

**METHODOLOGY AND ESTIMATION OF SOFTWARE
RELIABILITY FOR SAFETY SYSTEMS**

By

D.Thirugnana Murthy

Enrollment Number: ENGG 02200804015

Indira Gandhi Centre for Atomic Research, Kalpakkam, India

A thesis submitted to the Board of studies in Engineering Sciences

In partial fulfillment of requirements
For the Degree of

**DOCTOR OF PHILOSOPHY
of
HOMI BHABHA NATIONAL INSTITUTE
MUMBAI, INDIA.**

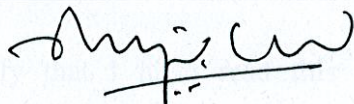


Dec 2013

Homi Bhabha National Institute

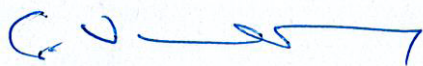
Recommendations of the Viva Voce Board

As members of the viva voce board, we certify that we have read the dissertation prepared by D.Thirugnana Murthy entitled "Methodology and Estimation of software reliability for safety systems" and recommend that it may be accepted as fulfilling the dissertation requirement for the degree of doctor of philosophy.



Date: 28-3-2014

Chairman - **Dr. T. Jayakumar**



Date: 28/3/14

Guide / Convenor- **Dr. K. Velusamy**



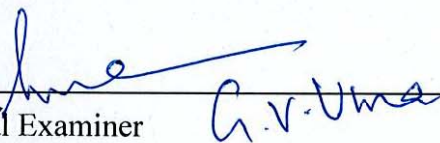
Date: 28/3/2014

Member 1 - **Dr. A.K. Bhaduri**



Date: 28.3.14

Member 2 - **Dr. M. Saibaba**



Date: 28/3/14

External Examiner

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to HBNI.

Date: 28.03.2014

Place: Indira Gandhi Centre for Atomic Research (IGCAR)
Kalpakkam

STATEMENT BY AUTHOR

CERTIFICATE

I hereby certify that I have read this dissertation prepared under my direction and recommend that it may be accepted as fulfilling the dissertation requirement.



**Dr.K.Velusamy
(Guide)**

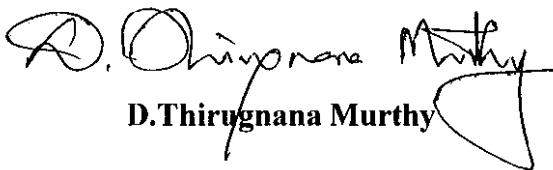
Date: 28/3/14

Place: Indira Gandhi Centre for Atomic Research (IGCAR)
Kalpakkam

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at Homi Bhabha National Institute (HBNI) and is deposited in the Library to be made available to borrowers under rules of the HBNI.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgement of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the Competent Authority of HBNI when in his or her judgment the proposed use of the material is in the interest of scholarship. In all other instances, however, permission must be obtained from the author.

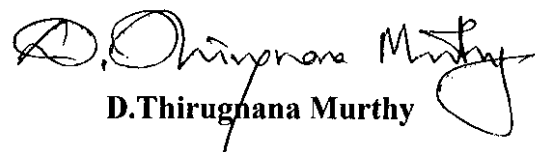

D. Thirugnana Murthy

Date : 24/12/2013

Place : Indira Gandhi Centre for Atomic Research (IGCAR)
Kalpakkam

DECLARATION

I, hereby declare that the investigation presented in the thesis has been carried out by me.
The work is original and has not been submitted earlier as a whole or in part for a degree /
diploma at this or any other Institution / University.


D.Thirugana Murthy

Date : 24/12/2013

Place : Indira Gandhi Centre for Atomic Research (IGCAR)
Kalpakkam

DEDICATIONS

To

K. Doraiswamy

(Friend, Philosopher and Father)

&

B. Mangalakshmi

(Mother, Who had more confidence in me than myself)

ACKNOWLEDGEMENTS

My Sincere thanks

to all those people

I came across

CONTENTS

Title	Page Number
SYNOPSIS.....	V
LIST OF FIGURES.....	XII
LIST OF TABLES.....	XIII
CHAPTER 1. INTRODUCTION.....	1
1.1 Foreword	1
1.2. Motivation	6
1.3. Objectives and Scope of the Present Research Work	7
1.4. Organisation of the thesis.....	8
CHAPTER 2. SOFTWARE LIFE CYCLE MODEL FOR SAFETY SYSTEMS.....	10
2.1 Introduction	10
2.2 Available software life cycle models	10
2.2.1 Waterfall model	10
2.2.2 Spiral model.....	11
2.2.3. Iterative and incremental development.....	11
2.2.4. Agile development.....	12
2.2.5. Code and fix.....	12
2.3. Model suggested for NPP.....	12
CHAPTER 3. ISSUES AND IMPORTANCE OF SOFTWARE TESTING WITH RESPECT TO RELIABILITY	17
3.1 Introduction	17
3.2 Software Testing	18
3.2.1 White Box Testing.....	18
3.2.2 The Nature of Software Defects	18
3.2.3 Basis Path Testing.....	19
3.2.4 Flow Graphs.....	19
3.2.5 The Basis Set	19

3.3 Deriving Test Cases	20
3.4 Loop Testing	20
3.5. Other White Box Techniques.....	21
3.6. Black Box Testing.....	21
3.7 Equivalence Partitioning	22
3.8 Boundary Value Analysis (BVA)	23
3.9 Mutation testing	23
3.10 Fault injection testing.....	25
3.11 Testing of Safety systems of Fast reactor	27
3.12 Inferences	33
CHAPTER 4. EVALUATION OF SOFTWARE METRICS AND TOOL	35
4.1 Evaluation of Software Metrics	35
4.1.1 Cyclomatic Complexity (CC)	35
4.1.2 Nesting Level.....	35
4.1.3 Comment to Code ratio.....	36
4.1.4 Ternary Operator.....	36
4.1.5 Dynamic Memory	36
4.1.6 Goto and Continue Statements	36
4.1.7 Number of code lines in a function.....	37
4.1.8 Recursive functions.....	37
4.1.9 Unused functions and Variables	37
4.2 Software Quality Metrics IEEE-1061 standard	37
4.3 Thirty Software Engineering Measures	39
4.3 Static Analyser Tool.....	40
4.4 Inferences	43
CHAPTER 5. VERIFICATION AND VALIDATION PROCEDURE.....	46

5.1 Introduction	46
5.2. Verification procedure for Custom-Built Systems (CBS)	47
5.2.1 Documents submitted	47
5.2.2 Procedure for Verification	48
5.2.3 System Requirements Specification (SyRS) Review	49
5.2.4 System Architecture Review (SAR)	50
5.2.5 Software Requirement Specification (SRS) Review	51
5.2.6 Software Design Description (SDD) Review	51
5.2.7 Software Implementation Review	51
5.2.8 System Integration Review	52
5.2.9 System Validation Review.....	53
5.3 Inferences	53
CHAPTER 6. SOFTWARE RELIABILITY MODELING.....	56
6.1 Introduction	56
6.2 Types and Approaches	63
6.3 Static Model	65
6.3.1 Phase-based Model: Gaffney and Davis	66
6.3.2 Predictive Development Life Cycle Model: Dalal and Ho	66
6.4 Dynamic Models: Reliability growth models for testing and operational use.....	67
6.4.1 A General Class of Models	67
6.4.2 Assumptions Underlying the Reliability Growth Models	67
6.5 Software Reliability Growth Modeling.....	68
6.5.1 A Generalized Non-homogeneous Poisson Process Model.....	68
6.6 A Reliability Model with Considerations of Random Field Environments	69
6.7 Precautions in Using Reliability Growth Models	70
6.8 Reliability Growth Modeling with Covariates	71
6.9 Time to Stop Software Testing	71

6.10 Inferences	72
CHAPTER 7. ESTIMATION OF SOFTWARE RELIABILITY.....	75
7.1 Introduction	75
7.2 Reliability Estimation.....	76
7.3 Typical Estimation for Safety Critical System.....	83
7.4 Inferences	88
CHAPTER 8. CONCLUSIONS AND FUTURE DIRECTIONS	89
8.1 Conclusions	89
8.2. Scope For Future Work	90
APPENDIX – I: Definitions and Abbreviations	91
APPENDIX – II: Checklist for System Requirement Review.....	95
APPENDIX – III: Checklist for System Architecture Review	101
APPENDIX – IV: Check list for Software Requirement Specification Review	106
APPENDIX – V: Checklist for Detailed Design Verification.....	111
APPENDIX – VI: Checklist for System Integration Verification.....	114
APPENDIX – VII: Checklist for System Validation Review	115
PUBLICATIONS BASED ON THE THESIS.....	117

SYNOPSIS

I. INTRODUCTION

Software design, development and testing have become very intricate with the advent of modern highly distributed systems, networks, middleware and interdependent applications. The demand for complex software systems has increased more rapidly than the ability to design, implement, test, and maintain them, and hence the reliability of software systems has become a major concern. Today software is being deployed in safety applications due to the advancement of technology. In nuclear power plants (NPP), many systems are being used in safety critical and safety related applications, which demand a very high reliability [1]. As software becomes an increasingly important part of many different types of systems that perform complex and critical functions in many applications, such as defense, nuclear reactors, etc., the risk and impacts of software-caused failures have increased. There is now a general agreement on the need to increase software reliability by eliminating errors made during software development and maintenance.

Software is a collection of instructions or statements in a computer language. It is also called a computer program, or simply a program. A software program is designed to perform a set of specified functions. Upon execution of a program, an input state is translated into an output state. An input state can be defined as a combination of input variables or a typical transaction to the program. When the actual output deviates from the expected output, a failure occurs. It is estimated that 60-90% of current computer errors is from software faults [2].

Software reliability is defined as the probability of failure-free software operations in a specified environment [3]. The software reliability field discusses ways of quantifying it and using it for improvement and control of the software development process. A number of standards have emerged in the area of developing reliable software consistently and efficiently [4]. The Software Engineering Institute has established a standard called the software Capability Maturity Model (CMM) that scores organizations on multiple criteria and gives a numeric grade from one to five.

The Software faults are most often caused by the requirement and design faults. The requirement fault can be incomplete requirement or interpreted in different or wrong way. The incomplete / missing of requirement may be covered under “Adequacy” check. An ambiguous statement may lead to wrong interpretation. It generally happens because of user’s “implicit specification”, i.e., user assumes it is obvious. Design faults occur when a designer either misunderstands a specification or simply makes a mistake. Software faults are common for the simple reason that the complexity in modern systems is often pushed into the software part of the system. Software reliability is operationally measured by the number of field failures, or failures seen in development, along with a variety of ancillary information. The ancillary information includes the time at which the failure was found, part of the software where it was found, the state of software at that time, the nature of the failure, time of deployment, *etc.* Most of the software quality improvement efforts are triggered by lack of software reliability. Thus, software managers recognize the need for systematic approaches to measure and assure software reliability, and devote a major share of project development resources to this. Formally, software reliability engineering is the field that quantifies the operational behavior of software based systems with respect to user

requirements with bearing on reliability. It includes data collection on reliability, statistical estimation, metrics and attributes of product architecture, design, software development, time of deployment and the operational environment. Besides its use for operational decisions like deployment, it includes guiding software architecture, design, development and testing. Most of the testing process is driven by software reliability concerns, and applications of software reliability models are to improve effectiveness of testing [5].

II. MOTIVATION

From the important software disasters, it is clear that software errors cost the country economy in rework, lost productivity and actual damages. Faulty software can also be expensive, embarrassing, destructive and deadly. It is well recognized that assessing the reliability of software applications is a major issue in reliability engineering [6]. Prediction of software reliability is highly involved. Perhaps the major difficulty is that we are concerned primarily with design faults, which is a very different situation from that tackled by conventional hardware theory. The input values to the software modules (functions) either internally or externally may be considered as arriving to the software randomly. So although software failure may not be generated stochastically, it may be detected in such a manner. Therefore, we can use stochastic models of the underlying random process that governs the software failure [7]. Hence, for safety critical software / highly dependable software systems the estimation of software reliability becomes prime important to evaluate the overall system reliability with the data available on hardware reliability. The hardware and software reliability together becomes more meaningful and useful for predicating the system performance and availability [8].

III. OBJECTIVES AND SCOPE OF THE PRESENT WORK

The main objective of the research is the estimation of software reliability for safety systems of Nuclear Power Plants. In case of NPP, the safety is of main concern and systems use mainly “Structured Programming” for safety systems instead of Object oriented or Commercially off the shelf (COTS) components [9].

In the process of assessment of reliability, the following sub-objectives are envisaged since software reliability is more concerned with design, methodologies, practices and tools used in the process of software development.

The sub-objectives are

(a) Development of software life cycle model for safety systems.

Assessing various life cycle models and the identification of a suitable software life cycle model for safety systems, which ensures highly reliable software delivery is one of the tasks of the current research.

(b) Determination of software metrics and development of software metric tool.

The identification of the software metrics that affect the reliability in terms of quality attribute and the development of tool to evaluate the metrics from the software code is very important.

(c) Development of Software Verification and Validation (V&V) methodology.

To evolve a comprehensive methodology to be followed for V&V throughout the life cycle to ensure quality artifacts are generated. The standards to be practiced and the detailed audit procedures and the checklist for each stage of the development, the role of V&V members and their independence in terms of evaluation are explained. This methodology is for the software fused in custom built computer based systems (CBS). This V&V procedure is developed from IEEE standard [10], NPCIL procedures on C&I system [11] and V&V procedure [12] so that it is suited for nuclear power plants.

(d) Development of Software Reliability Model.

Software reliability measurement includes two types of models namely static and dynamic reliability estimation, used typically in the earlier and later stages of development respectively. A key use of the reliability models is in the area of when to stop testing. One purpose of reliability models is to perform reliability prediction at an early stage of software development. This activity determines future software reliability based upon available software metrics and measures. Particularly when field failure data are not available (*e.g.* software is in the design or coding stage), the metrics obtained from the software development process and the characteristics of the resulting product can be used to estimate the reliability of the software upon testing or delivery.

(e) Procedure for Estimation of Software Reliability.

As a part of the research, a comprehensive procedure to estimate software reliability for a given software program coded in “C” language has been established. Robustness and

validation of the methodology has been demonstrated by applying it to software deployed in safety systems of a fast reactor.

The above said sub-objectives form the basis for the estimation of software reliability for safety systems of NPP. The current work is specifically for the software development on Instrumentation & Control systems employed in NPP. Due to large human contribution in the software development, the reliability calculation will have significant uncertainty. The variations among individuals need to be properly accounted in software development in particular during code development. The unreliability attached with Human Error Probability is assumed to be minimal which is basically the boundary within which the exercise has been carried out.

REFERENCES

1. Design safety guide on computer based systems, Atomic Energy Regulatory Board AERB/SG/D10 - Design Safety Guide on Safety Critical Systems, 2002.
2. Watts S. Humphrey, Managing the software process, SEI series in software engineering, Addison Wesley Longman Inc, ISBN-981-235-916-8, 1999.
3. Sommerville, I., Software Engineering, Pearson Education, 6th Ed., 2001.
4. IEEE Standards Software Engineering, 1999 Edition volume one – four, ISBN-0-7381-1559-2, ISBN-0-7381-1560-6, ISBN-0-7381-1561-4, ISBN-0-7381-1562-2.
5. IAEA-TECDOC-1335 Configuration Management in Nuclear Power Plants, Jan 2003, International Atomic Energy Agencies, Vienna, ISBN 92-0-100503-2.
6. Software reliability and safety in Nuclear reactor protection systems by J. Dennis Lawrence for U.S Nuclear Regulatory Commission. UCRL-ID-114839.
7. T.T.Soong, Fundamentals of Probability and Statistics for Engineers, State University of New York, USA, John Wiley & Sons Ltd, ISBN –0-470-86913-7, 2004.
8. Design safety guide on computer based systems, Atomic Energy Regulatory Board AERB/NPP – PHWR/SG/D20 - Safety Related Instrumentation & Control for pressurized heavy water reactor based nuclear power plants, 2003.
9. Design safety guide on computer based systems, Atomic Energy Regulatory Board AERB/SG/D25 - Computer Based systems of Pressurized Heavy water Reactor, 2001.
10. IEEE Std 1012-1998 - IEEE Standard for Software Verification and Validation, 1998.
11. NPCIL / ED-PROC - Engineering Procedure for Computer Based C&I Systems, 2003.
12. NPCIL / IVVC Proc. Version 1.0- Procedure for Independent Verification and Validation of Computer Based C&I systems, 2003.

LIST OF FIGURES

Figure 1.1 SEI - Capability Maturity Model levels	5
Figure 2.1 Software life cycle “V” Model	13
Figure 4.1 Typical output screen of the SA with Cyclomatic Complexity	40
Figure 4.2 Screen output of the SA detailing Comment to Code ratio	41
Figure 4.3 Typical variation of failure intensity.	43
Figure 6.1 Classifications of Software Reliability Model.....	57
Figure 6.2 Intensity of Failure in Normal development and Deployment.....	60
Figure 6.3 Concatenated failure rate function for Jelinski - Moranda mod.....	65
Figure 6.4 Testing in software development cycle	69
Figure 6.5 Intensity of Failure in Reliability Growth model.....	70
Figure 6.6 Typical System Un-Availability	72

LIST OF TABLES

Table 2.1 Lifecycle in this Study versus Recommended Life-cycle in IEEE610.....	14
Table 3.1 List of I &C Systems.....	28
Table 3.2 Errors found during inspection of NPP software	29
Table 3.3 Sample Test Cases for I&C of Air Conditioning and Ventilation System ..	32
Table 4.1 Recommended Limit Values on code	42
Table 5.1 Software Criticality Levels.....	47
Table 5.2 Review Taskforce Committee	48
Table 7.1 Mapping of II factors to “Category of Levels”	79
Table 7.2 Template for Quality Parameters of Function.....	79
Table 7.3 List of Failures for functions	81
Table 7.4 Software Failure Calculation for function	84
Table 7.5 Estimated reliability based on the number of input combinations	86
Table 7.6 Software Version history of SCS – Part I.....	87
Table 7.7 Software Version history of SCS – Part II	88

CHAPTER 1. INTRODUCTION

1.1 Foreword

Software design, development and testing have become very intricate with the advent of modern highly distributed systems, networks, middleware and interdependent applications. The demand for complex software systems has increased more rapidly than the ability to design, implement, test and maintain them and hence, the reliability of software systems has become a major concern. Today software is being deployed in safety applications due to the advancement of technology. In a nuclear reactor, many systems are being used in safety applications, which demand high reliability [1.1]. As software becomes an increasingly important part of many systems that perform complex and critical functions, such as military defense, nuclear reactors, etc., the risk and impacts of software caused failures have increased dramatically. There is now general agreement on the need to increase software reliability by eliminating errors made during software development [1.2].

Software is a collection of instructions or statements in a computer language also known as a program, which is designed to perform a set of specified functions. Upon execution of a program, an input state is translated into an output state. An input state can be defined as a combination of input variables or a typical transaction to the program. When the actual output deviates from the expected output, a failure occurs. The definition of failure, however, differs from application to application and should be clearly defined in specifications. For instance, a response time of 30s is a serious failure for air traffic system, but acceptable for a railway ticket reservation system.

Within the last decade of the 20th century and the first few years of the 21st century, many reported system outages or machine crashes were traced back to computer software failures. For example, in the health industry, the Therac-25 radiation therapy machine was hit by software errors in its sophisticated control systems and claimed several patients' lives in 1985 and 1986 [1.2]. In the telecommunications industry, known for its high reliability, the nation wide network of a major carrier suffered with embarrassing network outage on 15 January 1990, due to a software problem [1.3]. In 1991, a series of local network outages occurred in a number of US-cities due to software problems in central office switches [1.2]. Software failures have impaired several high visibility programs in space, telecommunications, defense and health industries. The Mars Climate Orbiter crashed in 1999. The Mars Climate Orbiter Mission Failure Investigation Board concluded, the 'root cause' of the loss of the spacecraft, was the failed translation of English units into metric units in a segment of ground-based, navigation-related mission software [1.3]. Besides the loss of money, it delayed the space program by more than a year. Current versions of the Osprey aircraft, developed at a cost of billions of dollars are not deployed because of software-induced field failures. The costly "Y2K" problem resulted because of a design failure, a problem that occupied tens of thousands of programmers and costs running to tens of billions of dollars [1.3].

