Part I

The context
Introduction

1.1 Background

Safety-critical systems are: "systems whose failure could result in loss of life, significant property damage, or damage to the environment" [1]. Software based systems are replacing pure hardware based systems for safety operations in areas such as: aerospace, automotive, medical, nuclear, etc. This is due to the advantages software based systems offer in terms of functionality, cost, flexibility, maintainability, reusability, etc.

However, the increase in the use of software for critical operations has increased the likelihood of failures occurring due to software faults. For example: an analysis of failures in launch vehicles worldwide shows such a trend (Table 1.1 [2] as cited by [3]).

<table>
<thead>
<tr>
<th>Subsystem</th>
<th>1980s</th>
<th>1990s</th>
<th>2000s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Propulsion</td>
<td>42 %</td>
<td>38 %</td>
<td>54 %</td>
</tr>
<tr>
<td>Guidance and navigation</td>
<td>6 %</td>
<td>16 %</td>
<td>4 %</td>
</tr>
<tr>
<td>Electrical</td>
<td>6 %</td>
<td>8 %</td>
<td>8 %</td>
</tr>
<tr>
<td>Operational ordnance</td>
<td>2 %</td>
<td>8 %</td>
<td>0 %</td>
</tr>
<tr>
<td>Software and computing</td>
<td>0 %</td>
<td>8 %</td>
<td>21 %</td>
</tr>
<tr>
<td>Structures</td>
<td>4 %</td>
<td>6 %</td>
<td>0 %</td>
</tr>
<tr>
<td>Pneumatics and hydraulics</td>
<td>4 %</td>
<td>2 %</td>
<td>0 %</td>
</tr>
<tr>
<td>Unknown</td>
<td>37 %</td>
<td>16 %</td>
<td>13 %</td>
</tr>
</tbody>
</table>

Table 1.1: Worldwide subsystem failures by decade in launch vehicles

This trend is a concern as software failures are usually mistakes in design which are often difficult to visualize, classify, detect, and debug [4]. Also, as software in future safety-critical systems are likely to be more common and powerful, it is necessary to study the dynamics behind building safe and reliable software.
1.2 The problem statement

Nuclear Power Plants (NPPs) are replacing analog equipment with computer based systems for their safety functions such as: reactor start-up, fuel handling, discordance supervision, control rod handling, emergency shutdown, decay heat removal, radioactive waste management, etc.

As software failures in critical systems could be life threatening and catastrophic [5–14]; the increase in software based controls for safety operations demand for a systematic evaluation of software reliability.

1.2.1 Research questions

For software in safety-critical system:

1. How can the rigor in software testing be quantified?

2. What is its probability of failure-free operation? (i.e. the software reliability)

3. What factors are likely to affect the software reliability?

4. How can the software reliability be improved to meet target reliability?

5. What is the relationship between software reliability and safety?

1.3 Motivation

Software reliability is one of the main attributes of software quality, and is popularly defined as:

1. "The probability of failure-free software operations for a specified period of time in a specified environment" [15].

2. "The reliability of a program P is the probability of its successful execution on a randomly selected element from its input domain" [16].

The first definition is made compatible with the hardware reliability definition; thus, making it possible to estimate the overall system reliability [17]. However, the fact
that the failures in software are mainly caused due to its design faults, and not due to its wearing off (i.e. software failures are not direct function of time), makes software and hardware reliability fundamentally different (Figure 1.1). Thus, the definition of software reliability with respect to time is arguable. The second definition, however is independent of time, and is used as the basis in the present study.

An interesting analogy of software reliability called the *minefield analogy* [18], questions whether software failures are probabilistic in nature. The analogy treats the input space of a program as a field, with hidden/unexplored mines; where, mines represent the faults in software, and the path represents software execution flow/path (Figure 1.2 on the next page). As the result of each run/path is deterministic in nature, the software failure must also be deterministic. However, the probabilistic nature of software reliability is due to its operational profile, and the difficulty in detecting
1. Introduction

Figure 1.2: The minefield analogy of software reliability (the mines represent faults in software, and the path represent a single execution flow of the software)

infeasible paths in the software.

Even before software reliability was formally defined, classical/hardware reliability was a well established field. Hence, most of the software reliability modeling and prediction techniques were influenced by hardware reliability modeling techniques. Unfortunately, such techniques have assumptions and limitations [19-21], which are questionable for safety and mission critical software applications. For example:

1. There are fixed number of faults in the software being tested.

2. Whenever a failure is found, it is removed instantaneously, without inducing a new fault.

3. Each fault has the same contribution to the unreliability of the software; and software with fewer faults is more reliable than the one with more faults.

4. The probability of two or more software failures occurring simultaneously is negligible.

5. Enough and accurate software failure data is available for analysis.

6. The execution time between failures is distributed in a known fashion.
7. The hazard rate for a single fault is constant.

8. Tests conducted represent the operational profile.

Assumptions, limitations, and applicability of defect prediction models have been well discussed in critical reviews [19–21] and experiments [22]. Moreover, choosing the right model suiting particular situation/software is also considered a complex task [23, 24]. Also, some of the models have been reported to be less useful in certain development methodologies such as the agile approach to software development [25].

1.4 Software in safety-critical systems

Most of the existing software reliability estimation techniques depend upon failure statistics to predict reliability. These techniques require enough and accurate failure data for analysis. Hence, unless enough software failures have been observed, the software reliability cannot be predicted accurately.

But, software built for safety-critical applications are different from business-critical or general purpose systems. Generally, safety systems are: (i) smaller and focused, (ii) rugged and have fault tolerant features, (iii) designed with defense in depth, (iv) Written in safe subset of programming languages, (v) expected to have lower failure rates, (vi) meant to fail in fail-safe mode, and (vii) not expected to rely on human judgment or intervention to initiate safety action.

Given the rigorous nature of safety-critical software development, a fundamental question may be asked:

"Whether a software system having experienced lot of failures, fit to be used in safety-critical system to begin with"?

