SOFTWARE RELIABILITY IN SAFETY CRITICAL SUPERVISION AND CONTROL OF NUCLEAR REACTORS

by

P. ARUN BABU
(ENGG02201004005)

Indira Gandhi Centre for Atomic Research, Kalpakkam

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

MAY — 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 P. ARUN BABU entitled: "SOFTWARE RELIABILITY IN SAFETY CRITICAL SUPERVISION AND CONTROL OF NUCLEAR REACTORS" and recommend that it may be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

M. Sai Baba Date: 10-5-14

Chairman : Dr. M. Sai Baba

Date: 10 May 2014

Guide/Convener : Dr. T. Jayakumar

R.S. Keshavamurthy Date: 10/5/14

Co-guide/Member : Dr. R. S. Keshavamurthy

S. Venugopal Date: 10-5-2014

Member : Dr. S. Venugopal

N. Murali Date: 10-05-2014

Member : Shri. N. Murali

Rajib Mall Date: 10-05-2014

External examiner : Prof. Rajib Mall (IIT, Kharagpur)

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

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.

_____ Date :

Guide: Dr. T. Jayakumar

Place:

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 judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

(P. Arun Babu)

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.

(P. Arun Babu)

Abstract

1. Context

Software based systems have several advantages over hardware based systems in terms of functionality, cost, flexibility, maintainability, reusability, etc. However, software is prone to failure. Poorly written safety-critical software may lead to catastrophic failures and life threatening situations. Hence, safety-critical software must be adequately tested; and the probability of occurrence of software failures must be studied.

Quantification of software reliability is considered an unresolved issue; and existing approaches and models have assumptions and limitations which are not acceptable for safety applications. Also, to build reliable software, it is necessary to study the factors which are likely to affect the software reliability.

2. Objectives

1. To propose an automated method to generate test cases, and to determine test adequacy in safety-critical software.
2. To propose an approach to quantify software reliability in safety-critical systems of nuclear reactors.
3. To study the factors affecting software reliability in such safety systems.
4. To understand the relationship between the software reliability and number of faults remaining in the software.
5. To understand the relationship between the software reliability and safety in safety critical systems.

3. Method

To quantify the software reliability, a hybrid approach using *software verification* and *mutation testing* is proposed. Techniques to solve related issues such as quantification of software test adequacy and detection of equivalent mutants are also presented. The steps proposed to quantify software reliability are:

1. Generation of large number of test cases, where each test case has a unique execution path. To achieve this, code coverage information and genetic algorithms are used.
2. Verification of test cases using a semi-formal model, which is traceable to requirements; and acts as a test oracle.
3. Calculation of test adequacy for the above generated test cases in the range [0,1] using mutation score and conservative test coverage.
4. Calculation of software reliability using the computed test adequacy and the amount of verification carried out.

The formulae for software reliability are derived, and the factors affecting software reliability are presented. The proposed methods are applied to software in the following instrumentation and control systems for fast breeder reactors:

1. Fresh Sub-assembly Handling System (FSHS)
2. Reactor Startup system (RSU)
3. Steam Generator Tube Leak Detection system (SGTLD)
4. Core Temperature Monitoring System (CTMS)
5. Radioactive Gaseous Effluent System (GES)
6. Safety Grade Decay Heat Removal system (SGDHR)

Also, for each case study, mutant characteristics during mutation testing, and the relationship between software reliability and safety are presented.

4. Major results

1. For the case studies, the proposed test case generation technique has resulted in high test adequacy. Using the generated test cases, the probability of software failure in the case studies has been demonstrated to be $< 10^{-5}$ for a random input from the input domain, with 95% confidence level.

2. In mutation testing, for an effective set of test cases, the unkilled mutants have been found to have lower variance in their properties when compared to the killed mutants.

3. Three factors: (i) test adequacy, (ii) the amount of verification carried out, and (iii) the amount of verified code reused; have been found to be affecting the software reliability.

4. The results of present study suggest that software reliability estimates based on the number of faults present in the software *alone*, are likely to be inaccurate for safety-critical software.

5. The empirical results indicate that: for safety-critical software, the required safety can be achieved by improving the reliability; however the vice-versa is not always true.

5. Conclusion

The methods and analysis presented in this thesis demonstrate the use of software testing to arrive at an estimate of the software reliability. The results on relationship between the software reliability and safety in safety-critical systems would be helpful in understanding the dynamics behind developing safer software based systems.

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

List of publications

Journals

1. **An intuitive approach to determine test adequacy in safety-critical software**,
   P. Arun Babu, C. Senthil Kumar, N. Murali, and T. Jayakumar,
   ACM Sigsoft software engineering notes, Volume 37, Issue 5 (Sept. 2012).

2. **A hybrid approach to quantify software reliability in safety systems of nuclear reactors**,
   P. Arun Babu, C. Senthil Kumar, and N. Murali,
   Annals of nuclear energy, Volume 50, December 2012, Pages 133—140.

3. **Properties of software reliability in safety systems of nuclear reactors**,
   P. Arun Babu, C. Senthil Kumar, N. Murali, and T. Jayakumar,
   Manuscript under review in Journal of systems and software.

Conferences/Symposiums/Articles

4. **Software reliability in safety-critical applications:**
   A case study from the nuclear industry,
   P. Arun Babu, C. Senthil Kumar, N. Murali and T. Jayakumar,
   International Applied Reliability Symposium, Chennai, India, Oct. 2012.

5. **Making formal software specification easy**,
   P. Arun Babu, N. Murali, P. Swaminathan, and C. Senthil Kumar,
   2nd International Conference on Reliability, Safety and Hazard,
   pages 511–516, Dec. 2010. doi: 10.1109/ICRESH.2010.5779603.

6. **Semi-formal property verification in games**,
   P. Arun Babu and N. Murali,
   Testing Experience, issue 15, pages 14–17, Sept. 2011.

Internal reports

Versions of the above publications have also appeared in the following internal reports:

1. **A hybrid approach to quantify software reliability in safety systems**
   AERB/SRI/2012/2007

2. **Method to determine adequacy of software testing for reliability estimation of computer based systems in NPPs**
   AERB/SRI/2013/2010

3. **Development of software reliability assessment methodology using model and mutation based testing for systems important to safety in FBRs**
   EIRSG/ICG/RTSD/PRIS(G)/2012-13/IV(8a)

4. **Test case adequacy assessment using mutation based testing and test coverage for computer based safety related systems in FBRs**
   EIRSG/ICG/RTSD/PRIS(G)/2012-13/IV(8b)

Acknowledgements

- To my guide Dr. T. Jayakumar (Director, MMG, IGCAR) and my co-guide Dr. Keshava Murthy (Head, RSDD, IGCAR) for being my source of guidance.
- To my other doctoral committee members: Dr. M. Sai Baba (Associate Director, RMG, IGCAR) (who also acted as my troubleshooter) and Dr. S. Venugopal (Associate Director, GRIP, IGCAR) for their guidance and mentoring.
- To Dr. P. Swaminathan (Former Group Director, EIG, IGCAR), for introducing and motivating me to take up the research topic.
- To N. Murali (Associate Director, ICG, EIRSG, IGCAR) for being my friendly technical advisor. His strong opinions, questions, and suggestions have significantly improved my work.
- To Dr. C. Senthil Kumar (Scientific Officer/G, Safety Research Institute, AERB) for being my collaborator. I am lucky to have worked with him, and I cannot thank him enough for his patience during discussions and reviews.
- This thesis is case study centric, and cannot be completed without the help of engineers (at EIRSG) associated with the systems, who have patiently helped me in understanding the safety systems. I would like to thank Anindya Bhattacharyya, A. Santhana Raj, M. Chandramouli Sharma (also my labmate), M. Manimaran, Manoj, MA. Sanjith, and Saritha Menon.
- Special thanks to S.A.V Satyamurthy (Group Director, EIRSG), R. Jehadeesan (Computer Division), and staff members for providing me the high performance computing facility, which has led to quicker results.

- To A. Shanmugam, Aditya Gour, Alok Gupta, B. Sasidhar Rao, D. Thirugnanamurthy, L. Srivani, M. Kasinathan, P. Parimalam, Patankar, R. P. Behra, Venkat Kishore, and other EIRSG staff members for all the encouragement, discussions, suggestions, and assistance.

- To the administrative staff of EIRSG for timely assistance.

- I am truly indebted to the contributors of free and open source software such as: LaTeX, OpenBSD, Vim, GCC, Erlang, Drakon, Python, Gnuplot, Graphviz, Linux, etc. which have made my work easier.

- Many thanks to Dr. Baldev Raj (Former Director, IGCAR), Brad Stewart (Developer Geeks), Prof. Dick Hamlet (Portland State University), and S. Kishore (FRTG, IGCAR) for allowing me to use their published materials in my thesis.

- To the reviewers and editors of journals, conferences, and symposiums for reviewing my work.

- To my dear parents and friends: Anil, Ashutosh, Biju, Bubathi, Deepak, Govindha, Hari Babu, Hemangi, Naveen, Paawan (also my labmate), Rajini, Ravikirna, Saptarishi, Sharath, Srihari, Subhra, and others for all the fun times we had and for *just being there* for me.

- To the Department of Atomic Energy (DAE) for providing me the generous DAE Graduate Fellowship Scheme (DGFS) Ph.D fellowship.

P. Arun Babu

Contents

List of figures

List of tables

List of equations

List of acronyms

BBN	Bayesian Belief Network
CSRDM	Control and Safety Rod Drive Mechanism
CTMS	Core Temperature Monitoring System
DSLs	Domain Specific Languages
DSRDM	Diversified Safety Rod Drive Mechanism
FSEP	Fresh Sub-assembly Entry Port
FSHS	Fresh Sub-assembly Handling System
FSPF	Fresh Sub-assembly Preheating Facility
FSRF	Fresh Sub-assembly Receiving Facility
FSU	Fuel handling Startup system
GES	Radioactive Gaseous Effluent System
IAEA	International Atomic Energy Agency
IEC	International Electro-technical Commission
KLOC	Kilo Lines of Code
LCSAJ	Linear Code Sequence And Jump
MC/DC	Modified Condition/Decision Coverage
MD5	Message Digest 5

MISRA	Motor Industry Software Reliability Association
NNS	Non-Nuclear Safety
NPPs	Nuclear Power Plants
PCA	Principal Component Analysis
PFBR	Prototype Fast Breeder Reactor
PFD	Probability of Failure on Demand
PSA	Probabilistic Safety Assessment
QSRM	Quantitative Software Reliability Method
RSU	Reactor Startup System
SC	Safety Critical
SCRAM	Safety Control Rod Axe Man
SGDHR	Safety Grade Decay Heat Removal system
SGTLD	Steam Generator Tube Leak Detection system
SIL	Safety and Integrity Level
SR	Safety Related
SRGMs	Software Reliability Growth Models
TMR	Triple Modular Redundancy
VDM	Vienna Development Method

Part I

The context

1

Introduction

1.1 Background

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

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

Subsystem	**1980s**	**1990s**	**2000s**
Propulsion	42 %	38 %	54 %
Guidance and navigation	6 %	16 %	4 %
Electrical	6 %	8 %	8 %
Operational ordnance	2 %	8 %	0 %
Software and computing	0 %	8 %	21 %
Structures	4 %	6 %	0 %
Pneumatics and hydraulics	4 %	2 %	0 %
Unknown	37 %	16 %	13 %

Table 1.1: *Worldwide subsystem failures by decade in launch vehicles*

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

1.2 The problem statement

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

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

1.2.1 Research questions

For software in safety-critical system:

1. How can the rigor in software testing be quantified ?
2. What is its probability of failure-free operation ? *(i.e. the software reliability)*
3. What factors are likely to affect the software reliability ?
4. How can the software reliability be improved to meet target reliability ?
5. What is the relationship between software reliability and safety ?

1.3 Motivation

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

1. " *The probability of failure-free software operations for a specified period of time in a specified environment*" [15].
2. "*The reliability of a program P is the probability of its successful execution on a randomly selected element from its input domain*" [16].

The first definition is made compatible with the hardware reliability definition; thus, making it possible to estimate the overall system reliability [17]. However, the fact

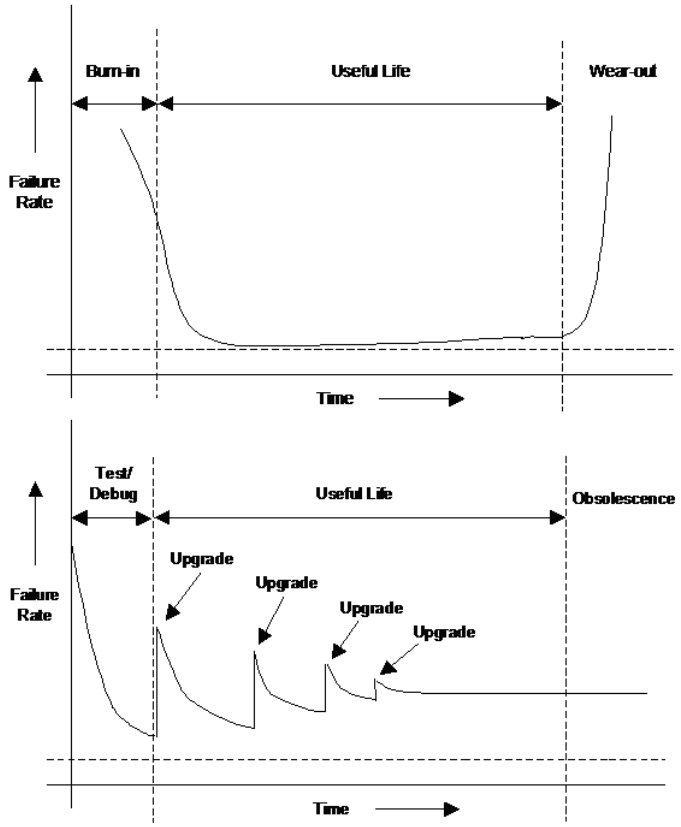


Figure 1.1: *Typical hardware and software failure rates over lifetime*

that the failures in software are mainly caused due to its design faults, and not due to its wearing off (i.e. software failures are not direct function of time), makes software and hardware reliability fundamentally different *(Figure 1.1)*. Thus, the definition of software reliability with respect to time is arguable. The second definition, however is independent of time, and is used as the basis in the present study.

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

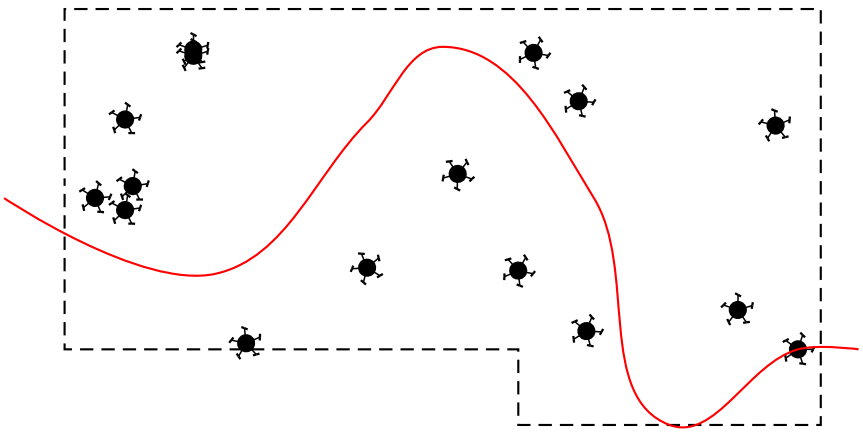


Figure 1.2: *The minefield analogy of software reliability (the mines represent faults in software, and the path represent a single execution flow of the software)*

infeasible paths in the software.

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

1. There are fixed number of faults in the software being tested.
2. Whenever a failure is found, it is removed instantaneously, without inducing a new fault.
3. Each fault has the same contribution to the unreliability of the software; and software with fewer faults is more reliable than the one with more faults.
4. The probability of two or more software failures occurring simultaneously is negligible.
5. Enough and accurate software failure data is available for analysis.
6. The execution time between failures is distributed in a known fashion.

7. The hazard rate for a single fault is constant.

8. Tests conducted represent the operational profile.

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

1.4 Software in safety-critical systems

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

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

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

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

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

1.5 Software in nuclear reactors

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

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

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

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

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

1.6 Software failures in nuclear industry

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

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

2. Files become inaccessible to the nuclear accountants using nuclear material tracking software at Kurchatov institute, Russia [**29**].

3. *Slammer* worm disabled safety parameter display system for 5 hours at Davis-Besse nuclear power station [**30**].

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

5. *Stuxnet* worm infects nuclear plants in Iran running Supervisory Control and Data Acquisition (SCADA) systems controlled by Siemens software [**32**].

and several others [**33**]. The main reasons for the failures include: improper/imprecise requirement specification, insufficient testing, use of untested Commercial Off the Shelf Software (COTS), incorrect reuse of older software, vulnerabilities in the software, etc. Hence, an *ideal* software reliability quantification approach must take such factors in to consideration.

1.7 Issues in software reliability quantification

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

As safety systems in a nuclear power plant are categorized based on their importance to safety; for computer based systems, the International Electro-technical Commission (IEC) standards give requirements in the form of Safety and Integrity Level (SIL) [**35**]. SIL is specified in the form of a number from one to four based on the probability of failure. SIL-1 represents the lowest safety integrity level with target average Probability of Failure on Demand (PFD) between 10^{-2} and 10^{-1}, whereas SIL-4 is the highest with PFD between 10^{-5} and 10^{-4}. Common safety functions in NPPs are governed

by defense in depth principles such as: reactivity control, maintenance of fuel integrity, control of pressure boundary, continuation of core cooling, and prevention of release of radioactivity. In view of the inherent complexity in such control software, it is difficult to assess the failure probability of software and quantify the influence of its safety function on core melt down frequency.

1.8 Need for a new approach

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

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

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

Some qualitative improvement in software reliability may be achieved with N-version programming [**41**]; however, it is costly and its benefits are arguable [**42**]. Hence, the current licensing procedure for computer based systems in nuclear reactors is based on deterministic criteria. For a risk-informed regulation, a procedure for software reliability estimation is not yet been satisfactorily developed [**43**, **44**].

An ideal way to demonstrate that the software meets a required reliability is through formal verification. Formal verification is a method of proving certain properties in the designed algorithm, with respect to its requirement specification written in mathematical language/notation. Approaches to formal verification include formal proof and model checking. Formal proof is a finite sequence of steps which proves or disproves a certain property of the software, whereas model checking achieves the same through exhaustive search of the state space of the model. Unfortunately, it is not always feasible to ensure complete formal verification of software due to the difficulties involved such as state space explosion and difficulties in practical implementation of formal methods [**45**]. Also, a major assumption in formal verification is that the requirements specification captures all the desired properties correctly. If this assumption is violated, the formal verification becomes invalid.

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

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

1.9 This thesis

1.9.1 Assumptions and limitations

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

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

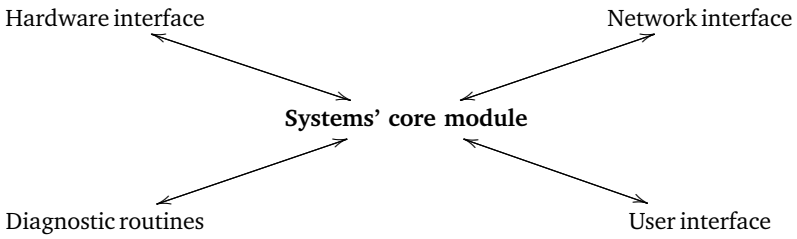


Figure 1.3: *Typical software architecture of safety applications in nuclear reactors*

temperature, pressure, flow etc.) and send outputs to final control elements such as: motors, relays, blowers, heaters, etc.

(b) A user-interface module, which interacts with the user.

(c) A network-interface module, which can share soft inputs/outputs with other connected systems.

(d) A diagnostic module which checks the state of the system at regular intervals.

(e) The main/core module which performs the systems' intended function.

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

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

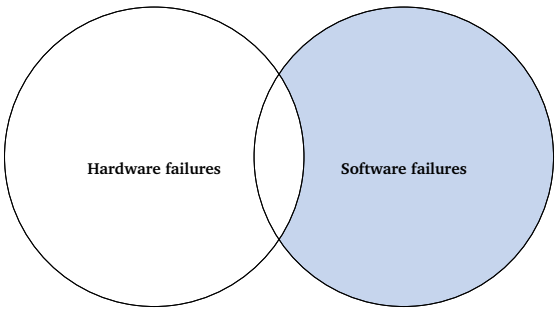


Figure 1.4: *Focus of the present study: failures caused due to software faults (indicated by the shaded portion of the venn-diagram)*

3. The software is written in portable C-programming language, adhering to Motor Industry Software Reliability Association (MISRA) standards.
4. The software is single-threaded and runs on bare hardware without any operating system support.
5. Software is testable, i.e. it has a test oracle, using which large number of test cases can be verified automatically.

1.9.2 Structure

The thesis is structured as follows:

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

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

– **Chapter-6**

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

– **Chapter-7**

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

– **Chapter-8**

  * Summarizes the thesis, and lists out some of the open problems

2

Related work

2.1 In formal methods

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

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

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

1. *Algebraic specification languages:* Algebraic specification is a formal process of writing specifications in mathematical structures and functions. Vienna Development Method (VDM) [**56**], Z (*zed*) notation [**57**] and B-Method [**58**] are

the most popular algebraic specification languages both in academia as well as in industries [**59**, **60**]. A detailed description and comparison of various specification languages can be found in [**61**]. Applications and features of VDM and Z are well discussed in [**62**]. Also, works such as [**52**] highlight the experiences with formal specification languages and formal methods in general.