Software faults are most often caused by requirement and design faults that occur when a designer either misunderstands a specification or simply makes a mistake. Software faults are common for the simple reason that the complexity in modern systems is often pushed into the software part of the system. It is estimated that 60-90% of current computer errors is from software faults [1.4]. Software faults may also occur from hardware where these

faults are usually transitory in nature and can be masked using a combination of current software and hardware fault tolerance techniques.

The current assumption is that software cannot be made without bugs. This assumption may be true, but software does not have to be as traditional buggy as it is now. It is well recognized that assessing the reliability of software applications is a major issue in reliability engineering. Predicting software reliability is not easy. Perhaps the major difficulty is that we are concerned primarily with design faults, which is a very different situation from that tackled by conventional hardware theory. A fault (or bug) refers to a manifestation in the code of a mistake made by the programmer or designer with respect to the specification of the software. Activation of a fault by an input value leads to an incorrect output. Detection of such an event corresponds to an occurrence of a software failure. Input values may be considered as arriving to the software randomly. So, although software failure may not be generated stochastically, it may be detected in such a manner. Therefore, this justifies the use of stochastic models of the underlying random process that governs the software failures.

Software reliability is defined as the probability of failure-free software operations in a specified environment. The software reliability field discusses ways of quantifying it, using it for improvement and control of the software development process. Software reliability is operationally measured by the number of field failures, or failures seen in development, along with a variety of ancillary information. The ancillary information includes the time at which the failure was found, the part of the software where it was found, the state of software at that time, the nature of the failure, etc. Most quality

improvement efforts are triggered by lack of software reliability. Thus, software companies recognize the need for systematic approaches to measure and assure software reliability, and devote a major share of project development resources to this. A number of standards have emerged in the area of developing reliable software consistently and efficiently. The Software Engineering Institute (SEI) has proposed an elaborate standard known as the software Capability Maturity Model (CMM) that scores software development organizations on multiple criteria and gives a numeric grade from one to five. The software reliability is being viewed more in terms of software quality, measurements and control. SEI has devised the CMM by adopting two parameters viz, level and key problem areas as shown in Figure 1.1 [1.5]. This methodology provides a step-by-step procedure to improve reliability from level 1 to level 5. SEI-CMM level certification is issued based on the process adopted in the organization.

The main limitation of software maturity model is that it does not relate to risk or reliability. The levels are arranged to a large variety of software developments and it is not tuned for specific software. It has the provision to accommodate all software organization in to the certification. On the other hand, for nuclear power plants, the safety is of main concern and systems use mainly the “Structured Programming” for safety systems instead of Object oriented or Commercially Off The Shelf (COTS) components [1.6]. So, this research work is aimed at developing a reliability model for the safety systems with software components deployed in Nuclear Power Plants.

Level	Key Problem Areas
Initial	Project Management Project Planning Configuration Management Quality Assurance
Repeatable	Training Technical Practices Process focus
Defined	Process Management Process Analysis Quantitative quality plans
Managed	Change in Technology Problem Analysis Problem Prevention
Optimizing	Automation

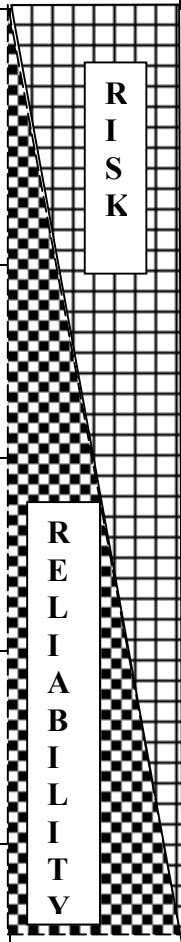


Figure 1.1 SEI - Capability Maturity Model levels

Software reliability engineering quantifies the operational behavior of software-based systems with respect to user requirements with bearing on reliability. It includes data collection on reliability, statistical estimation, metrics and attributes of product architecture, design, software development, and the operational environment. Besides its use for operational decisions like deployment, it includes guiding software architecture, design, development and testing. Testing process is driven by software reliability concerns and software reliability models to improve the effectiveness.

1.2. Motivation

From the software disasters, it is clear that software errors have a strong potential to cause serious damage to economy in terms of rework and productivity. It is well recognized that assessing the reliability of software applications is a major issue in reliability engineering [1.7]. Prediction of software reliability is highly involved. Perhaps the major difficulty is that we are concerned primarily with design faults, which is a very different situation from that tackled by conventional hardware theory. Input values may be considered as arriving to the software randomly. So although software failure may not be generated stochastically, it may be detected in such a manner. Therefore, we can use stochastic models of the underlying random process that governs the software failure [1.7].

Hence, for safety / highly dependable software systems the estimation of software reliability becomes prime important to evaluate the overall system reliability with the data available on hardware reliability. The hardware and software reliability together becomes more meaningful and useful for predicting the system performance and availability [1.8].

The University of Maryland (UMD) based its study on previous research carried out by Lawrence Livermore National Laboratory (LLNL) identified a pool of 78 software engineering measures. These measures related to software reliability and established a set of software engineering ranking criteria which is used to assess the metric's potential as software reliability indicator. This set was then reduced to 30 using importance considerations and these 30 software-engineering measures constitute the basis of the UMD study [1.9].

Four categories of models have been considered as “potential candidates” for modeling the reliability of software. The four categories include reliability growth models, input domain models, architectural models and early prediction models. The first category captures failure behavior during testing and extrapolates it to behavior during operation. Hence, this category of models uses failure data and trends observed in the failure data to derive reliability predictions. The second category of models uses properties of the input domain of the software to derive a correctness probability estimate from test cases that executed properly. The third category of model emphasizes on the architecture of the software and derives reliability estimates by combining estimates obtained for the different modules of the software. Finally, the fourth category of models uses characteristics of the software development process from requirements to test and extrapolates this information to behavior during operation.

1.3. Objectives and Scope of the Present Research Work

Further to the literature review and the motivation behind this research, the main objective of the research is the estimation of software reliability for safety systems of Nuclear Power Plants (NPP). In the process of assessment of reliability, the following sub-objectives are envisaged since software reliability is more concerned with design, methodologies, practices and the tools used in the process of software development.

The sub-objectives are

- (a) Development of software life cycle model for safety systems.
- (b) Determination of software metrics and development of software metric tool.
- (c) Development of Software Verification and Validation methodology.
- (d) Development of Software Reliability Model.

These form the basis for the Estimation of software reliability for safety systems of NPP. The present work is focussed towards software development on Instrumentation and Control systems employed in NPP. Around fifty software systems of I&C of fast reactor are taken for analysis. All these systems have been developed using “C” Programming language as part of the embedded systems to perform a specific task. The unreliability attached with Human Error Probability is assumed to be minimal which is basically the boundary within which the exercise has been carried out.

1.4. Organisation of the thesis

The thesis is arranged in eight chapters and they are

Chapter 1. Introduction

Chapter 2. Software life cycle model for safety systems

Chapter 3. Issues and Importance of Software Testing With Respect To Reliability

Chapter 4. Evaluation of Software Metrics and Tool Development

Chapter 5. Verification and Validation Methodology

Chapter 6. Software Reliability Modeling

Chapter 7. Estimation of Software Reliability

Chapter 8. Conclusions and Scope for Future Work

REFERENCES

- [1.1] P. Swaminathan, Design Aspects of Safety Critical Instrumentation of Nuclear Installations, International journal of Nuclear energy Science and Technology Vol.1, nos.2/3, p.254-263,2005
- [1.2] Design safety guide on computer based systems, Atomic Energy Regulatory Board AERB/NPP – PHWR/SG/D20 - Safety Related Instrumentation and Control for pressurized heavy water reactor based nuclear power plants, 2003.
- [1.3] www.devtopics.com/20-famous-software-disasters
- [1.4] Sommerville, I., Software Engineering, Pearson Education, 6th Ed., 2001.
- [1.5] Watts S. Humphery, Managing the software process, SEI series in software engineering, Addison wesley longman Inc, ISBN-981-235-916-8, 1999.
- [1.6] Design safety guide on computer based systems, Atomic Energy Regulatory Board AERB/SG/D25 - Computer Based systems of Pressurized Heavy water Reactor, 2001.
- [1.7] Soong.T.T., Fundamentals of Probability and Statistics for Engineers, John wiley and sons, ISBN –0-470-86913-7, 2004.
- [1.8] Hoang Pham, Handbook of Reliability Engineering, Springer – Verlag, London, ISBN-I-85233-453-3, 2003.
- [1.9] Software Engineering Measures for Predicting Software Reliability in Safety Critical Digital Systems.NUREG/GR-0019, UMD-RE-2000-23 University of Maryland, U.S. Nuclear Regulatory Commission, Washington, DC 20555-0001, 2000.

CHAPTER 2. SOFTWARE LIFE CYCLE MODEL FOR SAFETY SYSTEMS

2.1 Introduction

Software life cycle, is a structure imposed on the development of a software product. It is considered as a subset of systems development life cycle. There are several models for such processes, each describing the approach to a variety of tasks or activities that take place during the process. ISO/IEC 12207 is an international standard for software life-cycle processes. It aims to be the standard that defines all the tasks required for developing and maintaining software [2.1]. Assessing various life cycle models and the identification of a suitable software life cycle model for safety systems, which ensures high reliability. software delivery is one of the prime tasks of the current research.

2.2 Available software life cycle models

The Planning, implementation, testing, documenting, deployment and maintenance are the steps in the software life cycle development process. Several models exist to streamline the development process. Each one has its pros and cons, and it's up to the development team to adopt the most appropriate one for the project.

2.2.1 Waterfall model

The waterfall model depicts a process, where developers are to adopt the following phases in order:

1. Requirement Specification (Requirement Analysis)
2. Software Design
3. Integration

4. Testing (or Validation)
5. Deployment (or Installation)
6. Maintenance

In this model, after each phase is completed, it proceeds to the next stage. Reviews may occur before moving to the next phase, which allow for the possibility of changes. Waterfall discourages revisiting any prior phase once it is complete. This "inflexibility" is the main limitation of this model [2.2].

2.2.2 Spiral model

The main characteristic of a Spiral model is risk management at regular stages in the development cycle. Barry Boehm [2.3] proposed a formal software system development "spiral model", with emphasis on a key area of risk analysis that is neglected by other methodologies. The spiral model is visualized as a process passing through a number of iterations. The first stage is to formulate a plan, and then strive to find and remove all potential risks through careful analysis and, if necessary, by constructing a prototype. If some risks cannot be ruled out, the user has to decide whether to terminate the project or to ignore the risks.

2.2.3. Iterative and incremental development

Iterative development model prescribes the construction of initially small but ever-larger portions of a software project to help all those involved to uncover important issues early before problems or faulty assumptions can lead to disaster. Iterative processes can assist with revealing and refined definition of design goal [2.1].

2.2.4. Agile development

Agile software development uses iterative development as a basis but advocates a lighter and more people-centric viewpoint [2.4]. Agile processes use feedback as primary control mechanism. First, one writes automated tests, to provide concrete goals for development. The next step is coding by a pair of programmers, which is complete when all the tests are successfully passed. The incomplete but functional system is demonstrated for the users. At this point, the practitioners start again on writing tests for the next most important part of the system.

2.2.5. Code and fix

"Code and fix" development is not a deliberate strategy and schedule pressure on software developers. With incomplete design, programmers begin producing code. At some point, testing begins (often late in the development cycle), to fix the bugs before shipment [2.1].

2.3. Model suggested for NPP

Since the software requirements are very well defined and finalized in the case of software for NPP, the waterfall model may be suitable [2.5]. But there is no checking mechanism in each stage of life cycle, so, by introducing verification at the end of every stage [2.6] with the respective artifact and enforcing validation of the product (code) with the specification it can fulfill the safety requirements of NPP [2.7]. The modified model suitable for software to be deployed in NPP is shown in Figure 2.1.

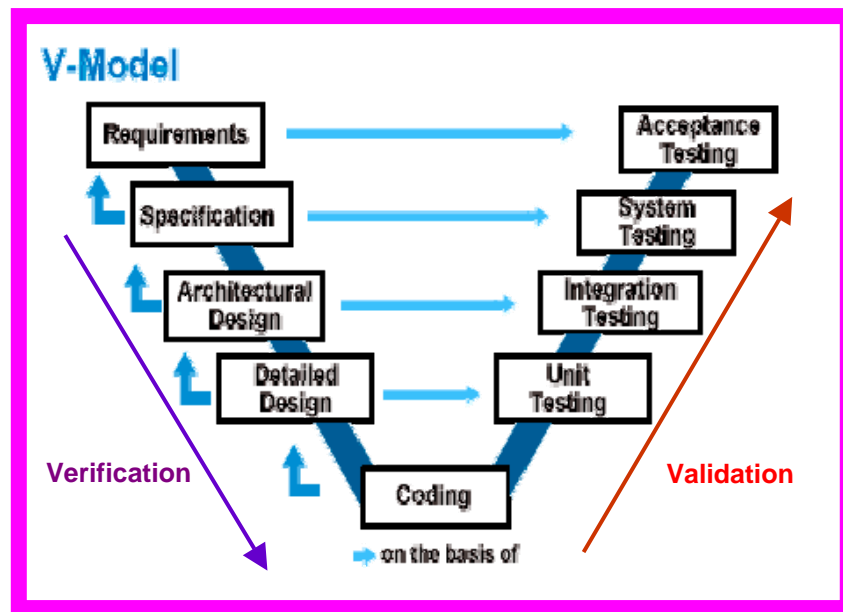


Figure 2.1 Software life cycle “V” Model

This model ensures verification at all the stages before moving on to the next phase and also the validation of the product. Each artifact is checked against the testing scheme, which ensures the completeness and correctness of the artifact. It also provides a complete control on the life cycle of the software, which in turn ensures well-matured and stable software product, which is the requirement for NPP software [2.8]. As the “test-cases” generated by an independent team in the presence of the developers is carrying out the testing, it ensures that the software product confirms to the specification of its indented functions. The software developed for safety system of fast reactor by following this “V” model resulted in better quality and phenomenal improvements in reliability. The only key parameter is that the testing team is an independent team but with the domain knowledge and its deployment with respect to the other systems and modes of operation. It should also be noted that an inherent assumption of the study is that the software described follows a waterfall life-cycle [2.9]. A waterfall life-cycle is typically characterized by the succession

of the phases from requirements to operation without too many backwards steps such as for instance the fact of going back from design to requirements. Other software development lifecycles exist such as for instance the spiral model [2.10], a lifecycle where development is driven by perceived risk areas and the resolution of these risks in an iterative fashion. Spiral development makes heavy use of prototyping and is typically used for software with a strong user interface component. Waterfall development on the other hand is recommended for programs with strong algorithmic component such as the software used in safety applications [2.11].

The Table 2-1 shows the mapping of the software development phases used in this study to the IEEE610 standard phases [2.12].

Table 2.1 Lifecycle in this Study versus Recommended Life-cycle in IEEE610.

Life-Cycle in this Study	Equivalent Life-Cycle in IEEE610 standard
Requirements Concept	Requirements
Design	Design
Implementation	Implementation
Testing	Test, Installation & Checkout Installation
Operation	Operation & Maintenance, Retirement

REFERENCES

- [2.1] http://en.wikipedia.org/wiki/Software_development_process.
- [2.2] Roger Pressman, Software Engineering: A Practitioner's Approach, Mc Graw Hill, Fifth Edition, 2001.
- [2.3] Sommerville, I., Software Engineering, Pearson Education, 6th Ed., 2001.
- [2.4] Watts S. Humphrey, Managing the software process, SEI series in software engineering, Addison Wesley Longman Inc, ISBN-981-235-916-8, 1999.
- [2.5] IAEA-TECDOC-1335 Configuration Management in Nuclear Power Plants, 2003, International Atomic Energy Agencies, Vienna, ISBN 92-0-100503-2.
- [2.6] Design safety guide on computer based systems, Atomic Energy Regulatory Board AERB/SG/D10 - Design Safety Guide on Safety Critical Systems, 2002.
- [2.7] Design safety guide on computer based systems, Atomic Energy Regulatory Board AERB/NPP – PHWR/SG/D20 - Safety Related Instrumentation and Control for pressurized heavy water reactor based nuclear power plants, 2003.
- [2.8] Design safety guide on computer based systems, Atomic Energy Regulatory Board AERB/SG/D25 - Computer Based systems of Pressurized Heavy water Reactor, 2001.
- [2.9] Schach, S. R., Software Engineering 2nd Edition, Richard D. Irwin, Inc., and Aksen Associates, Inc., Boston, 1993.
- [2.10] Boehm, B. W., "A Spiral Model of Software Development and Enhancement", IEEE Computer 21, pp. 61-72, 1988.
- [2.11] Software Engineering Measures for Predicting Software Reliability in Safety Critical Digital Systems. NUREG/GR-0019, University of Maryland, U.S. Nuclear Regulatory Commission, Washington, DC 20555-0001, 2000.

- [2.12] T.Sridevi, A.Shanmugam, D.Thirugnana Murthy, S.Ilango Sambasivan, P. Swaminathan, Software Lifecycle for Safety Critical Systems , International Conference on Trends in Intelligent Electronics System , Chennai , Nov 2007, vol.II, p.563-566

CHAPTER 3. ISSUES AND IMPORTANCE OF SOFTWARE TESTING WITH RESPECT TO RELIABILITY

3.1 Introduction

It is because of the human designer's vulnerability for errors and its own abstract and complex nature, software development must be accompanied by quality assurance activities. It is not unusual for developers to spend 40% of the total project time on testing. For life-critical software (e.g. flight control, reactor monitoring), testing can cost 3 to 5 times as much as all other activities combined. The destructive nature of testing requires that the developer discards preconceived notions of the correctness of his/her developed software. Since the advent of high-level languages, the practice of developing software in a different environment compared to the environment in which it will eventually be used has become common. The development environment is referred to as the host, and the environment in which the software will be used is referred to as the target. Such a development strategy is referred to as host-target development, and the associated testing practices as host-target testing or cross testing.

Traditionally, host-target development has been used for embedded systems, where a powerful multi-user host environment is used to develop software, which is ultimately executed in an embedded microprocessor target environment. The Personal Computer (PC) explosion has opened new avenues for host-target development, with PCs being used as a development host for embedded systems, and also as a host to develop software, which will eventually be executed on mini or mainframe systems.

3.2 Software Testing

Testing is a process of executing a program with the intent of finding an error. A good test case is one that has a high probability of finding an as yet undiscovered error. A successful test is one that uncovers an as yet undiscovered error. Testing should systematically uncover different classes of errors in a minimum duration of time and with a minimum amount of effort. A secondary benefit of testing is that it demonstrates that the software appears to be working as stated in the specifications. The data collected through testing can also provide an indication of the software's reliability and quality. But, testing cannot show the absence of defect. It can only show that software defects are present [3.1].

3.2.1 White Box Testing

White box testing is a test case design method that uses the control structure of the procedural design to derive test cases. Test cases can be derived to

1. Guarantee that all independent paths in a module have been exercised at least once,
2. Exercise all logical decisions on their true and false sides,
3. Execute all loops at their boundaries and within their operational bounds, and
4. Exercise internal data structures to ensure their validity.

3.2.2 The Nature of Software Defects

The logic errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed. General processing tends to be well understood that while special case processing tends to be prone to errors. It is often believed that a logical path is not likely to be executed when it may be executed on a regular basis. Unconscious

assumptions of the developer about control flow and data lead to design errors that can only be detected by path testing.

3.2.3 Basis Path Testing

This method enables the designer to derive a logical complexity measure of a procedural design and use it as a guide for defining a basis set of execution paths. Test cases that exercise the basis set are guaranteed to execute every statement in the program at least once during testing.