Too many software failures indicate that something is fundamentally wrong; and raises doubts on the development and verification processes being followed. Hence, the confidence on the reliability estimates based on historical failure rates for safety-critical systems would be low.
1.5 Software in nuclear reactors

Based on safety, systems in a nuclear reactor may be classified into three categories [26]:

1. **Safety Critical (SC):**
   Systems important to safety, provided to assure that under anticipated operational occurrences and accident conditions, the safe shutdown of the reactor followed by heat removal from the core and containment of any radioactivity is satisfactorily achieved.

2. **Safety Related (SR):**
   These are systems important to safety, which are not included in safety-critical systems, but are required for the normal functioning of the safety systems in the reactor.

3. **Non-Nuclear Safety (NNS):**
   Systems which do not perform any nuclear safety function.

For each category, the International Atomic Energy Agency (IAEA) as well as the atomic energy regulator in the respective countries issue guidelines [27, 28] on best practices in software requirement analysis, defense in depth design, safe programming practices, verification and validation processes, etc. The regulators expect a formal systematic review of the software and its associated hardware using requirement specifications and independent reviews.

1.6 Software failures in nuclear industry

Even though the nuclear industry is well guided and regulated, it is not immune to software failures. Documented software failures in the nuclear industry include:

1. Canada's Therac-25 radiation therapy machine delivered high radiation doses to patients [5].

2. Files become inaccessible to the nuclear accountants using nuclear material tracking software at Kurchatov institute, Russia [29].
3. *Slammer* worm disabled safety parameter display system for 5 hours at Davis-Besse nuclear power station [30].

4. Computer resets the control system after software patching and reboot at Edwin I. Hatch nuclear power plant [31].

5. *Stuxnet* worm infects nuclear plants in Iran running Supervisory Control and Data Acquisition (SCADA) systems controlled by Siemens software [32] and several others [33]. The main reasons for the failures include: improper/imprecise requirement specification, insufficient testing, use of untested Commercial Off the Shelf Software (COTS), incorrect reuse of older software, vulnerabilities in the software, etc. Hence, an ideal software reliability quantification approach must take such factors into consideration.

### 1.7 Issues in software reliability quantification

Difficulty in quantifying software reliability is due to the factors such as: software complexity, difficulty in identifying suitable metrics, difficulty in exhaustive testing, difficulty in quantifying effectiveness of test cases, etc. Also, there are difficulties in implementing high level guidance [34] and establishing a working consensus. Deterministic analysis such as hazard analysis and formal methods are generalization of the design basis accident methodology used in the nuclear industry. However, probabilistic analysis is considered more appropriate as software faults are by definition design faults.

As safety systems in a nuclear power plant are categorized based on their importance to safety; for computer based systems, the International Electro-technical Commission (IEC) standards give requirements in the form of Safety and Integrity Level (SIL) [35]. SIL is specified in the form of a number from one to four based on the probability of failure. SIL-1 represents the lowest safety integrity level with target average Probability of Failure on Demand (PFD) between $10^{-2}$ and $10^{-1}$, whereas SIL-4 is the highest with PFD between $10^{-5}$ and $10^{-4}$. Common safety functions in NPPs are governed
by defense in depth principles such as: reactivity control, maintenance of fuel integrity, control of pressure boundary, continuation of core cooling, and prevention of release of radioactivity. In view of the inherent complexity in such control software, it is difficult to assess the failure probability of software and quantify the influence of its safety function on core melt down frequency.

1.8 Need for a new approach

As the software developed for critical systems are different from traditional software systems; it is unclear if the traditional Software Reliability Growth Models (SRGMs) are suitable for critical applications. Studies such as [36, 37] suggest that the amount of time required in testing for demonstrating ultra-high reliability is in-feasible. Software testing with large number of test cases without analyzing the quality/effectiveness of test cases, cannot give confidence on the reliability estimate. The current methods to quantify the quality of test cases include: test coverage and mutation based testing [38]. However, as Littlewood quotes [39]: "most software testing is unlike operational use, and any reliability predictions based on this kind of classical testing will not give an accurate picture of operational reliability". Also, the principle findings of a U.S. Nuclear Regulatory Commission (NRC) report quotes [40]:

1. "Most of the existing quantitative software reliability methods were not developed specifically for supporting quantification of software failure rates and demand failure probabilities to be used in reliability models of digital systems".

2. "All methods are based on assumed empirical formulas that are not applicable in all situations."

Some qualitative improvement in software reliability may be achieved with N-version programming [41]; however, it is costly and its benefits are arguable [42]. Hence, the current licensing procedure for computer based systems in nuclear reactors is based on deterministic criteria. For a risk-informed regulation, a procedure for software reliability estimation is not yet been satisfactorily developed [43, 44].
An ideal way to demonstrate that the software meets a required reliability is through formal verification. Formal verification is a method of proving certain properties in the designed algorithm, with respect to its requirement specification written in mathematical language/notation. Approaches to formal verification include formal proof and model checking. Formal proof is a finite sequence of steps which proves or disproves a certain property of the software, whereas model checking achieves the same through exhaustive search of the state space of the model. Unfortunately, it is not always feasible to ensure complete formal verification of software due to the difficulties involved such as state space explosion and difficulties in practical implementation of formal methods [45]. Also, a major assumption in formal verification is that the requirements specification captures all the desired properties correctly. If this assumption is violated, the formal verification becomes invalid.

Hence, reliability estimates based on software testing has been adopted by many for decades. Repeated failure free execution of the software provides a certain level of confidence in the reliability estimate. However, it is well known that software testing can only indicate the presence of faults and not its absence.

Some of the existing defect prediction models predict the number of faults present in software based on the historical failure trend. However, they fail to pin point the remaining defects. For real world safety applications, predicting the reliability alone is not sufficient; hence, an ideal reliability estimation approach must also provide a way to improve the reliability. Hence, there is a need for a systematic and robust software reliability estimation method suitable for critical applications related to safety.

