three variables: (i) \( \text{adequacy}(T) \), the test adequacy; (ii) \( x \), the fraction of verified test cases; and (iii) \( y \), the fraction of code shared between \((N - P)\) un-verified paths and \( P \) verified paths, which is an indication of the software cohesion/reusability. The \( \text{adequacy}(T) \), \( x \), and \( y \) values are in range \([0, 1]\).

The case where \( P = N \) or \( x = 1 \), implies that all the given test cases have been verified and there are no un-verified paths left (i.e. \( y = 0 \)), hence \( \text{Reliability} = \text{adequacy}(T) \).

6.3.2 Achieving target reliability

![Contour graph showing the combination of \( x \) and \( y \) for various reliability values (0.05-0.99), when test adequacy is 0.99.](image)

The Equation (6.3) on the previous page helps in choosing the combination of \( x \) and \( y \)
6. Quantification of software reliability

values required to achieve target reliability (Figure 6.5 on the previous page). For a given reliability, as $x$ increases, the requirement for $y$ decreases. The decrease in required value of $y$ exhibits linear to exponential behavior as the target reliability increases, and becomes a step function as $\text{Reliability} \to \text{adequacy}(T)$ and $x \to 1$ (Figure 6.5 on the previous page). From the Equation (6.3) on page 80 and Figure 6.5 on the previous page, it can be seen that test adequacy is the major factor affecting software reliability.

6.3.3 Properties of the software

For software with high test adequacy (as required by most of the safety applications), the values of $x$ and $y$ help in understanding some properties of the software, and in making further recommendations to the development/testing team. For example:

1. When $x \approx 0$ and $y \approx 0$:
   Almost no verification has been carried out for the software. Hence, rigorous software verification must be recommended for such systems.

2. When $x \approx 0$ and $y \approx 1$:
   The software has very high reusability. Though, the estimated reliability seems to be high, to improve the confidence on the reliability estimate, more software verification must be recommended for such systems.

3. When $x \approx 1$ and $y \approx 0$:
   Nearly all the generated paths have been verified, and very few groups of un-verified paths have been left out, which do not share much code with the verified paths.

4. When $x \approx 1$ and $y \approx 1$:
   An ideal scenario where almost all the generated paths have been verified. Few small groups of un-verified paths have been left out, which share most of code with the verified paths.

   The values of $x$ and $y$ contribute equally to the estimated reliability. However, achieving 100% verification (i.e. $x = 1$) could be difficult, as it requires a true test
oracle. The value of \( y \) may be improved by reusing verified code. Hence, achieving high reliability for software with high reusability is relatively easier.

### 6.4 Results, discussions, and critical review

This chapter has proposed three approaches to estimate software reliability, and a method to improve software reliability to meet target reliability. However, it is a well known fact that software reliability of 100% can never be achieved; i.e. if the proposed approaches estimate the software reliability as 1, it only indicates that: the set of test cases which has resulted in 100% test coverage and mutation score = 1, has been verified; which implies that the software has very high reliability (\( \approx 1 \)). Hence, as no faults are found in the software after running \( N \) test cases, then:

\[
\text{The failure probability of software} < \frac{1}{N} \quad (6.4)
\]

The Equation (6.4) represents the failure probability of software (without considering any confidence level); for example: if \( 10^5 \) test cases with test adequacy = 1 are generated and verified; then, the failure probability of the software must be less than 1 in \( 10^5 \) (i.e.: < \( 10^{-5} \)).

However, it is important to provide a statistical confidence level on the estimated reliability. Many researchers have suggested statistical methods to elicit effective sample size (i.e. number of test cases in the present context) to attain certain confidence level [219–222]. The Wilks criteria [221] (Figure 6.6 on the next page) provides a logical procedure to arrive at the sample size at different statistical confidence levels. It suggests: to get \( 10^{-5} \) probability of failure with 95% confidence level; perform \( N \) experiments such that at least one experiment output falls in the failure domain \( \Omega \) with a probability of \( \beta \) (the confidence level = 95%).

Therefore:

- If the probability that a single experiment output will fail to fall in \( \Omega \) is \( \gamma \).

- Then, the probability that all \( N \) experimental outputs fail to fall in \( \Omega \) is \( \gamma^N \).
And, the probability that at least 1 experiment output is in $\Omega$ will be $1 - \gamma^N$.

Then:

$$\beta = 1 - \gamma^N$$

$$\gamma^N = 1 - \beta$$

$$N \log (\gamma) = \log (1 - \beta)$$

$$N = \frac{\log (1 - \beta)}{\log (\gamma)}$$

$$N = \frac{\log (1 - 0.95)}{\log (1 - 10^{-5})} \approx 3 \times 10^5 \text{ test cases}$$

Hence, at least $3 \times 10^5$ unique test cases have to be generated and verified (Table 5.2 on page 70) for all the case studies to gain 95% confidence level on the reliability estimate of probability of software failure $< 10^{-5}$. The confidence level may be further improved by:

1. Increasing the number of unique test cases.

2. Improving the effectiveness of the mutation testing by using good number of mutant operators which can induce realistic faults into the software.
3. Reducing the uncertainty in mutation score calculation by detecting equivalent mutants correctly.