3.2.4 Flow Graphs

Flow graphs can be used to represent control flow in a program and can help in the derivation of the basis set. Each flow graph node represents one or more procedural statements. The edges between nodes represent flow of control. An edge must terminate at a node, even if the node does not represent any useful procedural statements. A region in a flow graph is an area bounded by edges and nodes. Each node that contains a condition is called a predicate node. Cyclomatic complexity is a metric that provides a quantitative measure of the logical complexity of a program. It defines the number of independent executable paths in the basis set and thus provides an upper bound for the number of tests that must be performed [3.2].

3.2.5 The Basis Set

An independent executable path is any path through a program that introduces at least one new set of processing statements (move along at least one new edge in that path) [3.3].

The basis set is not unique. Any number of different basis sets can be derived for a given procedural design. Cyclomatic complexity, $V(G)$, for a flow graph G is equal to

1. The number of regions in the flow graph.
2. $V(G) = E - N + 2$ where E is the number of edges and N is the number of nodes.
3. $V(G) = P + 1$ where P is the number of predicate nodes.

3.3 Deriving Test Cases

Towards deriving a test case, the following procedure needs to be adopted.

1. From the design or source code, derive flow graph i.e-independent executable paths.
2. Determine the cyclomatic complexity of this flow graph.

Even without a flow graph, $V(G)$ can be determined by counting the number of conditional statements in the code.

3. Determine a basis set of linearly independent paths.

Predicate nodes are useful for determining the necessary paths.

4. Prepare test cases that will force execution of each path in the basis set.

Each test case is executed and compared with the expected results.

3.4 Loop Testing

The white box technique focuses exclusively on the validity of loop constructs. The following are the four different classes of loops that can be defined:

1. Simple loops
2. Nested loops
3. Concatenated loops and
4. Unstructured loops.

3.5. Other White Box Techniques

Other white box testing techniques include:

1. Condition testing that exercises the logical conditions in a program.
2. Data flow testing that selects test paths according to the locations of definitions and uses of variables in the program.

3.6. Black Box Testing

Black box testing attempts to derive sets of inputs that will fully exercise all the functional requirements of a system. It is not an alternative to white box testing [3.4]. This type of testing attempts to find errors in the following categories:

1. Incorrect or missing functions
2. Interface errors
3. Errors in data structures or external database access
4. Performance errors and
5. Initialization and termination errors.

Tests are designed to answer the following questions:

1. How is the function's validity tested?
2. What classes of input will make good test cases?
3. Is the system particularly sensitive to certain input values?
4. How are the boundaries of a data class isolated?
5. What data rates and data volume can the system tolerate?
6. What effect will specific combinations of data have on system operation?

White box testing should be performed early in the testing process, while black box testing tends to be applied during later stages. Test cases should be derived which

1. Reduce the number of additional test cases that must be designed to achieve reasonable testing and
2. Prompts us something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

3.7 Equivalence Partitioning

This method divides the input domain of a program into classes of data from which test cases can be derived. Equivalence partitioning strives to define a test case that uncovers classes of errors and thereby reduces the number of test cases needed. It is based on an evaluation of equivalence classes for an input condition. An equivalence class represents a set of valid or invalid states for input conditions.

Equivalence classes may be defined according to the following guidelines:

1. If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
2. If an input, condition requires a specific value, then one valid and two invalid equivalence classes are defined.
3. If an input, condition specifies a member of a set, then one valid and one invalid equivalence classes are defined.
4. If an input condition is Boolean, then one valid and one invalid equivalence, classes are defined.

3.8 Boundary Value Analysis (BVA)

This method leads to a selection of test cases that exercise boundary values. It complements equivalence partitioning since it selects test cases at the edges of a class. Rather than focusing on input conditions solely, BVA derives test cases from the output domain also. BVA guidelines include:

1. For input ranges bounded by a and b, test cases should include values a and b and just above and just below a and b respectively.
2. If an input condition specifies a number of values, test cases should be developed to exercise the minimum and maximum numbers and values just above and below these limits.
3. Apply guidelines 1 and 2 to the output.
4. If internal data structures have prescribed boundaries, a test case should be designed to exercise the data structure at its boundary.

3.9 Mutation testing

The Mutation testing, Mutation analysis and Program mutation are used to design new software tests and evaluate the quality of software tests. Mutation testing involves modifying a program's source code or byte code in small ways [3.5]. Each mutated version is called a mutant and tests detect and reject mutants by causing the behavior of the original version to differ from the mutant. This is called killing the mutant. Test suites are measured by the percentage of mutants that they kill. New tests can be designed to kill additional mutants. Mutants are based on well-defined mutation operators that either mimic typical programming errors such as using the wrong operator or variable name or force the creation of valuable tests such as driving each expression to zero. The purpose is

to help the tester to develop effective tests or locate weaknesses in the test data used for the program or in sections of the code that are seldom or never accessed during execution.

Tests can be created to verify the correctness of the implementation of a given software system, but the creation of tests still poses the question whether the tests are correct and sufficiently cover the requirements that have originated the implementation. Mutation testing is done by selecting a set of mutation operators and then applying them to the source program one at a time for each applicable piece of the source code.

For example, consider the following C code fragment:

```
if (a && b) { c = 1; } else { c = 0; }
```

The condition mutation operator would replace && with || and produce the following mutant:

```
if (a || b) { c = 1; } else { c = 0; }
```

Now, for the test to kill this mutant, the following three conditions should be met:

1. A test must reach the mutated statement.
2. Test input data should infect the program state by causing different program states for the mutant and the original program. For example, a test with $a = 1$ and $b = 0$ would do this.
3. The incorrect Program State, the value of 'c' must propagate to the program's output and be checked by the test.

However, it is not possible to find a test case that could kill some mutant. The resulting program is behaviorally equivalent to the original one and they are called equivalent mutants. Equivalent mutants detection is one of biggest obstacles for practical usage of

mutation testing. The effort needed to check if mutants are equivalent or not can be very high even for small programs [3.6].

Here are some examples of mutation operators for imperative languages:

Replace each boolean sub-expression with true and false.

Replace each arithmetic operation with another, e.g. + with *, - and /.

Replace each boolean relation with another, e.g. > with >=, == and <=.

Replace each variable with another variable declared in the same scope.

mutation score = number of mutants killed / total number of mutants

These mutation operators are also called traditional mutation operators.

3.10 Fault injection testing

In software testing, fault injection is a technique for improving the coverage of a test by introducing faults to test code paths, in particular error handling code paths, that might otherwise rarely be followed [3.7]. It is often used with stress testing and is widely considered to be an important part of developing robust software [3.8]. The propagation of a fault to an observable failure follows a well defined cycle. When executed, a fault may cause an error, which is an invalid state within a system boundary. An error may cause further errors within the system boundary. Therefore each new error acts as a fault, or it may propagate to the system boundary and be observable. When error states are observed at the system boundary they are termed failures. This mechanism is termed the fault-error-failure cycle and is a key mechanism in dependability.

Software Implemented fault injection techniques for software can be categorized into two types and they are compile time injection and runtime injection. Compile-time injection is an injection technique where source code is modified to inject simulated faults into a system. A simple example of this technique could be changing

$a = a + 1$ to $a = a - 1$

Code mutation produces faults, which are very similar to those unintentionally added by programmers. A refinement of code mutation is Code Insertion Fault Injection which adds code, rather than modifies existing code. This is usually done through the use of perturbation functions which are simple functions which take an existing value and perturb it via some logic into another value, for example

```
int pFunc(int value) { return value + 20; }

int main(int argc, char * argv[]) {

    int a = pFunc(aFunction(atoi(argv[1])));

    if (a > 20) { /* do something */ }

    else { /* do something else */ }
```

In this case, pFunc is the perturbation function and it is applied to the return value of the function that has been called introducing a fault into the system.

Runtime Injection techniques uses a software trigger to inject a fault into a running software system. Faults can be injected via a number of physical methods and triggers can be implemented in a number of ways. Runtime injection techniques can use a number of different techniques to insert faults into a system via a trigger like corruption of memory space i.e., corrupting memory, processor registers, and I/O map. These techniques are often based around the debugging facilities provided by computer processor architectures.

3.11 Testing of Safety systems of Fast reactor

The Safety systems of fast reactors are tested and verified by systematic approach employing the techniques listed in sub-sections 3.1 to 3.8. The Table 3.1 shows the selective list of instrumentation and control (I&C) systems [3.9], for which verification was carried out using the above techniques.

The I&C systems are basically Triplicated or Dual redundant fault tolerant systems with switch over logic configured as distributed control systems and connected to the plant backbone network. This network is in-turn connected to the high-end servers to view any of the plant parameter at one single place in different format. The servers also log the data for future analysis. Each system runs the application software written in “C” programming language. The independent Verification / Testing team generates the test case for each sub-system. The validation of the I&C system is carried out using the dynamic simulator which can generate the scenario of the running nuclear power plant. The simulator is capable of generating the disturbance / Event as it can occur in the plant.

The errors generally encountered during testing of software developed for deploying in Nuclear power plant installations are listed in Table 3.2. A typical sample “Test Cases” for Instrumentation and control of Air Conditioning and Ventilation (AC&V) is represented in Table 3.3.

Table 3.1 List of I &C Systems

Systems
Discordance Supervision
Reactor Startup Authorisation
Fuel Handling Startup Authorisation
Chilled water and Service Water system
Argon Supply and Distribution
Nitrogen Supply and Distribution
Supervision and control system for spent Subassembly Storage Bay
Biological Shield Cooling System
Reactor Assembly Components
Control Building, Steam Generator Building & Reactor Containment Building Air Conditioning and Ventilation system
Secondary Sodium Purification Circuit
Steam Generator Tube Leak Detection System
Safety Grade Decay Heat Removal system
Primary Sodium Main Circuit
Secondary Sodium Main Circuit
Top Shield Cooling System
Core Temperature Monitoring
Process Disturbance Analyser
Event Sequencer Recorder
Fuel Handling Systems
Radioactive Effluent System

Table 3.2 Errors found during inspection of NPP software

System	Symptom	Error
Process Disturbance Analyzer	Running continuously for 40 hours and then stopping	Stack Overflow because of the Interrupt Service Routine
	Storing 10 samples of same value	Array index not incremented properly
Generator Temperature Monitoring	Gray shade of alarm panel appears on CRT and even other display comes up on the screen	Alarm panel display is the default display and it occupies same place on CRT
Attendance System	It stops working in the beginning hours of first working day of the week	When no entry for more than one day, it tries to send empty record continuously to server
	It shows busy (CPU 99%) and not recording the proximity card entries	The Antivirus was trying to establish connection over internet but the system is kept in isolation
SCADA	In the code, comment was written in confusing way to run redundant code	The execution of the code produces delay and the comment was not written properly
Guide tube profile	Software works fine when it is on Single Stepping	Timing mismatch happens when it is running in full speed
Control System	Over writing on other channel parameters	Usage of old library files

	Check on alarms inconsistent	Checking with “single &”, instead of “double &&” on the flags
Pump Interlock	Interlock works for some combination and not for other combinations	Usage of “and” & “or” operation is one “if” without brackets and precedence is not taken care of.
Permissive check for Startup	The condition never happens	Checking for equality in floating point numbers
	Code and comment mismatch leads to ambiguity	Code is modified but comment was not modified
Rotating plug alignment system	Software reaches some undefined space and stops working	Variable of one element is declared as POSITION[1] but used as POSITION[1] instead of POSITION[0]
	When Plug Rotates, "Unlock" was removed and hence it Stopped. But when "Normalized" it started rotating.	Resetting the output after anyone condition is violated was not done.
	Repetition rate sometimes becomes large	If execution takes more time than the repetition rate, the timer counter reaches maximum value and takes time

Sodium fill and Drain system	Switching to standby pump was not happening	when standby started it trips since no flow is there. Therefore delay is introduced in sensing
Decay Heat Removal System	Switching from Auto / Manual and vice versa was not bump-less	Resetting the Values and not carrying the “state information” in switchover caused the problem
Alarm System	Periodically readings become empty and comes back to normal	Circular Buffer end point to beginning not synchronized

Table 3.3 Sample Test Cases for I&C of Air Conditioning and Ventilation System

Cabinet No : SPCsg115			Code version Number: 1.1			
No.	Test Case Description	Procedure / Input	Expected Outputs/ Behaviour	Observed outputs/ Behaviour	Test Pass or fail	Remarks
1	To check the DI channels (Level switches)	Change the state of DI-102 CH 1 to 12	Respective channel on GUI has to go ON	Same as expected	PASS	
2	To check the DO channels (Remote/Local selection of Pumps) DO 103 ch 1-12	When the remote selection is selected ie. When SSW is ON (DI102 ch 7-12)	STR,STP of the corresponding pumps should be enabled to operate.	Same as expected	PASS	
3	To check the DO channels (Remote/Local selection of Pumps) DO 103 ch 1-12	When the remote selection is selected ie. When SSW is OFF (DI 102 ch 7-12)	STR,STP of the corresponding pumps should be disabled to operate.	Same as expected	PASS	
4	Auto mode operation of the pumps	DO 103 ch13,14, DO-108 ch 1-4 should be ON for remote operation	DO 103 ch13,14, DO-108 ch 1-4 should be ON for remote operation	Same as expected	PASS	
5	Analog inputs (Pressure Transmitter)	Feed 4mA, 12mA and 20mA in AI-106 ch 1-4	0MPa, 0.55MPa 1.1MPa respectively	0MPa, 0.549MPa and 1.098MPa	PASS	
6	To check the DI channels (Pressure switches)	Change the state of DI-107 CH 1 to 6	Respective channel on GUI has to go ON	Same as expected	PASS	
	Prepared by :					
	Testing carried out by:					

3.12 Inferences

This chapter covered the testing techniques deployed in the I&C systems of fast reactor that are listed in table 3.1. These systems are basically embedded systems with MC68020 as the processor, which uses the VME bus. The table 3.2 shows the errors found during the inspection of the software code written in “C” language. A typical sample of “Test Cases” is also shown in Table 3.3. Elaborate “Test Cases” were written for each of I&C systems. Planned systematic software testing and the error removal increase the reliability.

REFERENCES

- [3.1] Roger Pressman, Software Engineering: A Practitioner's Approach ,Mc Graw Hill, Fifth Edition, 2001.
- [3.2] Sommerville, I., Software Engineering, Pearson Education, 6th Ed., 2001.
- [3.3] Watts S. Humphery, Managing the software process, SEI series in software engineering, Addison wesley longman Inc, ISBN-981-235-916-8, 1999.
- [3.4] IEEE Standards Software Engineering, vol 1 – 4, 1999.
- [3.5] http://en.wikipedia.org/wiki/Mutation_testing.
- [3.6] A Practical System for Mutation Testing: Help for the Common Programmer by A. Jefferson Offutt.
- [3.6] http://en.wikipedia.org/wiki/Fault_injection.
- [3.7] J. Voas, "Fault Injection for the Masses," Computer, vol. 30, pp. 129–130, 1997.
- [3.8] N. Sridhar, B. Krishnakumar and S. Ilango Sambasivan ,Computer Based Systems for Prototype Fast Breeder Reactor, (ANIMMA, June 7-10, 2009) Advancements in Nuclear Instrumentation, Measurement Methods and their Applications, (ISBN: 978-1-4244-5207-1, DOI: 10.1109/ANIMMA.2009. 5503832
- [3.9] M.A. Sanjith, K.Kameswari, B.Ramasamy Pillai, S.Ilango Sambasivan & P. Swaminathan , Real Time Computer Based Control Systems for Prototype Fast Breeder Reactor, National Symposium on Applications of computer and Embedded Technology(SACET09), BARC, Mumbai

CHAPTER 4. EVALUATION OF SOFTWARE METRICS AND TOOL

4.1 Evaluation of Software Metrics

Identification of the software metrics that affect the reliability in terms of quality attribute and the development of tool to evaluate the metrics from the software code are very important. The various quality parameters that contribute to reliability [4.1] are described in the following subsections.

4.1.1 Cyclomatic Complexity (CC)

This attribute measures the complexity of each function in terms of number of independent executable paths, i.e, number of branches. Less the number of branches, better for the reliability. CC is a software metric developed by Thomas J. McCabe [4.2] and is the indicator of complexity. It is computed using the control flow graph of the program, where the nodes of the graph correspond to indivisible groups of commands, and a directed edge connects two nodes. It can also be applied to individual functions within a program. The number of test cases in the Basis Path Testing strategy is proportional to the cyclomatic complexity of the program.

4.1.2 Nesting Level

The depth of “condition branches” or the “loops” is the measure for each function. When the depth is less, the reliability is better, “test cases” are less and the testability and coverage are good.

4.1.3 Comment to Code ratio

Comments in the code is one way of documentation that goes with the code. More the comments is an indication of good understanding of the internal details of the function. Comments play a major role as part of maintenance in terms of fixing the problem or upgrading the software.

4.1.4 Ternary Operator

Programming languages use the feature, ternary operator, “?:,” which defines a conditional expression. It is sometimes referred to as "the ternary operator", though it is more accurately referred to as the “conditional operator”. The functional programming does not need such an operator as their regular conditional expression, e.g., if (a > b) {result = x;} else { result = y; } can be rewritten as the following ternary statement: result = a > b ? x: y; This kind of convention is prone to error forcing difficulties during software maintenance.

4.1.5 Dynamic Memory

Dynamic memory allocation also known as heap-based memory allocation is the allocation of memory storage for use during the run-time. It can also be seen as a way of distributing ownership of limited memory resources among many pieces of data and code. Hence it is vulnerable for memory overflow and resulting in failure [4.3].

4.1.6 Goto and Continue Statements

The use of the ‘go to’ statement has an immediate consequence that it becomes hard to find a meaningful set of coordinates. The ‘go to’ statement as it stands is too primitive and could be a potential source of error. The “Continue” statement is used to add delay and it is

used to keep the label number to jump in. This leads to program un-structured and sensitive to location.

4.1.7 Number of code lines in a function

Less the number of lines of code, it will be easier to understand, test and maintain. On the other hand, when the code is lengthy and complex it is prone to error. So the number of code lines should be less and it should be simple in logic [4.4].

4.1.8 Recursive functions

This creates ambiguity and in-turn it is vulnerable for failures because of the unknown number of iterations which may result in stack overflow [4.5].

4.1.9 Unused functions and Variables

The unused functions and variables are called as “Dead Codes”. It will unnecessarily occupy memory and leading to ambiguity during maintenance and in impact analysis.

4.2 Software Quality Metrics IEEE-1061 standard

In developing a Set of Metrics, IEEE Standard 1061 [4.6] lays out a methodology for developing metrics for software quality attributes. The standard defines an attribute as "a measurable physical or abstract property of an entity". A quality factor is a type of attribute, "a management-oriented attribute of software that contributes to its quality". A metric is a measurement function, and a software quality metric is "a function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software adhere to a given attribute that affects its

quality”. To develop a set of metrics for a project, one creates a list of quality factors that are important for it. Associated with each quality factor is a direct metric that serves as a quantitative representation of a quality factor. For example, a direct metric for the factor reliability could be mean time to failure (MTTF). Identify one or more direct metrics and target values to associate with each factor, such as an execution time of 1 sec, which is set by project management. Otherwise, there is no way to determine whether the factor has been achieved [4.6].

For each quality factor, assign one or more direct metrics to represent the quality factor to serve as quantitative requirements for that quality factor. For example, if "high efficiency" was one of the quality requirements, assign a direct metric "actual resource utilization / allocated resource utilization" with a value of 90%. Use direct metrics to verify the achievement of the quality requirements [4.7]. Use only validated metrics (i.e., either direct metrics or metrics validated with respect to direct metrics) to assess current and future product and process quality. The Section 4.5 of IEEE Standard 1061 lays out several criteria for validation, which are summarized as follows:

- 1) Correlation: The metric should be linearly related to the quality factor as measured by the statistical correlation between the metric and the corresponding quality factor.
- 2) Consistency: Let F be the quality factor variable and Y be the output of the metrics function, $M: F \rightarrow Y$. M must be a monotonic function. That is, if $f_1 > f_2 > f_3$, then we must obtain $y_1 > y_2 > y_3$.
- 3) Tracking: For metrics function, $M: F \rightarrow Y$. As F changes from f_1 to f_2 in real time, $M(f)$ should change promptly from y_1 to y_2 .