1.9 This thesis

1.9.1 Assumptions and limitations

1. Software for safety systems may be divided into five basic modules (Figure 1.3 on the next page):

   (a) A hardware-interface module, which can take inputs from sensors (e.g. for
1. Introduction

1. Hardware interface → Network interface

   Systems' core module

   Diagnostic routines ← User interface

**Figure 1.3:** Typical software architecture of safety applications in nuclear reactors

- temperature, pressure, flow etc.) and send outputs to final control elements such as: motors, relays, blowers, heaters, etc.
- (b) A user-interface module, which interacts with the user.
- (c) A network-interface module, which can share soft inputs/outputs with other connected systems.
- (d) A diagnostic module which checks the state of the system at regular intervals.
- (e) The main/core module which performs the systems' intended function.

The main/core module of various safety systems are used as case studies in the present study, for which source code is available.

2. The focus of the thesis is on pure software failures (indicated by the shaded portion in - Figure 1.4), and not on system failures arising due to hardware or hardware-software interaction.

**Figure 1.4:** Focus of the present study: failures caused due to software faults (indicated by the shaded portion of the venn-diagram)
3. The software is written in portable C-programming language, adhering to Motor Industry Software Reliability Association (MISRA) standards.

4. The software is single-threaded and runs on bare hardware without any operating system support.

5. Software is testable, i.e. it has a test oracle, using which large number of test cases can be verified automatically.

1.9.2 Structure

The thesis is structured as follows:

• Part - I (The context)
  
  – Chapter-1
    
    * Outlines the context, motivation, goals, and contributions of this thesis.
  
  – Chapter-2
    
    * Reviews related work in formal methods, model checking, software testing, software reliability estimation methods, etc.
  
  – Chapter-3
    
    * Provides background information on the case-studies used in the present study.

• Part - II (Studies on software reliability)
  
  – Chapter-4
    
    * Describes the research methodology being followed.
  
  – Chapter-5
    
    * Proposes an approach to determine the test adequacy in safety-critical software.
1. Introduction

- **Chapter-6**
  - Proposes an approach to quantify the software reliability in safety-critical systems.

- **Chapter-7**
  - Presents some empirical results on properties of software reliability in safety-critical systems.

- **Chapter-8**
  - Summarizes the thesis, and lists out some of the open problems
Related work

2.1 In formal methods

Natural languages such as English have been widely used in the requirement specification of software, popularly known as the Software Requirement Specification (SRS) document. The advantages of using natural languages include: (i) better understand-ability by large and diverse audiences, (ii) search-able using keywords, and (iii) ability to specify large projects. However, natural languages are easily prone to ambiguity and imprecision. These problems were very early recognized and well discussed [46].

Experience [47–50] indicates that the errors in requirement specification is the major cause for software failures, and are the costliest to fix. In this regard, formal methods are being adopted in critical areas to prove that the software meets its functional requirements [51–55]. Formal methods are techniques based on mathematics to prove/disprove certain properties in software or specification.

As a precise and clear specification is the first step in developing reliable and fault free software; the use of formal methods with sound mathematical base and notations seemed to be the right way to solve these problems. Hence, a lot of research has been done in developing formal specification languages. Various types of languages/techniques include:

1. Algebraic specification languages: Algebraic specification is a formal process of writing specifications in mathematical structures and functions. Vienna Development Method (VDM) [56], Z (zed) notation [57] and B-Method [58] are
the most popular algebraic specification languages both in academia as well as in industries [59, 60]. A detailed description and comparison of various specification languages can be found in [61]. Applications and features of VDM and Z are well discussed in [62]. Also, works such as [52] highlight the experiences with formal specification languages and formal methods in general.

2. Object oriented modeling techniques: Due to rise in popularity of object-oriented paradigm, and limitations of Z and VDM to model object-oriented systems; VDM++ [63] and Object-Z [64], the object-oriented extensions for VDM and Z respectively were released. But, the Unified modeling language (UML) became the most widely used notation for object-oriented modeling. However, UML does not support specification of constraints in the model. As constraints make a model precise and complete, languages such as: Object Constraint Language (OCL) [65], Java Modeling Language (JML) [66], and Spec# [67] were developed. OCL is an Object Management Group (OMG) standard language, used to specify pre-conditions, post-conditions and invariants in UML diagrams. Whereas JML is a behavioral interface specification language developed to specify Java classes and interfaces. JML specifications are written as Java annotation comments in the source files, and tools such as jmlc [68] compile JML annotated Java files with runtime assertion checks. Spec# is a formal language for API contracts; it is a super-set of C# with constructs for non-null type variables, class contracts and method contracts like pre-conditions and post-conditions. It leverages on the popular C# programming language and .NET framework; for easier adoption by programmers.

3. Special purpose languages: Eiffel [69], originally developed by Eiffel Software is an object-oriented programming language which has introduced and popularized the set of principles such as: command-query separation, design by contract, open-closed principle, option-operand separation, single-choice principle, and uniform-access principle. Some of these principles were later adopted by many other specification and programming languages. The goal of Eiffel programming method is to enable programmers to create reliable, reusable, and correct software. The
Prototype verification system (PVS) [70], developed at the Computer Science Laboratory of SRI International, California, USA, is a framework for writing formal logical specifications and constructing proofs. PVS has been successfully used in specification and verification of various critical applications [71] in organizations such as NASA for Cassini aircraft [72] and Space Shuttle [73].

4. **Functional programming languages:** Even though general purpose programming languages offer a lot of flexibility to the specifier, they are not suitable to be used as a formal specification language. Only pure functional programming languages such as Lisp [74] and Haskell [75]; which offer referential transparency [76], can be considered suitable for the purpose. For example: Haskell has been used to formally verify a micro-kernel seL4 [77].

5. **Domain Specific Languages (DSLs):** Domain-Specific Languages [78] are special purpose languages that allow specification or development of applications for a specific domain. Unlike general-purpose programming languages, DSLs contain fewer programming constructs, and are easier to learn. As they are used by people well aware of the domain, writing and reviewing specification is also easier. Also, DSLs with visual programming interfaces helps domain experts with little/no programming background to write specification. However, due to their fewer programming constructs, DSLs lack flexibility, and may require frequent additions or modifications as the domain evolves/changes.