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

3. *Special purpose languages:* Eiffel [**69**], originally developed by Eiffel Software is an object-oriented programming language which has introduced and popularized the set of principles such as: command-query separation, design by contract, open-closed principle, option-operand separation, single-choice principle, and uniform-access principle. Some of these principles were later adopted by many other specification and programming languages. The goal of Eiffel programming method is to enable programmers to create reliable, reusable, and correct software. The

Prototype verification system (PVS) [**70**], developed at the Computer Science Laboratory of SRI International, California, USA. is a framework for writing formal logical specifications and constructing proofs. PVS has been successfully used in specification and verification of various critical applications [**71**] in organizations such as NASA for Cassini aircraft [**72**] and Space Shuttle [**73**].

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

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

Critical review of specification languages

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

1. *An expert is required to get started, and should be always available:* Successful use of formal methods requires selection of suitable notation, right tools and fair amount

of discrete mathematical skills. Hence, a team of experts is required to get started [**45**].

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

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

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

understand, verify, and review the specification. Thus, these languages may not be suitable for large and complex applications, where requirements may change frequently.

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

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

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

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

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

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

4. Proving large, complex, or non-linear properties using formal methods is difficult, time consuming, and sometimes impractical.

5. Formal methods cannot indicate if enough properties have been proven.

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

2.2 In model checking

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

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

Critical review of model checking

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

As exhaustive model checking may not scale well for large problems due to the state space explosion problem; Bounded model checkers (BMC) were proposed, which can

check a given model without the state space explosion problem. But, due to the k bound, completeness cannot be achieved.

2.3 In safety-critical software development, V&V

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

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

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

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

Critical review of safety-critical software development and V&V techniques

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

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

2.4 In software testing and test coverage

Testing is a process of giving a set of inputs to the software under test and match its output with the expected output. Software in safety and mission critical applications often require proof that they have been thoroughly tested. Hence, programmers and testers are expected to write good test cases [**118**] which can verify the behavior of the entire system. However, as exhaustive testing is impractical in real world applications, the amount of testing is quantified through test coverage.

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

1. The MC/DC criterion is satisfied only when:
   (a) Every point of entry and exit in the program has been invoked at least once.
   (b) Every condition in a decision has taken all possible outcomes at least once.
   (c) Every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect the outcomes of that decision. A condition is shown to independently affect the

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

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

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

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

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

As large number of test cases may require a lot of time to execute (especially during regression testing), varieties of ways to reduce test cases and to prioritize them have

been proposed [**131–134**]. Some studies [**135**, **136**] indicate that test case reduction by keeping the test coverage constant does not have significant effect on the effectiveness of the test suite. A systematic survey on test case minimization and prioritization may be found in [**137**].

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

Critical review of software testing and test coverage

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

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

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

Fuzz testing is an effective testing technique to detect memory leaks, buffer

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

2.5 In mutation testing and test adequacy

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

$$\text{Mutation score} = \frac{K}{G - E} \tag{2.1}$$

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

Critical review of mutation testing and test adequacy

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

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

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

(a)

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

(b)

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

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

                if ( d || e || f )
                        // statement 2
        }
}
```

(a)

```
bool function ( int  a, bool b, bool c, bool d, bool e, bool f )
{
        bool a_is_equal_to_100  = a == 100 ;

        bool b_or_c             = b ||c ;
        bool d_or_e_or_f        = d || e || f ;

        if ( a_is_equal_to_100 )
        {
                if ( b_or_c )
                        // statement 1