4) Predictability: For metrics function, $M: F \rightarrow Y$. If we know the value of Y at some point in time, we should be able to predict the value of F .

5) Discriminative power: A metric shall be able to discriminate between high-quality software components (e.g., high MTTF) and low-quality software components (e.g., low MTTF). The set of metric values associated with the former should be significantly higher (or lower) than those associated with the latter.

6) Reliability: A metric shall demonstrate the correlation, consistency, predictability, tracking and discriminative power properties of the application metric.

The validation criteria are expressed in terms of quantitative relationships between the attribute being measured (the quality factor) and the metric. The IEEE Standard 1061 recommends the use of direct metrics. A direct metric is a metric that does not depend upon a measure of any other attribute.

4.3 Thirty Software Engineering Measures

The set of thirty measures considered in the University of Maryland (UMD) study is listed below. The resulting measures constitute the basis of their study [4.8].

Bugs per line of code (Gaffney estimate),	Cause & effect graphing
Code defect density	Cohesion
Completeness	Cumulative failure profile
Cyclomatic complexity	Data flow complexity
Design defect density	Error distribution
Failure rate	Fault density
Fault-days number	Feature point analysis
Function point analysis	Functional test coverage

Graph-theoretic static arch. complexity	Human hours per major defect detected
Mean time to failure	Minimal unit test case determination
Modular test coverage	Mutation testing (error seeding)
Number of faults remaining (error seeding)	Requirements compliance
Requirements specification change requests	Requirements traceability
Reviews, inspections and walkthroughs	Software capability maturity model
System design complexity	Test coverage

4.3 Static Analyser Tool

This tool evaluates the software quality parameters and generates the report on the quality attribute metrics [4.9]. The report of Cyclomatic complexity with respect to each function of a typical safety system of NPP is shown in Figure 4.1. The right side of the screen shows the cyclomatic complexity of all the functions with the scroll bar.

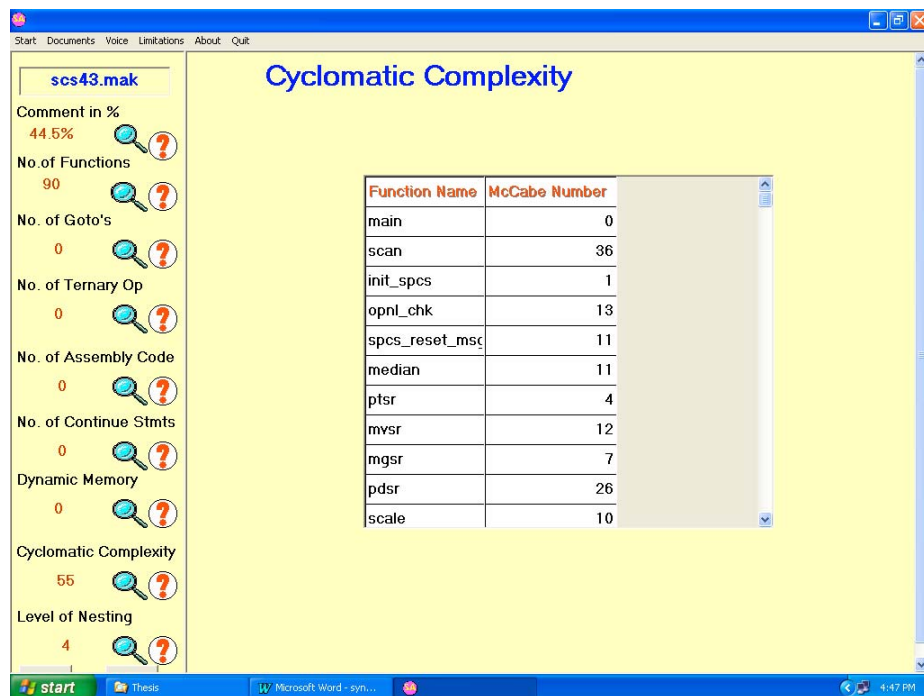


Figure 4.1 Typical output screen of the SA with Cyclomatic Complexity

The Figure 4.1 shows the typical output screen with cyclomatic complexity detailing. All the parameters are listed on the left side of the screen. User can view any one parameter in detail by clicking the “Lens Icon”. The details of that parameter will appear on the right side of the screen. Here it is shown that “scan” and “pdsr” cyclomatic complexity is much higher than the prescribed limits. Each parameter “Help” is provided which explains the algorithm used in calculating the parameter and how to use the application software.

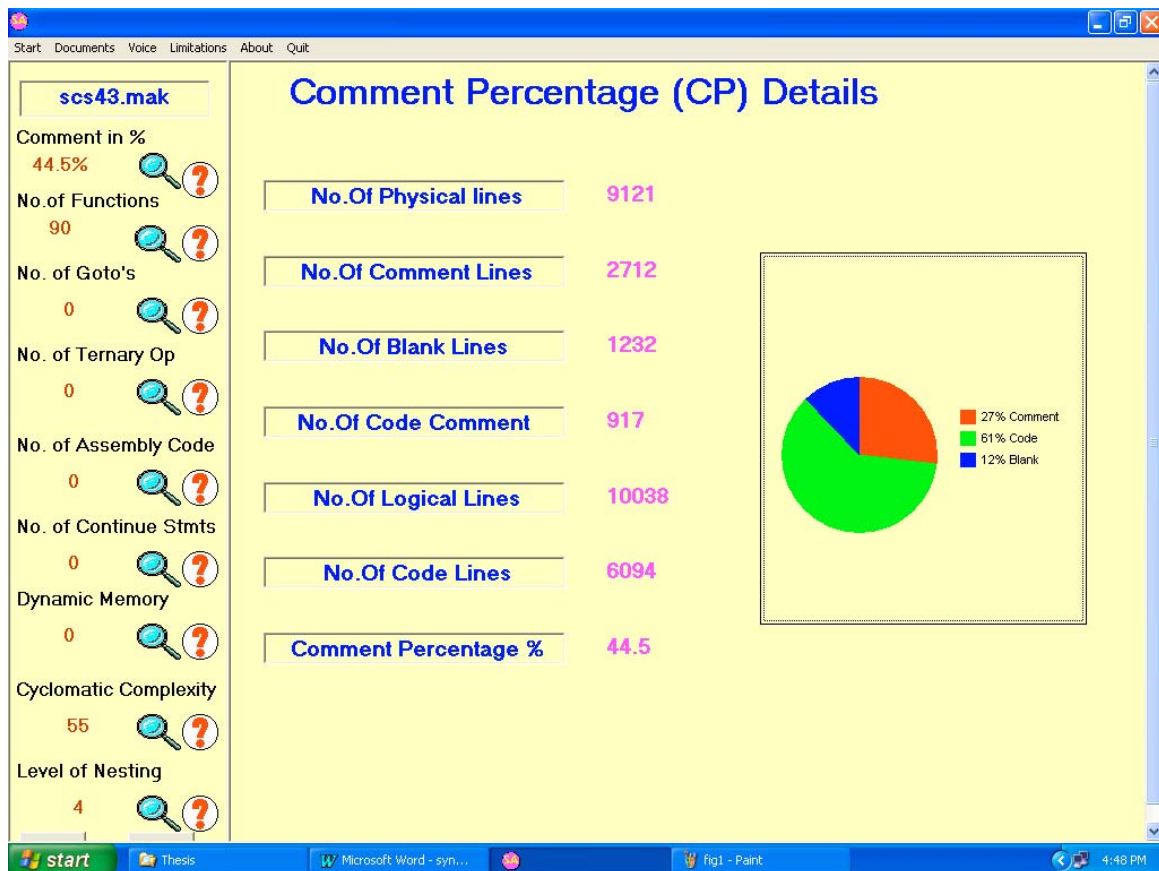


Figure 4.2 Screen output of the SA detailing Comment to Code ratio

The Figure 4.2 is detailing the Comment to Code ratio of the safety critical system software, which comprises of the complete project file. The comment percentage is only 44%, limit values shows that it should be minimum 50%. Table 4.1 presents the recommended limit values of Quality parameters on the code. This data is used to measure in terms of maintainability and the project development time [4.10]. The SA software is

developed using “Microsoft Visual Basic” as the programming language. The typical screen output of the developed tool of “Static Analyser” is shown in Figure 4.2.

Table 4.1 Recommended Limit Values on code

Parameter	Limit Value	Remarks
Comment to Code ratio	Greater Than or Equal to 50%	i) As per number of lines ii) Comments should be fairly distributed iii) Quality comments
Cyclomatic Complexity	Less than 20 for SC II Less than 15 for SC I	As per CASE tool
Nesting Level	Less than 4	As per CASE tool
Ternary Operator	Not Allowed	As per MISRA guidelines
Dynamic Memory	Not allowed	As per MISRA guidelines
Goto, Continue	Not Allowed	As per MISRA guidelines
Number of code lines in a function	Less than 50 lines	Including comment lines
Recursive functions	Not allowed	Vulnerable to Stack Overflow
Unused functions	Not allowed	Dead code, As per MISRA guidelines
Variable number of arguments in a function	Not allowed	
Unused Variables	Not allowed	As per MISRA guidelines

4.4 Inferences

The in-house developed Static Analyzer tool lists out the quality parameters with the measurements for each function of application software, which will be useful in calculating the software reliability. The recommended limit values on code based on the systems with the safety level criteria, which we have analyzed in almost 50 software modules of fast reactor, is also listed. Figure 4.3 shows the typical variation of failure intensity (Number of incorrect code segments) observed over the time (in hours) for three different embedded software, viz., a, b and c.

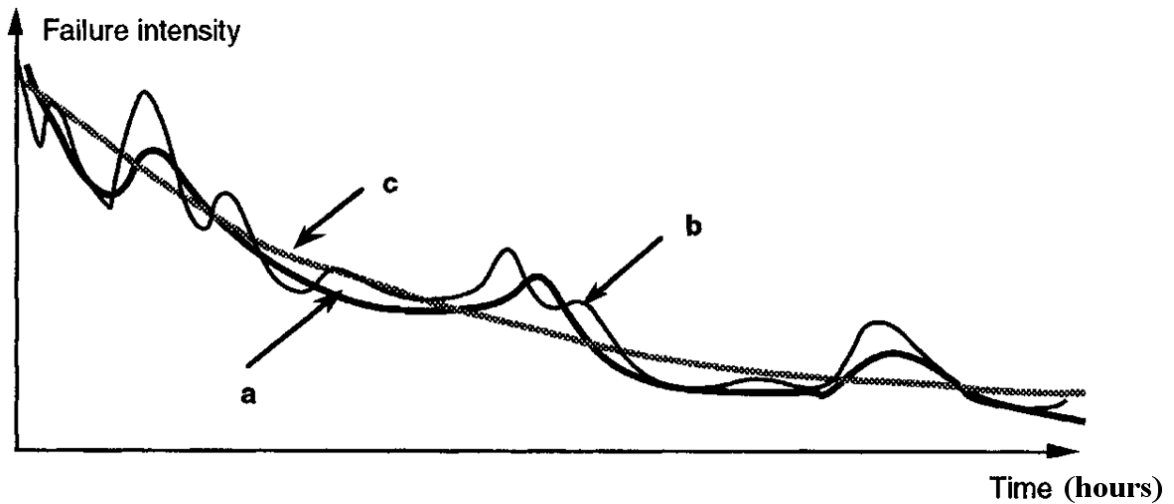


Figure 4.3 Typical variation of failure intensity.

REFERENCES

- [4.1] Watts S. Humphrey, Managing the software process, SEI series in software engineering, Addison Wesley Longman Inc, ISBN-981-235-916-8, 1999.
- [4.2] Sommerville, I., Software Engineering, Pearson Education, 6th Ed., 2001.
- [4.3] Roger Pressman, Software Engineering: A Practitioner's Approach, Mc Graw Hill, Fifth Edition, 2001.
- [4.4] Design safety guide on computer based systems, Atomic Energy Regulatory Board AERB/NPP – PHWR/SG/D20 - Safety Related Instrumentation and Control for pressurized heavy water reactor based nuclear power plants, 2003.
- [4.5] Design safety guide on computer based systems, Atomic Energy Regulatory Board AERB/SG/D25 - Computer Based systems of Pressurized Heavy water Reactor, 2001.
- [4.6] IEEE, "IEEE Std. 1061-1998, Standard for a Software Quality Metrics Methodology, revision." Piscataway, NJ,: IEEE Standards Dept., 1998.
- [4.7] Software Engineering Metrics: What Do They Measure and How Do We Know? Cem Kaner, Senior Member, IEEE, and Walter P. Bond, 10th International Software Metrics Symposium, METRICS, 2004.
- [4.8] Software Engineering Measures for Predicting Software Reliability in Safety Critical Digital Systems. NUREG/GR-0019, University of Maryland, U.S. Nuclear Regulatory Commission, Washington, DC 20555-0001, 2000.
- [4.9] T.Sridevi, A.Shanmugam, D.Thirugnana Murthy, S.I. Sambasivan and P.Swaminathan, Static Analyzer for Computer Based Safety Systems, Journal of the Instrument Society of India (ISOI), Vol. 37(1), pp. 40-48, 2007.

- [4.10] T.Sridevi, D.Thirugnana Murthy, B. Krishnakumar, S.A.V. SatyaMurty and P.Swaminathan, Software Quality Assessment for Safety Systems of Nuclear Reactor, 2nd International Conference on Asian Nuclear Prospects (ANUP-2010), Oct 11-13, Mahabalipuram, Tamilnadu

CHAPTER 5. VERIFICATION AND VALIDATION PROCEDURE

5.1 Introduction

Software verification and validation (V&V) is a technical discipline of systems engineering. The purpose of software V&V is to help the developer to build quality into the software during the software life cycle. The software V&V processes determine if development products of a given activity confirm to the requirements of that activity, and if the software satisfies the intended use and user needs. The determination includes assessment, analysis, evaluation, review, inspection and testing of software products and processes. The software V&V is performed in parallel with the software development, not at the end of the software development [5.1]. The complete framework of V&V procedure developed as part of research work encompassing artifacts, checklists, traceability matrix are detailed here. The V&V procedure evaluates on the following key concepts. They are software safety levels, V&V tasks for each software safety level, systems viewpoint and Compliance with International and IEEE standards. The V & V procedure is developed to

- Establish a common framework for V&V processes, activities, and tasks in support of all software life cycle processes.
- Define the V&V tasks, required inputs, and required outputs.
- Identify the minimum V&V tasks corresponding to software safety levels using a three-level scheme [5.2].

This V&V procedure shall use the following three-level software integrity scheme as a method to define the minimum V&V tasks that are assigned to each software criticality level as listed in Table 5.1 [5.3]

Table 5.1 Software Criticality Levels

Criticality	Description	Level
Safety Critical	Initiates the shutdown of the reactor when the signal crosses the safety limit.	1
Safety Related	Selected function affects important system performance.	2
Non Nuclear Safety	Systems not related to nuclear safety.	3

5.2. Verification procedure for Custom-Built Systems (CBS)

5.2.1 Documents submitted

The following design documents along with Traceability matrix have to be submitted by the designer / developer to the V&V committee as and when required as the verification process progresses. The documents are generated as per the standards [5.4]

Design Basis Document	System Requirements Specification (SyRS)
System Design Guidelines	System Architectural Design (SAD)
System Integration & Test Procedure(SysITP)	Software Requirements Specification (SRS)
Software Design Description (SDD)	Software Integration & Test Procedure (SITP)
Programming Guidelines (PG)	Software Implementation (source and Object code)
User Documentation (UD)	System Build

During the process of V&V, all the documents are reviewed for clarity, completeness, correctness, consistency and compliance to standards. They are verified for complete traceability [5.5].

The following test reports have to be submitted to V&V Committee during system validation [5.6].

Software Unit Test Report (SUTR) Software Integration and Test Report (SITR)
 System Integration and Test Report (SyITR) System Acceptance Test Report
 System Safety Analysis Report (SSAR)

5.2.2 Procedure for Verification

The procedure for verification has the following stages of review. They are System Requirements, System Architecture, Hardware / software specifications, Hardware / software Design, Hardware / software Implementation, System Integration and System Validation [5.7]. The V& V process is initiated once the SyRS for a particular system is submitted to the V&V Committee and the committee forms a review task force, which comprises the following members as minimum requirement as shown in Table 5.2.

Table 5.2 Review Taskforce Committee

S.No	Responsibility	Resource
1	Convenor	From V&V
2	Co-convenor	From V&V
3	Member	From V&V
4	Subject Matter Expert (SME) / Domain Expert	Invitee
5	Inspectors / Peers	Invitee
6	Author	Designer / Developer

The task force reviews the document submitted / presented at various stages by the designer / developer and verifies for its compliance with the baselined documents and checks for the consistency of the document and produces verification report. In case of no anomaly, the document may be cleared and in case of any anomaly or controversy with the any document, a revision will be recommended to the designer. In that case, the designer has to re-submit the document again with the compliance report that states the reason for anomaly. After verification, the task force, issues verification report through V&V Committee. Once a document baseline, any change to that will initiate the review process to documents submitted thereafter [5.8].

5.2.3 System Requirements Specification (SyRS) Review

The Purpose of SyRS review is to verify that SyRS covers all of the I & C design requirements of the corresponding system as described in Design Basis and all the requirements are unambiguous, complete, consistent, traceable and verifiable. The documents to be submitted are Design Basis, System Requirements Specification as per IEEE standard 1233 [5.9] and other specific requirements, if any. The documents generated are Review comments along with Compliance Report and Traceability matrix report to track defect closure, which contains total defects, review efficiency and defect density. The System Requirements Specification document is base-lined after review.

When the review task force identifies any defect, it has to be recorded by Convener for the traceability purpose. The consolidated defects identified in the review process should be circulated to the review task force. The checklist for the system requirement review is detailed in Appendix-II. The review process may be repeated iteratively, once the defects

are fixed. The review process should ensure System consistency, safety, reliability and maintainability. The V&V effort shall perform, as appropriate for the selected system integrity level Traceability Analysis, System Requirements Evaluation, Interface Analysis, Criticality Analysis, System V&V Test Plan Generation and Verification, Acceptance V&V Test Plan Generation and Verification, Configuration Management Assessment and Safety Analysis.

5.2.4 System Architecture Review (SAR)

The documents to be submitted are baselined System Requirements Specification (SyRS), Software Design Guidelines, Software Integration Test Procedure and System Architecture Design. The documents generated are Review comments along with compliance report and traceability matrix report to track defect closure, which contains total defects and defect density. The System Architecture Design document is base-lined after review.

The baselined SyRS document forms the starting point for this review. This review is to ensure the completeness of the architecture design document. The review task force conduct the review process as per the given architecture review checklist and consolidate the defects in the compliance report. The same is communicated to the designer for the completeness of the document. This review process ensures the forward and backward traceability of the system. The checklist for system architecture review is provided in Appendix-III. The objectives of V&V are to demonstrate that the design is a correct, accurate, and complete transformation of the system requirements and no unintended features are introduced.

5.2.5 Software Requirement Specification (SRS) Review

The Purpose is to verify that the software requirements specification confirms to the system requirement specification, system architecture design and traceability. The documents submitted are baselined SyRS, SAD and SRS as per IEEE 830 [5.10], Traceability matrix with SyRS and SAD. The documents generated are Verification report of SRS, SRS document Check List and System V & V test plan. The V&V effort shall perform, as appropriate for the selected system integrity level. The checklist for the software requirement review is presented in Appendix-IV.

5.2.6 Software Design Description (SDD) Review

The purpose is to verify that the software design confirms to the software requirements and it is traceable. The documents submitted are baselined SRS with SDD as per IEEE 1012 [5.11] and Traceability matrix with SRS and SDD. The documents generated are Verification report of SDD and SDD checklist. The design V&V activity addresses software architectural design and software detailed design. The objectives of V&V are to demonstrate that the design is a correct, accurate, and complete transformation of the software requirements and that no unintended features exists. The checklist for the software design review is provided in Appendix-V.