**Critical review of specification languages**

Formal methods ensure systematic software development by ensuring correctness at early stages of software development. Formal methods when applied correctly have been found to be successful in certain applications [60,72,79-81]. However, formal methods have not been widely adopted by software engineering practitioners due to the following reasons:

1. **An expert is required to get started, and should be always available:** Successful use of formal methods requires selection of suitable notation, right tools and fair amount
of discrete mathematical skills. Hence, a team of experts is required to get started [45].

2. No specification language is suitable for all kinds of systems: There are too many specification languages; each one suitable for a particular kind of application. For example: Z and VDM are well suited for structured systems, Object-Z and VDM++ for object-oriented modeling, and Lustre [82] for modeling reactive systems. Also, as no one specification language can be used to specify all aspects of a large system, one may have to mix two or more specification languages to achieve desired results. In such cases, the difference in syntax among specification languages adds up to the complexity [83].

3. Formal specification languages have poor readability: Early formal specification languages like Z are heavily based on mathematical notations, and use lot of non-ASCII symbols in their syntax, which require special tools for writing specification. These languages seem to have ignored readability and usability to achieve precision and expressiveness. Due to their complex syntax, testers may find it difficult to write test cases from specification. Also, most of the programmers are not well trained in these notations, which could easily lead to incorrect implementation and imprecise verification and validation (V&V). Although, automatic code generators which generate target code from the specification, try to address this problem. However, testing is usually performed at the model level, and unless the generated code is clean and understandable programs; it is difficult to test and debug the code.

4. Once written, they are often difficult to maintain: It is a well known fact that: in real-world scenarios, software requirements and specifications are not frozen. Software may require frequent additions, modifications, and deletions to meet new user demands and comply with new standards and regulations. Once written, the complex mathematical notations are difficult to modify, and requires an expert to do the work. Also, other concerned members like managers, testers, certifiers, and end users may not be mathematically inclined; and may find it difficult to
understand, verify, and review the specification. Thus, these languages may not be suitable for large and complex applications, where requirements may change frequently.

5. **High cost is required in training the staff**: Training each and every member on formal methods takes a lot of time and money due to scarcity of experts in these areas; making it very difficult for small and medium sized organizations, as well as projects with tight budget and time constraints to apply formal specifications successfully.

6. **Formal methods usually focus on functional specification**: Most of the formal specification languages focus on functional specification, but not well on non-functional specifications such as: performance, security, maintainability, testability, etc.

A study in nuclear software development [84,85] conducted by University of Virginia on staff at University of Virginia reactor (UVAR); consisting of nuclear engineers, computer scientists, and developers; revealed similar barriers in practical implementation of formal methods. Improvements in tools such as graphical formal notations such as Safety Critical Application Development Environment (SCADE) [86] attempts to make specification simpler and readable, and have been used in various safety and mission critical applications [87–92]. However, formal methods in general have the following limitations:

1. Formal methods assume accurate transformation of formal specification or the model to implementation code.

2. Formal methods do not have information about the operating environment such as: underlying hardware, operating system, network configurations, etc.

3. Results of formal methods can be negated by faults in compilers.

4. Proving large, complex, or non-linear properties using formal methods is difficult, time consuming, and sometimes impractical.
5. Formal methods cannot indicate if enough properties have been proven.

6. The result of formal methods is qualitative in nature, and thus cannot be directly used to quantify software reliability (i.e. formal methods are not Quantitative Software Reliability Method (QSRM)).

2.2 In model checking

Model checking \[93, 94\] refers to exhaustively checking if a given model of a system satisfies a given property. The system is usually modeled in the form of a finite-state machine, and is checked if the given property is valid for all states and transitions of the model. If the given property fails, counterexamples can be generated. This model also enables automatic test case generation for the system. Further research in model checking through symbolic model checking using Binary Decision Diagrams (BDDs) \[95\] and satisfiability (SAT) solvers \[96\] has improved the speed of model checking. However, these techniques may not be scalable for large and complex systems.

Bounded model checking (BMC) \[97\] is an efficient technique to verify the given property in a bound of \(k\) steps. The main advantage of BMC is that it does not suffer from state space explosion problem, hence is likely to be a practical technique; and work such as \[98\] highlights the benefits of BMC in an industrial setting. Systematic survey on model checking and its associated tools can be found in \[99, 100\].

Critical review of model checking

Model checking has been used successfully in practice \[101–105\] to prove safety and liveliness properties. However, the model checking can only check finite state systems. Also, creating a good mathematical model for a large and complex system is a challenging task \[106\]. Research on automatic extraction of states to build a model from a given program is in progress \[107\].

As exhaustive model checking may not scale well for large problems due to the state space explosion problem; Bounded model checkers (BMC) were proposed, which can
check a given model without the state space explosion problem. But, due to the \( k \) bound, completeness cannot be achieved.

2.3 In safety-critical software development, V&V

Software in safety systems are built with utmost care, and are written in safe subset of programming languages such as: MISRA C/C++ \([108, 109]\), JSF++ \([110]\), SPARK Ada \([80]\), etc.

Also, the tools used for testing safety related software are expected to be dependable. Earlier works such as \([111]\) has reviewed and evaluated software correctness and security assessment tools under various categories such as: static analysis, source code fault injection, dynamic analysis, binary fault injection, byte-code analysis, etc. Among them, static source code analysis tools have been proven to be the most mature, as they are found useful in multiple phases of the software development life cycle. Source code fault injection tools provide mechanism through which source code can be instrumented to induce the code to follow control paths that would be otherwise difficult to test. A detailed analysis on benefits and drawbacks of each of the tools under the respective categories has also been described \([111]\).

Safety-critical industries often receive guidelines from their regulators on software verification and validation processes (e.g. ISO-26262 \([112]\) for automotive, DO-178B \([113]\) for avionics, EN-50128 \([114]\) for railways, IEC-61508 \([115]\) for nuclear power plants, etc.). Compliance with specific safety standards and guidelines is mandatory to ensure the quality of software used in safety-critical industries.