                if ( d_or_e_or_f )
                        // statement 2
        }
}
```

(b)

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

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

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

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

(a)

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

(b)

Figure 2.4: *An example of mutant program: (a) the original program, (b) the mutant program (the induced fault is indicated by the red color).*

mutation score will always be < 1. Automatic detection of equivalent mutant is in general considered as an undecidable problem [**149**]; nevertheless, many attempts have been made [**150–153**] to detect them with certain accuracy. Also, results of mutation testing could be misleading if faults are not induced at all paths of the code.

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

2.6 In software reliability growth models (SRGM)

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

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

Critical review

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

However, as with any black box technique, the software testing methodology is *adhoc* in nature, and may not be sufficient to test safety-critical software. Also, choosing a

useful model for a given situation/software is a complex task [**23**, **24**].

2.7 In Bayesian belief network

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

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

Critical review

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

2.8 In architecture based approaches

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

made out of components" [**166**].

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

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

Critical review

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

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

A systematic review of architecture based models can be found in [**167**, **174**].

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

Table 2.1: *Summary of the related work - I*

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

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

2.9 Summary

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

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

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

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

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

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

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

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

3

Background information

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

3.1 Instrumentation and control in nuclear reactors

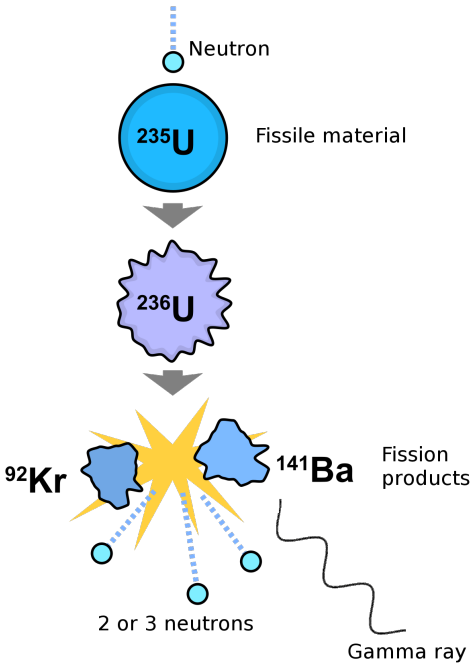


Figure 3.1: *A fission reaction*

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

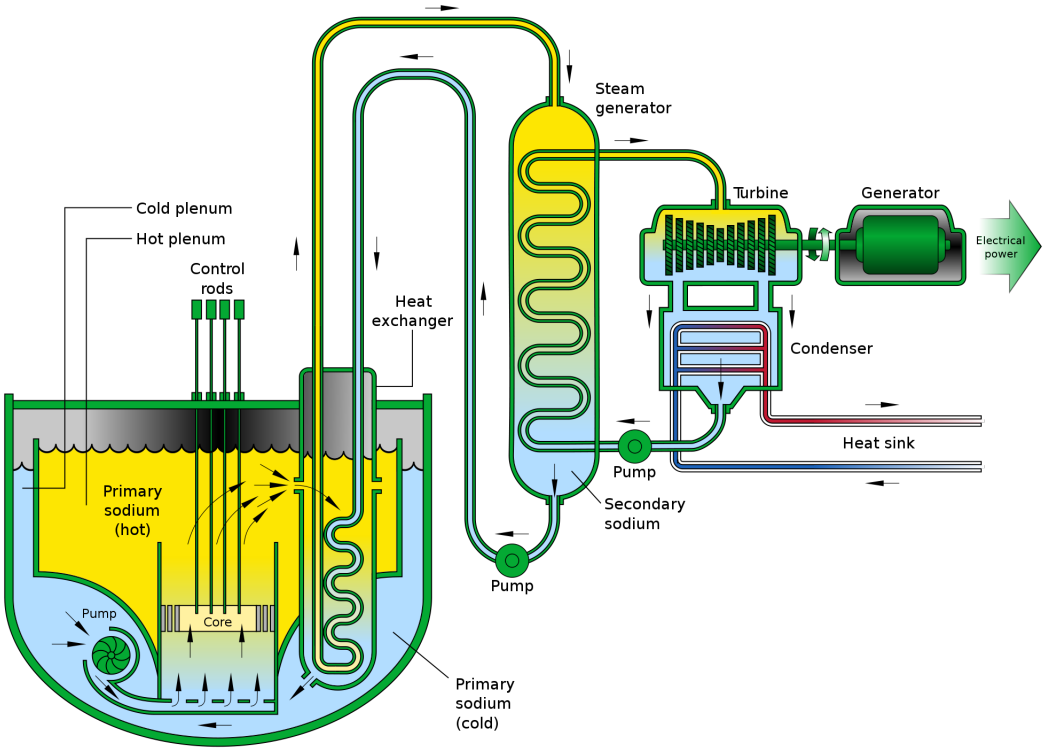


Figure 3.2: *A typical sodium-cooled, pool-type fast reactor*

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

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

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

3.2 Case studies used in the present study

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

3.2.1 Fresh subassembly handling system

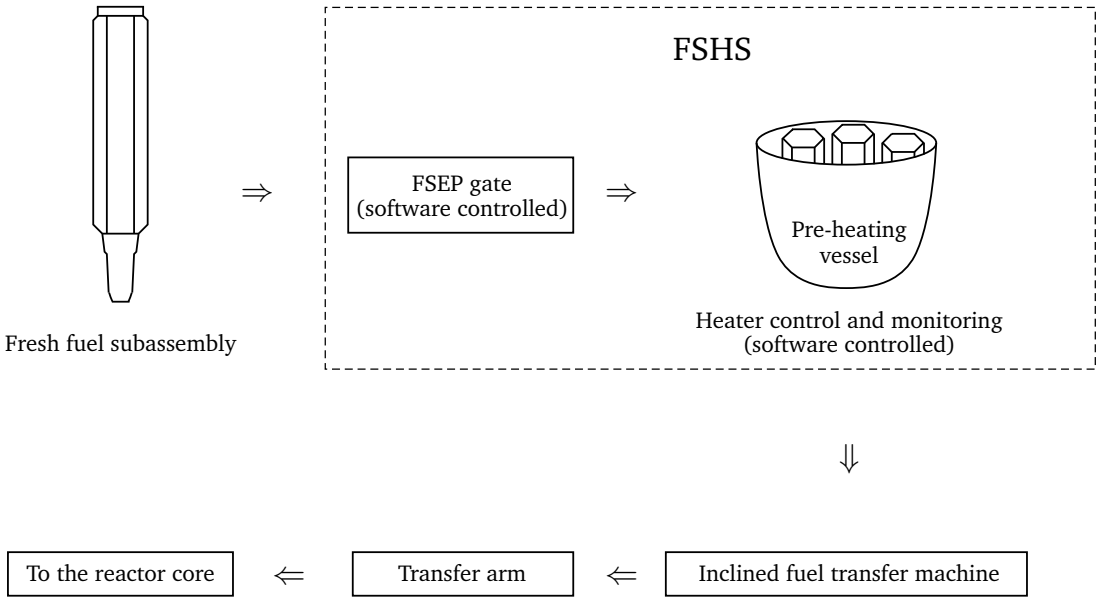


Figure 3.3: *The flow of fresh fuel subassembly*

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

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

3.2.2 Reactor start-up system

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

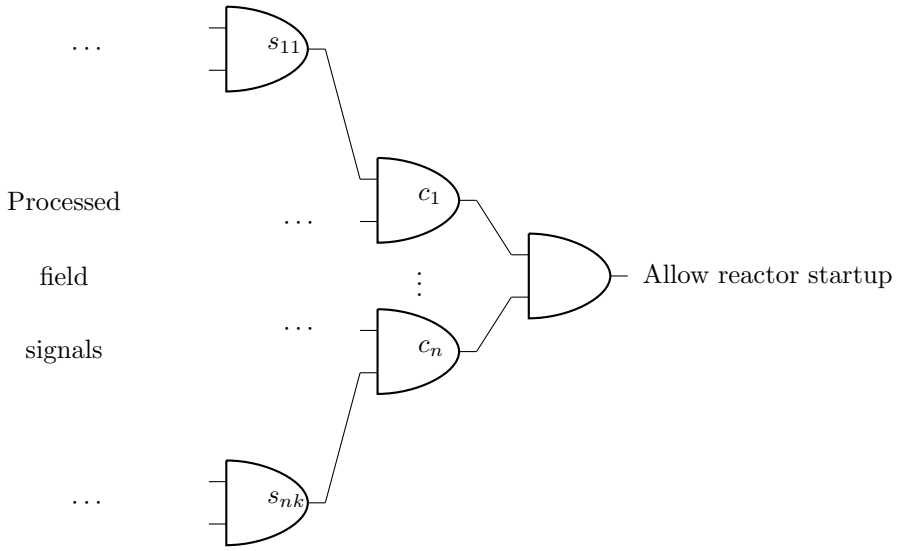


Figure 3.4: *Logic diagram of the reactor startup system (c_i is one of the condition to be satisfied for the reactor startup, and s_{ij} is the j^{th} sub-condition of c_i)*

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

3.2.3 Steam generator tube leak detection system

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

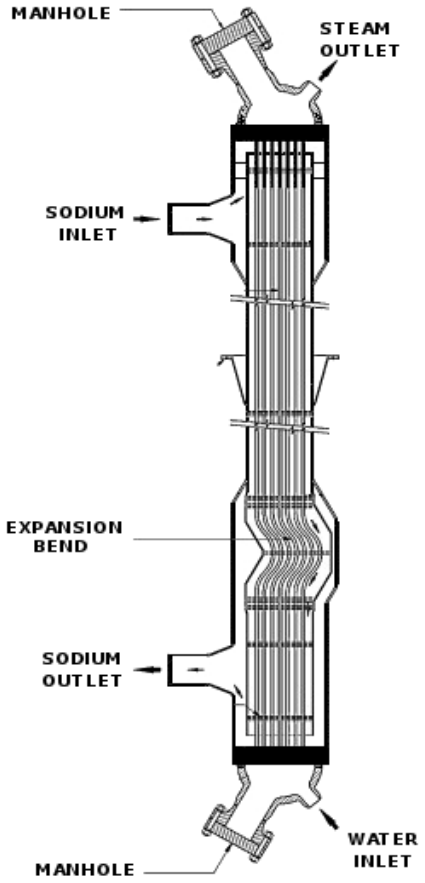


Figure 3.5: *Steam generator in a sodium cooled fast reactor*

3.2.4 Core temperature monitoring system

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

These signals are generated when the computed parameters cross their respective threshold limits. The alarms generated are sent to the control room for the operator, and the SCRAM signal is sent to Control and Safety Rod Drive Mechanism (CSRDM)

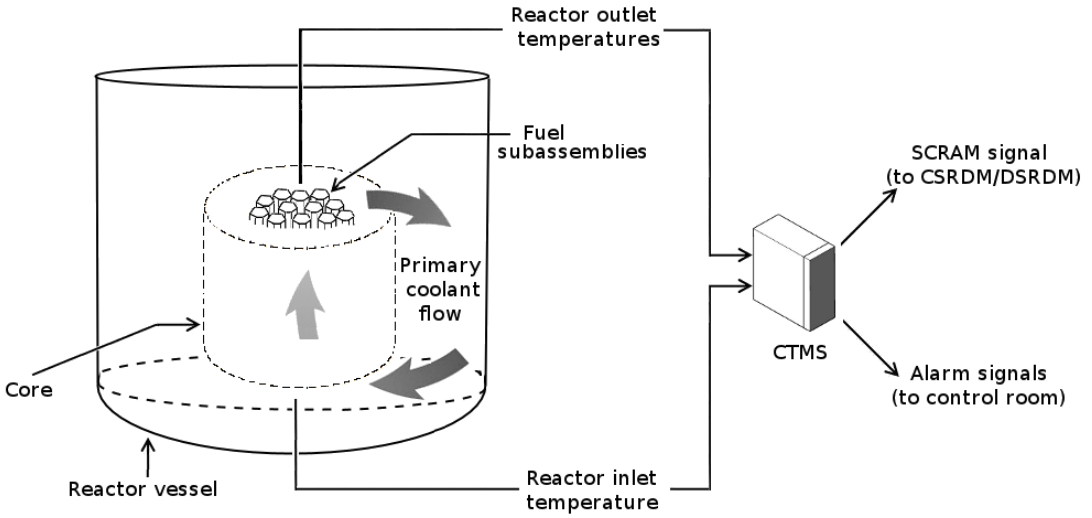


Figure 3.6: *Schematic of the software based CTMS*

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

3.2.5 Radioactive gaseous effluent system

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

Radioactive Gaseous Effluent System (GES) [**181**] software processes the system signals and produces the required control and alarm signals for achieving safe radioactive effluent handling. The control actions mainly include start/stop of compressors and open/close of valves.

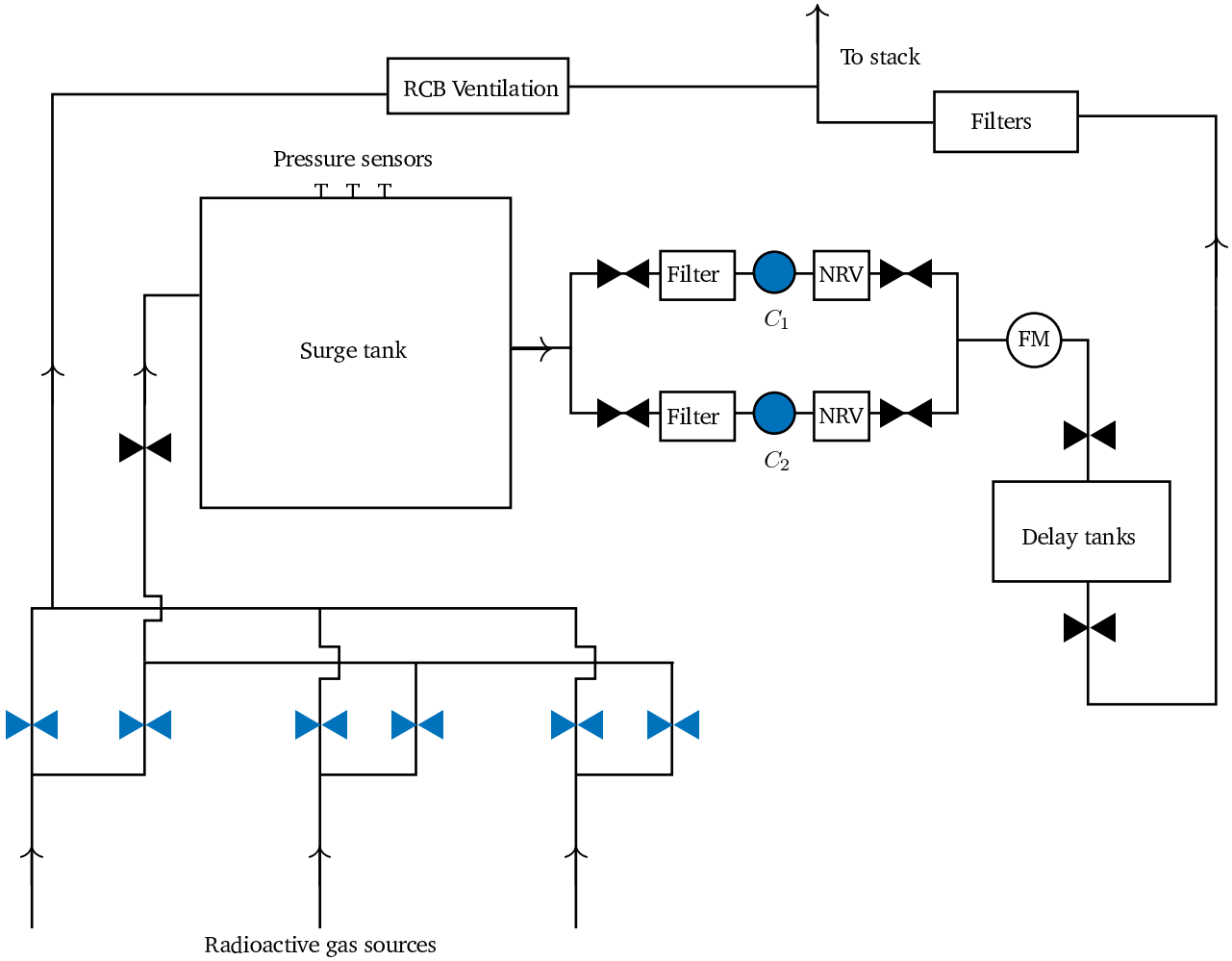


Figure 3.7: *The schematic of Radioactive Gaseous Effluent System (GES) (Here the symbol ⋈ indicates a pneumatic valve, NRV indicates a non-return valve, FM indicates the flow meter,* C_1 *and* C_2 *are the compressors, and the items controlled by the software are indicated by the blue color)*

3.2.6 Safety grade decay heat removal system

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

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

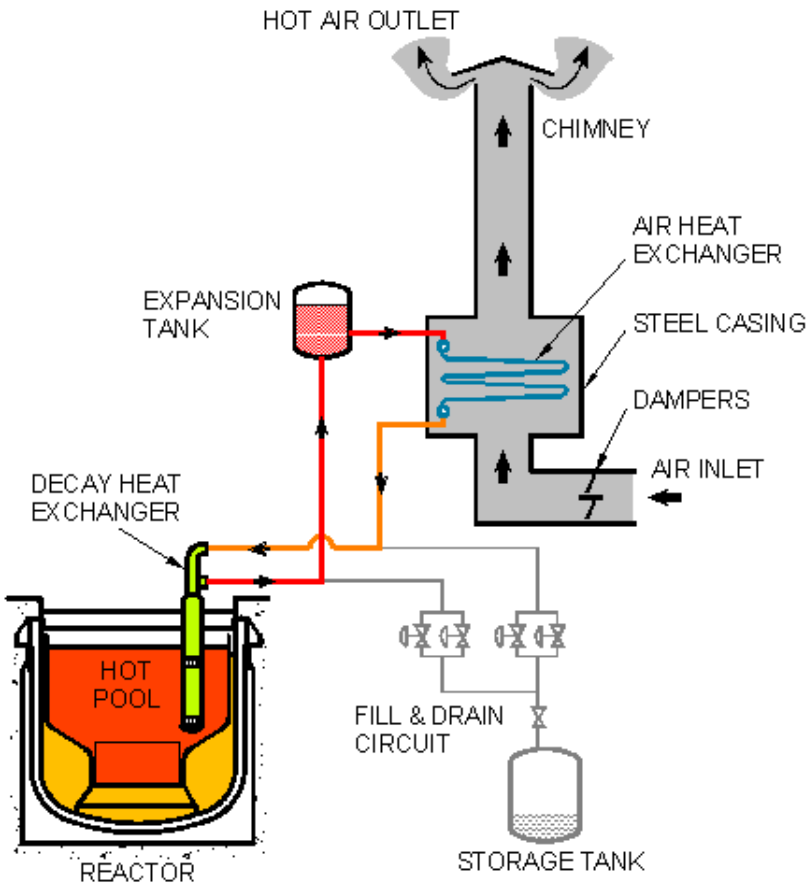


Figure 3.8: *Schematic of one of the four independent and identical loops of safety grade decay heat removal system*

Part II

Studies on software reliability

4

Research methodology

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

4.1 Software reliability definition

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

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

study, the software reliability is assumed to remain constant *wrt.* time as long as the environment remains the same.

4.2 Choice of case-studies

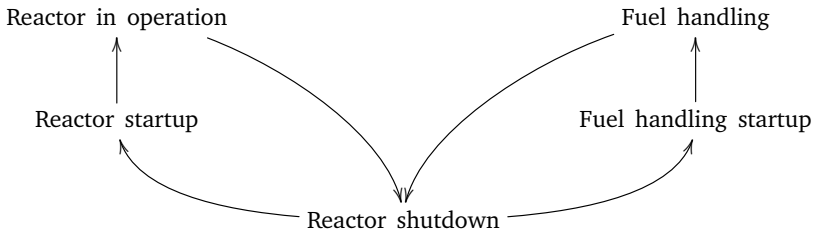


Figure 4.1: *Various states of a nuclear reactor*

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

Table 4.1: *Case studies chosen in the present study*

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

4.3 Method

The present study uses the results of software testing to quantify software reliability. The method involves the following steps:

1. Creation of a model of the software:

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

2. Generating effective test cases:

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

3. Calculation of software test adequacy:

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

4. Quantification of software reliability:

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

4.4 Experimental details

4.4.1 Software under test

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

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

4.4.2 Software testing

1. On host:

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

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

2. On target:

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

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

4.4.3 Parallel processing

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

5

Test adequacy in safety-critical software

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

5.1 Introduction

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

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

5.2 Challenges

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