5.2.7 Software Implementation Review

The documents submitted are software (both source code and object files), baselined Software Design Description (SDD), Traceability Matrix (SDD to Source Code), Observed Programming Guidelines, Software Unit Test Report (SUTR), Software Integration Test Report (SITR) and System Build Settings.

The documents generated include the code non-compliance report with respect to MISRA ‘C’ guidelines and to the observed programming guidelines [5.12], static and dynamic analysis report and the manual code walk-through report. The software coding, testing, and build settings are evaluated [5.13].

The code is subjected to static analysis to verify if the structure of code is as per the programming guidelines. For example, if Cyclomatic complexity of safety critical function software is defined to be within 10, and if the code does not meet this requirement then the code will be returned for correction. Only when the code passes the static analysis check, the software shall be taken for further verification process. The compliance of the source code to the observed standard is also checked before it is taken up for further verification process. Safety systems software will undergo manual code walk-through besides static analysis. V&V team shall evaluate the source code for correctness, consistency, completeness, accuracy, readability and testability [5.14].

5.2.8 System Integration Review

The documents submitted are baselined System Requirement Specification, System Architectural Design, Interface Requirement Specification and System Integration Test Report. The document generated is System Integration Verification Report including the check list given in Appendix-VI. SyITR is reviewed for its completeness and consistency.

5.2.9 System Validation Review

The documents submitted are baselined System Requirement Specification, Traceability Matrix SyRS to Implementation (Software functions / Hardware) and System Acceptance Test Report [5.15]. The document generated is System Validation Report including the checklist for the system validation review as given in Appendix-VII.

5.3 Inferences

The framework of V&V procedure developed as part of research work encompassing artifacts, checklists, traceability matrix are applied on the I&C systems of nuclear power plant, resulted in phenomenal increase in quality, reliability and confidence on these systems.

REFERENCES

- [5.1] IEEE Std 1012-1998 - IEEE Standard for Software Verification and Validation.
- [5.2] NPCIL / IVVC Proc. Version 1.0- Procedure for Independent Verification and Validation of Computer Based C&I systems.
- [5.3] P. Swaminathan, Design Aspects of Safety Critical Instrumentation of Nuclear Installations, International journal of Nuclear energy Science and Technology ,Vol.1, nos.2/3, p.254-263,2005
- [5.4] Software Standards and Qualification for Safety Related Systems of PFBR, (NSNI, Feb 24-26, 2010) DAE-BRNS National Symposium on Nuclear Instrumentation-2010, K.Kameswari, B.Ramasamy Pillai, S.Ilango Sambasivan & P. Swaminathan
- [5.5] AERB/SG/D25 - Computer Based systems of Pressurized Heavy water Reactor.
- [5.6] IEC 880 – Software for computers in safety systems of nuclear power stations.
- [5.7] C.K.Pithawa, Fault Tolerant Safety Related Computer based Process control system for TAPP3&4. Proceedings of SACI-2005, p 56-68
- [5.8] S.G.Bhandarkar “Design of hardware for Computer Based Systems, Proceedings of National Symposium on Nuclear Instrumentation-2004” ,pp 84-86.
- [5.9] IEEE Std 1233, 1998 Edition, IEEE Guide for Developing System Requirements Specifications.
- [5.10] IEEE Std 830-1998, IEEE Recommended Practice for Software Requirements Specifications.
- [5.11] IEEE Std 1016-1998, IEEE Recommended Practice for Software Design Descriptions.

- [5.12] BARC/1999/E/021 - Programming Guidelines for Computer Systems of NPPs, BARC, Mumbai.
- [5.13] NPCIL / ED-PROC - Engineering Procedure for Computer Based C&I Systems.
- [5.14] D. Thirugnana Murthy, T. Sridevi, A. Shanmugam, P. Swaminathan, Verification & Validtion for Safety Critical Real Time Computers (ISSN 0973-9238), International Journal on Intelligent Electronic Systems, Vol.1, No.1 Nov 2007, p15-21
- [5.15] D.Thirugnana Murthy ,T.Sridevi, SAV Satyamurty , P.Swaminathan, Verification and Validation of Computer based Safety Systems for Nuclear Reactors, International Conference on Peaceful uses of Atomic Energy, Delhi, Sep 2009, p158-159

CHAPTER 6. SOFTWARE RELIABILITY MODELING

6.1 Introduction

It is well recognized that assessing the reliability of software applications is a major issue in reliability engineering. Predicting software reliability is not an easy task. The major difficulty is concerned primarily with design faults, which is a very different situation from that handled by conventional hardware theory. A fault refers to a manifestation in the code of an error made by the programmer or designer with respect to the specification of the software. Activation of a fault by an input value leads to an incorrect output. Detection of such an event corresponds to an occurrence of a software failure [6.1]. The input values to the software modules (functions) either internally or externally may be considered as arriving to the software randomly. So although software failure may not be generated stochastically, it may be detected in such a manner. Therefore, it justifies the use of stochastic models of the underlying random process that governs the software failures.

Six categories of models have been considered as potential candidates for modeling the reliability of software. Classification of software reliability models is presented according to software development life cycle phases as shown in Figure 6.1. The six categories include early prediction models, architectural based models, hybrid white box approach, hybrid black box approach, reliability growth models and input domain models. The early prediction model uses characteristics of the software development process from requirements to test and extrapolates this information to behavior during operation. The architectural based models put emphasis on the architecture of the software and derive reliability estimates by combining estimates obtained for the different modules of the software. The reliability growth model captures failure behavior during testing and

extrapolates it to behavior during operation. Hence, this category of models uses failure data and trends observed in the failure data to derive reliability predictions. The input domain model uses properties of the input domain of the software to derive a correctness probability estimate from test cases that executed properly.

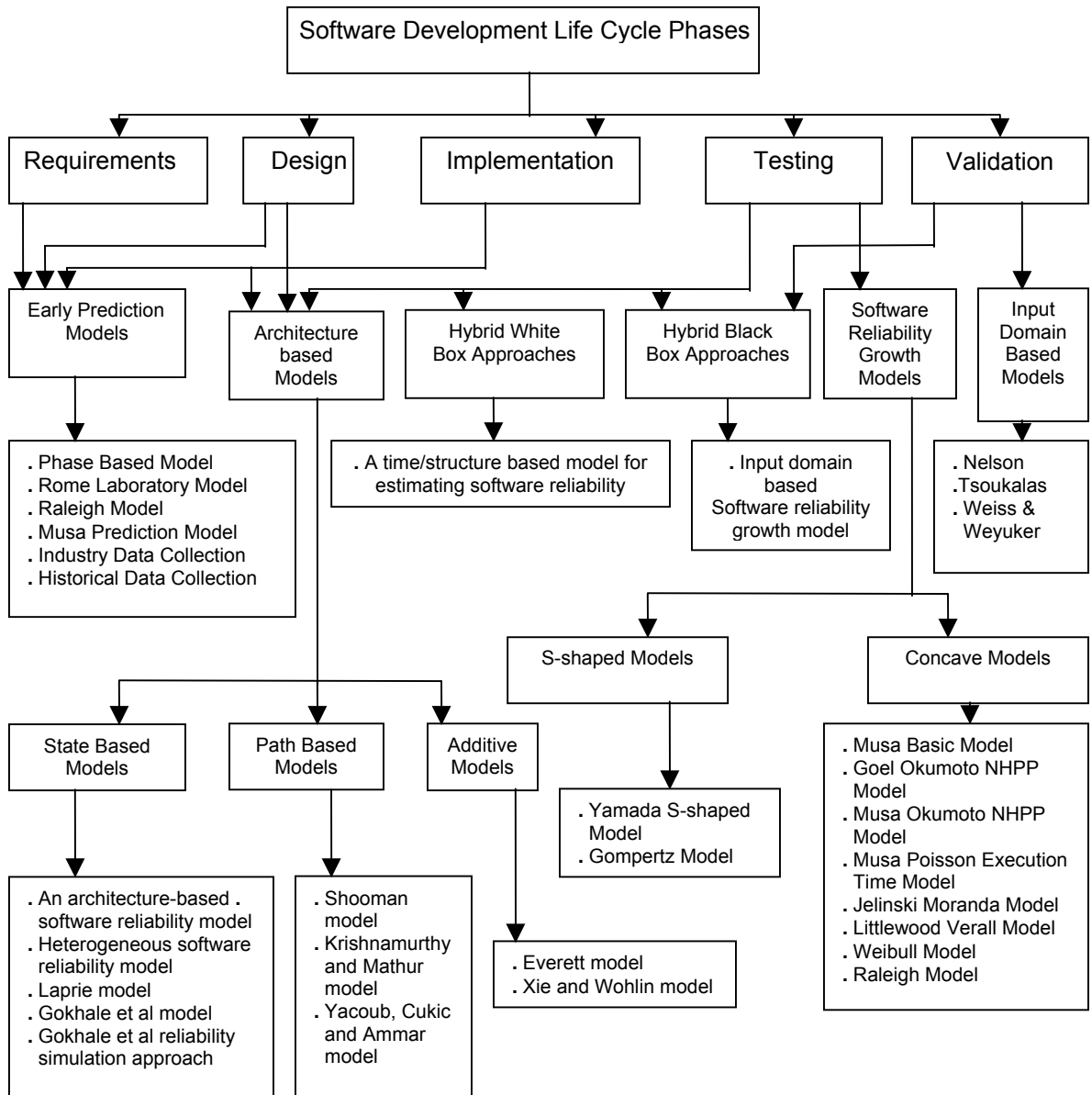


Figure 6.1 Classifications of Software Reliability Model

Hybrid black box models combine the features of input domain models and software

reliability growth models. Hybrid white box models use selected features from both white box models and black box models. However, each group of models has its inherent flaws when applying them to safety systems [6.2].

The traditional input domain model, Nelson model [6.3], is as simple as a point estimate of failure rate based on the number of failures and number of total test cases. This model needs exorbitant amounts of testing for safety systems. Nelson and other researchers introduced the concept of equivalence class, which significantly reduces the amount of testing required. These models started with a problematic assumption that the input domain can be thoroughly identified and classified into equivalence classes. Extreme caution should be exercised when applying this model to safety systems. The inputs of such a system are generally infinite and unpredictable. The structural models are widely used in fault-tolerant systems [6.4] [6.5]. The failure rate (or transition rate) from the normal state to the abnormal state (or vice versa) is assumed to be available. However, how to estimate this parameter is not known. In essence, this rate is the failure rate of a subsystem in the fault-tolerant system. The estimation of the rate requires the failure data to be available.

Ramamoorthy and Bastani model [6.6] is an Input domain based model. This pertains to critical, real time and process control programs. In such systems, no failures should be detected during the reliability estimation phase, such that the reliability estimated is unity. Thus, the important metric of concern is the confidence level in the reliability estimate. This model provides an estimation of the conditional probability that the program is correct for all possible inputs given that it is correct for a specified set of inputs.

Several early prediction models exist [6.7 – 6.9]. The Gaffiney and Davis model [6.7] is

based on the assumption that the size of the system in lines of code is available (or predictable) at the time the prediction is made. Then the number of discovered faults is given by an empirical relationship. Unfortunately, there is still a long way to go from the number of discovered faults to reliability prediction. The Rome laboratory model [6.8] derives reliability from copious data sources by means of some unexplainable empirical relationships. No research shows that this model is applicable to safety systems [6.9]. Software reliability models have their genesis in hardware reliability models, but there are differences between hardware and software reliability models. Failures in hardware are typically based on the age of hardware and the stress of the operational environment, whereas failures in software are due to incorrect requirements, design, coding, or the inability to inter-operate with other related software. Software failures typically manifest when the software is operated in an environment for which it is not designed. Typically, except for the infant mortality factor in hardware, hardware reliability decreases with age, whereas software reliability can increase with age (due to fault fixes).

A software reliability model specifies the general form of the dependence of the failure process on the principal factors that affect it namely fault introduction, fault removal, and the operational environment. During the test phase, the failure rate of a software system is generally decreasing due to the identification and correction of software faults. With record-keeping procedures, it is used to analyze the historical record.

The purpose of this analysis is two folds:

- ✓ To predict the additional time needed to achieve a specified reliability objective.
- ✓ To predict the expected reliability when testing is completed.

The assumption is that the software system being tested remains fixed throughout (except

for the removal of faults as they are found). This assumption is frequently violated. As we move in the life of the software, the number of failures generally follows the profile depicted in Figure 6.2.

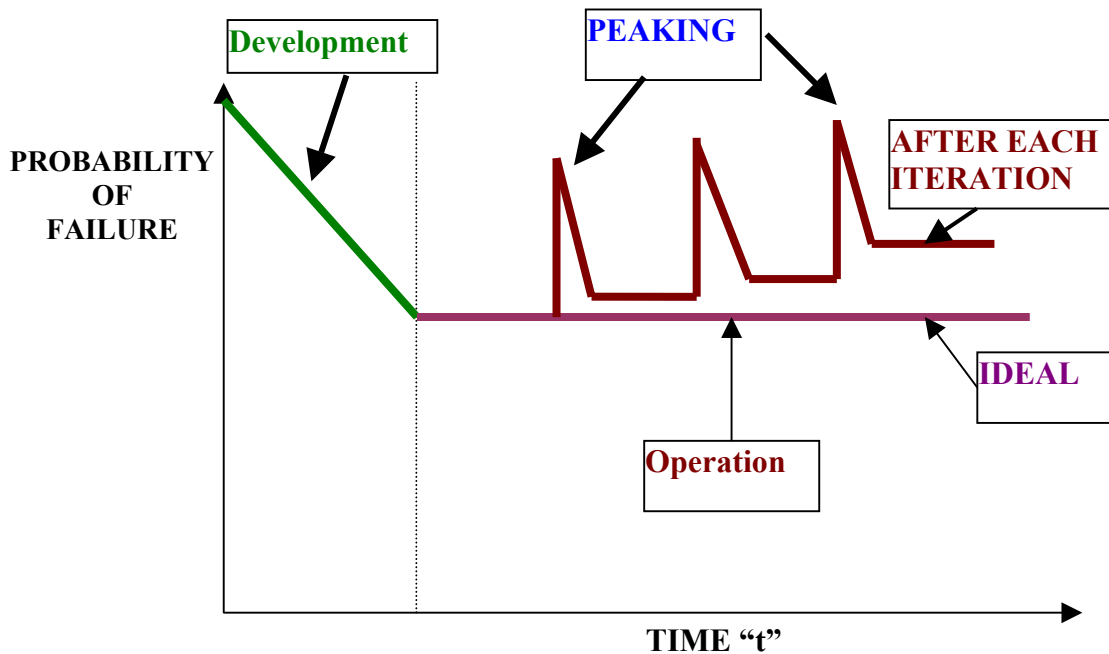


Figure 6.2 Intensity of Failure in Normal development and Deployment

The major goal of the Software Reliability modeling is to predict the future value of metrics from the gathered failure data. A central problem in Software Reliability is in selecting a model. The predictive quality of a Software Reliability model may be drastically improved by using preprocessing of data. In particular, statistical tests have been designed to capture trends in data. Thus, reliability trend analysis allows the use of software reliability models that are adapted to reliability growth and stable reliability.

The failure intensity decay parameter \emptyset is considered as taking an unknown but fixed value. Two basic methods to estimate the value of \emptyset are the method of maximum

likelihood (ML) and the least square method. That is, we have to optimize with respect to θ an objective function and collected failure data to get a point estimate. Another standard procedure, namely, interval estimation, gives an interval of values as estimate of θ .

The problem of imperfect debugging may be addressed in a Bayesian framework. Reliability growth is captured through a deterministically non-increasing sequence of failure rates r_i . In a Bayesian framework, parameters of r_i are considered as random. Hence, with stochastically decreasing sequence of random variables r_i , which allows one to take into account the uncertainty on the effect of a corrective action can be dealt. Assume that each fault detected has a probability ρ to be removed from the software. The hazard rate after $(i - 1)$ repairs is $\lambda_{i-1} = \theta [N - \rho (i - 1)]$6.1

But the problem of eventual introduction of new faults is not addressed. Addressing the problem of eventual insertion of new faults is concerned with finite Non-Homogenous Poisson Process (NHPP) models.

The complexity of software is an influencing factor of the reliability attributed. Typically, the computation of Halstead and McCabe metrics, which are respectively program size and control flow measures, is performed. Most software complexity metrics is strongly related to the structure of software code. Thus, including a complexity factor in software reliability may be thought of as a first attempt to take into account the architecture of the software in reliability assessment.

Other empirical evidence suggests that the higher the test coverage, then the higher the reliability of the software would be. Thus, a model that incorporates information on

functional testing as soon as it is available is of value. This issue is addressed in an NHPP model proposed by Gokhale and Trivedi [6.10]. It consists of defining an appropriate parameterization of a finite NHPP model, which relates software reliability to the measurements that can be obtained from the code during functional testing. Let “a” be the expected number of faults that would be detected in a given infinite time testing. The intensity function λ is assumed to be proportional to the expected number of remaining failures:

$$\lambda(t)=[a-\Delta(t)] \phi(t) \dots\dots\dots 6.2$$

where $\phi(t)$ is the hazard rate per fault.

Finally, the time-dependent function $\phi(t)$ is of the form

$$\phi(t) = (dc(t)/dt) / (1 - c(t)) \dots\dots\dots 6.3$$

where $c(t)$ is the coverage function. It is, the ratio of the number of potential faults covered by time t divided by the total number of potential faults under consideration during testing. Function $c(t)$ is assumed to be continuous and monotonic as a function of testing time. Specific forms of function $c(t)$ allow the retrieving of some well-known finite failure models.

The software reliability models generally ignore the factors affecting software reliability. Imperfect debugging is related to the fact that new faults may be inserted during a repair. The complexity attributes of software are strongly correlated to its fault-proneness. Empirical investigations show that the development process, testing procedure, programmer skill, human factors, the operational profile and many others factors affect the reliability.

6.2 Types and Approaches

Software reliability measurement includes two types of model, namely, static and dynamic reliability estimation, used typically in the earlier and later stages of development respectively. A key use of the reliability models is in the area of when to stop testing. Two approaches are used in software reliability modeling. The most prevalent is the black-box approach, in which only the interactions of the software with the environment are considered. Self-exciting point processes as a basic tool to model the failure process. A second approach, called the white-box approach, incorporates information on the structure of the software in the models and proposes basic techniques for calibrating black-box models. Fault prevention, fault removal, fault tolerances are three methods to achieve reliable software. The reliability of the software is a measure of the continuous delivery of the correct service by the software under a specified environment. This is a measure of the time to failure. The first failure time is a random variable T with distribution function

$$F(t) = P\{T \leq t\} \quad t \in \mathbb{R} \quad \dots\dots\dots 6.4$$

If F has a probability density function f , then we define the hazard rate of the random variable T by

$$r(t) = \frac{f(t)}{R(t)} \quad t \geq 0 \quad \dots\dots\dots 6.5$$

with $R(t) = 1 - F(t) = P\{T > t\}$. Function $R(t)$ is called the survivor function of the random variable T . The hazard rate function is interpreted to be

$$r(t) dt \approx P\{t < T \leq t + dt | T > t\} \dots\dots\dots 6.6$$

$$\approx P\{a \text{ failure occurs in } [t, t + dt] \text{ given that no failure occurred up to time } t\}$$

Thus, the phenomenon of reliability growth is represented by a decreasing hazard rate.

When F is continuous, the hazard rate function characterizes the probability distribution of T through the exponentiation formula

$$R(t) = \exp \left(- \int_0^t r(s) ds \right) \dots\dots\dots 6.7$$

Finally, the mean time to failure (MTTF) is the expectation of the waiting time of the first failure. A basic way to represent time evolution with confidence in software is as follows:

At instant zero, the first failure occurs at time t_1 according to a random variable $X_1 = T_1$ with hazard rate r_1 . Given time $T_1 = t_1$ it is observed a second failure at time t_2 at rate r_2 . Function r_2 is the hazard rate of the inter-failure random variable $X_2 = T_2 - T_1$ given $T_1 = t_1$. The choice of r_2 is based on the fact that one fault was detected at time t_1 .