Apart from following the respective standards and procedures, the safety critical software undergoes rigorous testing. International standards limit the rate for catastrophic failures to be less than \(10^{-8}\) failures per hour for continuous control systems and less than \(10^{-4}\) failures per demand for protection systems such as emergency shutdown systems \([116]\).
Critical review of safety-critical software development and V&V techniques

Safe subsets of programming languages reduces the likelihood of dangerous faults in software [117], hence are recommended for building safety applications. Also, popular safe subsets such as MISRA-C/C++ are regularly reviewed and updated. However, no specific safe-subset standards exist for nuclear applications.

Good development practices, reviews and independent V&V helps in building reliable and safe software. However, results of reviews and V&V are deterministic and are usually check-list based, hence cannot be directly used to quantify software reliability.

2.4 In software testing and test coverage

Testing is a process of giving a set of inputs to the software under test and match its output with the expected output. 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, as exhaustive testing is impractical in real world applications, the amount of testing is quantified through test coverage.

In general purpose applications, statement coverage and branch are the two popular test coverage criteria. For safety applications, Modified Condition/Decision Coverage (MC/DC) [119] and Linear Code Sequence And Jump (LCSAJ) [120, 121] coverage are also recommended.

1. The MC/DC criterion is satisfied only when:

(a) Every point of entry and exit in the program has been invoked at least once.

(b) Every condition in a decision has taken all possible outcomes at least once.

(c) Every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect the outcomes of that decision. A condition is shown to independently affect the
2. Related work

outcomes of a decision by varying just that condition while holding all other possible conditions fixed [119].

2. LCSAJ (aka. jump-to-jump path/JJ-path) coverage criterion is satisfied when all the LCSAJs are executed at least once. LCSAJ is a linear sequence consisting of three linear jumps/points [120]: (i) the start point, (ii) the end point, and (iii) the jump-to point, which marks the end of the linear sequence/flow.

Achieving 100% MC/DC and LCSAJ criteria often requires large number of test cases, for which automatic test case generation may be used. Random testing [122, 123] and model based testing [124] are the two popular techniques to generate test cases automatically.

Random testing though is the quickest and easiest test case generation technique, it generates redundant test cases; and may not satisfy specific requirements. As the main goal of testing is to generate a test case which has maximum probability of finding an error; techniques involving Adaptive random testing (ART) [125], directed random testing [126, 127], and genetic algorithms [128] have been proposed [129, 130]. ART attempts to spread inputs evenly over the input domain using distance calculations, where as directed random testing combines symbolic execution and test coverage information of the current input (test case) to generate the next input (test case). On the other hand, genetic algorithm is a search technique which uses an initial set of random test cases as the initial population, and mimics natural evolution by producing better test-cases based on a fitness function.

Model based test case generation requires building a model of the system, and a test case generation criteria; using which, test cases are generated for the actual system. The main advantage of this approach is that it forces the designers to create a precise behavior of the system at the requirement stage itself, thus ensuring quality at the early stages of development. After a model has been validated, automatic code generators may be used to generate the implementation code [86].

As large number of test cases may require a lot of time to execute (especially during regression testing), varieties of ways to reduce test cases and to prioritize them have
been proposed [131–134]. Some studies [135, 136] indicate that test case reduction by keeping the test coverage constant does not have significant effect on the effectiveness of the test suite. A systematic survey on test case minimization and prioritization may be found in [137].

Another interesting testing technique is fuzz-testing [138, 139]. Fuzz-testing involves giving random and malformed/invalid inputs to the program to analyze its behavior. The technique is usually automated; and is effective in detecting security faults, crashes (including assertion failures), and memory leaks.

**Critical review of software testing and test coverage**

Testing is an important part of V&V of a system, and any safety related software must be rigorously tested. However, exhaustive software testing in real world applications is usually impractical. Also, the number of execution paths a program may take is exponential to the number of conditions (branches), and can be infinite if the program contains loops. Hence, it is also impractical to test all paths in large and complex applications. Thus, as Dijkstra quotes [140]: “program testing can be a very effective way to show the presence of faults, but is hopelessly inadequate for showing their absence”.

The MC/DC and LCSAJ coverage are very effective coverage metrics and are used in various safety-critical applications. Unfortunately, generating 100% LCSAJ can be difficult for large programs. Hence, techniques such as genetic algorithms and model based testing are used for generating large number of test cases. However, genetic algorithms have two issues: (i) how to generate the initial population/test-cases? (ii) how to choose two parents to generate new test cases? Also, large number of test cases cannot be verified manually, hence use of automatic test oracles is a must [141, 142]. However, it is challenging to build a true test oracle.

Control coverage of the code is popularly used to quantify the amount of testing carried out. However, single control coverage criteria alone such as 100% MC/DC could be misleading in certain situations (Figures 2.1 to 2.3 on pages 24–25), and may not be sufficient to ensure the test adequacy in safety-critical software [143, 144].

Fuzz testing is an effective testing technique to detect memory leaks, buffer
overflows, null pointer dereference, uncontrolled format string issues, denial of service, assertion failures, out of memory faults, etc. Traditionally, fuzz testing has depended on random number generation for generating inputs; but, combining fuzzing and symbolic execution has also been reported to be very effective and scaleable for production use \cite{145,146}. However, fuzz testing is not a QSRM, and cannot be directly used to quantify software reliability.

2.5 In mutation testing and test adequacy

Mutation testing \cite{147,148} is a fault injection technique, where realistic faults are induced intentionally into the source code. The fault induced program is known as a mutant (Figure 2.4 on page 25), and the result of mutation testing is the mutation score, defined as:

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

(2.1)

where: \(K\) is the number of mutants killed by the test cases (i.e. at least one of the test cases has failed while executing the mutant), \(G\) is the number of mutants generated and \(E\) is the number of equivalent mutants. The value of mutation score is in range \([0,1]\); and it indicates effectiveness of the test cases to catch faults (higher the mutation score, higher is the effectiveness); and is an indication of test adequacy. Ideally, a good set of test cases must have a mutation score = 1 (i.e. should be able to detect/kill all the mutants).