5.3 Software in the case studies

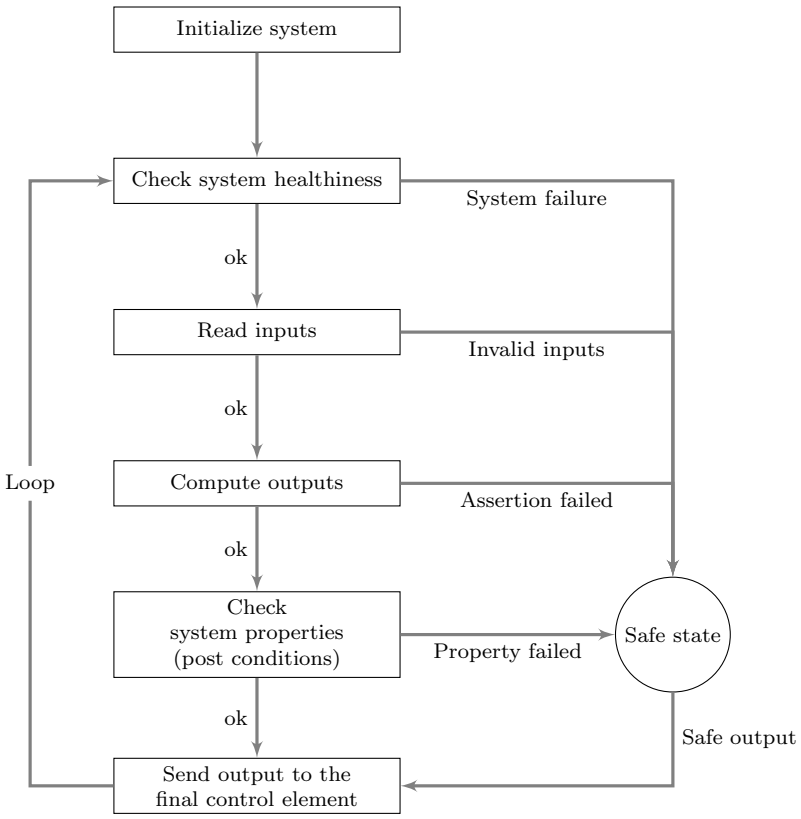


Figure 5.1: *Execution flow in safety-critical software*

As mentioned in *Section 3.2 on page 34*, *six* safety systems in a nuclear reactor are taken up as case studies. The execution flow of the software in case studies is illustrated in *Figure 5.1*. And the software in case studies has the following characteristics:

1. Software is written in portable C programming language, following important MISRA [**108**] guidelines.

2. Unless required, signed integers are avoided.

3. Function-like macros are avoided.

4. Only fixed bounded *for* loops are used.

5. No dynamic memory allocations are used.

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

7. The software passes the following static, dynamic, and security checkers:

   (a) No warnings with static analyzers: *Clang* [**192**], and *Cppcheck* [**193**] with —*enable=all* as argument.

   (b) No warnings with *Splint* [**194**] static analyzer using *-checks*, *-strict-lib*, and *-realrelatecompare* as arguments.

   (c) Final score = 0 using *BogoSec* [**195**] code security scanner. The scanners include: *FlawFinder* [**196**], *RATS* [**197**], and *Lintian* [**198**].

   (d) No warnings or errors found with dynamic analyzers: *Valgrind* [**199**] with *–leak-check=full* as argument and *Electric-Fence* [**200**] for the generated test cases *(Section 5.4.1 on the next page)*.

8. Assertions have been used to validate inputs and to check impossible conditions during execution. Functions which do not have assertions are either very simple/have error handling code/return the error code to the caller.

9. Apart from assertions in functions, system properties (as post-conditions) in the form of assertions must be met (*Figure 5.1 on the previous page*).

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

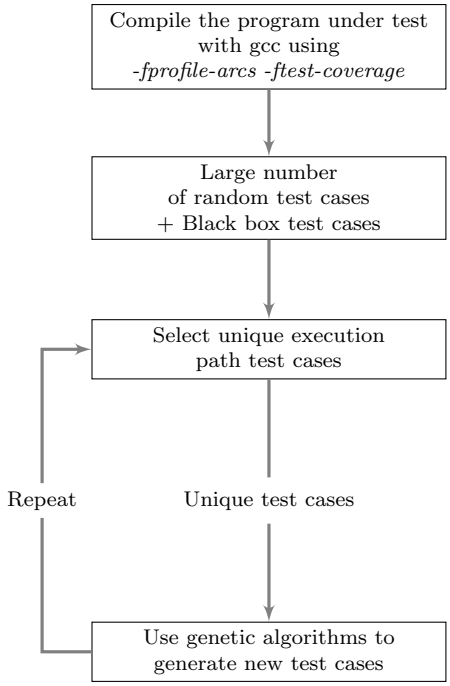


Figure 5.2: *Test case generation using coverage information and genetic algorithms. (The unique execution path test case selection - genetic algorithm cycle is repeated till the required code coverage is achieved).*

5.4 Test generation, verification, and coverage

5.4.1 Test case generation

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

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

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

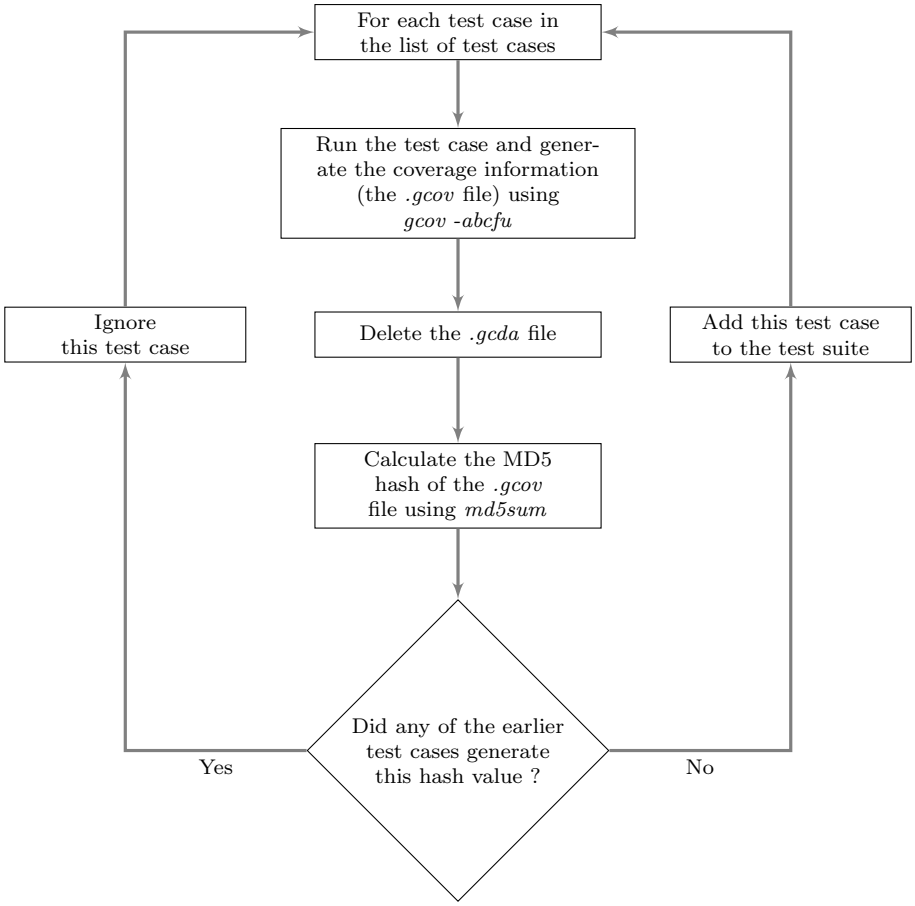


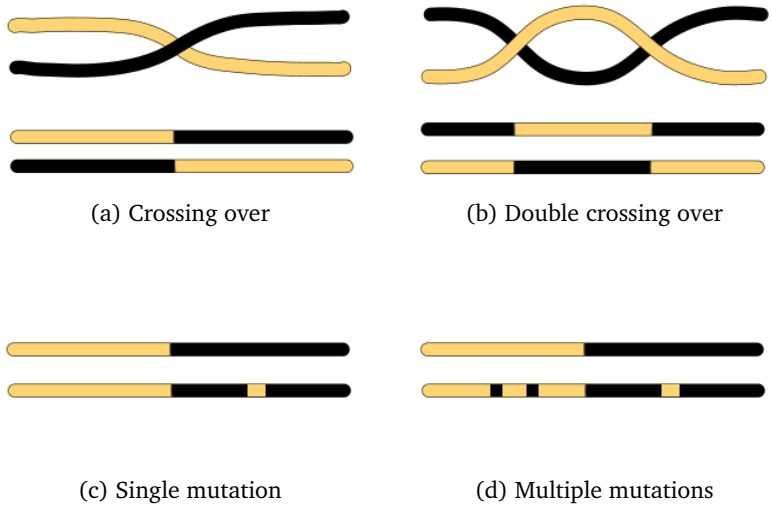
Figure 5.3: *Technique to select unique execution path test cases using gcc, gcov and md5sum. (The -abcfu arguments to gcc implies to display coverage information of: all blocks, branch probabilities, branch counts, function summaries, and unconditional branches. The .gcov file consist of the coverage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

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

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

in each iteration is called a generation. The algorithm attempts to generate new and better individuals from the population for the next generation, based on a predefined fitness function.

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



(a) Crossing over (b) Double crossing over

(c) Single mutation (d) Multiple mutations

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

5.4.2 Verification of test cases

The generated test cases are verified using a model written using the *Drakon* editor [**184**]. The Drakon notations were developed for the Buran space project in Russia [**203–205**] to provide simple and clean graphical notations for program writing. The

Drakon notations can also be used for requirements modeling, and the resultant model is a semi-formal specification of the software. An example of semi-formal specification in drakon for FSHS is shown in *Appendix – A on page 101*.

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

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

5.4.3 Conservative test coverage

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

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

It must be noted that the above metric (*Equation* (5.1)) indicates the test coverage

achieved during system testing, and not during unit testing of a function.

5.5 Mutation testing

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

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

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

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

5.5.1 Mutant properties

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

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

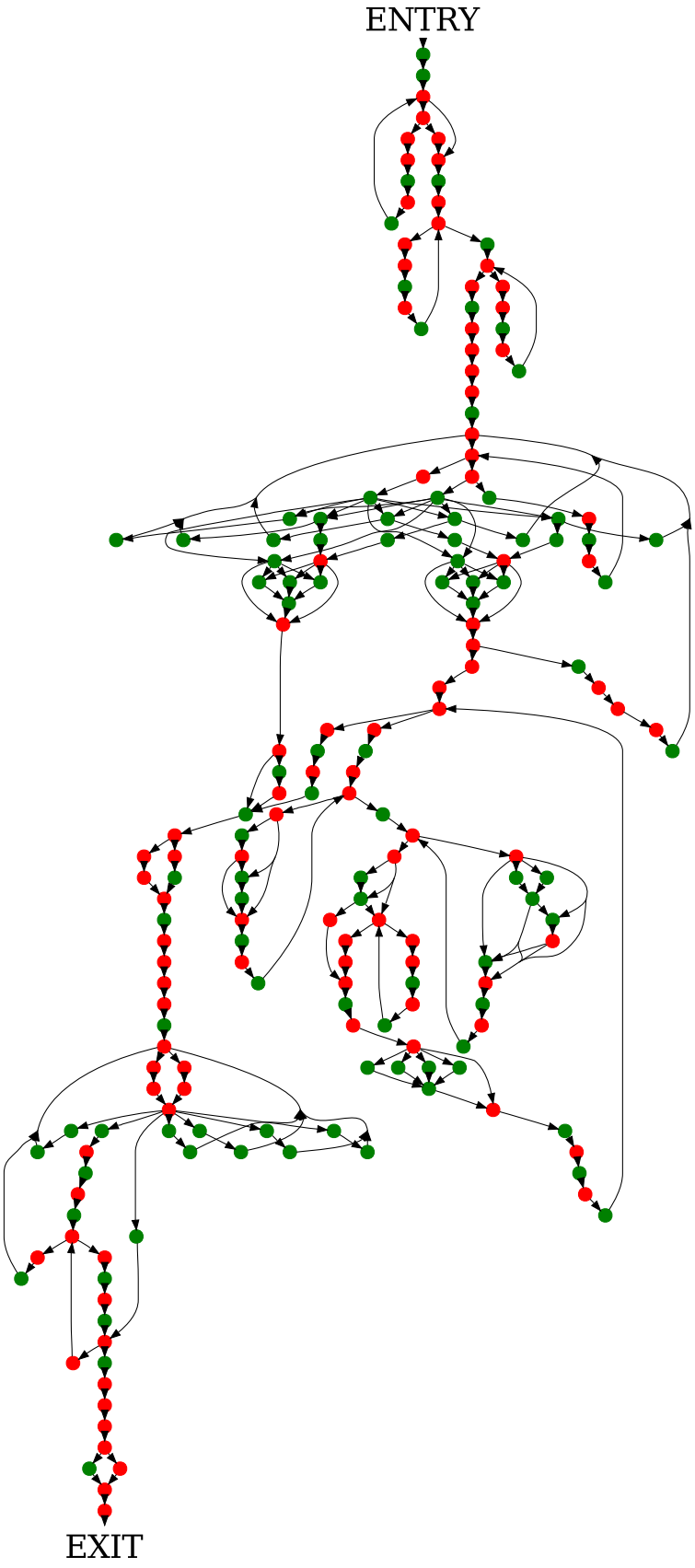


Figure 5.5: *Concatenated LCSAJs for the FSHS. (The green colored nodes indicate the LCSAJ points where faults have been induced and caught; the red colored nodes indicate otherwise)*

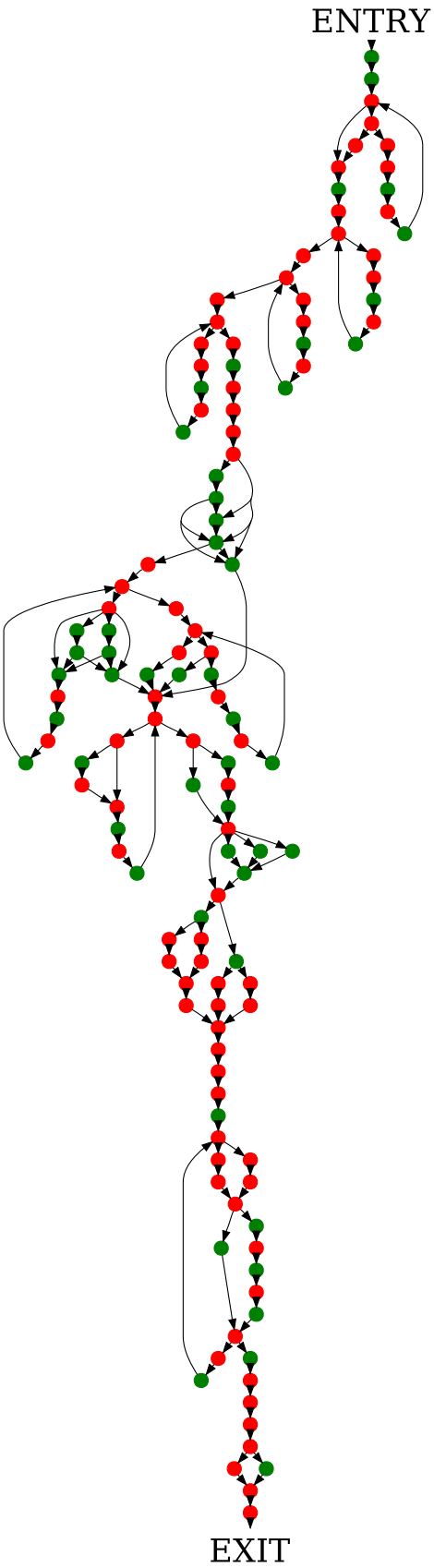


Figure 5.6: *Concatenated LCSAJs for the RSU. (The green colored nodes indicate the LCSAJ points where faults have been induced and caught; the red colored nodes indicate otherwise)*

Figure 5.7: *Concatenated LCSAJs for the SGTLD. (The green colored nodes indicate the LCSAJ points where faults have been induced and caught; the red colored nodes indicate otherwise)*

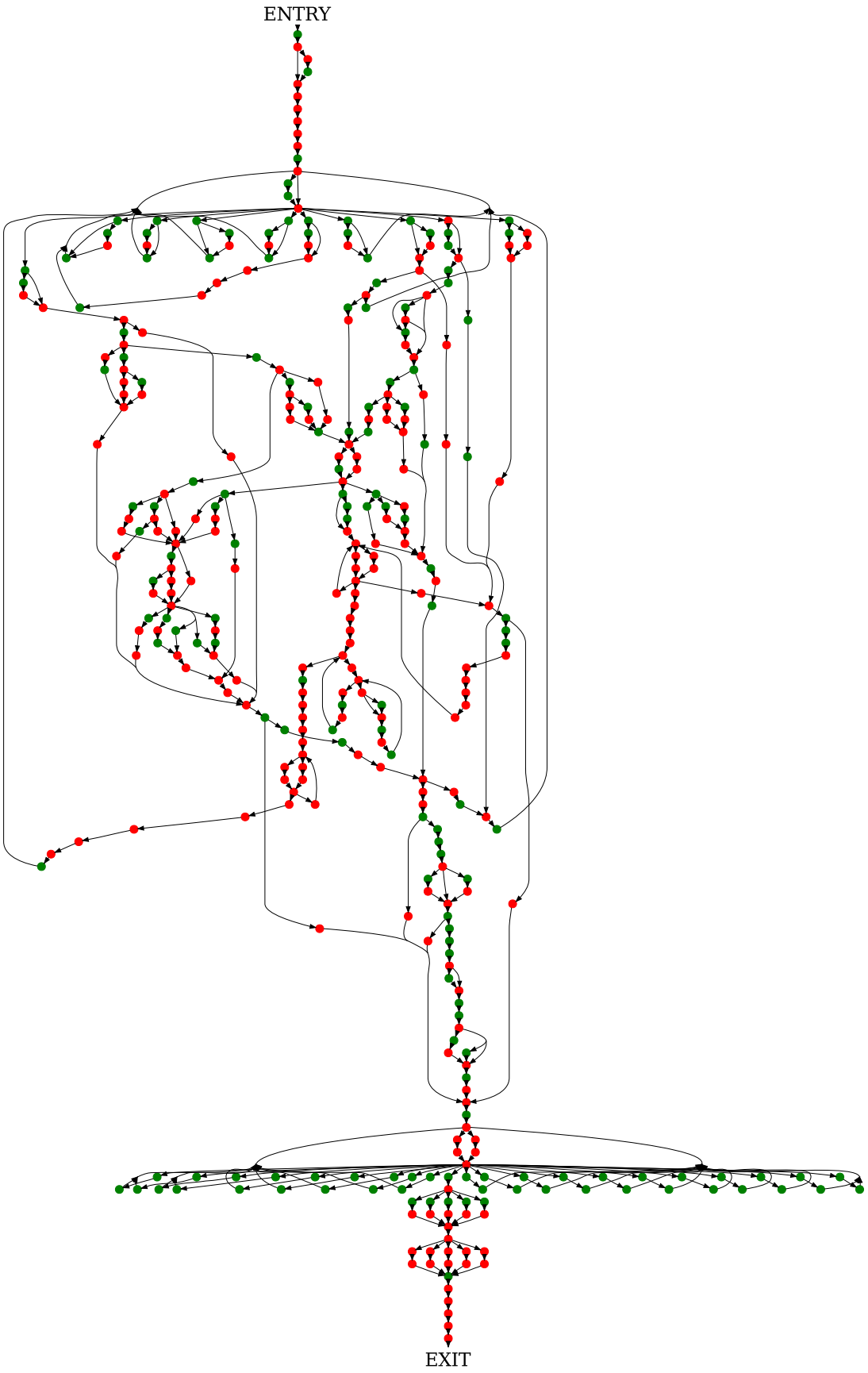


Figure 5.8: *Concatenated LCSAJs for the CTMS. (The green colored nodes indicate the LCSAJ points where faults have been induced and caught; the red colored nodes indicate otherwise)*

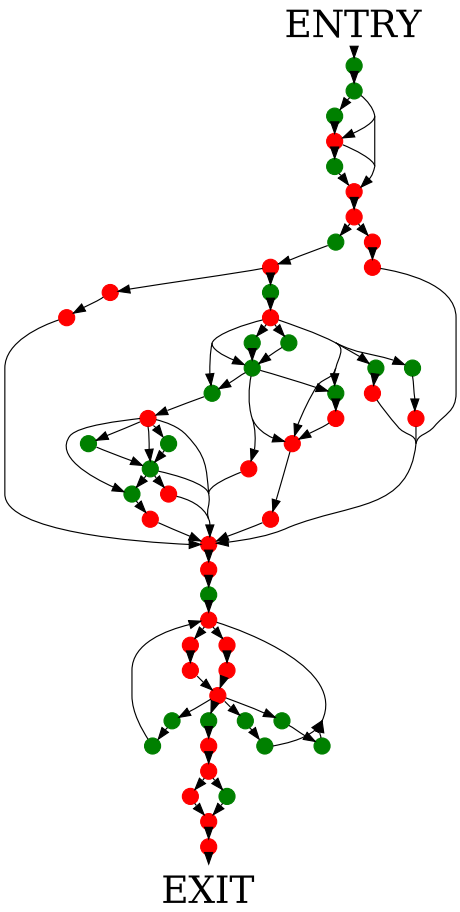


Figure 5.9: *Concatenated LCSAJs for the GES. (The green colored nodes indicate the LCSAJ points where faults have been induced and caught; the red colored nodes indicate otherwise)*

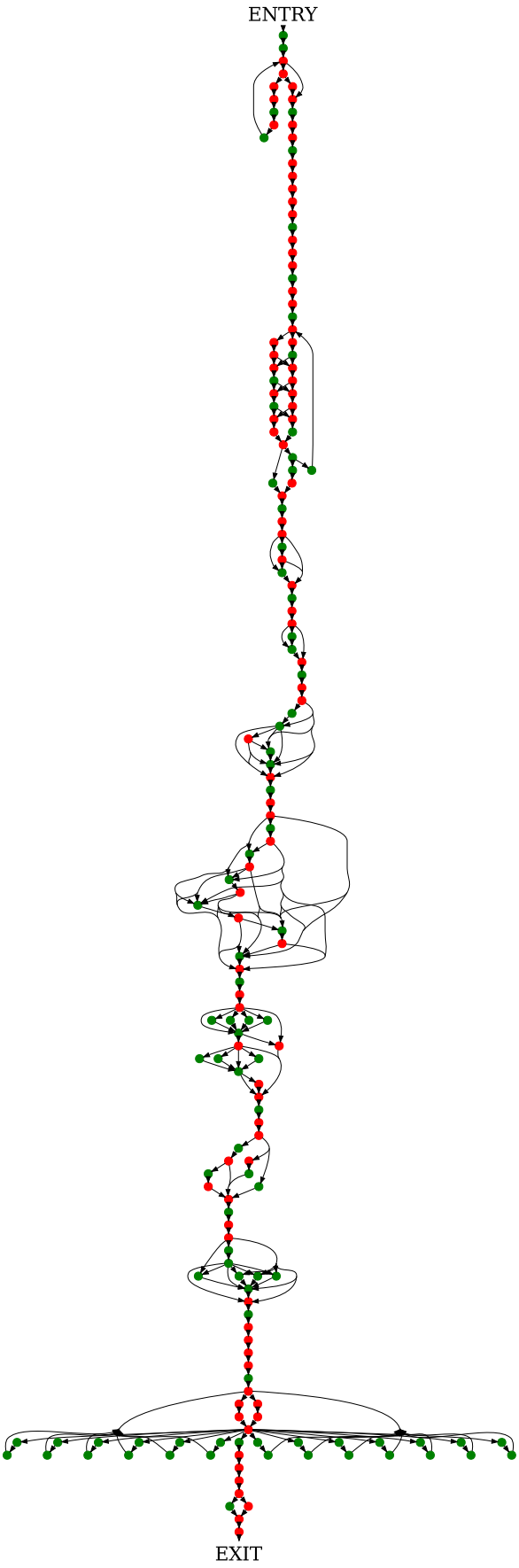


Figure 5.10: *Concatenated LCSAJs for the SGDHR. (The green colored nodes indicate the LCSAJ points where faults have been induced and caught; the red colored nodes indicate otherwise)*

differentiate between killed and unkilled mutants. Whereas, the dynamic analysis (using *Valgrind* [**199**] and *Electric-Fence* [**200**]) indicate that the unkilled mutants are not likely to have any memory corruptions or leaks. Also, the coverage impact (calculated as the average change in the number of times a statement/branch/jump/function-call was executed in a mutant program with respect to the original program) suggests that the majority of unkilled mutants have little or no change in their code coverage.

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

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

5.5.2 Calculating mutant score

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

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

where: K is the number of mutants killed, G is the number of mutants generated, and E is the number of equivalent mutants. And unless all the equivalent mutants are detected, the mutation score will always be < 1.

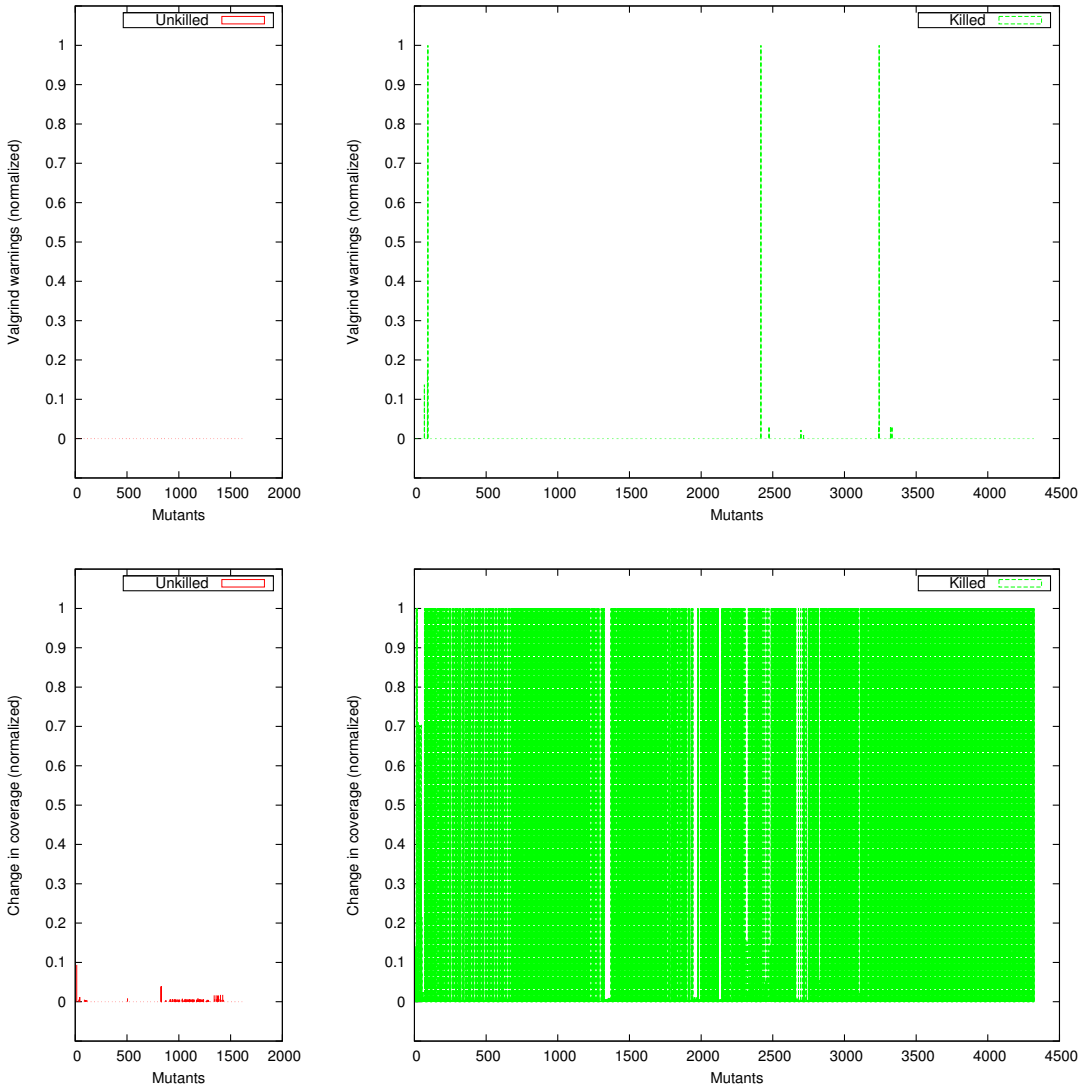


Figure 5.11: *Dynamic analysis of CTMS mutants using: Valgrind and Change in coverage*

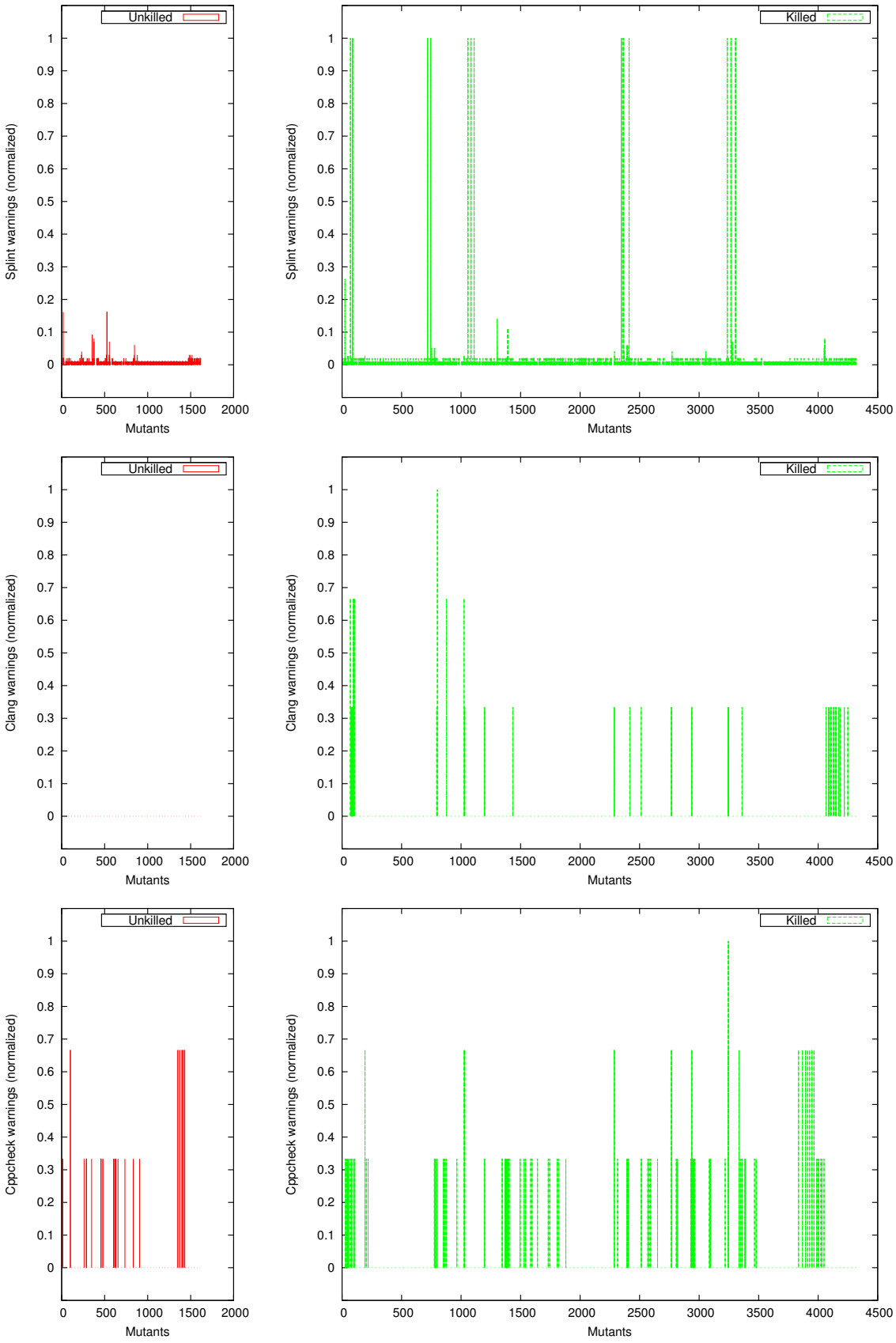


Figure 5.12: *Static analysis of CTMS mutants using: Splint, Clang, and Cppcheck*

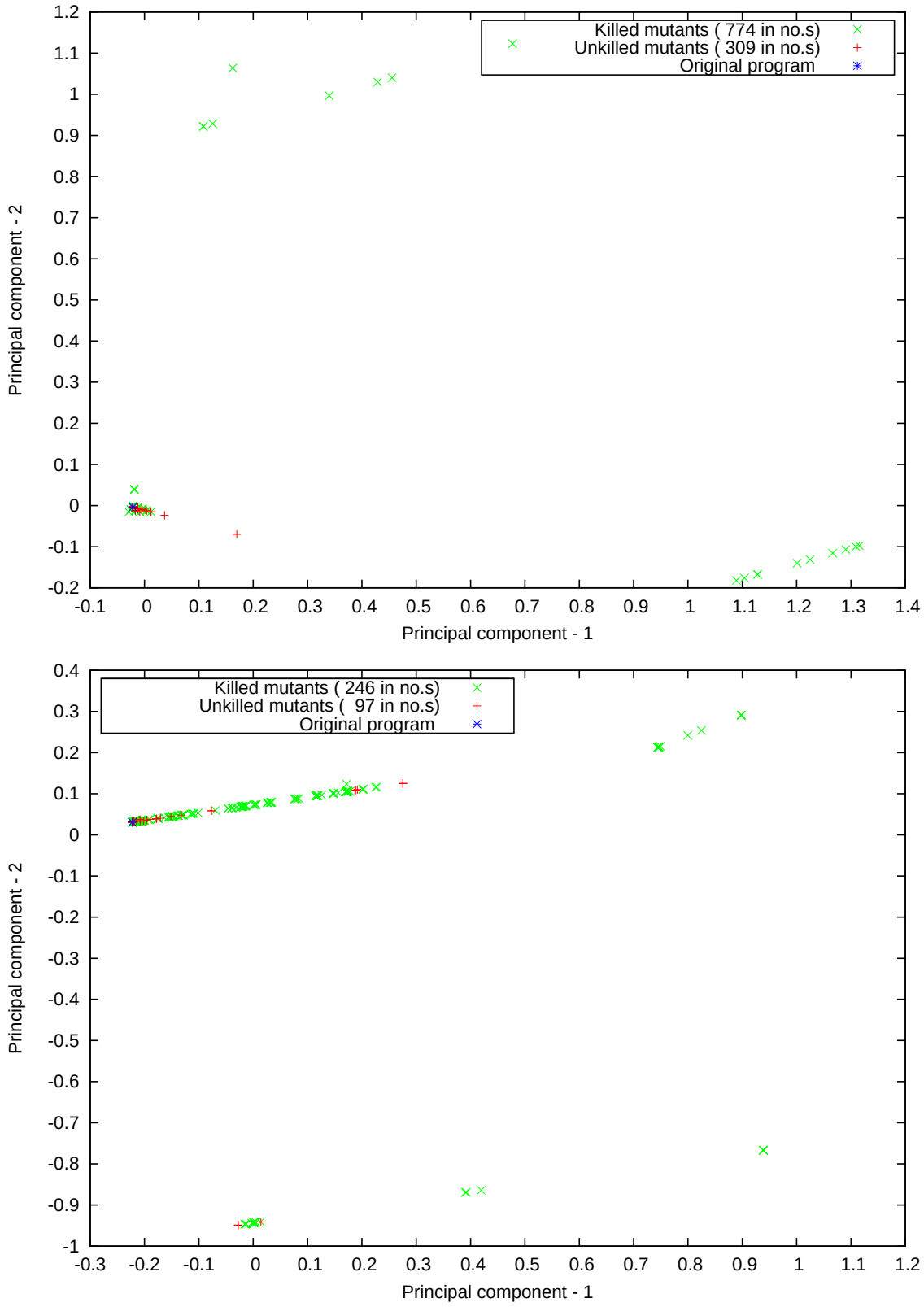


Figure 5.13: *Principal component analysis (PCA) of static, dynamic, and coverage analysis of mutants for: FSHS and RSU*

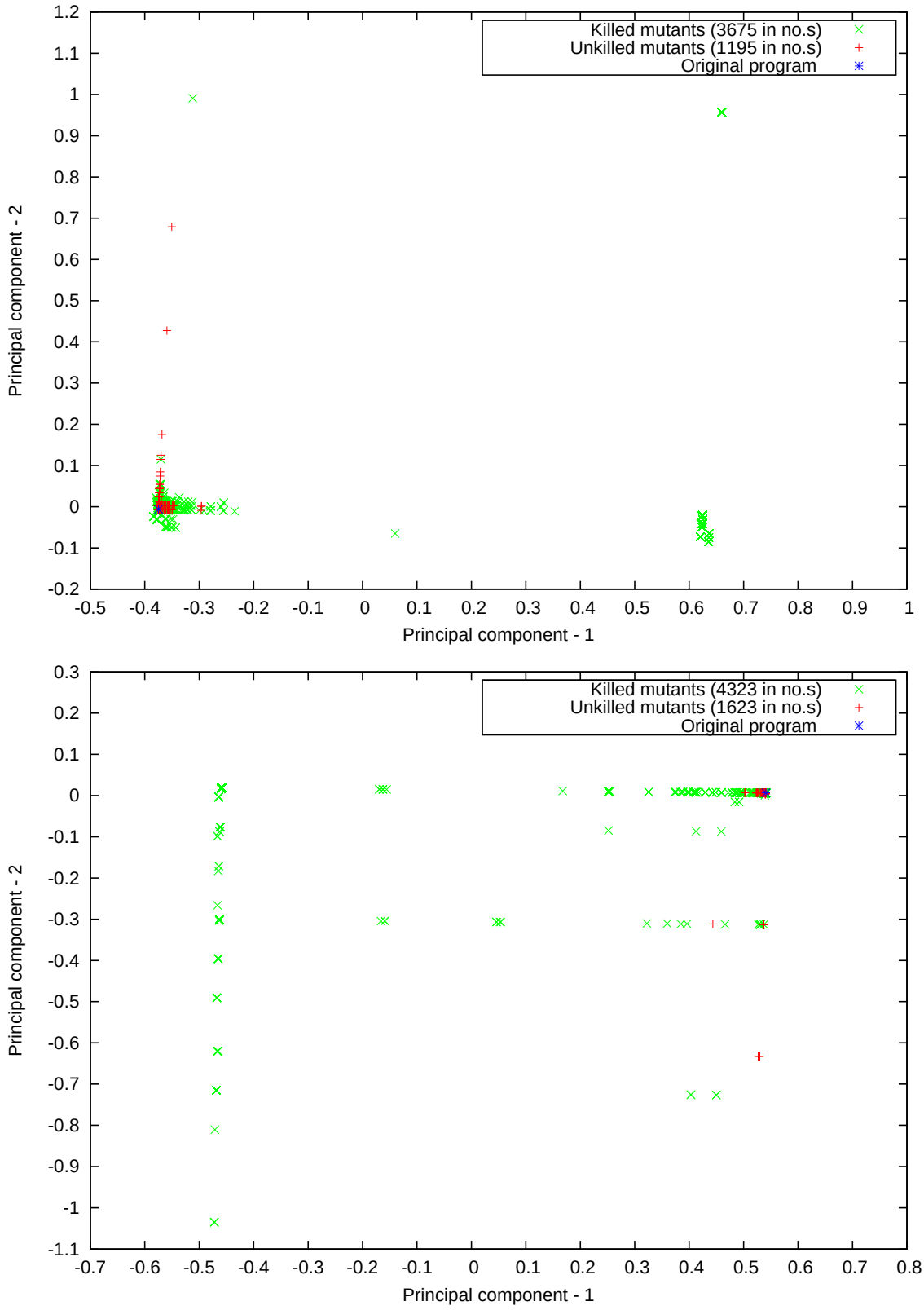


Figure 5.14: *Principal component analysis (PCA) of static, dynamic, and coverage analysis of mutants for: SGTLD and CTMS*

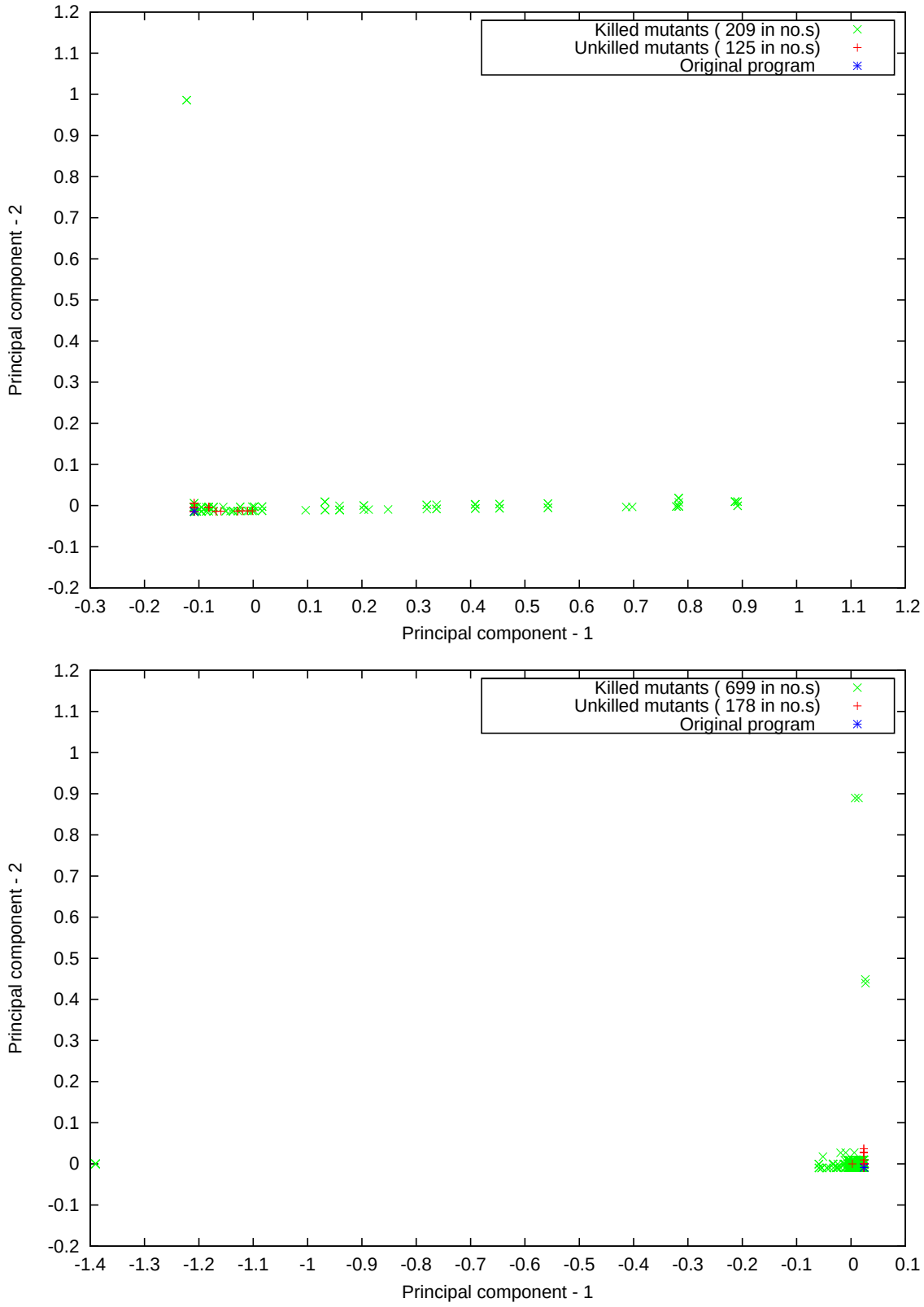


Figure 5.15: *Principal component analysis (PCA) of static, dynamic, and coverage analysis of mutants for: GES and SGDHR*