A third failure occurs at t_3 with failure rate r_3 . Function r_3 is the hazard rate of the random variable $X_3 = T_3 - T_2$ given $T_1 = t_1$, $T_2 = t_2$ and is selected according to the “past” of the failure process at time t_2 . The two observed failures at times t_1 and t_2 and so on. It is expected that, due to a fault removal activity, confidence in the software’s ability to deliver a proper service will be improved during its life cycle. Therefore, a basic model in Software Reliability has to capture the phenomenon of reliability growth. Reliability growth basically follows a sequence of inequalities of the following form

$$r_{i+1}(t - t_i) \leq r_i(t_i) \text{ on } t \geq t_i \dots\dots\dots 6.8$$

From selection of decreasing hazard rates r_i . It illustrates this “modeling process” on the Jelinski - Moranda model (JM). It is assumed that software includes only a finite number N of faults [6.11]. The first hazard rate is $r_1(t; \emptyset, N) = \emptyset N$ 6.9

where \emptyset is some non-negative parameter. From time $T_1 = t_1$, a second failure occurs with the hazard rate $r_2(t; \emptyset, N) = \emptyset (N-1)$ 6.10

In a more formal setting, the two parameters N and \emptyset will be encompassed as background history f_0 , which is any background information about the software. Then “the failure rate” is called the concatenated failure rate function. The graphical display of a path of this stochastic function is given in Figure 6.3.

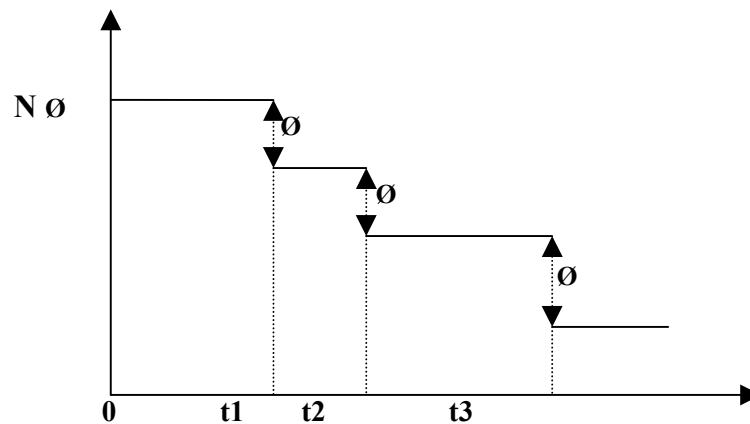


Figure 6.3 Concatenated failure rate function for Jelinski - Moranda mod

6.3 Static Model

One purpose of reliability models is to perform reliability prediction in an early stage of software development. This activity determines future software reliability based upon available software metrics and measures. Particularly when field failure data are not available (*e.g.* software is in the design or coding stage), the metrics obtained from the software development process and the characteristics of the resulting product can be used to estimate the reliability of the software upon testing or delivery. Two prediction models,

the phase-based model by Gaffney and Davis and a predictive development life cycle model from Dalal and Ho exists [6.11].

6.3.1 Phase-based Model: Gaffney and Davis

This model assumes that code size estimates are available during the early phases of development. Further, it assumes that faults found in different phases follow a Raleigh density function when normalized by the lines of code. This model is clearly motivated by the corresponding model used in hardware reliability, and the predictions are hardwired in the model based on one parameter. This model is one of the first to leverage information available in earlier development life cycle phases.

6.3.2 Predictive Development Life Cycle Model: Dalal and Ho

This model does not postulate a fixed relationship between the numbers of faults discovered during different phases. Instead, it leverages past releases of similar products to determine the relationships. The relationships are not postulated beforehand, but are determined from data using only a few releases per product. Similarity is measured by using an empirical hierarchical Bayes framework. The number of releases used as data is kept minimal , typically, only the most recent one or two releases are used for prediction.

The basic assumptions behind this model are as follows:

- (i). Defect rates from different products in the same product life cycle phase are samples from a statistical universe of products coming from that development organization.
- (ii). Different releases from a given product are samples from a statistical universe of releases for that product.

The first assumption reflects the fact that the products developed within the same

organization at the same life cycle maturity are more or less homogeneous.

6.4 Dynamic Models: Reliability growth models for testing and operational use

Software reliability estimation determines the current software reliability by applying statistical inference techniques to failure data obtained during system test or during system operation. Since reliability improves over time during the software testing and operation periods because of removal of faults, the models are also called reliability growth models.

6.4.1 A General Class of Models

In binomial models the total number of faults is N , the number found by time t has a binomial distribution with mean $\mu(t) = NF(t)$, where $F(t)$ is the probability of a particular fault being found by time t . Thus, the number of faults found in any interval of time (including the interval (t, ∞)) is also binomial [6.10]. $F(t)$ could be any arbitrary cumulative distribution function. Then, a general class of reliability models is obtained by appropriate parameterization of $\mu(t)$ and N .

Letting N be Poisson (with some mean ν) gives the related Poisson model, wherein the number of faults found in any interval is Poisson, and for disjoint intervals these numbers are independent. Denoting the derivative of F by F' , the hazard rate at time t is

$$r_t = F'(t) / [1 - F(t)] \dots\dots\dots 6.11$$

These models are Markovian but not strongly Markovian, except when F is exponential.

6.4.2 Assumptions Underlying the Reliability Growth Models

Most of the models are based on common underlying assumptions as listed

1. The system being tested remains essentially unchanged throughout testing, except for

the removal of faults as they are found.

2. Removing a fault does not affect the chance that a different fault will be found.
3. The model is Markovian. The future evolution of the testing process depends only on the present state (the current time, the number of faults found and remaining, and the overall parameters of the model) and not on the past history.
4. All faults are of equal importance (contribute equally to the failure rate).
5. At the start of testing, there is some finite total number of faults, which maybe fixed or random. If random, its distribution may be known or of known form with unknown parameters. Alternatively, the “number of faults” is not assumed finite, so that, if testing continues indefinitely, an ever-increasing number of faults will be found.
6. Between failures, the hazard rate follows a known functional form. This is a constant.

6.5 Software Reliability Growth Modeling

6.5.1 A Generalized Non-homogeneous Poisson Process Model

The only environmental factor available in this application is the testing team size. Team size is one of the most useful measures in the software development process. It has a close relationship with the testing effort, testing efficiency, and the development management issues. From the correlation analysis of the 32 environmental factors, team size is the only factor correlated to the program complexity, which is the number one significant factor according to our environmental factor study [6.11]. Since the testing team size ranges from one to eight, we categorize the team size factor two levels as follows and denoted as z_1 .

$$z_1 = \begin{cases} 0 & \text{team size ranges from 1 to 4} \\ 1 & \text{team size ranges from 5 to 8} \end{cases} \dots\dots\dots 6.12$$

6.6 A Reliability Model with Considerations of Random Field Environments

Once a software product is released, it can be used in many different locations, applications, tasks and industries. The field environments for the software product are quite different from place to place. Therefore, the random effects of the end-user environment will affect the software reliability in an unpredictable way. Software reliability model with consideration of random field environments unified software reliability model that covers both testing and operation phases in the software life cycle. The model also allows to remove faults if a software failure occurs in the field and can be used to describe the common practice of so-called 'beta testing' in the software industry. During beta testing, software faults will still be removed from the software after failures occur, but beta testing is conducted in an environment that is the same as the end-user environment, and it is different from the in-house testing environment. The quality of the software depends on how much time testing takes and what testing methodologies are used. Figure 6.4 shows the entire software development life cycle considered in this software cost model, the in-house testing, beta testing, and operation. The beta testing and operation are conducted in the field environment, which is commonly quite different from the environment where the in-house testing is conducted.

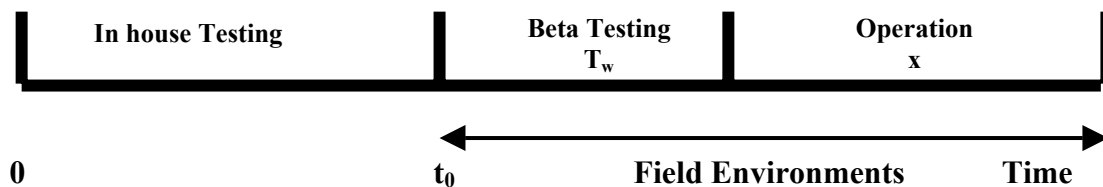


Figure 6.4 Testing in software development cycle

Let us consider the following.

1. There is a set-up cost at the beginning of the in-house testing, it is assumed constant.
2. The cost to do testing is a linear function of in-house testing time.

3. The cost to remove faults during the in-house testing period is proportional to the total time of removing all faults detected during this period.
4. The cost to remove faults during the beta-testing period is proportional to the total time of removing all faults detected in the time interval $[t_0, t_0 + T_w]$.
5. It takes time to remove faults and it is assumed that the time to remove each fault follows a truncated exponential distribution.
6. There is a penalty cost due to the software failure after formally releasing the software.

As we move in the life of the software the number of failures keeps on decreasing as shown in Figure 6.5 as per reliability growth model where rigorous V&V is carried out during development and rigorous impact analysis after every updating of the software.

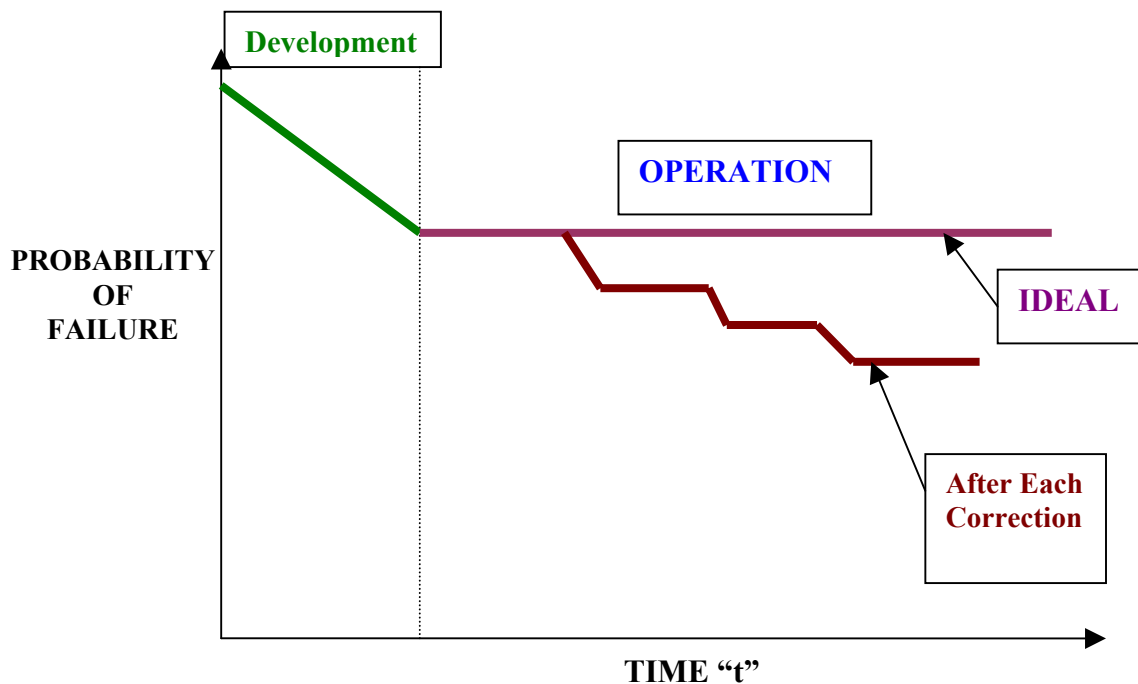


Figure 6.5 Intensity of Failure in Reliability Growth model

6.7 Precautions in Using Reliability Growth Models

In fitting any model to a given data set, due consideration to be given for the assumptions.

For example, a model may assume that a fixed number of software faults will be removed

within a limited period of time. But in the observed process the number of faults is not fixed (*e.g.* new faults are inserted due to imperfect fault removal, or new code is added), then one should adopt a model that does not suffer from this assumption. Then the limitation and implementation issue of the model concerns future predictions. If the software is being operated in a manner different from the way it is tested the failure history of the past will not reflect these changes, and poor predictions may result. Then another issue is related to the software development environment. Most reliability growth models are primarily applicable from testing onwards. The software is assumed to have matured to the point that extensive changes are not being made.

6.8 Reliability Growth Modeling with Covariates

It is testing process, for all but small systems involving short development and life cycles. For large systems there are variables, other than time, that are very relevant. For example, it is typically assumed that the number of faults (found and unfound) in a system under test remains stable during testing. This implies that the code remains frozen during testing. However, this is rarely the case for large systems, since aggressive delivery cycles force the final phases of development to overlap with the initial stages of system test. Thus, the size of code and the number of faults in a large system can vary widely during testing. If these changes in code size are not considered as a covariate, then it is likely to have an increase in variability. The associated a loss in predictive performance resulting in a poor fitting model with unstable parameter estimates.

6.9 Time to Stop Software Testing

Dynamic reliability growth models can be used to make decisions about when to stop testing. Software testing is a necessary but expensive process, consuming one-third to one-half the cost of a typical development project [6.12]. More testing can lead to a product

that is overpriced and late to market, whereas fixing a fault in a released system is usually an order of magnitude more expensive than fixing the fault in the testing laboratory [6.13]. Thus, the question of how much to test is an important economic parameter.

6.10 Inferences

The Various reliability models were discussed in this section and it is found that “Reliability Growth Model” is suitable for NPP software. This model is very much applicable because of rigorous verification and validation as well as the impact analysis during software maintenance. It is observed in the practice of software for NPP that the reliability increases after correction because of the many mandatory criteria imposed on the safety. Figure 6.6 shows the system Un-availability X_{UA} with respect to time when we adopt the reliability growth models [6.14].

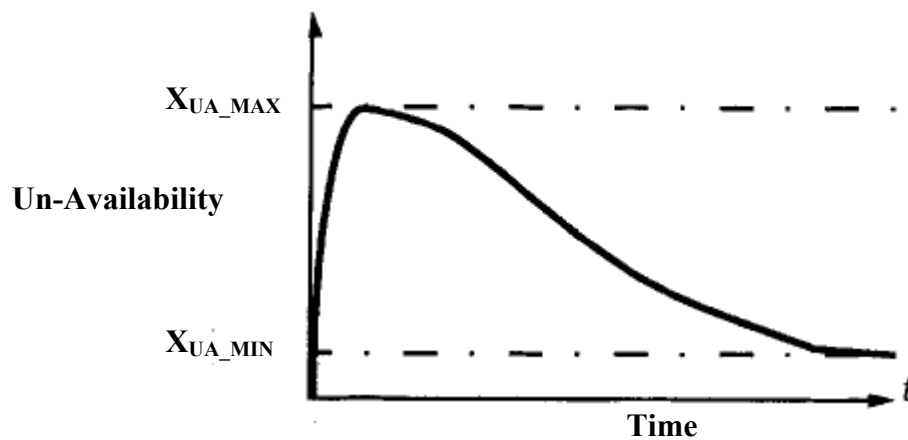


Figure 6.6 Typical System Un-Availability

REFERENCES

- [6.1] Software reliability and safety in Nuclear reactor protection systems by J. Dennis Lawrence for U.S Nuclear Regulatory Commission. UCRL-ID-114839.
- [6.2] Software Engineering Measures for Predicting Software Reliability in Safety

- Critical Digital Systems.NUREG/GR-0019, University of Maryland, U.S. Nuclear Regulatory Commission, Washington, DC 20555-0001.
- [6.3] E. Nelson, Estimating Software Reliability from Test Data, Microelectronics and Reliability, Vol 17, pp. 67- 74, Pergamon, New York, 1978.
 - [6.4] J.B.Dugan, M.Lyu. Dependability Modeling for Fault Tolerant Software and Systems, Software Fault Tolerance, John Wiley & Sons Ltd., 1995.
 - [6.5] R.K. Scott, J. W. Gault, D. F. McAllister, Fault-Tolerant Software Reliability Modeling, IEEE Transactions on Software Engineering, vol. SE-13, No.5, May 1987.
 - [6.6] CV Ramamoorthy, FB Bastani, Software Reliability—Status and Perspectives, IEEE Transactions on Software Engineering, 1982.
 - [6.7] J. E. Gaffney, C. F. Davis, An approach to estimating software errors and availability, SPC-TR88-077, version 1.0, March 1988.
 - [6.8] Rome Laboratory (RL), Methodology for Software Reliability Prediction and Assessment, Technical Report RL-TR-92-52, Vol. 1 and 2, 1992.
 - [6.9] M. Stutzke, C. Smidts, A Stochastic Model of Fault Introduction and Removal during Software Development, Probabilistic Safety Assessment and Management -PSAM'4, Vol 1, Springer, pp 1111-1116, 1998.
 - [6.10] Fundamentals of Probability and Statistics for Engineers, T.T.Soong, State university of New York, USA, John wiley and sons Ltd, ISBN –0-470-86913-7, 2004.
 - [6.11] Handbook of Reliability Engineering by Hoang Pham, Springer - Verlag London limited press, ISBN-I-85233-453-3, 2003.

- [6.12] Managing the software process, Watts S. Humphery, SEI series in software engineering, Addison Wesley Longman Inc, ISBN-981-235-916-8, 1999.
- [6.13] Sommerville, I., Software Engineering, Pearson Education, 6th Ed., 2001.
- [6.14] Handbook of Software Reliability Engineering, Edited by Michael R. Lyu, Published by IEEE Computer Society Press and McGraw-Hill Book Company
<http://www.cse.cuhk.edu.hk/~lyu/book/reliability/>.

CHAPTER 7. ESTIMATION OF SOFTWARE RELIABILITY

7.1 Introduction

This chapter explains the step by step procedure involved in calculating software reliability for the safety systems of NPP. A comprehensive procedure to estimate software reliability for a given software program coded in “C” language is detailed. The safety systems of NPP are embedded systems, implemented in “C” language. The software metrics required for each function is obtained by running the “Static Analyser” tool developed as a part of this research or by any Computer Aided Software Engineering tools. It is also assumed that the software has undergone vigorous V&V procedure developed as part of this research, which is detailed in chapter 5. It is also assumed that the subsequent version improves reliability i.e. changes fixes the bug or new functionality added without introducing error in the existing working code. In view of this the reliability growth model is assumed.

It is from the experience of the software deployed in NPP and Heavy water plants, a set of parameters influencing the software reliability is derived. The reliability estimation is based on the following parameters:

- a) Lines of code
- b) Number of functions
- c) Cyclomatic Complexity
- d) Local and Global Variables
- e) Nesting Levels
- f) Number of times functions invoked
- g) Number of distinct operators
- h) Dynamic Memory usage

- i) Adherence to the standards
- j) V&V iterations - Versions
- k) Comment to code ratio
- l) Number of interrupts
- m) Criticality of the software (Safety Critical, Safety Related and Non nuclear Safety)

7.2 Reliability Estimation

This procedure is evolved with the data collected on the successful operation of software systems from operating nuclear power plants and other Atomic Energy establishments like Heavy Water Plants over the last two decades. This procedure involves basically five steps. The detailed explanation of these steps is as follows:

Step 1: List out the “functions” used in the software

The “C” language basically deals with only the “functions” as its basic sub-units of the program. The first step for calculating reliability starts with finding out the “functions” for the complete project. The complete project includes Source files, Library files and Header files. For the software deployed in safety systems the complete source code is available including library and header files. All the necessary library functions are developed instead of using object files so that it is amenable for Verification and Validation at source code level.

List out the names of all the functions used in the complete software project including the function “main ()”

Step 2: Measure the parameters and tabulate for all the functions

The parameters pertaining to each function of the complete software is measured using the “Static Analyser”. It is basically the software metrics pertaining to each of the functions are evaluated and tabulated. These parameters directly influence the reliability of the software. In terms of severity of parameter’s influence on reliability, it is categorized as different levels with “Level 0” as the highest severity of levels are tabulated in Table 7.1. The next sequence is to measure the following parameters and fill up the Table 7.2.