Considering the fact that all safety-critical software undergo rigorous testing and verification to ensure correctness; the proposed approach is suitable for any safety-critical software.

### 6.5 Summary of results

This chapter shows how results of software testing can be used to estimate software reliability. The main observations of the study are:

1. The test adequacy is the major factor in determining the software reliability in systems related to safety.

2. The estimated software reliability is a function of test adequacy (\( \text{adequacy}(T) \)), the amount of verification carried out (\( x \)), and the amount of verified code reused (\( y \)).

3. For a given software reliability target, as the value of \( x \) increases, the requirement for \( y \) decreases. The decrease in required value of \( y \) exhibits linear to exponential behavior as the target reliability increases, and becomes a step function as the \( \text{Reliability} \rightarrow \text{adequacy}(T) \) and \( x \rightarrow 1 \).

4. The proposed approach re-iterates the fact that: achieving high reliability for software with high reusability is relatively easier.

5. For software with high test adequacy, values of \( x, y \) may give some insights on properties of the software.

6. The probability of software failure in the case studies have been found to be lesser than \( 10^{-5} \) for a random input from the input domain.
Some properties of software reliability

This chapter attempts to generalize the following relationships, observed in the case studies using mutation based testing: (i) Relationship between software reliability and number of faults in the software, (ii) Relationship between software reliability and results of static/dynamic analysis, (iii) Relationship between software reliability and safety. In the current study, for each case study, 7500 mutants (500 mutants for each faults ranging from 1 to 15 in number) are generated randomly from Tables B.1 to B.2 on pages 109–110. And the reliability, warnings/errors during static/dynamic analysis, impact on safety of the mutants are analyzed.

The current study uses the approach based on Section 6.2.1 on page 74 to study the above properties, as the model (the test oracle) built for the current study is a true oracle, and no failures were found during software testing. Also, as the approach in Section 6.2.1 on page 74 is practical and easier to use when compared to other methods proposed in the current study, it is likely to be adopted to quantify real life software.

7.1 Software reliability vs. number of faults in the software

In literature, lot of methods has been proposed to estimate number of faults/defects remaining in the software [223–227], from which one may estimate the software reliability. This section aims to study whether the number of faults is a good indicator of software reliability in safety-critical systems.

For each mutant in all the case studies, the software reliability is estimated, and relationship between the average estimated software reliability vs. defect-density/the
number of faults induced in the software are plotted. The results (Figure 7.1 on the next page) indicate a broad gap between $\pm \sigma$ limits of estimated reliability in the results of defect density vs. the software reliability, for all the case studies combined. Similarly, the results of number of faults vs. the software reliability in 7500 mutants (Figures 7.2 to 7.4 on pages 89–91) indicate: though, on an average the software reliability decreases exponentially wrt. the number of faults in the software, it may not be a good indicator of software reliability. This can be concluded by the broad gap of $\pm 1\sigma$ limit of estimated reliability results. This $\pm 1\sigma$ gap was found to be broad in all the case studies; and the gap seems to reduce only when the reliability $\to 0$, which is not a characteristic of safety-critical software.

As the standard deviation of estimated reliability in the present study was found to be high (Figures 7.1 to 7.4 on pages 88–91), it indicates that the reliability depends upon the severity of faults rather than the number of faults. Hence, reliability estimates based only on the defect-density/number of faults present in the software is likely to be inaccurate for safety-critical software.

### 7.2 Software reliability vs. results of static, dynamic analysis

Static and dynamic analyses are an important part of V&V of software. Static analysis analyzes the software without executing it, and reports warnings and errors; whereas dynamic analysis executes the program with code instrumentation or in a controlled environment, and reports errors/warnings during the program execution.

To understand if there exists any relationship between static/dynamic analysis and the estimated reliability, for all the 7500 mutants in each cases study; the average of warnings during static analysis using splint [194], clang [192], and cppcheck [193]; and errors/warnings during dynamic analysis using Valgrind [199] and Electric-Fence [200] are analyzed. The results (Figures 7.5 to 7.6 on page 92) indicate that the warnings/errors found during dynamic analysis decreases exponentially as the reliability increases. However, no significant relationship could be found between static analysis and the estimated software reliability.
Figure 7.1: Estimated reliability vs. the defect density (in KLOC) for all the case studies. As the software under test is \( \approx 1 \) KLOC, the results for defect density < 1 Defects/KLOC cannot be plotted. (The upper and lower bounds indicate the \( \pm 1 \sigma \) limit, and the software reliability is in the range \([0,1]\))
Figure 7.2: Estimated reliability vs. the number of induced faults — for FSH and RSU (The upper and lower bounds indicate the ± 1σ limit, and the software reliability is in the range [0,1])
Figure 7.3: Estimated reliability vs. the number of induced faults – for SGTLD and CTMS (The upper and lower bounds indicate the \( \pm 1\sigma \) limit, and the software reliability is in the range \([0,1]\))
7. Some properties of software reliability