```
unsigned int Sum (unsigned int array[], size_t length) {

        unsigned int i = 0, sum = 0 ;

        assert ( length > 0 ) ;

        for (i = 0; i < length; ++i) {
                sum += array[i] ;
        }

        return sum ;
}
```

(a)

```
unsigned int Sum (unsigned int array[], size_t length) {

        unsigned int i = 0, sum = 0 ;

        assert ( length > 0 ) ;

        for (i == 0; i < length; ++i) {
                sum += array[i] ;
        }

        return sum ;
}
```

(b)

```
unsigned int Sum (unsigned int array[], size_t length) {

        unsigned int i = 0, sum = 0 ;

        assert ( length > 0 ) ;

        for (i <= 0; i < length; ++i) {
                sum += array[i] ;
        }

        return sum ;
}
```

(c)

Figure 5.16: *Example of mutants with similar static, dynamic, and coverage properties: (a) The original program, (b) Mutant-1, and (c) Mutant-2 (the induced faults are indicated by red color). Both Mutant-1 and Mutant-2 share the same coordinates on the PCA plot.*

Hence, to detect likely equivalent mutants, a technique is proposed, which is based on the principle that: if P is a program, and M is its equivalent mutant created by injecting a fault F in the statement S, then P' (mutant of P) and M' (mutant of M) created by injecting fault(s) F' in statement(s) succeeding S, must also be equivalent *(Figures 5.17 to 5.18 on pages 66–67)*. Assuming an effective set of test cases, if several such equivalent P' and M' are generated; then P and M are likely to be equivalent.

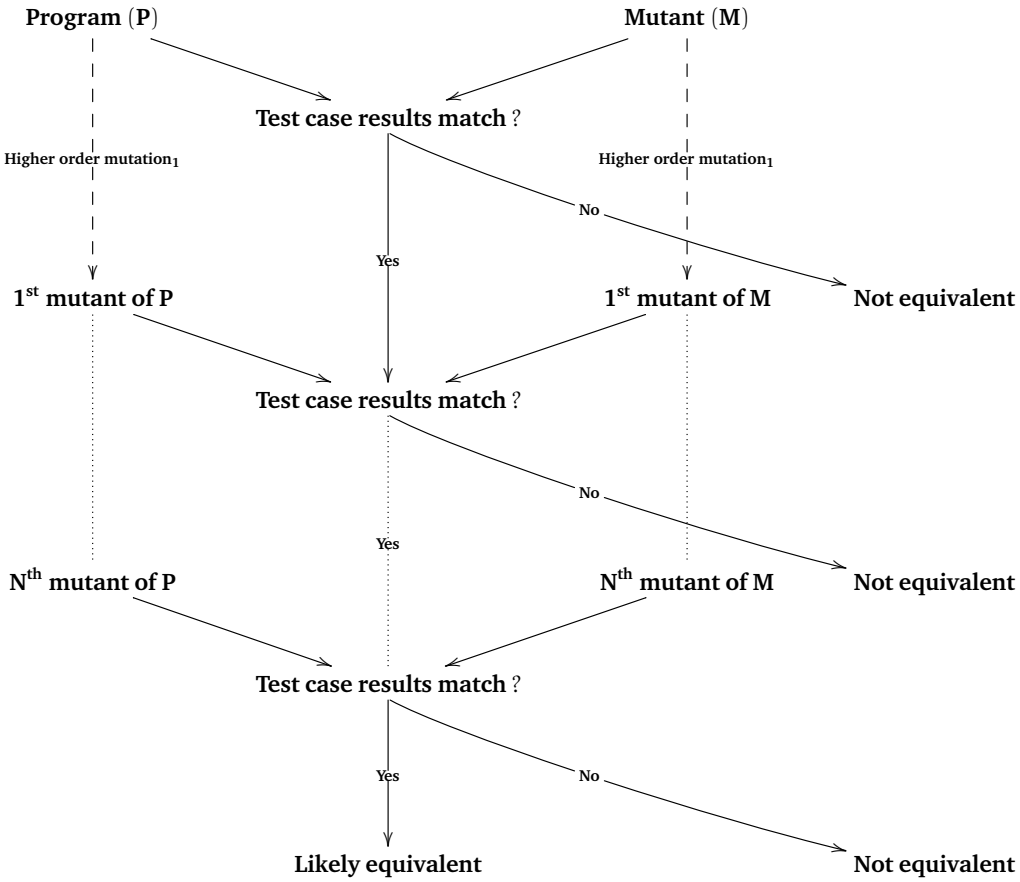


Figure 5.17: *Algorithm for detecting equivalent mutants*

As the above detection algorithm requires creating large number of mutants, it is computationally intensive. To improve the speed of detection, higher order mutation [**212**] is used. Also, as each mutant can be executed in parallel, the algorithm is run on *Intel Xeon X7460 2.66GHz - 24 core* machine using the *multiprocessing* module [**190**] in *Python* [**191**].

```
int Max ( int *array,
          size_t length ) {

    size_t i ;
    int max ;

    assert (length > 0);
    assert (array != NULL);

    max = array[0] ;

    for (i = 1; i < length; ++i)
    {
        if ( array[i] > max )
                max = array [i];
    }

    return max ;
}
```

(a)

```
int Max ( int *array,
          size_t length ) {

    size_t i ;
    int max ;

    assert (length > 0);
    assert (array != NULL);

    max = array[0] ;

    for (i = 1; i != length; ++i)
    {
        if ( array[i] > max )
                max = array [i];
    }

    return max ;
}
```

(b)

```
int Max ( int *array,
          size_t length ) {

    size_t i ;
    int max ;

    assert (length > 0) ;
    assert (array != NULL);

    max = array[0] ;

    for (i = 1; i < length; ++i)
    {
        if ( array [i-1] < max )
                max = array [i];
    }

    return -1 * max ;
}
```

(c)

```
int Max ( int *array,
          size_t length ) {

    size_t i ;
    int max ;

    assert (length > 0) ;
    assert (array != NULL);

    max = array[0] ;

    for (i = 1; i != length; ++i)
    {
        if ( array [i-1] < max )
                max = array [i];
    }