The parameters identified are

a) Lines of Code

Less the number of lines of code, it will be easier to understand, test and maintain. On the other hand, when the code is lengthy and complex it is prone to error. So the number of code lines should be less to improve the reliability. This forms the basis for the base failure rate of the function.

b) Cyclomatic Complexity (CC)

This attribute measures the complexity of each function in terms of number of independent paths, i.e., number of branches. Less the number of branches, better is the reliability. It is applied to individual functions within a program. The number of “test cases” in the Basis Path Testing strategy is proportional to the cyclomatic complexity of the program. This falls in the category of Level 0, which means that its contribution is significant.

c) Local Variables

Usage of more variables results in occupying more memory and particularly the stack in case of usage of “static” variables under Local Variables. Usage of local variables is preferred compared to Global Variables because of vulnerability in affecting and becomes concern in “Impact Analysis” on top of being occupying memory throughout the life of the

program. This falls in the category of Level 3, which means that its contribution is minimum in affecting the reliability.

d) Global Variables

It is always better to minimise global variables but still some values are required to be of scope covering the complete project. Usage of local variables is preferred compared to Global Variables because of vulnerability in affecting and becoming concern in “Impact Analysis” on top of being occupying memory throughout the life of the program. This falls in the category of Level 2, which means that its contribution is moderate and affects reliability to some extent.

e) Number of times invoked / used

This parameter is to give weight-age as per the usage of the functions. Less the frequency of usage less will be the combination of inputs that result in better reliability. This falls in the category of Level 3, which means that its contribution is minimal in affecting the reliability.

f) Level of nesting

The depth of “condition branches” or the “loops” is the measure for each function. When the depth is less, the reliability is better, number of test cases is less and hence, the testability and coverage are good. This falls in the category of Level 1 which means that its contribution is large and affects reliability to large extent.

g) Dynamic Memory usage (Number of times)

Dynamic memory allocation also known as heap-based memory allocation is the allocation of memory storage for use during the run-time. It can also be seen as a way of distributing ownership of limited memory resources among many pieces of data and code. Hence, it is vulnerable for memory overflow resulting in failure. When the number of occurrences is

more it is more vulnerable for failure resulting in decrease in reliability. This falls in the category of Level 1 which means that its contribution is large and affects reliability to a large extent.

h) Number of distinct operators

The complexity of arithmetic is directly related to the number of distinct operators used in the algorithms. As the number of operators increases the reliability decreases. This falls in the category of Level 3, which means that its contribution is minimal in affecting the reliability.

The Π factors are derived depending on their “Category of Level” assigned to the parameters. Table 7.1. Shows the mapping of level to Π factor equation. In the next step using this table Π factors are evaluated and used in calculation for its contribution to reliability.

Table 7.1 Mapping of Π factors to “Category of Levels”

Severity Levels	Π factor equation	Description
0	$1.0 \times \text{Parameter}$	Significant
1	$1 + (1.0 \times \text{Parameter})$	Large
2	$1 + (0.1 \times \text{Parameter})$	Moderate
3	$1 + (0.01 \times \text{parameter})$	Minimum
4	$1 / (1 + (0.01 * \text{parameter}))$	Small improvement
5	$1 / (1 + (0.1 \times \text{parameter}))$	Large improvement

Table 7.2 Template for Quality Parameters of Function

Function Name	Lines Of code	Cyclomatic Complexity	Local Variable	Global Variable	Times used	Nesting	Dyn. Memory	operators
Func 1								
Func 2								
Func 3								
Func 4								
Func 5								

Step 3: Apply the equation to calculate failure per combination for all the functions

The equation for failure of a function per input combination is defined as Φ_f

The Φ_f is calculated as, $\Phi_f = \Phi_b \Pi_C \Pi_{LV} \Pi_{GV} \Pi_{TU} \Pi_N \Pi_D \Pi_O$ (7.1)

where

- Φ_b Base failure / Combination = Number of lines x 10^{-8} For Non Nuclear Safety
= Number of lines x 10^{-9} For Safety Related systems
= Number of lines x 10^{-10} For Safety Critical systems
- Π_C Cyclomatic Complexity Factor = 1.0 x Cyclomatic Complexity
- Π_{LV} Local Variable Factor = 1 + (0.01 x Number of Local Variable)
- Π_{GV} Global Variable Factor = 1 + (0.1 x Number of Global Variable)
- Π_{TU} No. of Times Used Factor = 1 + (0.01 x Number of Times used)
- Π_N Nesting Factor = 1 + (1.0 x Nesting Level)
- Π_D Dynamic Memory Factor = 1 + (1.0 x Dynamic Memory usage)
- Π_O Operators Factor = 1+(0.01 x number of operators used)

The weights for each factor is arrived from analyzing around fifty software modules of fast reactor and with the methodology applied in the organization. If necessary these factors can be tuned as per the organization methodology and its perception. The Failure / Combination Φ_f of all the functions shall be calculated and tabulated as in the Table 7.3.

Table 7.3 List of Failures for functions

Function Name	Failure / Combination $\Phi_f \times 10^{-10}$
Func 1	
Func 2	
Func 3	
Func 4	

Step 4: Apply the equation to calculate failure per combination for software

Now for the complete software the failure is calculated by arriving at subtotal and then by multiplying the common factor affecting the complete software to achieve the total failure / input combination. The sub-total of failure / input combination is calculated using the Equation 7.2 by summing up all the functions failures since all the functions have to operate to achieve the required functionality of the software.

$$\Phi_{\text{SubTotal(ST)}} = \sum_{i=1}^N \Phi_{fi} \dots\dots\dots(7.2)$$

Once the sub-total is calculated the total failure / input combination for the complete software is calculated using Equation 7.3 by normalizing with the common quality factors pertaining to the project.

$$\Phi_{TOTAL} = \Phi_{ST} \Pi_{Ver} \Pi_{CC} \Pi_{STD} \Pi_{int} \dots\dots\dots(7.3)$$

Where

$$\Pi_{Ver} \text{ Version Number factor} = 1 / (1 + (0.1 * \text{number of versions}))$$

$$\Pi_{CC} \text{ Comment to Code Factor} = 1 / (1 + (0.1 * \text{percentage of comment to code}))$$

$$\Pi_{CC} \text{ Conformance to Standards Factor} = 1 + (10 - (10 * (0.01 * \text{percentage of conformance})))$$

$$\Pi_{int} \text{ Number of Interrupts used factor} = 1 + 10 * \text{Number of interrupts}$$

Once the sub total is calculated, the factors which are affecting / increasing the reliability are considered to arrive the total failure / input combinations. The global factors are

- a) Version Number : As the version number increases, the new releases covers the “bug” fixing and covers the requirements overlooked. In essence the new releases increase the reliability to certain extent.
- b) Comment to Code Ratio : The Comments in the code is like document embedded in the code. This helps in large extend to understand the algorithm and flow of the software
- c) Conformance to Standards: The degree of conformance to standards (like MISRA) greatly enhances reliability because we will be avoiding error prone and complex way of expression. It also facilitates good practices, which can trap invalid combinations (like “default” in Case statement).
- d) Number of Interrupts: As the number of interrupts used in the software increases, the software becomes complex. It becomes difficult to do the testing since the simulation of asynchronous interrupts during the normal courses of action and also simultaneous

triggering of multiple interrupts. Hence the reliability tends to decreases as the number of interrupts increases.

Step 5: Estimation of reliability

The reliability over the number of input combinations is given by

$$R(c) = e^{-(\Phi * COMBINATIONS)} \dots\dots\dots(7.4)$$

7.3 Typical Estimation for Safety Critical System

As a case study, the software developed for a fast reactor, which does the supervision of sodium temperature at inlet and outlet of fuel subassemblies and initiating shutdown of the nuclear plant on finding abnormalities is taken up. This software falls in the category of Safety Critical System. The step by step procedure for estimation of the software reliability is shown next

Step 1: List out the functions used in the software

The Safety Critical System software has the following functions

main	Main module
scan	Scan Module
median	To Calculate Median
ptrs	Pre Treatment Sub Routine
mvsr	Mean Value Sub Routine
mgssr	Mean Gradient Subroutine
spcs	Supervision of Core Subassembly
init	Initialization

Step 2: Measure the parameters and tabulate them for all the functions

The parameters are calculated for each functions and filled in the Table 7.3. Then Φ_f is calculated using Eq. 7.1. and filled in Table 7.4 as explained in step 3.

Table 7.4 Software Failure Calculation for function

Function Name	Lines Of code	CC	Local Var	Global Var	Times used	Nesting	Dyn. mem	operators	Failure / combination $\Phi_f \times 10^{-10}$
Init	26	2	1	21	1	1	0	4	342.0354
Scan	240	36	19	10	1	2	0	10	68537.1456
Median	55	11	6	6	1	1	0	9	2259.2229
Ptsr	35	4	7	8	1	1	0	8	588.2466
Mvsr	87	12	8	17	1	2	0	7	6579.8586
Mgsr	63	7	4	20	1	2	0	5	13002.4440
Spcs	78	12	2	15	1	2	0	6	7665.9242
Main	98	8	6	20	1	3	0	5	10575.8150

CC – Cyclomatic Complexity

Sub Total 108550.9623 x 10⁻¹⁰

Step 3: Apply the equation to calculate failure / combination for all the function

The equation for failure of a function per input combination is computed as Φ_f

$$\Phi_f = \Phi_b \Pi_C \Pi_{LV} \Pi_{GV} \Pi_{TU} \Pi_N \Pi_D \Pi_O$$

For the “Init” function

$$\Phi_b = 26 \times 10^{-10} \quad \Pi_C = 1.0 \times 2 \quad \Pi_{LV} = 1 + (0.01 \times 1)$$

$$\Pi_{GV} = 1 + (0.1 \times 21) \quad \Pi_{TU} = 1 + (0.01 \times 1) \quad \Pi_N = 1 + (1.0 \times 1)$$

$$\Pi_D = 1+(1.0 \times 0) \quad \Pi_O = 1+(0.01 \times 4)$$

Φ_f of “Init” function is 342.0354×10^{-10} failures / combination

In the same way, Φ_f is calculated for all the functions and entered on to the Table 7.4.

Step 4: Apply the equation to calculate failure / combination for software

Now for the complete software the failure is calculated by arriving at subtotal and then by multiplying the common factor affecting the complete software to achieve the total failure / input combination as

$$\Phi_{\text{SubTotal(ST)}} = \sum_{i=1}^N \Phi_{fi} = 108550.9623 \times 10^{-10} \text{ failures / combination}$$

Once the sub-total is calculated the total failure / input combination for the complete software is calculated using Eq. 7.3 by normalizing with the common quality factors.

$$\Phi_{\text{TOTAL}} = \Phi_{\text{ST}} \Pi_{\text{Ver}} \Pi_{\text{CC}} \Pi_{\text{STD}} \Pi_{\text{Int}}.$$

The Safety Critical System software has the data as follows:

$$\text{Number of Versions} = 39 \quad \text{Comment to Code Ratio} = 63\%$$

$$\text{Conforming to Standards} = \text{Fully, 100 \%} \quad \text{No. of Interrupts} = \text{Nil}$$

$$\Phi_{\text{TOTAL}} = 108550.9623 \times 10^{-10} * [1.0/(1+(0.1*39))] * [1.0/(1+(0.1*63))] * [1+(10-10*(0.01*100))] * [1+(10 * 0)]$$

$$\Phi_{\text{TOTAL}} = 108550.9623 \times 10^{-10} * [0.204] * [0.136] * [1.0] * [1] \\ = 3034.6928 \times 10^{-10} \text{ failures / combination}$$

Step 5: Estimation of reliability

The reliability over the number of input combinations is given by Eq. 7.4

$$R(c) = e^{-(\Phi * \text{COMBINATIONS})}$$

The Table 7.5 lists the estimated reliability based on the number of input combinations for the Φ_{TOTAL} of 3034.6928 failures / combination with the variation of criticality levels as Safety Critical, Safety Related and Non-nuclear safety. The reliability of the sample system for 5000 combinations becomes $R(5000) = 0.9984838041$

Table 7.5 Estimated reliability based on the number of input combinations

Reliability with input Combinations	Safety Critical (SC1)	Safety Related (SR)	Non Nuclear Safety (NNS)
R(100)	$e^{-(0.0000003034 * 100)}$ = 0.9999696535	$e^{-(0.000003034 * 100)}$ = 0.9996965767	$e^{-(0.00003034 * 100)}$ = 0.9969699072
R(1000)	$e^{-(0.0000003034 * 1000)}$ = 0.9996965767	$e^{-(0.000003034 * 1000)}$ = 0.9969699072	$e^{-(0.00003034 * 1000)}$ = 0.970115638
R(5000)	$e^{-(0.0000003034 * 5000)}$ 0.9984838041	$e^{-(0.000003034 * 5000)}$ = 0.9849410729	$e^{-(0.00003034 * 5000)}$ = 0.859246015
R(10000)	$e^{-(0.0000003034 * 10000)}$ 0.9969699072	$e^{-(0.000003034 * 10000)}$ = 0.97010891721	$e^{-(0.00003034 * 10000)}$ = 0.738303715

The typical safety critical software deployed in the reactor is taken for reliability estimation. This software basically does the core supervision, initiates alarm if the value exceeds the Alarm limits. It also gives command to shutdown whenever the parameter is approaching to design limits. These functions help to keep the core integrity and safe operation of the plant. The version history of the software is tabulated in Table 7.6 and 7.7

Table 7.6 Software Version history of SCS – Part I

Version No. & Date	obs	Description of important observations(obs)
31.5.2005 Ver 1.0	30	Comments were less - 23%, Three Functions Cyclomatic Complexity (CC) > 15, One function Nesting is greater than 5, Initialisation of some variables missing, Multiple Returns in functions, Usage of Library, Power Function is wrong i.e $10^{1.9} = 79.4$ instead of 10, Compute was wrong, cr_count ++ - No Reset keeps on incrementing
7.1.2006 Ver 4.0	20	Comments 21%, CC > 15 : 5 functions, Nesting > 3 : 2 functions, unused variables, Multiple Returns, Library Used Float equality check
17.1.2006 Ver 4.1	15	Comments 46%, CC > 15 : 3 functions, Nesting>3 : 2 functions, unused code, Library Used, typecasting
2.2.2006 Ver 4.2	10	Comments 46%, CC > 15 : 3 functions, Nesting>3 : 1 fn, Library Used, Hard-coding of numbers
14.2.2006 Ver 4.21	5	Comments 46%, CC > 15 : 3 functions, Nesting>3 : 1 function, Library Used

Table 7.7 Software Version history of SCS – Part II

Version No. & Date	obs	Description of important observations(obs)
12.3.2007 Ver 4.21	5	Comments 46%, CC > 15 : 3 functions, Nesting>3 : 1 function
13.1.2009 Ver 4.3 Constants modified	5	Comments 46%, CC > 15 : 3 functions, Nesting>3 : 1 function, Unused Macros & Variables
30.1.2009 Ver 4.4 LOR Modification	3	Comments 46%, CC > 15 : 3 functions, Nesting>3 : 1 function, This version is temporary
2.9.2009 Ver 4.3.1 New 15 Channel ROP	3	Comments 46%, CC > 15 : 3 functions, Nesting>3 : 1 function
13.12.2011 Ver 4.3.1 1 SPGS Group added	3	Comments 46%, CC > 15 : 3 functions, Nesting>3 : 1 function
10.2.2012 Ver 4.4 Removal of FIT	2	CC > 15 : 2 functions, Nesting >2 : 1 function Ref. Tolerance count is high
15.1.2013 Ver 4.4.1 Test SA modification	2	CC > 15 : 1 function, Nesting>2 : 1 function This version is temporary
17.10.2013 Ver 4.5 Pac Assembly added	2	CC > 15 : 1 function, Nesting>2 : 1 function Separate PDSR for sphere Pac Fuel Sub Assembly

7.4 Inferences

A comprehensive procedure to estimate software reliability for a given software program coded in “C” language has been established. The data required for each function is obtained by running the “Static Analyser” tool developed as a part of this research.

CHAPTER 8. CONCLUSIONS AND FUTURE DIRECTIONS

8.1 Conclusions

The present research work is directed towards development of a robust methodology for developing reliable software and estimation of reliability of software used in safety systems for nuclear reactor.

In the process of developing reliable software, the following artifacts are developed.

- ❑ Development of software life cycle model for safety systems – “V” model is developed to ensure the development process free from un-intentional errors with the checking mechanism at each stage.
- ❑ Determination of software metrics and development of software metric tools – Listing out the software metrics which affect the quality of the software and development of a “Static Analyser” tool to evaluate the quality parameters which have bearing on the reliability.
- ❑ Development of software Verification and Validation methodology – The complete methodology as step-by-step procedure to practice Verification and Validation of software is evolved to ensure the software reliability.

Although the most important sources of information in predicting software reliability are known as software engineering measures, limited study systematically demonstrates how software-engineering measures determine software reliability. The study in this report was a constructive attempt towards the establishment of a relationship between software engineering measures and software reliability.

In the process of Estimating Reliability of the software the following artifacts are developed.

- ❑ Development of Software Reliability Model – The software reliability growth model suitable for safety systems of nuclear reactor has been developed.
- ❑ Procedure for estimation of software reliability – A comprehensive procedure to estimate software reliability for a given software program coded in “C” language has been established.

Robustness and validation of the methodology has been demonstrated by applying it to software deployed in safety critical and safety related systems of fast reactor.

8.2. Scope For Future Work

Strong presence of human element and variations among individuals are envisaged in software development in particular during code development. The unreliability attached with the Human Error Probability is assumed to be minimal which is basically the boundary within which the research has been carried out.

Generally the application software for Human Machine Interface are developed using object oriented concepts. The same methodology can be extended for “Object Oriented Programming” considering the impact of Class, polymorphism and inheritance. It can also be extended for standard reuse library with which the object oriented system works.

APPENDIX – I: Definitions and Abbreviations

Definitions

Acceptance testing: Testing conducted in an operational environment to determine whether a system satisfies its acceptance criteria (i.e., initial requirements and current needs of its user) and to enable the customer to determine whether to accept the system.

Component testing: Testing conducted to verify the correct implementation of the design and compliance with program requirements for one software element (e.g., unit, module) or a collection of software elements.

Criticality: A subjective description of the intended use and application of the system. Software criticality properties may include safety, security, complexity, reliability, performance, or other characteristics.

Criticality analysis: A structured evaluation of the software characteristics (e.g., safety, security, complexity, performance) for severity of impact of system failure, system degradation, or failure to meet software requirements or system objectives.

Designer: The agency that generates the detailed specifications of the system interacts with the developer, manufacturer and regulator and is responsible for the final acceptance of the system.

Developer: The agency that does all project management activities, designs the system architecture, carries out detailed development and testing of the system, generates all relevant documents, has the relevant documents subjected to Quality Analysis and internal verification and carries out need based interaction with manufacturer and regulator.

Integration testing: An orderly progression of testing of incremental pieces of the software program in which software elements, hardware elements, or both are combined and tested until the entire system has been integrated to show compliance with the program

design, and capabilities and requirements of the system.

Interface Design Document (IDD): Documentation that describes the architecture and design of interfaces between system and components. These descriptions include control algorithms, protocols, data contents, formats, and performance.

Interface requirement specification (IRS): Documentation that specifies requirements for interfaces between systems, which include constraints on formats and timing.

Life cycle process: A set of inter-related activities that result in the development or assessment of software products. Each activity consists of set of tasks. The life cycle processes may overlap one another. For V&V purposes, no process is concluded until its products are verified and validated according to the defined tasks as per procedure.

Minimum tasks: Those V&V tasks required for the software safety level assigned to the software to be verified and validated.

Optional tasks: Those V&V tasks that may be added to the minimum V&V tasks to address specific application requirements.

Required inputs: The set of items necessary to perform the minimum V&V tasks mandated within any life cycle activity.

Required outputs: The set of items produced as a result of performing the minimum V&V tasks within any life cycle activity.

Software Design Description (SDD): A representation of software created to facilitate analysis, planning, implementation and decision-making. The software design description is used as a medium for communicating software design information and may be thought of as a blueprint or model of the system.