Critical review of mutation testing and test adequacy

Mutation testing is one of the most effective techniques to determine the test adequacy. But is considered difficult in practice, as it is computationally expensive and suffers from the equivalent mutants problem. Systematic reviews on mutation testing, the equivalent mutant problem, and test adequacy may be found in \cite{38}.

While calculating the result of mutation testing (i.e. the mutant score — Equation (2.1)), if few of the mutants could not be killed (i.e. \(K < G\)), then: unless the equivalent mutants are detected, the value of \(E\) is assumed to be 0. Thus, the
2. Related work

```c
bool function ( bool a, bool b, bool c, bool d, bool e, bool f )
{
    return ( a && ( b || c ) && ( d || e || f ) );
}
```
(a)

```c
bool function ( bool a, bool b, bool c, bool d, bool e, bool f )
{
    return ( ( d || e || f ) && ( b || c ) && a ) ;
}
```
(b)

**Figure 2.1:** An example of two functionally same programs having difference in MC/DC (calculated through code instrumentation), due to short-circuit evaluation by the compiler. For a given set of test cases: function (a) is likely to have lower MC/DC than function (b).

```c
bool function ( int a, bool b, bool c, bool d, bool e, bool f )
{
    if ( a == 100 )
    {
        if ( b || c )
            // statement 1
        if ( d || e || f )
            // statement 2
    }
}
```
(a)

```c
bool function ( int a, bool b, bool c, bool d, bool e, bool f )
{
    bool a_is_equal_to_100 = a == 100 ;
    bool b_or_c = b || c ;
    bool d_or_e_or_f = d || e || f ;
    if ( a_is_equal_to_100 )
    {
        if ( b_or_c )
            // statement 1
    }
    if ( d_or_e_or_f )
        // statement 2
}
```
(b)

**Figure 2.2:** An example of two functionally same programs having difference in MC/DC by manipulating the way conditions are written. For a given set of test cases: function (a) is likely to have a lower MC/DC than function (b).
2. Related work

```c
bool function ( bool true_condition )
{
    if ( true_condition )
    {
        // 1 statement
    }
    else
    {
        // 100 statements
    }
}
```

**Figure 2.3:** An example where MC/DC and LCSAJ coverage (50%) is greater than the statement coverage (≈ 1%).

```c
bool can_the_car_start (bool door_is_closed, bool seat_belt_is_on)
{
    if ( door_is_closed && seat_belt_is_on )
        return true ;
    else
        return false ;
}
```

(a)

```c
bool can_the_car_start (bool door_is_closed, bool seat_belt_is_on)
{
    if ( door_is_closed || seat_belt_is_on )
        return true ;
    else
        return false ;
}
```

(b)

**Figure 2.4:** An example of mutant program: (a) the original program, (b) the mutant program (the induced fault is indicated by the red color).
mutation score will always be < 1. Automatic detection of equivalent mutant is in general considered as an undecidable problem [149]; nevertheless, many attempts have been made [150–153] to detect them with certain accuracy. Also, results of mutation testing could be misleading if faults are not induced at all paths of the code.

Not much work is available on mutant characteristics, i.e. how do unkilled mutant programs (a mutant program which when tested, gave the same results as the original program) differ from the killed mutant programs (a mutant program which when tested, gave at least one result different from the original program). Work such as [154] suggests that the mutants with high coverage impact are likely to be non-equivalent, and are likely to be killed easily.

2.6 In software reliability growth models (SRGM)

SRGMs are statistical techniques to estimate the reliability of a given system using the past software failure data trend. Every time the software fails, it is corrected, and the software experiences reliability growth. Thus the reliability is expected to grow as the software matures. The failure data is expected to be accurate and correct; also, each time the software fails it is corrected without inducing new faults.

A variety of SRGMs have been proposed and applied to various projects [17, 155–159]. However, lot of assumptions and limitations has also been reported [19–21, 40].

Critical review

SRGMs are black box techniques which can be used without understanding the design or code of the software under test. It is particularly useful for large projects where understanding the design or code is difficult, or the full design or source of the software is not available. The main advantage of SRGMs is its ease of use. Once the failure data is available, an appropriate model is selected and the failure trend can be easily plotted, using which the reliability can be assessed or predicted.

However, as with any black box technique, the software testing methodology is ad hoc in nature, and may not be sufficient to test safety-critical software. Also, choosing a
useful model for a given situation/software is a complex task [23,24].

2.7 In Bayesian belief network

Bayesian Belief Network (BBN) defined as [160]: "A directed acyclic graphs (DAGs) in which the nodes represent variables of interest and the links represent informational or causal dependencies among the variables", is considered one of the potential technique to estimate software reliability [40,161,162] for safety-critical systems.

Building an useful BBN requires a group of experts and information from various sources of reliability evidence such as: design documents, expert knowledge, operating experience, testing, etc. [40,163]. Use of BBNs in safety software in nuclear industry has been highlighted in [164,165].

Critical review

BBN allows estimation of software reliability using existing knowledge, and displays the relationship between variables in a graphical form. The two main advantages of BBN are: (i) use of various kinds and sources of information to get a reliability estimate, (ii) allows uncertainties in parameters to be taken in to account. However, the main challenge in creating an effective BBN include : (i) collecting enough and accurate data for newly built products, (ii) qualifying experts for BBN development, (iii) resolving disagreements among experts.

2.8 In architecture based approaches

As more and more functionality is being added in to the software; the present software systems are growing large and complex. Hence, software reusability and component based software engineering is emphasized to reduce cost and V&V effort. Hence, for large and complex systems, black box based software reliability estimation techniques may not be appropriate. "Instead, there is a need for a white-box approach which estimates system reliability taking into account the information about the architecture of the software
made out of components" [166].

In architecture based models, clear knowledge of the structure of the software and/or past experience must be available to model the reliability of each software component and its interactions with other components. Architecture based approaches require an expert with thorough knowledge in the software architecture. Architecture based approaches can be divided into path based and state based approaches [167].