    return -1 * max ;
}
```

(d)

Figure 5.18: *Example of equivalent mutant detection: (a)* P*, the original program; (b)* M*, the equivalent mutant of* P*; (c)* P′*, the mutant of* P*; and, (d)* M′*, the mutant of* M*. (The induced faults are indicated by red color) and keywords by blue color)*

For every unkilled mutant, 10 higher order mutants (each with 10 faults) are generated and are checked for equivalence. The equivalent mutant detection algorithm has detected several non-equivalent mutants and few false positives (i.e. equivalent mutant detected as non-equivalent) (*Table 5.1*). The false positives are identified manually, and further test cases are added to kill the identified non-equivalent mutants.

5.5.3 Threat to validity

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

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

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

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

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

System under test	Number of mutants	No. mutants unkilled	No. non-equivalent mutants detected	False positives
FSHS	1083	309	19	1
RSUP	343	97	9	1
SGTLD	4870	1195	58	5
CTMS	5946	1623	8	7
GES	334	125	4	1
SGDHR	877	178	17	15

Table 5.1: *Results of the equivalent mutant detection algorithm*

the test cases, fails in an unsafe state, then the function in which fault was induced is *safety-critical in nature*. Such functions must have high test coverage.

5.6 Assurance of rigorous testing through test adequacy

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

$$\text{Test coverage} = \frac{\sum t_i \, w_i}{\sum w_i} \tag{5.2}$$

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

$$\begin{aligned} w_i \quad &= \text{No. of statements} \\ &\times \text{Cyclomatic complexity} \\ &\times \text{Frequency of the function call} \\ &\times \text{Safety-critical nature} \end{aligned} \tag{5.3}$$

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

$$\text{Test adequacy} = \text{Test coverage} \times \text{Mutation score} \tag{5.4}$$

5.7 Results

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

System under test	No. of unique test cases	No. of unique execution path test cases	Test adequacy
FSHS	$> 3 \times 10^5$	302174	0.9001
RSUP	$> 3 \times 10^5$	10772	0.8824
SGTLD	$> 3 \times 10^5$	378554	0.9806
CTMS	$> 3 \times 10^5$	311002	0.9922
GES	$> 3 \times 10^5$	95	0.8600
SGDHR	$> 3 \times 10^5$	98118	0.9655

Table 5.2: *Test adequacy achieved in case studies*

5.8 Summary of results

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

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

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

3. To detect equivalent mutants — a technique to identify equivalent mutants has been demonstrated. The proposed technique when applied to the case study has resulted in mutation score ≈ 1.

Using the proposed method, high test adequacy in the case studies has been achieved. The computed test adequacy (ignoring the unfeasible MC/DCs and LCSAJs), indicate the rigor in software testing carried out. The regulators in safety-critical industries may require the software reliability estimate before permitting the software to be used in the field. The test adequacy value serves as one of the inputs for the software reliability estimate.

6

Quantification of software reliability

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

6.1 Prerequisites for the approach

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

6.1.1 Set of test cases

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

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

6.1.2 Set of mutants

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

6.1.3 A test oracle

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

6.1.4 Test adequacy computation

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

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

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

6.1.5 Compiler correctness

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

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

To achieve the above, *three* compilers are used to match results of test cases covering all MC/DCs, all LCSAJs of the program under test, and with mutation score ≈ 1. And, if

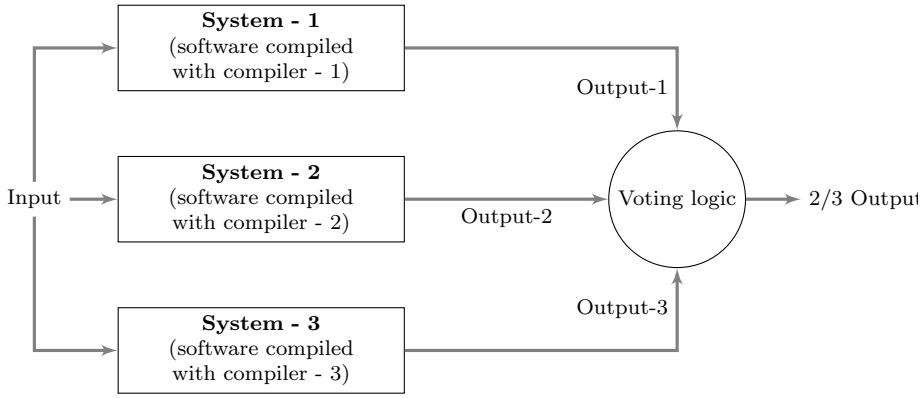


Figure 6.1: *A 3-version system, where each system runs the same software, but compiled by three different compilers*

all the *three* compilers (in the present study: *gcc* [**215**], *llvm* [**192**], and *pcc* [**216**, **217**]) agree to the outputs of all the test cases, then: the likelihood of incorrect code produced by any of the *three* compilers is ≈ 0. Also, the redundant architectures using multiple compilers (*Figure 6.1*) may provide little diversity similar to the N-version programming [**41**].

6.2 Software reliability estimation

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

6.2.1 Approach $-$ 1

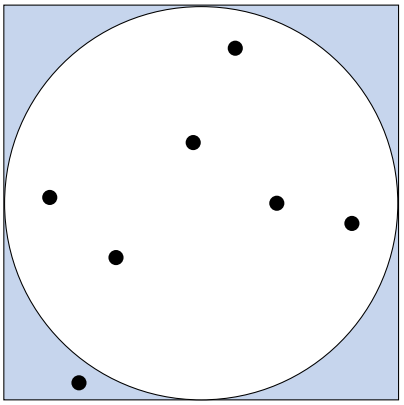
If the test oracle or model is an exact representation of the system under test, then: the reliability of the software can be estimated by:

$$\text{Test adequacy} \times \frac{\text{No. of test cases verified}}{\text{Total no. of test cases}} \tag{6.1}$$

The above approach requires true oracle or pseudo oracle [**142**]. However, in most of the real life applications, the software under test is large and complex. Hence, true/pseudo oracle may not be available. For such systems: *Approach-2* and *3* are appropriate (*Sections 6.2.2 to 6.2.3 on pages 75–76*).

6.2.2 Approach − 2

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



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

$$\frac{Area\ of\ circle}{Area\ of\ square} = \frac{\pi}{4}$$

$$\pi = 4 \times \frac{Area\ of\ circle}{Area\ of\ square}$$

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

Figure 6.2: *Monte-Carlo method of determining the value of* π

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

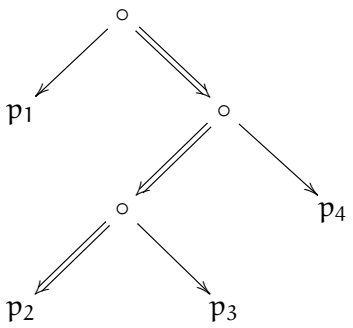


Figure 6.3: *Example of paths in a program, where reliability of the path* p_2 *is known (indicated by* $\Rightarrow$*)*

In the method of π value calculation, a random dart may either fall inside the circle or outside it; similarly, an induced fault may have three possible outcomes (i.e. result of the mutant execution): (i) it fails at least one of the verified test cases, indicating that the fault has been induced in a verified path; (ii) passes all verified test case but fails at least one of the un-verified test cases, indicating that fault has been induced in an un-verified path; (iii) does not fail any of the test cases (*the unkilled mutant*), indicating that the induced fault may not have any effect on the program.

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

$$\text{Test adequacy} \times \frac{\text{No. of times at least one of the verified test cases failed}}{\text{No. of mutants killed}} \tag{6.2}$$

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

6.2.3 Approach − 3

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

6.2.3.1 Estimating fraction of shared code

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

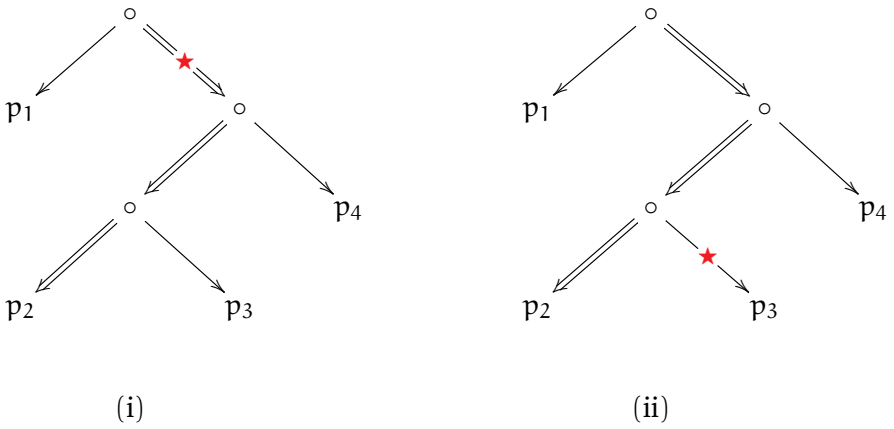


Figure 6.4: *Faults induced in path* p_3. *(The symbol* $\Rightarrow$ *indicates a path whose reliability is known, and* ★ *indicates an induced fault.)*

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

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

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

$$\text{Fraction of code shared between } p_i \text{ and } p_2 = \frac{F_{i2}}{F_{22}}$$

where, F_{i2} is number of times test cases running through path p_i has failed, given that a fault was induced in path p_2; and F_{22} is number of times test cases running through path p_2 has failed, given that a fault was induced in path p_2.

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

6.2.3.2 Pseudocode of the approach

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