Software Requirements Specification (SRS): Documentation of the essential requirements (i.e., functions, performance, design constraints and attributes) of the

software and its external interfaces. The software requirements are derived from the system specification.

Software Verification and Validation Plan (SVVP): A plan describing the conduct of software V&V.

Software Verification and Validation Report (SVVR): Documentation of V&V results and software quality assessments.

System testing: The activities of testing an integrated hardware and software system to verify and validate whether the system meets its original objectives.

Test case: Documentation that specifies input, predicted results, and a set of execution conditions for a test item.

Test design: Documentation that specifies the details of the test approach for a software feature or combination of software features and identifying the associated tests.

Test plan: Documentation that specifies the scope, approach, resources, and schedule of intended testing activities.

Test procedure: Documentation that specifies a sequence of actions for execution of tests.

Validation: The process of verifying whether the software was built as per the requirements or not is called as Validation, e.g., system and integration testing, user acceptance-testing etc.

Verification: The process of verifying whether the software is being built as per the requirements of the previous phase or not is called as verification, e.g, peer review, unit test, code walkthrough, software inspection etc.

Verification and Validation Committee (V&VC): V&V processes performed by an organization with a specified degree of technical, managerial, and financial independence from the development organization.

Abbreviations

The following abbreviations appear in this document:

AERB	: Atomic Energy Regulatory Board
AC&V	: Air Conditioning and Ventilation
COTS	: Commercial Off The Shelf
IDD	: Interface Design Document
IEC	: International Electro-technical Commission
IRS	: Interface Requirements Specification
IV&V	: Independent Verification and Validation
MVF	: Mean value function
NNS	: Non Nuclear Safety
PFBR	: Prototype Fast Breeder Reactor
QA	: Quality Assurance
SC-I / SCS	: Safety Class I / Safety Critical Systems
SC-II / SR	: Safety Class II / Safety Related Systems
SDD	: Software Design Description
SRE	: Software reliability engineering.
SRGM	: Software reliability growth model
SRS	: Software requirement specification
SSE	: Sum of squared errors
SVVP	: Software Verification and Validation Plan
SVVR	: Software Verification and Validation Report
V&V	: Verification and Validation

APPENDIX – II: Checklist for System Requirement Review

Sl. No.	Items to be examined	Yes/ No/ NA	Comments of the examiner
1.0	General		
1.1	Does the system requirements bring out the following		
1.1.1	Role of the system in nuclear power plants		
1.1.2	Salient features		
1.1.3	Safety Class (SCS, SRS, NNS)		
1.1.4	Whether context diagram exists		
1.2	Whether various operating modes are listed		
1.3	Are the requirements free of duplication and conflict with other requirements?		
1.4	Is each requirement written in consistent, clear and concise language?		
1.5	Does each requirement have only one interpretation? If a term could have multiple meanings, is it defined?		
1.6	Is each requirement verifiable by testing, demonstration, review, or analysis?		
1.7	Are there measurable acceptance criteria for all functional & non-functional requirement?		
1.8	Is each requirement uniquely and correctly		

	identified?		
1.9	Is each requirement traceable to its source (including derived requirements)?		
1.10	Is each requirement in scope for the project?		
1.11	Have appropriate requirements documentation standards been followed?		
1.12	Are all figures, tables, and diagrams labeled and referenced?		
1.13	Are all terms and units of measure defined?		
1.14	Has full life cycle support been addressed, including maintenance?		
2.0	Functional Requirements		
2.1	Are the requirements unambiguous?		
2.2	Are they consistent?		
2.3	Are they verifiable?		
2.4	Whether all inputs required to achieve a function listed?		
2.5	Whether the safety class of the function specified?		
2.6	Are there any priorities? If so, are they specified?		
3.0	Performance Requirements		

3.1	Whether following attributes of output specified for all operating modes? Accuracy, Resolution and Response time		
3.2	If the performance requirements are different under any hardware/ software failure condition, are they specified?		
4.0	Interface Requirements With Other Computer based systems		
4.1	Are the functions provided and required by the system at the interfaces described?		
4.2	Whether the nature of information to be transferred to/ from other systems specified?		
4.3	Frequency of interactions and protocol to be followed specified ?		
4.4	Clock synchronization with other systems specified ?		
4.5	Speed of data transfer specified ?		
4.6	Isolation requirements specified?		
4.7	Action to be taken on fault specified?		
5.0	Interface Requirements With Field I/O		
5.1	Whether the type of all inputs / outputs from/ to plant processes/ equipments and operator panels specified?		
5.2	Whether static/ dynamic characteristics of		

	sensors and actuators, if required, are mentioned?		
5.3	Signal conditioning requirements for inputs and type of output?		
5.4	Validation checks on inputs / outputs?		
6.0	Human Machine Interface (HMI) Requirement		
6.1	No. of interface points		
6.2	Nature of interfaces		
6.3	Refresh rates of displays		
6.4	Details of operator function required at each interface		
6.5	Precision of outputs		
6.6	Ergonomic requirements of controls and displays		
6.7	System response time to operator requests		
6.8	Scheme for use of colours on displays		
6.9	Menu navigational requirements for normal as well as emergency operations		
6.10	Error messages for HMI operator's convenience		
6.11	Help facility		
7.0	Power Supply Requirements		

7.1	Does power supply requirements cover the type of supply available, short and long term variations, noise level, interruption time during change over to back up supply?		
8.0	Testing, Diagnostics and Self Supervision Requirements		
8.1	Whether online and offline diagnostics are covered?		
8.2	Identification of system components to be checked?		
8.3	Periodicity of testing		
8.4	Fault annunciation methods		
9.0	Data Archival and Retrieval Requirements		
9.1	Whether data archival requirements in different modes of operation mentioned		
9.2	Has the format and frequency of archiving specified?		
9.3	What is the lifetime of archived information required?		
10.0	Safety requirements		
10.1	Whether fail-safe state of system outputs specified?		
11.0	Access Control Requirements		

11.1	Is the access control policy defined?		
11.2	Whether method to indicate security breach is mentioned?		
12.0	Environmental Requirements		
12.1	Whether the environmental conditions to which the system will be subjected is specified?		
13.0	Reliability/ maintainability Requirements		
13.1	The Mean Time Between Failure and Mean Time To Repair requirements specified?		
13.2	Demand failure probability specified if the safety class desires so		
14.0	Whether installation, cabling and grounding requirements included?		
15.0	Acceptance criteria specified?		

APPENDIX – III: Checklist for System Architecture Review

Sl. No	<i>Items to be examined</i>	Yes/ No/ NA	Comments of the Examiner
1.0.	Verification of Architectural Design		
1.1	Structure		
1.1.1	Does the architecture allow for implementation of all of the requirements?		
1.1.2	Has the architecture been adequately decomposed?		
1.1.3	Have the system functions been appropriately allocated to components?		
1.1.4	Does the architecture provide an adequate base for subsequent design work?		
1.1.5	Is the architecture feasible for implementation?		
1.1.6	Have maintainability issues been adequately addressed?		
1.1.7	Can the program set be integrated		

	and tested in an incremental fashion?		
1.2	Correctness		
1.2.1	Does the architecture avoid unnecessary redundancy?		
1.2.2	Have all reliability and performance requirements been addressed?		
1.2.3	Have all security considerations been addressed?		
1.2.4	Does the architecture consider all existing constraints?		
1.2.5	Are all necessary, and only the necessary, data structures defined?		
1.2.6	Will the proposed architecture satisfy all specified quality attributes and performance goals?		
1.3	Clarity		
1.3.1	Is the architecture, including the data flow, control flow, and interfaces, clearly represented?		
1.3.2	Are multiple representations of the design consistent with each other?		

1.3.3	Are all of the decisions, dependencies, and assumptions for this design documented?		
1.4	Is the interface with other system shown in the context diagram?		
1.5	Are all the subsystems/ packages (hardware as well as software) individually named?		
1.6	Are the interfaces external to the system identified and mapped to the subsystem/ package?		
1.7	Are the following available for each hardware		
1.7.1	Purpose		
1.7.2	Functional requirements		
1.7.3	Performance requirements		
1.7.4	Resource requirements		
1.7.5	Interfaces external to the system		
1.7.6	Field interfaces		
1.7.7	Human machine interface		
1.7.8	Interfaces with other computer based systems		
1.7.9	Interfaces with other subsystems		
1.7.10	Dependency with other subsystems		

1.7.11	Hardware related parameters required for software design		
1.7.12	Data validation criteria for each input and output		
1.8	Are the following available for each software		
1.8.1	Purpose		
1.8.2	Functional requirements		
1.8.3	Performance requirements		
1.8.4	Resource requirements		
1.8.5	Interfaces external to the system		
1.8.6	Field interfaces		
1.8.7	Human machine interface		
1.8.8	Interfaces with other computer based systems		
1.8.9	Interfaces with other software		
1.8.10	Dependency with other packages		
1.8.11	Software related parameters required for hardware design		
1.8.12	Data validation criteria for each I/O		
1.9	Is there a consistent and complete description of what hardware is expected to do within the proposed implementation		

1.10	Is there a consistent and complete description of what software is expected to do within the proposed implementation		
1.11	Have the following been demonstrated for the system		
1.11.1	Fault tolerance		
1.11.2	Fail safe action		
1.11.3	Reliability		
1.11.4	Security		
1.11.5	Redundancy		
2.0	Traceability		
2.1	Backward traceability		
2.1.1	Are the functions of each hardware/software traceable to the requirements stated in SyRS		
2.2	Forward traceability		
2.2.1	Is each and every requirement stated in SyRS covered as a function of one or the other hardware/ software		

APPENDIX – IV: Check list for Software Requirement Specification Review

Sl. No.	<i>Items to be examined</i>	Yes/ No/ NA	Comments of the Examiner
1.0	Standards Compliance		
1.1	Have the standards/guidelines and naming conventions been established for the document?		
1.2	Does the document format conform to the specified standard/guideline?		
1.3	Are the standards and naming conventions established followed throughout the document?		
2.0	Document Content		
2.1	Is there a high-level system overview?		
2.2	Do the high-level system diagrams depict the internal and external interfaces and data flows?		
2.3	Is the system's functional flow clearly and completely described?		
2.4	Has the software environment been specified (i.e., hardware, software resources, users)?		
2.5	Are the communication interfaces to other systems or devices such as LAN, serial devices clearly defined?		
2.6	Are all referenced documents listed?		

2.7	Are all definitions, acronyms, and abbreviations included?		
2.8	Is there a general description of the software system and operational concepts?		
2.9	Are the software functions described at a high-level?		
2.10	Are the user characteristics defined?		
2.11	Are general design and implementation constraints noted?		
2.12	Are general assumptions that affect implementation been stated?		
2.13	Are general dependencies noted?		
2.14	Is each function defined separately?		
2.15	Does each function fully define its purpose and scope?		
2.16	Have the functional requirements been stated in terms of inputs, outputs, and processing?		
2.17	Are the functional requirements clear and specific enough to be the basis for detailed design and functional test cases?		
2.18	Is there a description of the performance requirements for each function?		
2.19	Are the operational hardware limitations discussed for each function?		
2.20	Are any software limitations discussed for each		

	function?		
2.21	Are safety-critical software requirements uniquely identified?		
2.22	Are security requirements identified?		
2.23	Are software quality requirements identified (e.g., reliability, portability, reusability, maintainability)?		
2.24	Are personnel-related requirements identified?		
2.25	Are environmental requirements and conditions identified?		
2.26	Are all packaging requirements identified?		
2.27	Are all delivery requirements identified?		
2.28	Are requirements provided for the operational computer hardware?		
2.29	Are computer software resources identified (e.g., operating system, network software, databases, test software)?		
2.30	Have overall integration, test and acceptance criteria been established?		
2.31	Have test methods been identified for requirements?		
3.0	Traceability		
3.1	Is each functional requirement uniquely and correctly identified?		
3.2	Can each software functional requirement be traced to		

	one or more high-level system requirements?		
4.0	General		
4.1	Are all common functions identified?		
4.2	Are interface requirements to other major functions or external entities clearly identified?		
4.3	Are the requirements stated so that they are discrete, unambiguous, and testable?		
4.4	Has each decision, selection, and computational function that the software must perform been clearly defined?		
4.5	Is a dictionary for all data elements provided?		
4.6	Is the data dictionary complete?		
5.0	Information - Concise, Complete and Consistent		
5.1	Is the document concise and easy to follow?		
5.2	Does the level of detail provided reflect a level of detail appropriate to the purpose of the document?		
5.3	Are requirements stated consistently without contradicting themselves or other requirements?		
5.4	Is there evidence of documentation control?		
5.5	Was the document baseline prior to the Software Requirements Review?		
6.0	System/Node Integrity		

6.1	Do the requirements cover system/node integrity checks?		
6.2	Have the diagnostic tests on hardware that have to be executed while the system is running been specified?		
6.3	Have the integrity checks to be performed on software been specified?		
6.4	Have data integrity checks been specified?		
7.0	Performance		
7.1	Have the precision, accuracy of outputs and frequency of update been specified?		
7.2	Have the response times for important functions been specified?		
7.3	Have the time outs for communication failures and recovery time for the various software functions been specified?		
7.4	Are memory requirements provided?		
7.5	Are the timing and memory limits compatible with hardware constraints?		
7.6	Are all limits and restrictions on software performance defined?		

APPENDIX – V: Checklist for Detailed Design Verification

Sl. No.	<i>Items to be examined</i>	Yes/ No/ NA	Comments of the Examiner
1.0	Standards Compliance		
1.1	Were standards/guidelines and naming conventions established for the document?		
1.2	Does the document format conform to the specified standard/guideline?		
1.3	Are the standards and naming conventions established followed throughout the document?		
2.0	General		
2.1	Have similar solutions within the common application domain been considered?		
2.2	Has the architecture been exercised against existing usage scenarios?		
2.3	Have the interfaces to software, hardware and user been accounted for in the architecture?		
2.4	If diagrams are used, do they help in understanding the design?		
2.5	Is there unnecessary text or unnecessary pictures in the design document?		
2.6	Is the description of external interfaces correct?		

2.7	Are there any multiple descriptions of the same interfaces between program units?		
3.0	Modularity		
3.1	Are modules/packages/subsystems that have been defined highly cohesive within themselves, while they are loosely coupled to others?		
3.2	Does each module have a distinct purpose?		
3.3	Have the interfaces between modules been adequately defined?		
3.4	Is the structure of program units easy to understand?		
3.5	Is the design easy to modify?		
3.6	Are all program units sufficiently described?		
4.0	Error Recovery		
4.1	Is there a consistently applied policy for handling exceptional situations?		
4.2	Is there a consistently applied policy for handling data corruption in the database/files etc.?		
4.3	Is there is a defined strategy for handling "I/O queue full" or "buffer full" conditions, if any?		

5.0	Reliability		
5.1	Have strategy for achieving required availability been worked out?		
6.0	Performance		
6.1	Have nominal and maximal performance requirements for various operations been specified so as to meet the overall performance?		
7.0	Traceability		
7.1	Is there specified what requirements specification to meet?		
7.2	Is the correct version of the requirements specification referred?		
7.3	Does the design document cater for all requirements?		
7.4	Are all the architectural entities traceable back to the SRS?		
7.5	Does each architectural entity have a clear identifier for forward traceability?		

APPENDIX – VI: Checklist for System Integration Verification

Sl. No.	<i>Items to be examined</i>	Yes/ No/ NA	Comments of the Examiner
1	Is the system integration test report complete and correct?		
2	Is the communication across the system tested?		
3	Whether exceptions are handled if the communication fails?		
4	Whether common mode/cause failures are avoided in case of redundant systems?		
5	Whether the failure of any system is indicated in case of the redundant systems?		
6	Whether fail safe mode of the system is asserted?		
7	Whether the signal wiring/connectivity diagram is matching to the database list in the software?		
8	Whether appropriate class of power supply is fed to the system?		
9	Whether system is located in the corresponding cabinet with appropriate safety class?		
10	Whether system is connected to the appropriate safety class data highway for network connectivity?		

APPENDIX – VII: Checklist for System Validation Review

Sl. No.	<i>Items to be examined</i>	Yes/ No/ NA	Comments of the Examiner
1	Is the Traceability Matrix complete?		
2	Are there any features that cannot be tested directly?		
3	Is the test setup complete for each type of requirement?		
4	<p>Are the test cases and procedures complete, consistent and traceable to requirement specification?</p> <ol style="list-style-type: none"> 1. Functional requirements 2. Performance requirements 3. Interface requirements 4. External interface 5. Human machine interface 6. Power supply requirements 7. Fault tolerance requirement 8. Testing, diagnostics and self supervision requirement 9. Data archival and retrieval requirement 10. Safety requirement 11. Security requirement 12. Reliability requirement 13. Maintainability requirement 		

	14. Environmental requirement		
	15. Grounding, cabling and installation requirement		
5	Are the acceptance criteria traceable to requirement specifications?		
6	Is the verified and validated software loaded in the system?		
7	Is there a physical / administrative protection in place to safeguard that only verified and validated software is running in the system?		

PUBLICATIONS BASED ON THE THESIS
JOURNALS

1. **D. Thirugnana Murthy, T. Sridevi, A. Shanmugam and P. Swaminathan, Verification & Validation for Safety Critical Real Time Computers (ISSN 0973-9238)**
International Journal on Intelligent Electronic Systems, pp 15 -21 , November 2007.
2. T.Sridevi, A.Shanmugam, **D.Thirugnana Murthy, S.I. Sambasivan and P.Swaminathan, Static Analyzer for Computer Based Safety Systems**, Journal of the Instrument Society of India (ISOI), Vol. 37(1), pp. 40-48, 2007.
3. **D.Thirugnana Murthy, N.Murali, T.Sridevi, S.A.V. Satya Murty, K.velusamy, Software Reliability Growth Model For Safety Systems of Nuclear Reactor** , accepted by Journal of Life Cycle Reliability and Safety Engineering, Society for Reliability and Safety (SRESA), ISSN 2250 0820
4. **D.Thirugnana Murthy, Software Reliability Assessment and Modeling for Safety Systems of Nuclear Reactor** Communicated to International Conference on Advances in Communication, Network and Computing www.easychair.org/conferences/?conf=cnc2012 – CONC 2012, The Proceedings will be published by Springer and it will be available in the Springer Digital Library.
5. **D.Thirugnana Murthy, Verification and Validation of Safety Systems for Nuclear Reactors**, Communicated to Elsevier Nuclear Engineering and Design, NED-D-11-00652, <http://ees.elsevier.com/ned>.

6. **D.Thirugnana Murthy, K,Velusamy, The Methodology for Software Quality Assessment and Control of Safety Systems in Nuclear Reactor**, Communicated to “TechnoHub”, International Journal of Engineering and Technology , ISSN 2277 – 7708.

CONFERENCE PROCEEDINGS

1. T.Sridevi, A.Shanmugam, **D.Thirugnana Murthy**, S.Ilango Sambasivan and P.Swaminathan, “**Software Lifecycle for Safety Critical Systems**” , Proc. Int. Conf. on Trends in Intelligent Electronics System, Satyabama University, Chennai, Nov 2007.
2. **D.Thirugnana Murthy** , T.Sridevi, SAV Satyamurty and P.Swaminathan, **Verification and Validation of Computer Based Safety Systems for Nuclear Reactors** , Proc. Int. Conf. on Peaceful Uses of Atomic Energy, New Delhi, September 2009.
3. T.Sridevi, **D.Thirugnana Murthy**, B. Krishnakumar, S.A.V. SatyaMurty and P.Swaminathan, “**Software Quality Assessment for Safety Systems of Nuclear Reactor**”, 2nd International Conference on Asian Nuclear Prospects (ANUP-2010), Oct 11-13, Mahabalipuram, Tamilnadu.
4. L.Srivani, **D.Thirugnana Murthy**, N.Murali and S.A.V.Satyamurty, “**Reliability Analysis of Safety Critical I&C Systems in Nuclear Power Plants**”. Proc. International Applied Reliability Symposium, Chennai, Oct 10 -12, 2012.