Path based approach is one of the architecture based approaches, and involves generating/identifying paths in the software and testing/simulating the paths to estimate the software reliability by averaging all the path reliabilities [168]. On the other hand, Markov models [169-173] consist of system states, possible transitions between them, and its associated probabilities. The model calculates the system reliability using transition probability matrix. The characteristic of markov model is: the future behavior of the system is only dependent on the current state.

Critical review

White-box based software reliability modeling techniques such as architecture based models allow analysis of software reliability at early stages of software development life cycle. Two major limitations of path based approaches are: (i) difficulty in detecting unfeasible paths, (ii) the number of paths a program may have increases exponentially with number of conditions, and can be infinite if a program contains loops.

Markov analysis is a very useful technique in modelling time dependent failures, and describes the failure of an item and its subsequent repair. Also, it shows the probability of an event resulting from a sequence of sub-events. However, markov models are usually difficult to construct for large and complex systems and suffers from state-space explosion problem.

A systematic review of architecture based models can be found in [167, 174].
<table>
<thead>
<tr>
<th>#</th>
<th>Technique</th>
<th>Major advantages</th>
<th>Gaps / difficulties / disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Formal methods</td>
<td>i) Is rigorous and systematic in nature.</td>
<td>(i) Is labor intensive, difficult in practice for large projects.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ii) Focuses on correctness at the early stages of software development.</td>
<td>(ii) Proof/Specification may also contain faults/errors.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>iii) Supports automatic code generation.</td>
<td>(iii) Generally, does not consider the factors associated with the target compiler/hardware/environment.</td>
</tr>
<tr>
<td>2</td>
<td>Verification and validation</td>
<td>(i) Can be performed by an independent agency.</td>
<td>(i) Process is usually manual.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(ii) Focuses on functional correctness.</td>
<td>(ii) Results are usually check-list based and qualitative.</td>
</tr>
<tr>
<td>3</td>
<td>Classical software testing</td>
<td>(i) Results reflect the real environment.</td>
<td>(i) Exhaustive testing is impractical.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(ii) Amount of testing is quantifiable through test coverage.</td>
<td>(ii) Cannot prove absence of faults.</td>
</tr>
<tr>
<td>4</td>
<td>Mutation based testing</td>
<td>(i) An effective method to assess the quality of test cases.</td>
<td>(i) Is computationally expensive.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(ii) Its result (the mutation score) is an indication of test adequacy.</td>
<td>(ii) Suffers from the equivalent mutants problem.</td>
</tr>
<tr>
<td>5</td>
<td>Model checking</td>
<td>(i) Exhaustively searches for all nodes and transitions.</td>
<td>(i) Is computationally expensive.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(ii) Automatic test cases can be generated.</td>
<td>(ii) Requires model to be represented in the form of a state diagram.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(iii) Can generate counter examples for failed properties.</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Summary of the related work - I
<table>
<thead>
<tr>
<th>#</th>
<th>Technique</th>
<th>Major advantages</th>
<th>Gaps / difficulties / disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Fuzz testing</td>
<td>(i) Effective in detecting security/safety related faults. (ii) Attempts to detect faults/crashes which are often difficult in manual testing.</td>
<td>(i) Relies heavily on random numbers.</td>
</tr>
<tr>
<td>7</td>
<td>Reliability growth models</td>
<td>(i) Is black-box based approach, and is independent of the source code/architecture of the system. (ii) Gives quick assessment of reliability.</td>
<td>(i) Requires enough and accurate failure data. (ii) Based on assumptions which may not be acceptable for critical software.</td>
</tr>
<tr>
<td>8</td>
<td>Markov models</td>
<td>(i) Clearly describes both the failure of an item and its subsequent repair. (ii) Can handle probability of an event resulting from a sequence of sub-events</td>
<td>(i) Practical limitation due to state space explosion.</td>
</tr>
<tr>
<td>9</td>
<td>Bayesian belief networks (BBN)</td>
<td>(i) Allows combining different kinds/sources of data. (ii) Allows uncertainties in parameters to be taken in to account.</td>
<td>(i) Requires expert BBN developers. (ii) Qualification of experts could be an issue. (iii) Difficulty in collecting enough and accurate data for new products.</td>
</tr>
<tr>
<td>10</td>
<td>Architecture based models</td>
<td>(i) Based on through analysis of the software architecture.</td>
<td>(i) Requires expert/experienced personnel. (ii) Generally, does not consider the factors associated with the target compiler/hardware/environment.</td>
</tr>
</tbody>
</table>

Table 2.2: Summary of the related work - II
2.9 Summary

Literature review revealed some of the limitations in existing methods, and also the difficulties in using them to estimate software reliability (*Tables 2.1 to 2.2 on pages 29–30*). The main gaps or limitations observed in existing methods are:

1. Results of existing Verification & Validation (V&V) techniques are qualitative in nature, and are difficult to be integrated with the Probabilistic Safety Assessment (PSA) of a safety-critical system.

2. Difficulty in practical implementation of formal methods for large and complex applications.

3. Difficulty in practical implementation of model checking techniques due to the state space explosion problem.

4. Some of the software test coverage criteria were found to be misleading in certain situations.

5. The equivalent mutants problem limits the use of mutation testing in practice.

6. Results obtained through software reliability estimation techniques, which are based on the historical data or expert judgment/opinion may not be accurate for new products.

7. As software systems grow large and complex, reusability becomes an important factor. Hence, for large and complex systems, black box based software reliability estimation techniques may not be appropriate.
Background information

This chapter provides a brief background about instrumentation and control systems in nuclear reactors and the case studies used in the present study.