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

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

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

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

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

7. **for each** mutant m **in** M :

   (a) $I_m = \phi$

   (b) **for each** un-verified test case t_i **in** T :

   if t_i kills the mutant m **then**:

   $$U_i \leftarrow U_i + 1$$
   $$I_m \leftarrow I_m \cup \{i\}$$

   (c) **if** $I_m = \phi$ **then**

   ignore the mutant m **and continue** with next mutant in $step - 7$.

else if $\exists\ t$ **in** T such that t **is** a verified test case **and** kills the mutant m **then**

$\forall i$ **in** I_m :

$V_i \leftarrow V_i + 1$

end if

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

$$9.\ \text{Reliability} = \begin{cases} adequecy(T) \times \dfrac{\sum_{i=1}^{N} F_i}{N} \\ \quad \text{(if all paths are equally likeley to be executed)} \\ adequecy(T) \times \dfrac{\sum_{i=1}^{N} (F_i \times O_i)}{N} \\ \quad \text{(if the path } p_i \text{ has the probability } O_i \text{ of execution} \\ \quad \text{i.e. the operational profile)} \end{cases}$$

6.3 Theoretical results

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

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

$$
\begin{aligned}
\text{Reliability} &= adequacy(T) \times \frac{\sum_{i=1}^{N} F_i}{N} \\
&= adequacy(T) \times \left(\frac{P + \sum_{i=1}^{N-P} F_i}{N} \right) \\
&= adequacy(T) \times \left(\frac{P}{N} + \frac{\sum_{i=1}^{N-P} F_i \times (N-P)}{(N-P) \times N} \right) \\
&= adequacy(T) \times \left(x + y \times \frac{(N-P)}{N} \right) \\
&= adequacy(T) \times \left(x + y \times \left(1 - \frac{P}{N} \right) \right) \\
&= adequacy(T) \times (x + y \times (1-x)) \\
&= adequacy(T) \times (x + y - xy) \qquad (6.3)
\end{aligned}
$$

6.3.1 Factors affecting the estimated reliability

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

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

6.3.2 Achieving target reliability

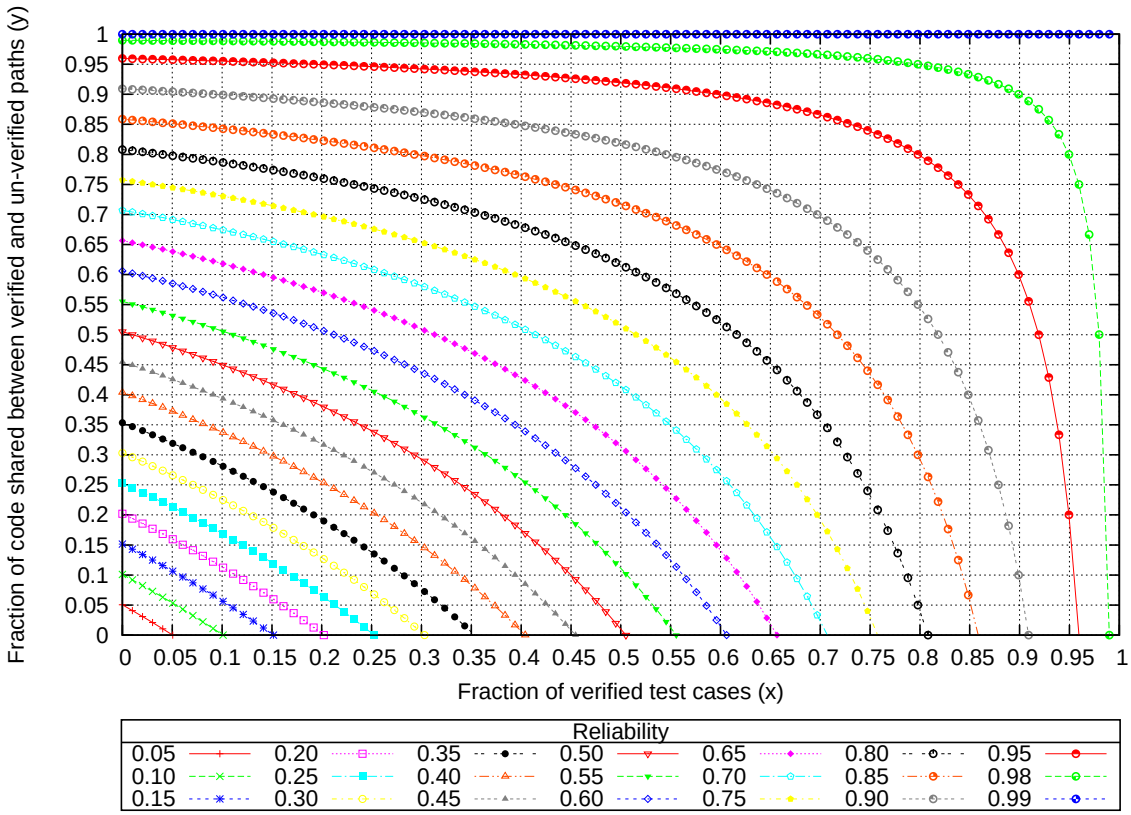


Figure 6.5: *Contour graph showing the combination of x and y for various reliability values (0.05-0.99), when test adequacy is 0.99.*

The *Equation* (6.3) *on the previous page* helps in choosing the combination of x and y

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

6.3.3 Properties of the software

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

1. When $x \approx 0$ and $y \approx 0$:

   Almost no verification has been carried out for the software. Hence, rigorous software verification must be recommended for such systems.

2. When $x \approx 0$ and $y \approx 1$:

   The software has very high reusability. Though, the estimated reliability seems to be high, to improve the confidence on the reliability estimate, more software verification must be recommended for such systems.

3. When $x \approx 1$ and $y \approx 0$:

   Nearly all the generated paths have been verified, and very few groups of un-verified paths have been left out, which do not share much code with the verified paths.

4. When $x \approx 1$ and $y \approx 1$:

   An ideal scenario where almost all the generated paths have been verified. Few small groups of un-verified paths have been left out, which share most of code with the verified paths.

The values of x and y contribute equally to the estimated reliability. However, achieving 100% verification (i.e. $x = 1$) could be difficult, as it requires a true test

oracle. The value of γ may be improved by reusing verified code. Hence, achieving high reliability for software with high reusability is relatively easier.

6.4 Results, discussions, and critical review

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

$$\text{The failure probability of software} \quad < \quad \frac{1}{N} \tag{6.4}$$

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

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

Therefore:

- If the probability that a single experiment output will fail to fall in Ω is γ.
- Then, the probability that all N experimental outputs fail to fall in Ω is γ^N.

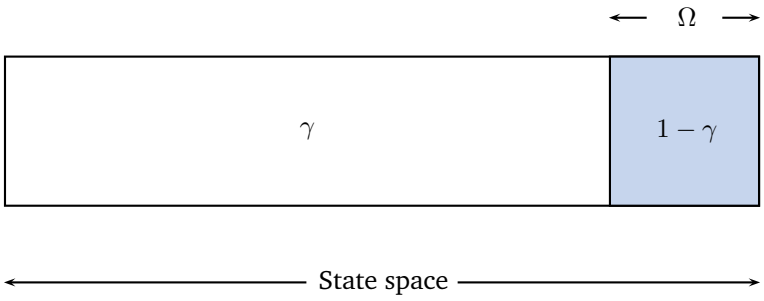


Figure 6.6: *Wilks criteria (Here γ indicates the probability that a single experiment output will not fall in the failure domain (Ω))*

- And, the probability that at least 1 experiment output is in Ω will be $1-\gamma^N$.

Then:

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

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

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

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

$$N = \frac{\log(1-0.95)}{\log(1-10^{-5})} \approx 3\times10^5 \text{ test cases}$$

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

1. Increasing the number of unique test cases.
2. Improving the effectiveness of the mutation testing by using good number of mutant operators which can induce realistic faults into the software.

3. Reducing the uncertainty in mutation score calculation by detecting equivalent mutants correctly.

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

6.5 Summary of results

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

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

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

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

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

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

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

7

Some properties of software reliability

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

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

7.1 Software reliability *vs.* number of faults in the software

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

For each mutant in all the case studies, the software reliability is estimated, and relationship between the average estimated software reliability *vs.* defect-density/the

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

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

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

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

To understand if there exists any relationship between static/dynamic analysis and the estimated reliability, for all the 7500 mutants in each cases study; the average of warnings during static analysis using splint [**194**], clang [**192**], and cppcheck [**193**]; and errors/warnings during dynamic analysis using Valgrind [**199**] and Electric-Fence [**200**] are analyzed. The results (*Figures 7.5 to 7.6 on page 92*) indicate that the warnings/errors found during dynamic analysis decreases exponentially as the reliability increases. However, no significant relationship could be found between static analysis and the estimated software reliability.

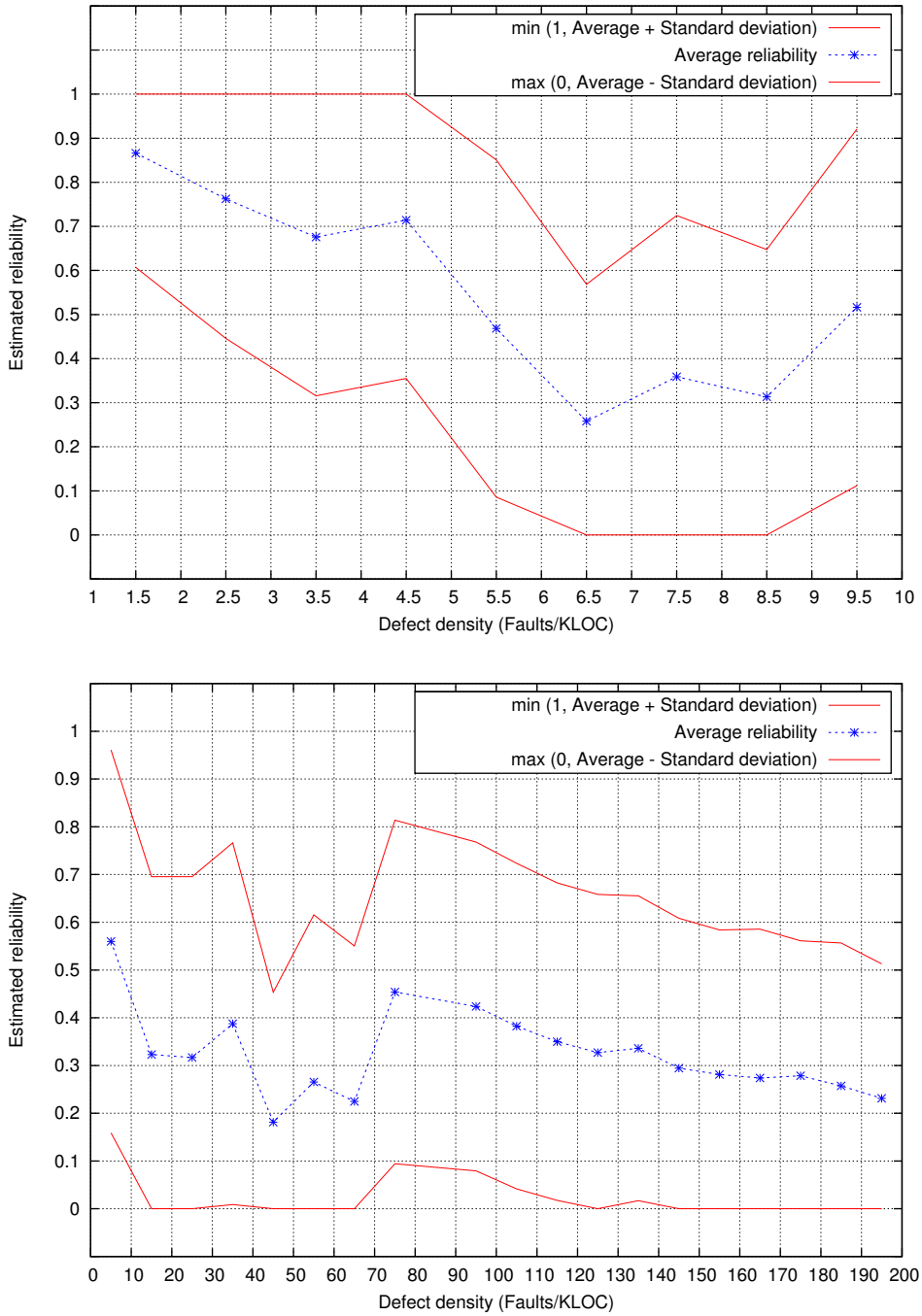


Figure 7.1: *Estimated reliability vs. the defect density (in KLOC) for all the case studies. As the software under test is ≈ 1 KLOC, the results for defect density < 1 Defects/KLOC cannot be plotted. (The upper and lower bounds indicate the ± 1σ limit, and the software reliability is in the range [0,1])*

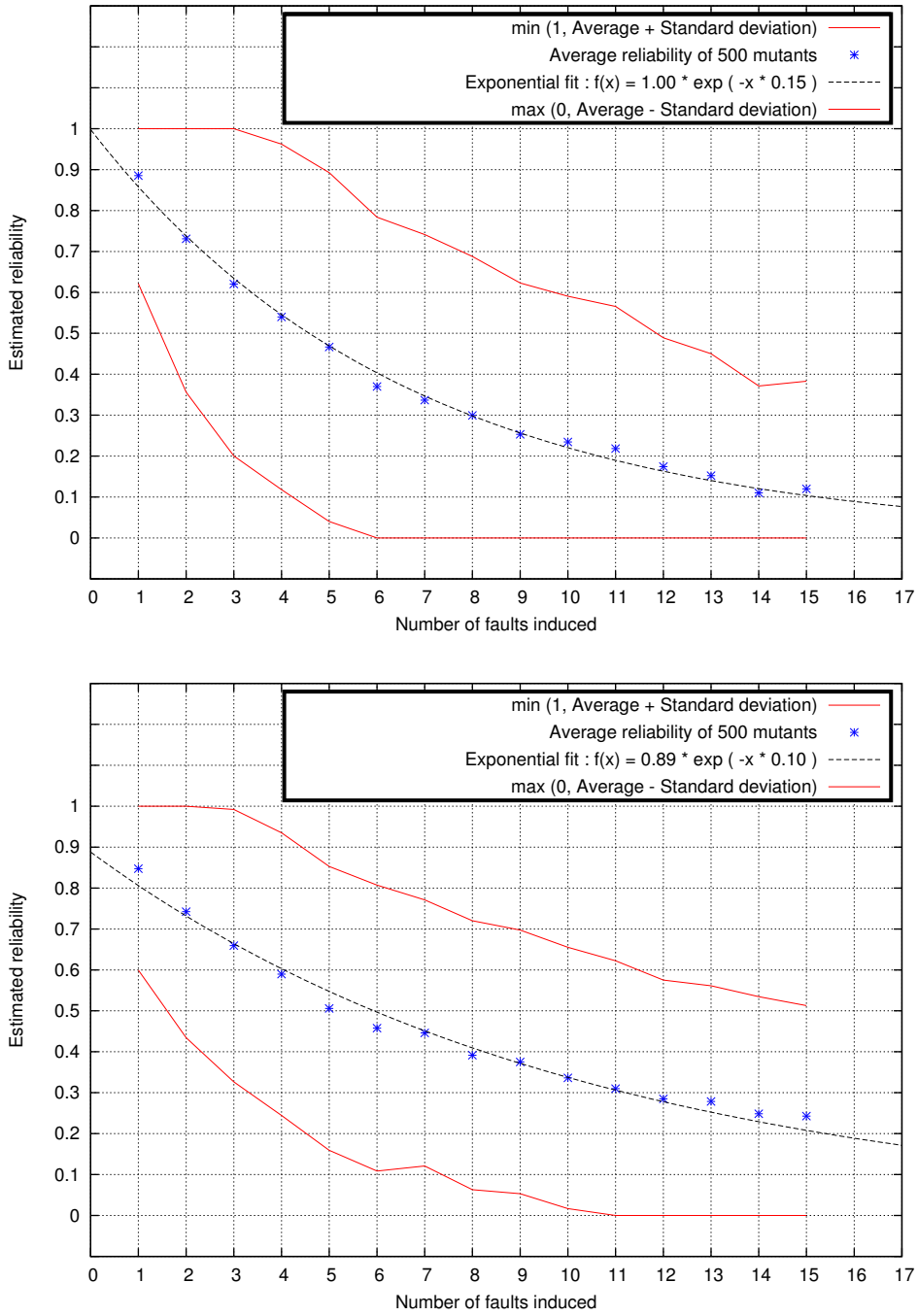


Figure 7.2: *Estimated reliability vs. the number of induced faults — for FSH and RSU (The upper and lower bounds indicate the $\pm$ 1σ limit, and the software reliability is in the range [0,1])*

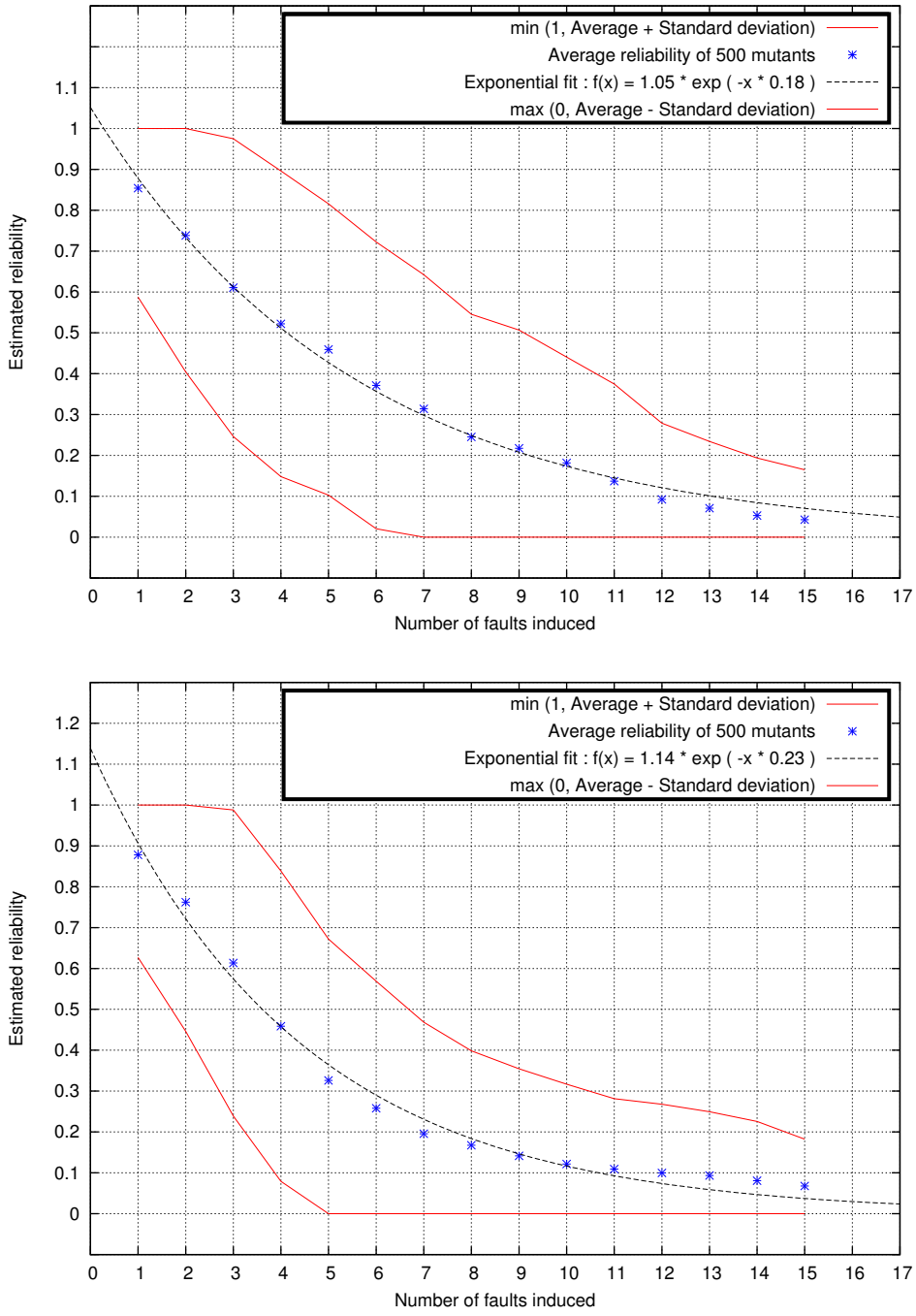


Figure 7.3: *Estimated reliability vs. the number of induced faults — for SGTLD and CTMS (The upper and lower bounds indicate the $\pm$ 1σ limit, and the software reliability is in the range [0,1])*

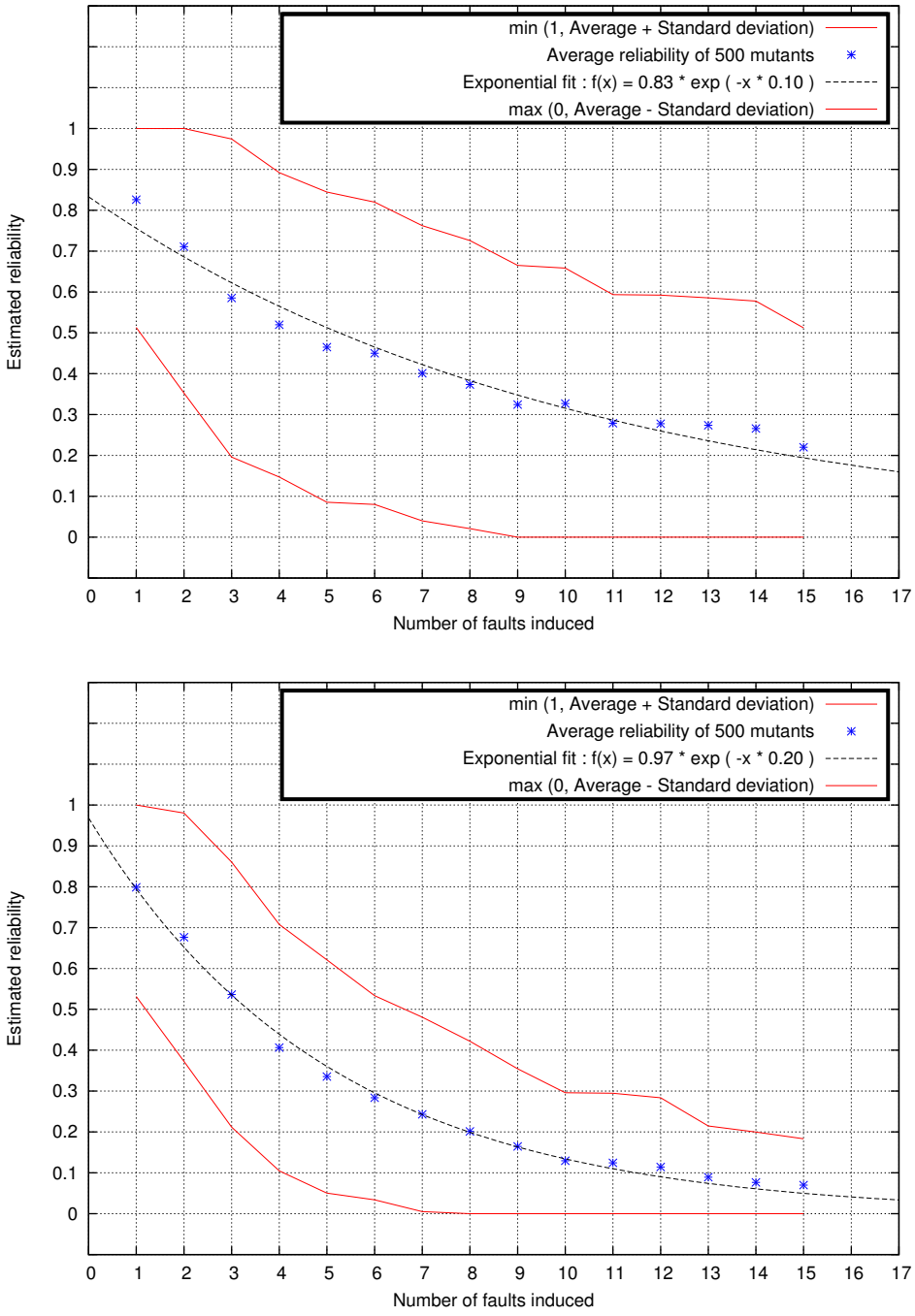


Figure 7.4: *Estimated reliability vs. the number of induced faults — for GES and SGDHR (The upper and lower bounds indicate the $\pm$ 1σ limit, and the software reliability is in the range [0,1])*

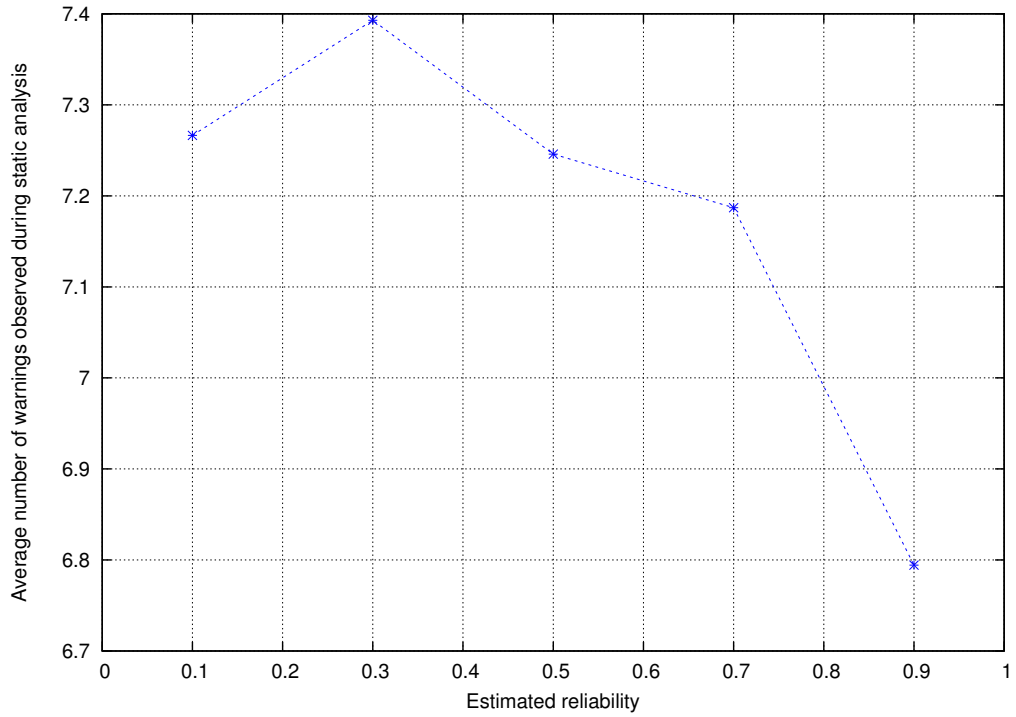


Figure 7.5: *Estimated reliability vs. the number of warnings found during static analysis for the all case studies*