Figure 7.4: Estimated reliability vs. the number of induced faults — for GES and SGDHR (The upper and lower bounds indicate the ± 1σ limit, and the software reliability is in the range [0,1])
7. Some properties of software reliability

Figure 7.5: Estimated reliability vs. the number of warnings found during static analysis for the all case studies

Figure 7.6: Estimated reliability vs. the number of errors found at dynamic analysis for the all case studies
7.3 Software reliability vs. safety

Software safety can be defined as [228]: "ensuring that software will execute within a system context without resulting in unacceptable risk", and is the most important aspect of a safety-critical system. However, the relationship between software reliability and safety has not been extensively studied in the literature. To establish a relationship between the two, for each mutant generated in Section 7.1 on page 86, a parameter called safety-indicator defined as:

\[
\text{Safety indicator} = \frac{|T_w|}{|E_w|} \tag{7.1}
\]

where \(T_w\) is the weighted safety vector observed while testing, and \(E_w\) is the weighted safety vector expected as per the software requirements using the executable semi-formal model of the system written in Erlang programming language (Section 4.4.1 on page 43). A weighted safety vector \((S_w)\) indicates the amount of safety provided by the software, and is defined as:

\[
S_w = (s_1 \times w_1, s_2 \times w_2, s_3 \times w_3, \ldots, s_n \times w_n), \quad \text{where:}
\]

\[
\begin{align*}
  s_i & \quad \text{is the number of times } i^{\text{th}} \text{ output has led to a safety action, and} \\
  w_i & \quad \text{is the weight associated with the safety-critical nature of the output } i
\end{align*}
\tag{7.2}
\]

For ideal safety-critical software, the safety indicator should be 1. If it is < 1, then the system provides lesser safety than specified in the software requirement which is a potentially dangerous situation. If the safety indicator is > 1, it indicates that the system often shows spurious failures, and provides more safety than required.

For each mutant in the all case studies, the safety indicator (Equation (7.1)) and its corresponding estimated software reliability based on Section 6.2.1 on page 74 is estimated and plotted. To plot the relationship between software reliability and safety, the software reliability is averaged at every 0.2 intervals of safety (indicated by the blue line in Figure 7.7 on page 95). As an ideal safety-critical software has safety = 1 and
reliability = 1 (indicated by the yellow colored line in Figure 7.7 on the next page), it is also shown along with the experimental results. Also, as the magnitude alone cannot distinguish between two vectors, the angle between \( T_w \) and \( E_w \) defined as:

\[
\text{Angle between } T_w \text{ and } E_w = \cos^{-1} \left( \frac{\sum (T_i \times E_i)}{\sqrt{\sum (T_i)^2} \times \sqrt{\sum (E_i)^2}} \right)
\] (7.3)

is also indicated (by color) for each mutant (Figure 7.7 on the next page). The angle between the two vectors is in range \([0^\circ, 90^\circ]\); and indicates the angular similarity between them (aka. the cosine similarity), where \(0^\circ\) indicates that both test output and expected output overlap with each other, whereas \(90^\circ\) indicates that both are perpendicular to each other. The obtained results (Figure 7.7 on the next page) indicate that: as the software reliability increases the safety also increases as reliability \(\rightarrow 1\), hence by improving reliability, safety can be improved. However, as the spurious failures increase, the reliability decreases. Hence, improvement in safety does not guarantee improvement in reliability.

### 7.4 Summary of results

The study on mutant programs of the case study has revealed the following results:

1. Reliability estimates based on number of faults present in the software is likely to be inaccurate for safety-critical software.

2. Safety-critical software should not show any warnings or errors during dynamic analysis. In the present study, it was found that: the average warnings/errors observed during dynamic analysis decreases exponentially as the reliability increases.

3. However, no conclusive relationship was found between software reliability and warnings observed during static analysis.

4. For safety-critical software, the required safety can be achieved by improving the reliability; however vice-versa is not always true.
7. Some properties of software reliability

Figure 7.7: Estimated reliability vs. the safety indicator, for all the case studies.
Summary and open problems

The present work aims to quantify software reliability, and to study properties of software reliability in safety-critical systems. The contributions, observations, and future scope of present study are summarized below:

8.1 Contributions

The present study makes the following contributions:

1. Empirical results on characteristics of mutant programs during mutation testing are presented. Using which, a systematic method to prioritize unkillled mutants is proposed (Section 5.5.1 on page 52).