3.1 Instrumentation and control in nuclear reactors

Nuclear Power Plants (NPPs) are power stations which use fissile material such as Uranium-235 or Plutonium-239 as its fuel (Figure 3.1). NPPs use the heat produced during the fission reaction to generate electricity. Nuclear reactors may be divided into
3. Background information

Figure 3.2: A typical sodium-cooled, pool-type fast reactor

thermal reactors and fast reactors. Thermal reactors employ slow moving neutrons for the fission reaction, whereas fast reactors use fast moving neutrons. An example of a fast reactor is the Prototype Fast Breeder Reactor (PFBR) [175], which is a 500MWe sodium cooled fast breeder reactor. The Figure 3.2 shows the schematic of a typical sodium-cooled fast reactor.

To ensure the smooth functioning of the plant during: reactor start-up, operation, fuel handling, shutdown, and maintenance; a lot of hardware and software based systems are used to monitor and control various plant parameters. These instrumentation and control systems are safety systems running on real-time computers, with fault tolerant features such as: redundant power supplies, redundant network connections, switch-over logics, etc. [176].

Also, safety-critical systems usually use Triple Modular Redundancy (TMR) architecture, where as safety related systems use dual hot standby architecture [176].
3. Background information

3.2 Case studies used in the present study

Six systems, which are representative of safety systems in nuclear reactors, are used as case studies in the present study. Below is the brief description of each system:

3.2.1 Fresh subassembly handling system

In nuclear reactors, fuel is replenished at approximately once every year. The spent fuel sub-assemblies are replaced with fresh fuel sub-assemblies during the refuelling campaign of the reactor. A fresh fuel sub-assembly is received at the Fresh Sub-assembly Receiving Facility (FSRF), and after initial inspections, it is sent to the Fresh Sub-assembly Preheating Facility (FSPF) through the Fresh Sub-assembly Entry Port (FSEP) gate. After pre-heating, the fresh fuel sub-assembly is sent to the reactor core using the Inclined fuel transfer machine (IFTM) and Transfer arm (TA) (Figure 3.3).

The main purpose of the Fresh Sub-assembly Handling System (FSHS) [177] software is to collect necessary plant information, generate interlocks, and to automate the process of fresh fuel handling.
3.2.2 Reactor start-up system

To smoothly start a reactor from reactor shutdown to reactor in operation state, several conditions have to be satisfied. Reactor Startup System (RSU) [178] (Figure 3.4) checks all these conditions and gives authorization for starting up the reactor. To check the list of conditions, the RSU scans hard wired inputs from different plant systems and process computer (which stores soft inputs given by other systems).

![Logic diagram of the reactor startup system](image)

Figure 3.4: Logic diagram of the reactor startup system ($c_i$ is one of the condition to be satisfied for the reactor startup, and $s_{ij}$ is the $j^{th}$ sub-condition of $c_i$)

The RSU software scans all the conditions to be satisfied for the reactor startup ($c_1 - c_n$ in Figure 3.4), and sends alarms for conditions which could not be satisfied. Also, while reactor startup if proper authorization is given, few conditions can be inhibited; for which RSU software sends respective alarms to the operator.

3.2.3 Steam generator tube leak detection system

As fast breeder reactors use liquid sodium as its coolant, a leak in the steam generator tubes (Figure 3.5 on the next page) causes violent sodium-water reaction, followed by hydrogen release. Hence, the Steam Generator Tube Leak Detection system (SGTLD) [179] is provided to detect leaks, send alarm signals to the operator, and to isolate steam generators to prevent further leaks and reaction. The SGTLD software also indicates the leaks as: small, medium, or large; and takes the appropriate safety action.
3.2.4 Core temperature monitoring system

The software based CTMS [180] (Figure 3.6 on the next page) continuously keeps track of nuclear reactors’ core temperature through thermocouples. The main purpose of the system is to detect anomalies such as plugging of fuel sub-assemblies, error in core loading and uncontrolled withdrawal of control and safety rods. The software scans reactor core inlet temperatures and sub-assembly outlet temperatures periodically; validates the inputs and calculates various parameters required for generating alarms and Safety Control Rod Axe Man (SCRAM) (emergency reactor shutdown) signals.

These signals are generated when the computed parameters cross their respective threshold limits. The alarms generated are sent to the control room for the operator, and the SCRAM signal is sent to Control and Safety Rod Drive Mechanism (CSRDM)
and/or Diversified Safety Rod Drive Mechanism (DSRDM) to drop the control rods into the reactor core, to stop the fission reaction. CTMS is classified as safety-critical system, and has two main failure modes: (i) failure to initiate SCRAM signal when parameters exceed their threshold value; which places demand on the hardware based CTMS and other diversified shutdown systems, (ii) generation of spurious SCRAM signals; which affects the plant availability.

### 3.2.5 Radioactive gaseous effluent system

Radioactive gas effluents are collected from various sources of the reactor, and are stored in delay tanks. After certain delay, depending upon the radioactivity level of the effluent, it is discharged to the environment after filtering through the stack (Figure 3.7 on the next page).

Radioactive Gaseous Effluent System (GES) [181] software processes the system signals and produces the required control and alarm signals for achieving safe radioactive effluent handling. The control actions mainly include start/stop of compressors and open/close of valves.
Figure 3.7: The schematic of Radioactive Gaseous Effluent System (GES) (Here the symbol △ indicates a pneumatic valve, NRV indicates a non-return valve, FM indicates the flow meter, C₁ and C₂ are the compressors, and the items controlled by the software are indicated by the blue color)
3.2.6 Safety grade decay heat removal system

After reactor shutdown, the heat produced by the fuel due to radioactive decay is called the decay heat. The Safety Grade Decay Heat Removal system (SGDHR) \(^{182}\) is used to remove the decay heat from the reactor core (Figure 3.8). To ensure sufficient cooling, a reactor may have more than one independent and identical SGDHR.

The instrumentation and control (I&C) system of the SGDHR monitors/controls sodium temperature, sodium flow, sodium level, sodium leak, argon pressure, air pressure, and valve positions signals. The control actions of the system include: open/close of valves, heater control, blower control, pump trip control, etc. Also, the system generates appropriate alarms.

![Figure 3.8: Schematic of one of the four independent and identical loops of safety grade decay heat removal system](image)

Figure 3.8: Schematic of one of the four independent and identical loops of safety grade decay heat removal system