2. A method for detecting likely equivalent mutants during mutation testing is proposed (Section 5.5.2 on page 59).

3. An intuitive and conservative approach to determine software test adequacy in safety-critical systems is proposed (Section 5.6 on page 69).

4. A hybrid approach using software verification and mutation testing to quantify software reliability in safety-critical systems is proposed (Section 6.2 on page 74).

5. Theoretical results on factors that may affect the software reliability are presented (Section 6.3.1 on page 81).

6. Empirical results on relationship between software reliability and safety in safety-critical systems are presented (Section 7.3 on page 93).
8.2 Observations

The following observations are made during the study:

1. Use of single control coverage criterion alone may not be sufficient to test safety-critical software (Section 2.4 on page 22).

2. For software with high test adequacy, the unskilled mutants have been found to have lesser variation in their characteristics when compared to the killed mutants (Section 5.5.1 on page 52).

3. The estimated software reliability is a function of test adequacy \( \text{adequacy}(T) \), the amount of verification carried out \( x \), and the amount of verified code reused \( y \) (Section 6.3 on page 80).

4. The test adequacy is the major factor in determining the software reliability in systems related to safety (Section 6.3.2 on page 81).

5. For a given software reliability, as the value of \( x \) increases, the requirement for \( y \) decreases. The decrease in required value of \( y \) exhibits linear to exponential behavior as the target reliability increases, and becomes a step function as \( \text{Reliability} \rightarrow \text{adequacy}(T) \) and \( x \rightarrow 1 \) (Section 6.3.2 on page 81).

6. Achieving high reliability for software with high reusability is relatively easier (Section 6.3.3 on page 82).

7. For software with high test adequacy, values of \( x \) and \( y \) may give insights on the properties of the software which may help in making further recommendations to the development/testing team (Section 6.3.3 on page 82).

8. Reliability estimates based on number of faults present in the software is likely to be inaccurate for safety-critical software (Section 7.1 on page 86).

9. The average warnings/errors observed during dynamic analysis decreases exponentially as the reliability increases. However, no conclusive relationship was found between software reliability and warnings observed during static analysis (Section 7.2 on page 87).
10. For safety-critical software, the required safety can be achieved by improving the reliability; however vice-versa is not always true (Section 7.3 on page 93).

8.3 Open problems

The following scope exists for further work:

1. **Enhancing the proposed testing techniques:**
   The proposed testing techniques may be further enhanced by making it change aware. As software evolves over time or is refactored, research in change aware testing can save time and resources.

2. **Software reliability issues on multi threaded/core safety systems:**
   The present study has focused on single-threaded software running on single core processors. However, future safety systems are likely to be powerful and may run on multi-core/threaded environment. Thus, issues pertaining to deadlocks, priority inversion, etc. have to be addressed; and their implication on safety of the system must be studied.

3. **Compiler verification:**
   Compiler verification is one of the grand challenges in computer science and offers great scope for research, as trusted compilers are must for safety applications. Embedded systems often use optimized compilers; hence a lot of challenges and research scope exist in verification of optimizing compilers.

4. **Operating system verification:**
   Verification of Operating system is a complex and challenging task. Research areas such as Just Enough Operating System (JEOS) are interesting and are likely to find practical applications in safety-critical industries.

5. **HMI/ Human factors in plant operations:**
   Human error is one of the major reasons for accidents in safety applications, and hence must be modeled accurately for Probabilistic Risk Assessment (PRA).
8. Summary and open problems

Research to reduce human errors in software based systems is one of the important areas of research.

6. Inclusion of digital I&C systems’ reliability into PSA of nuclear reactors:
Calculating system reliability of digital I&C systems using the software and hardware reliability is still an ongoing topic of research. And scope for research exist to integrate the I&C system reliability into PSA of a nuclear reactor.

8.4 Conclusion

There is an urgent need to demonstrate the safety and reliability of computer based systems in nuclear plants. The lack of commonly accepted methods on the assessment of software reliability may hinder the licensing process of such safety systems.

The methods and analysis presented in this thesis demonstrate the use of software testing to arrive at an estimate of software reliability. Also, the results obtained in this thesis gives insight on the dynamics of building safe and reliable software.

The proposed approaches could be used by safety-critical software developers to improve the software reliability. Further, the regulators may also use the techniques to verify reliability, safety, and dependability claims.

This thesis has proposed a set of methods to quantify software reliability and presented results on properties of software reliability for safety-critical systems. The present study can be enhanced by improving the proposed testing techniques. Further scope exists in issues related to multi-core/threaded safety applications, system software verification (compiler and operating system), human factors in plant operations, and inclusion of system reliability of digital I&C systems in PSA studies of nuclear reactors.