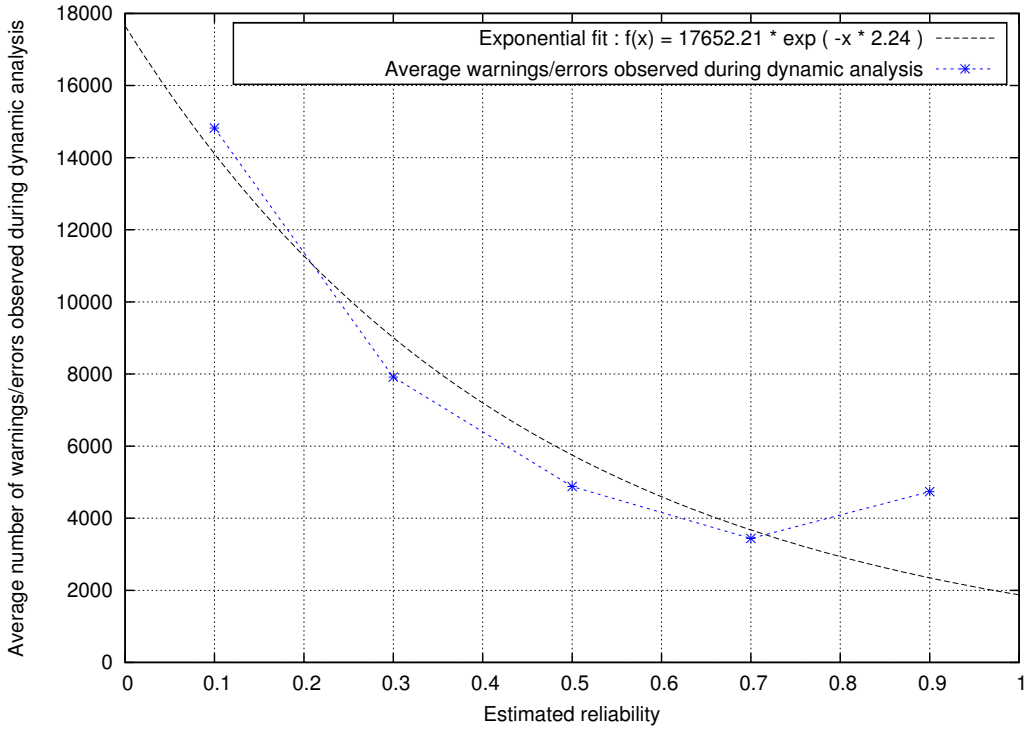


Figure 7.6: *Estimated reliability vs. the number of errors found at dynamic analysis for the all case studies*

7.3 Software reliability *vs.* safety

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

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

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

$$\left.\begin{array}{ll} S_w & = (s_1 \times w_1,\ s_2 \times w_2,\ s_3 \times w_3\ ,\ ....\ ,\ s_n \times w_n), \quad \text{where:} \\ & \\ s_i & \text{is the number of times } i^{th} \text{ output has led to a safety action, and} \\ & \\ w_i & \text{is the weight associated with the safety-critical nature of the output } i \end{array}\right\} \tag{7.2}$$

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

For each mutant in the all case studies, the safety indicator (*Equation* (7.1)) and its corresponding estimated software reliability based on *Section 6.2.1 on page 74* is estimated and plotted. To plot the relationship between software reliability and safety, the software reliability is averaged at every 0.2 intervals of safety (indicated by the *blue* line in *Figure 7.7 on page 95*). As an ideal safety-critical software has safety = 1 and

reliability = 1 (indicated by the *yellow* colored line in *Figure 7.7 on the next page*), it is also shown along with the experimental results. Also, as the magnitude alone cannot distinguish between two vectors, the angle between T_w and E_w defined as:

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

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

7.4 Summary of results

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

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

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

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

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

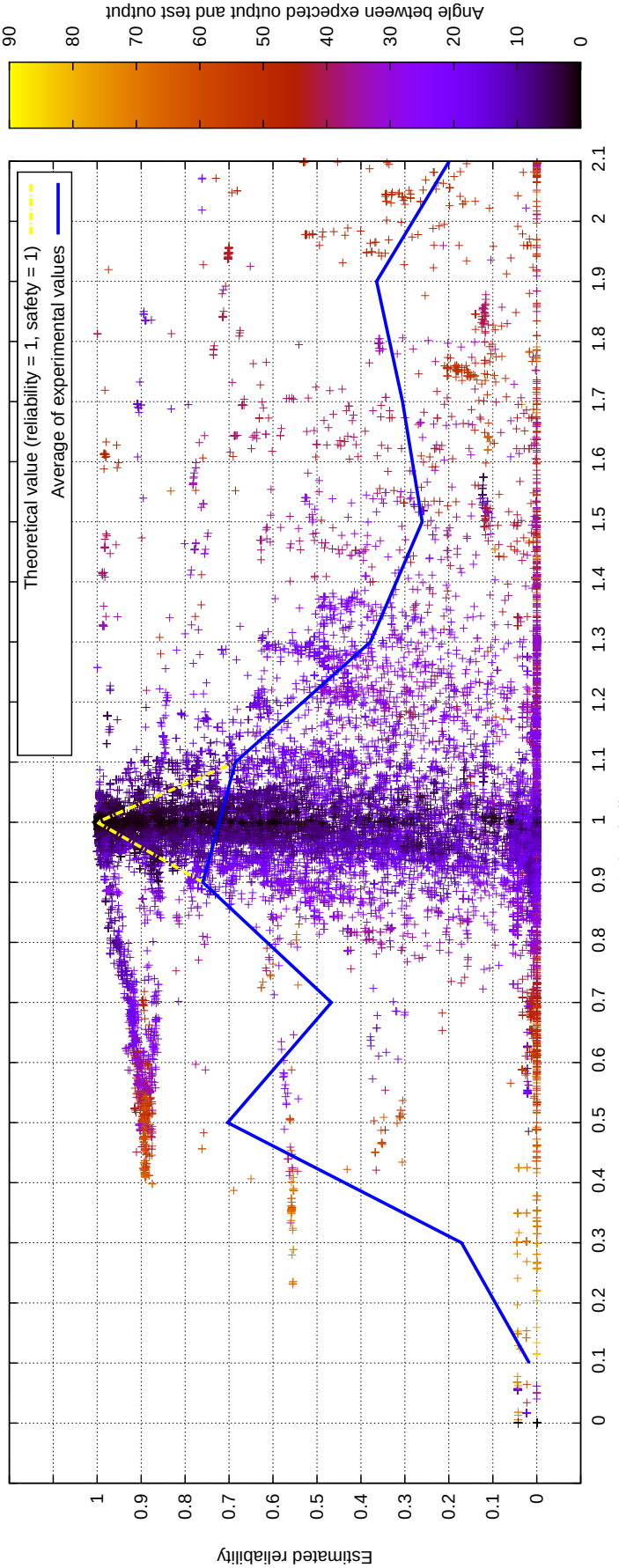


Figure 7.7: *Estimated reliability vs. the safety indicator, for all the case studies*

8

Summary and open problems

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

8.1 Contributions

The present study makes the following contributions:

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

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

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

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

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

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

8.2 Observations

The following observations are made during the study:

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

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

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

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

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

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

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

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

9. The average warnings/errors observed during dynamic analysis decreases exponentially as the reliability increases. However, no conclusive relationship was found between software reliability and warnings observed during static analysis (*Section 7.2 on page 87*).

10. For safety-critical software, the required safety can be achieved by improving the reliability; however vice-versa is not always true (*Section 7.3 on page 93*).

8.3 Open problems

The following scope exists for further work:

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

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

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

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

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

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

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

8.4 Conclusion

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

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

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

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

Part III

Appendices

A

Semi-formal software specification

As mentioned in *Section 5.4.2 on page 50*, *Drakon* [**184**] notations are used in the present study to build test oracle. Below is the list drakon notations used in the present work:

A.1 List of *Drakon* notations

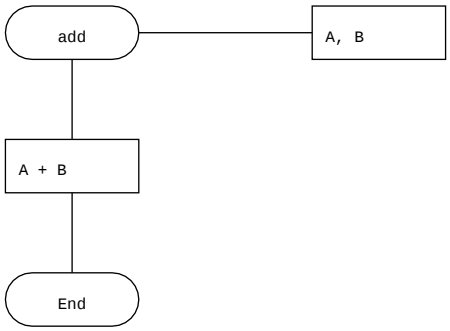


Figure A.1: *An example of a function "add", returning the sum of its parameters: A and B*

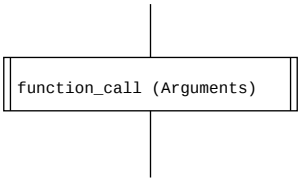


Figure A.2: *A function call*

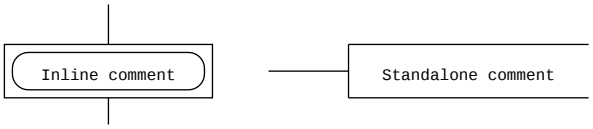


Figure A.3: *Inline and standalone comments*

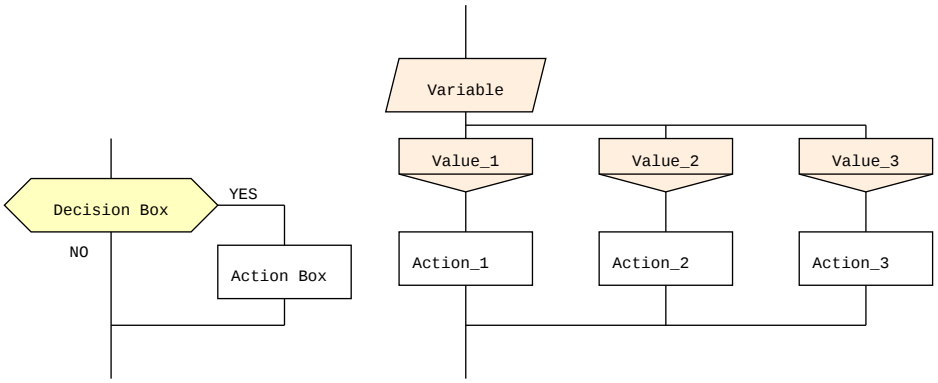


Figure A.4: *Control flow: (a) decision box, (b) switch case*

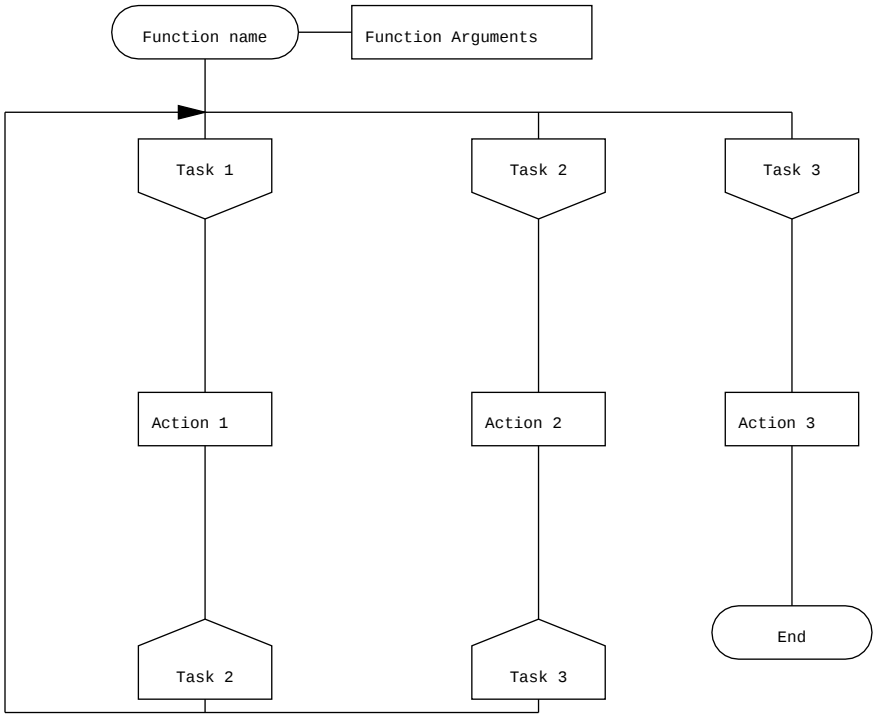


Figure A.5: *An example of branches. The order of execution is Task 1,2,3*

Although, very few notations in Drakon are suitable for Erlang; the flexibility in specification is achieved through functional programming constructs of Erlang programming language such as: list comprehension, map, filter, fold, recursions, etc.

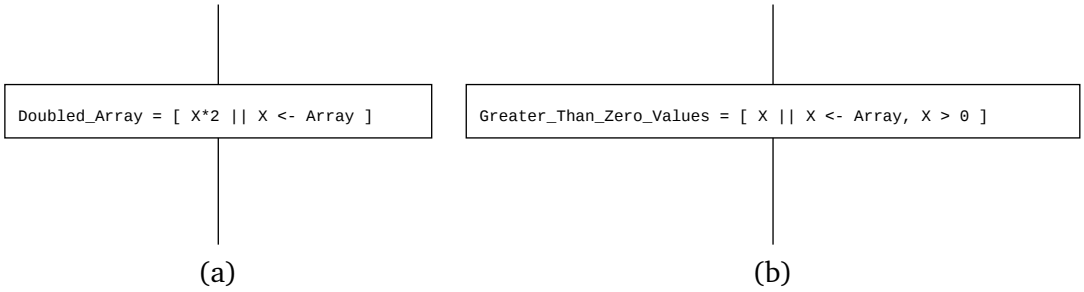
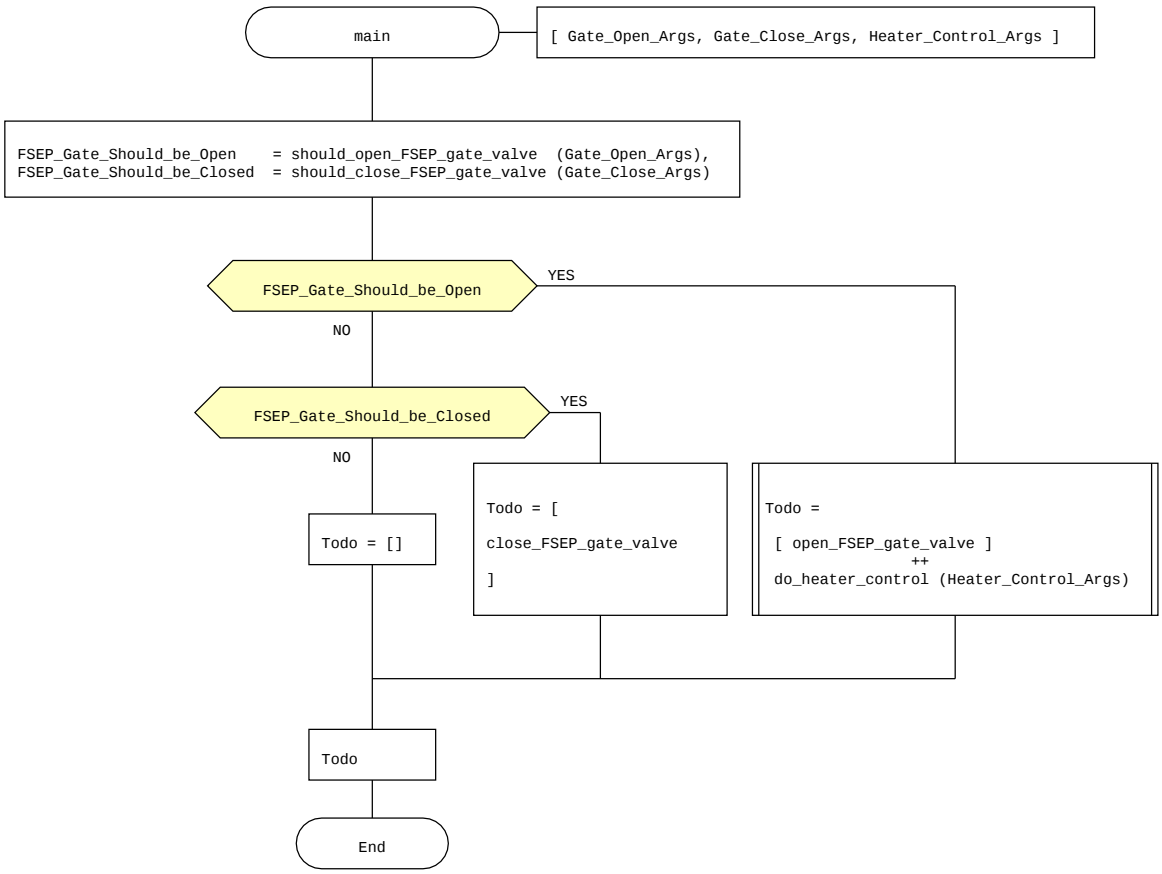
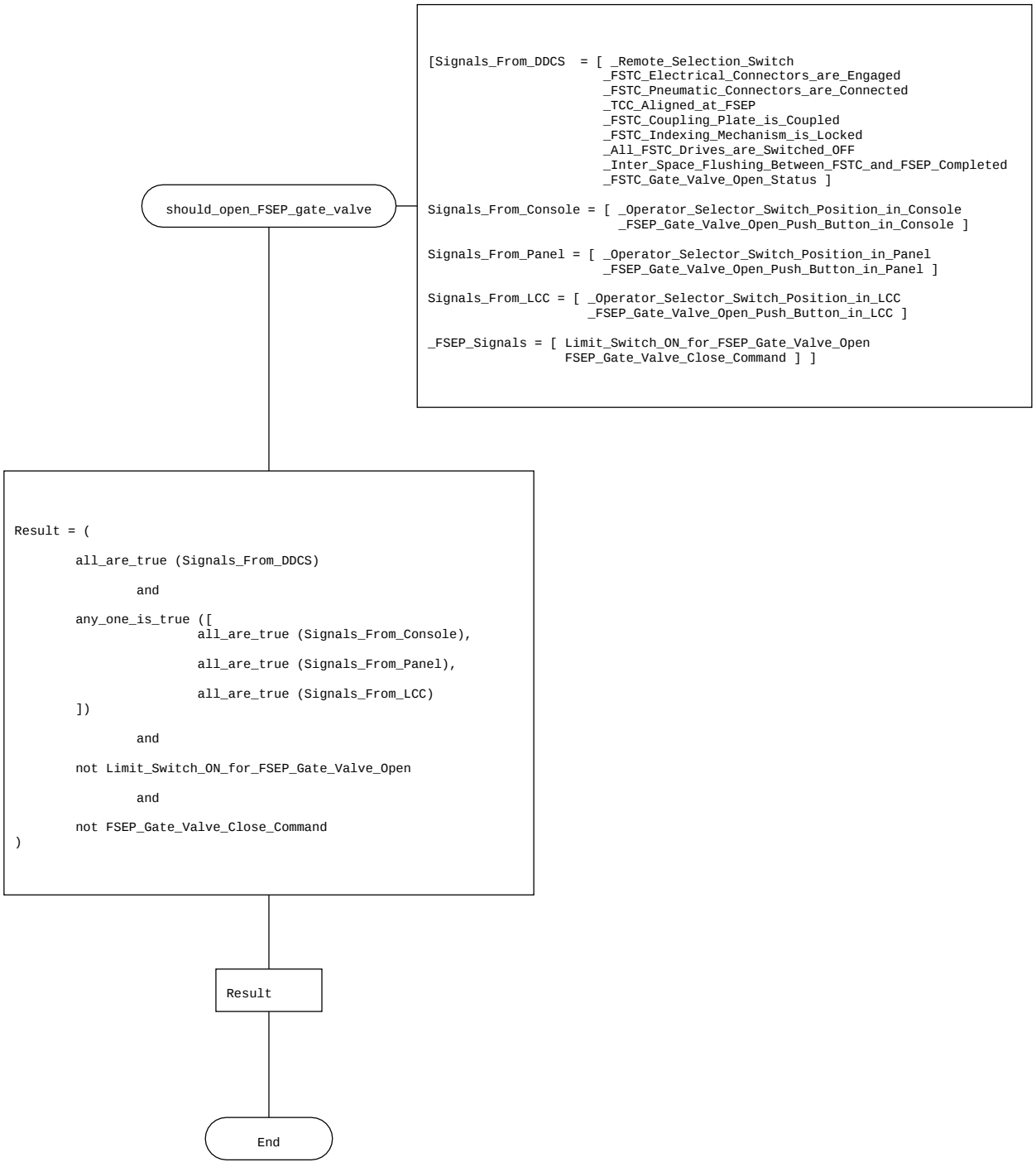


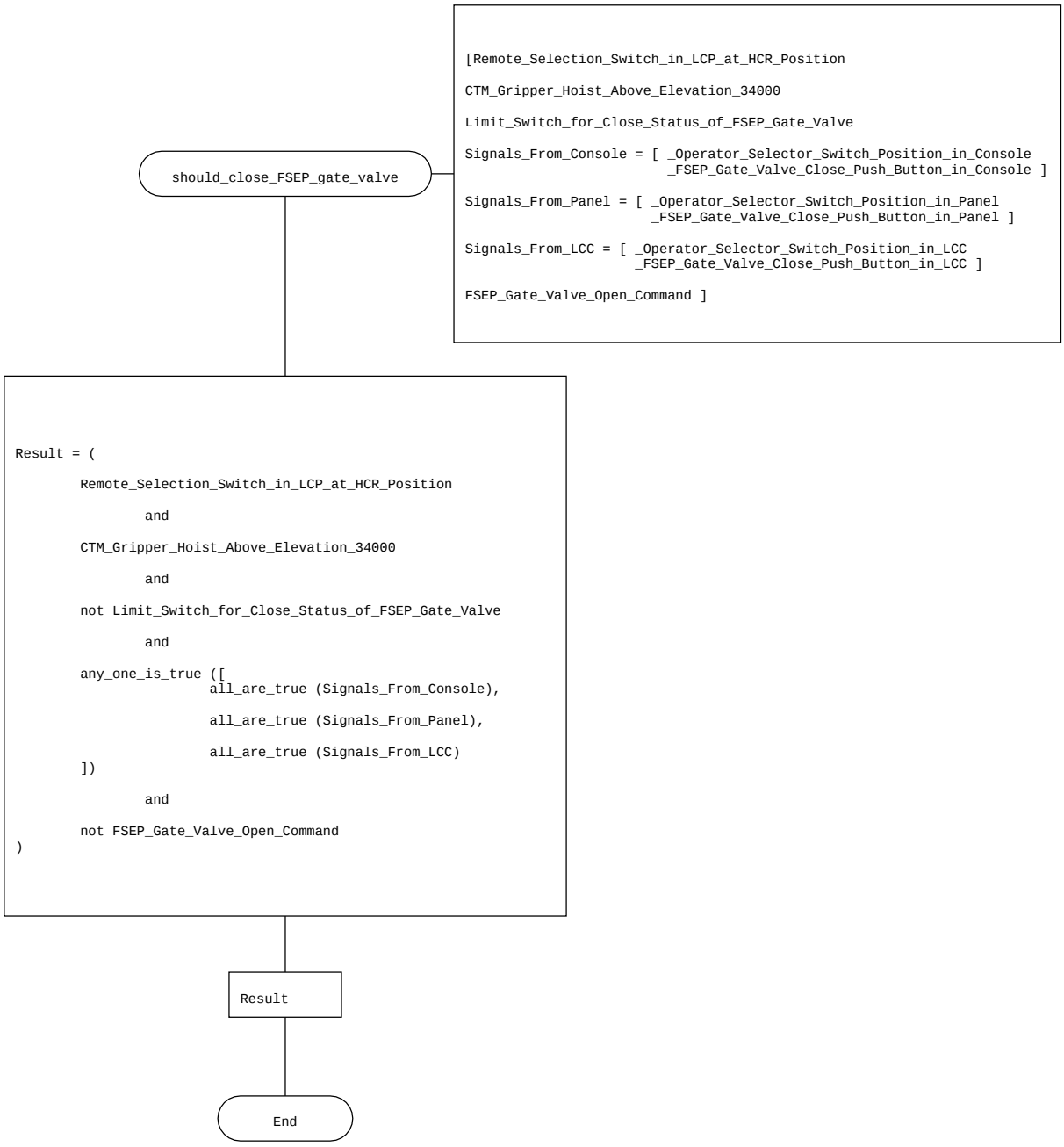
Figure A.6: *Map and filter using list comprehension: (a) Map and (b) Filter*

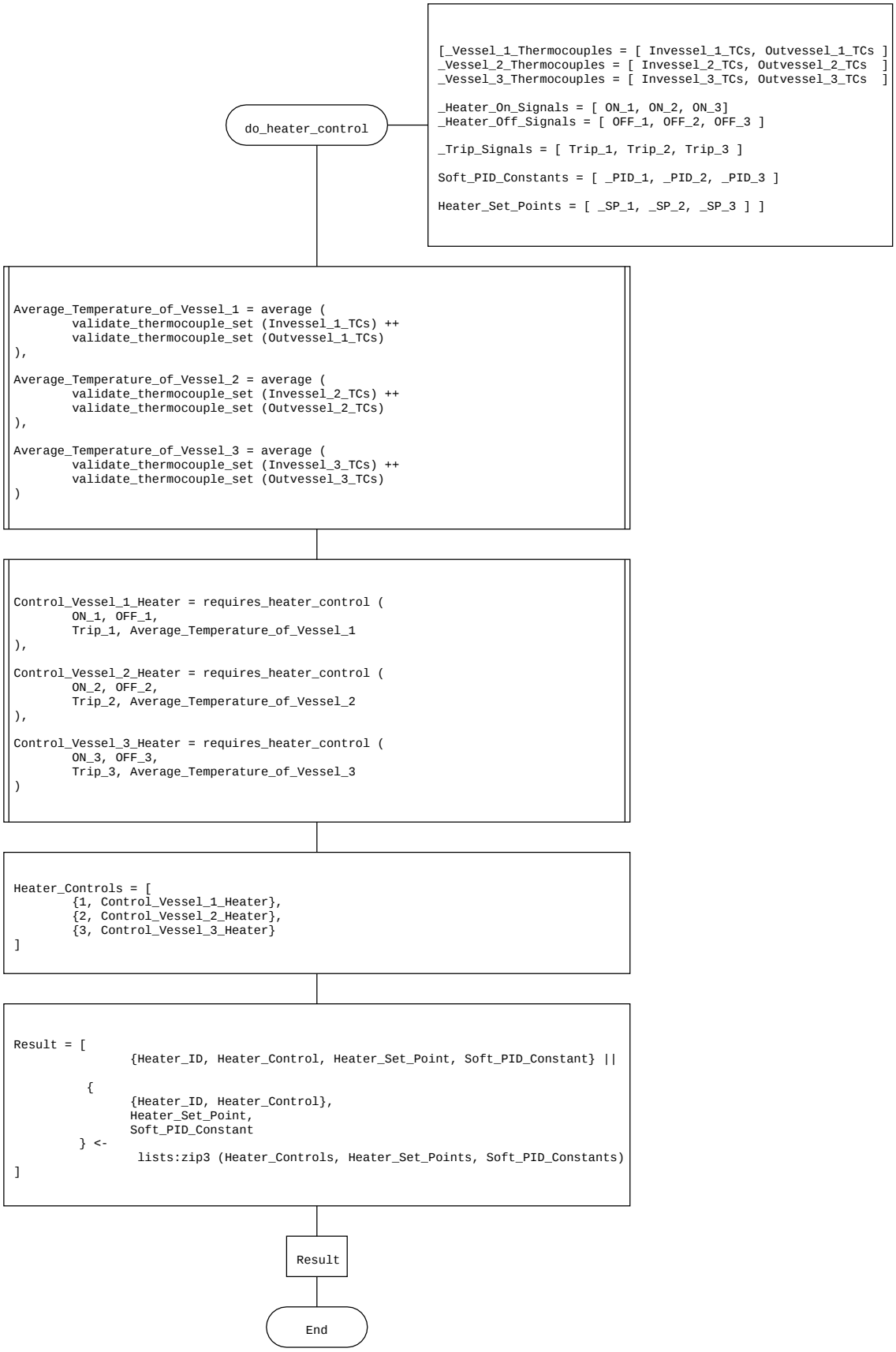
A.2 An example of semi-formal specification

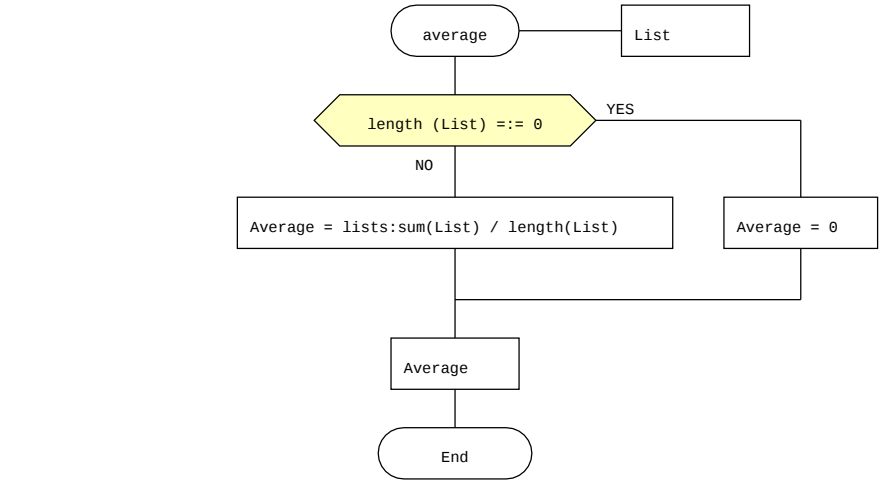
The graphical semi-formal specification in *Drakon* is converted to the *Erlang* [**185**] programming language, which acts as an executable specification. Below is an example of FSHS specification:

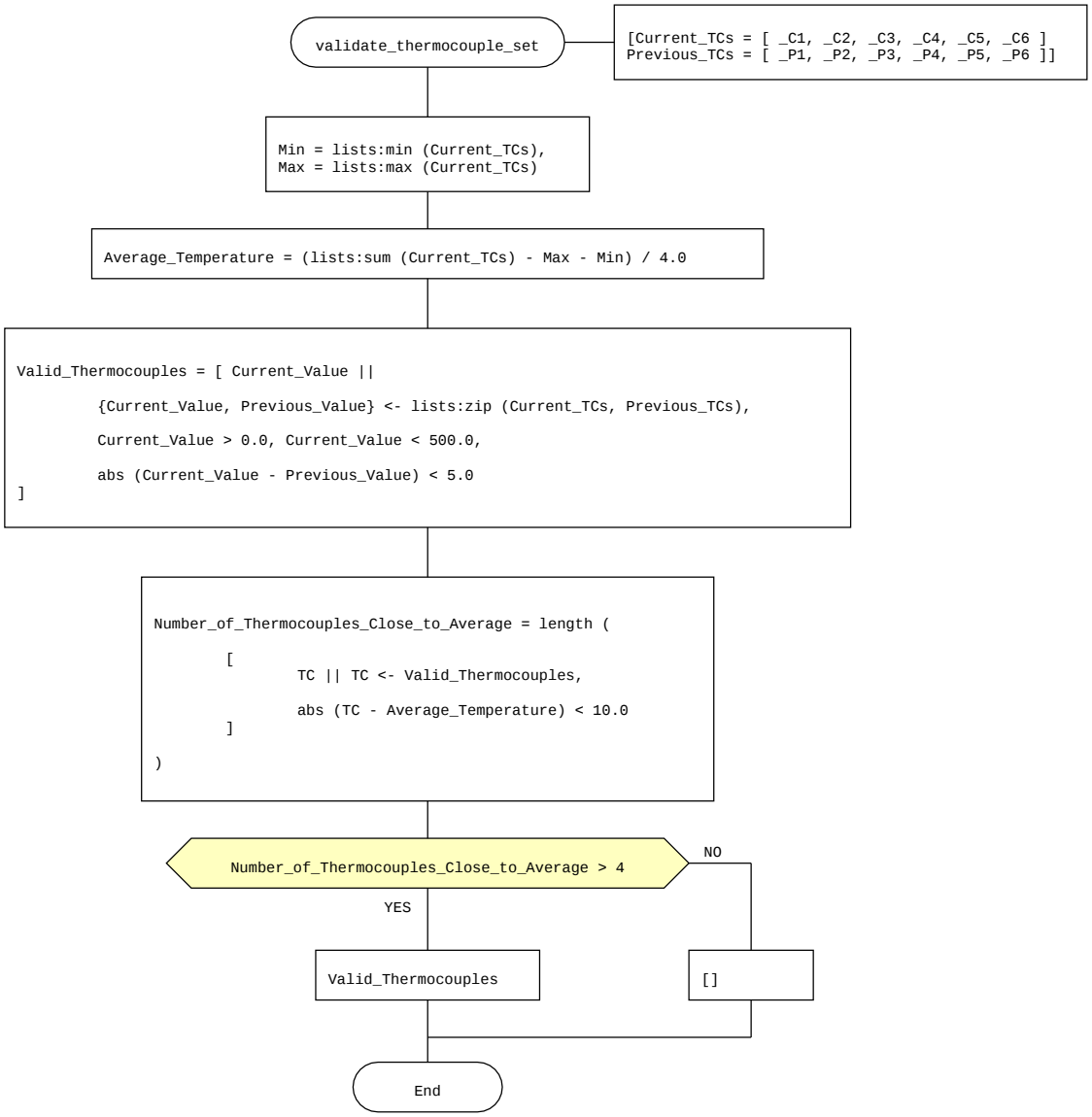


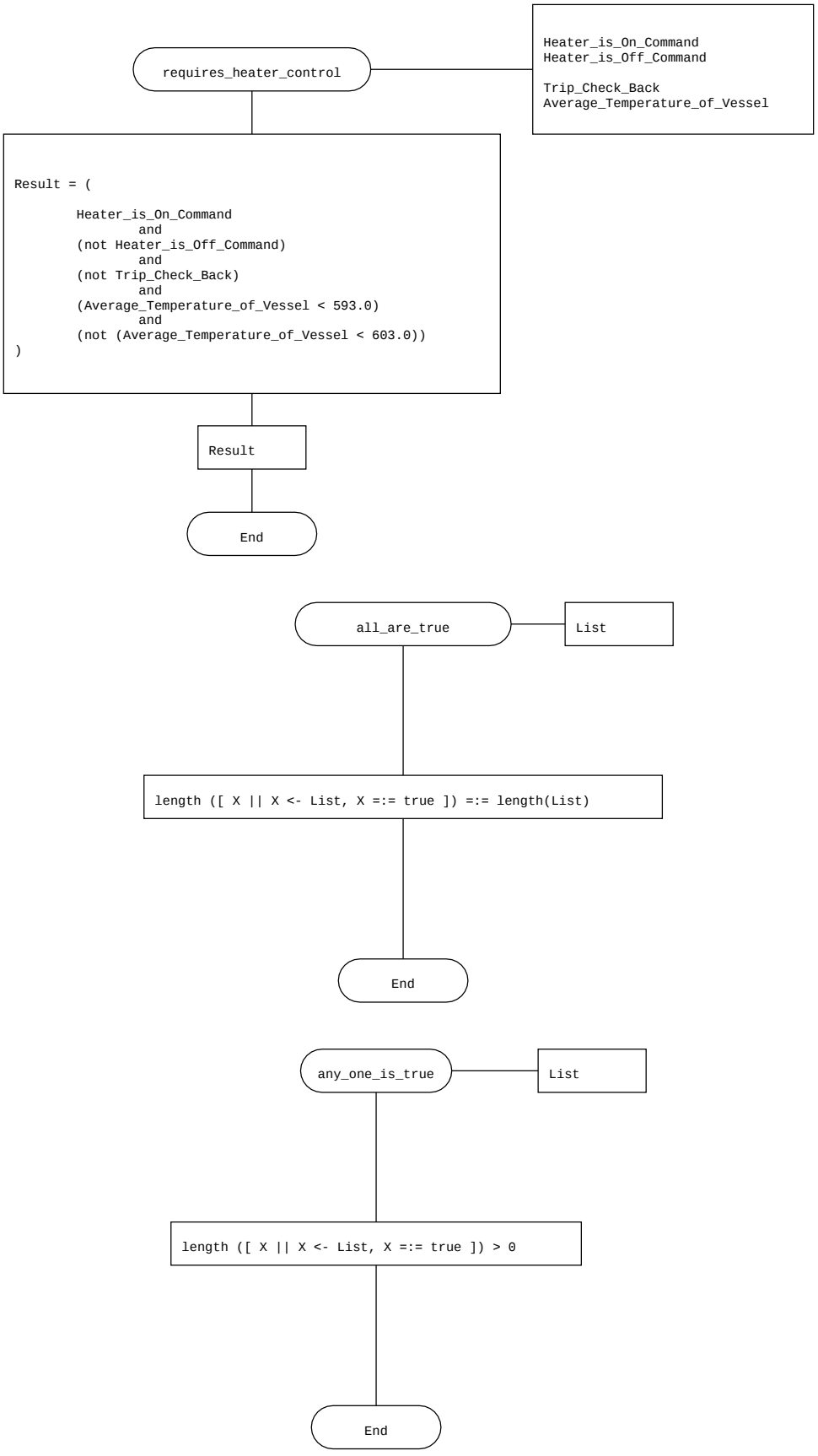

[Signals_From_DDCS = [_Remote_Selection_Switch
_FSTC_Electrical_Connectors_are_Engaged
_FSTC_Pneumatic_Connectors_are_Connected
_TCC_Aligned_at_FSEP
_FSTC_Coupling_Plate_is_Coupled
_FSTC_Indexing_Mechanism_is_Locked
_All_FSTC_Drives_are_Switched_OFF
_Inter_Space_Flushing_Between_FSTC_and_FSEP_Completed
_FSTC_Gate_Valve_Open_Status]
Signals_From_Console = [_Operator_Selector_Switch_Position_in_Console
_FSEP_Gate_Valve_Open_Push_Button_in_Console]
Signals_From_Panel = [_Operator_Selector_Switch_Position_in_Panel
_FSEP_Gate_Valve_Open_Push_Button_in_Panel]
Signals_From_LCC = [_Operator_Selector_Switch_Position_in_LCC
_FSEP_Gate_Valve_Open_Push_Button_in_LCC]
_FSEP_Signals = [Limit_Switch_ON_for_FSEP_Gate_Valve_Open
FSEP_Gate_Valve_Close_Command]]
should_open_FSEP_gate_valve
Result = (
all_are_true (Signals_From_DDCS)
and
any_one_is_true ([
all_are_true (Signals_From_Console),
all_are_true (Signals_From_Panel),
all_are_true (Signals_From_LCC)
])
and
not Limit_Switch_ON_for_FSEP_Gate_Valve_Open
and
not FSEP_Gate_Valve_Close_Command
)
Result
End


[Remote_Selection_Switch_in_LCP_at_HCR_Position
CTM_Gripper_Hoist_Above_Elevation_34000
Limit_Switch_for_Close_Status_of_FSEP_Gate_Valve
Signals_From_Console = [_Operator_Selector_Switch_Position_in_Console
_FSEP_Gate_Valve_Close_Push_Button_in_Console]
Signals_From_Panel = [_Operator_Selector_Switch_Position_in_Panel
_FSEP_Gate_Valve_Close_Push_Button_in_Panel]
Signals_From_LCC = [_Operator_Selector_Switch_Position_in_LCC
_FSEP_Gate_Valve_Close_Push_Button_in_LCC]
FSEP_Gate_Valve_Open_Command]
should_close_FSEP_gate_valve
Result = (
Remote_Selection_Switch_in_LCP_at_HCR_Position
and
CTM_Gripper_Hoist_Above_Elevation_34000
and
not Limit_Switch_for_Close_Status_of_FSEP_Gate_Valve
and
any_one_is_true ([
all_are_true (Signals_From_Console),
all_are_true (Signals_From_Panel),
all_are_true (Signals_From_LCC)
])
and
not FSEP_Gate_Valve_Open_Command
)
Result
End


do_heater_control
[_Vessel_1_Thermocouples = [Invessel_1_TCs, Outvessel_1_TCs]
_Vessel_2_Thermocouples = [Invessel_2_TCs, Outvessel_2_TCs]
_Vessel_3_Thermocouples = [Invessel_3_TCs, Outvessel_3_TCs]
_Heater_On_Signals = [ON_1, ON_2, ON_3]
_Heater_Off_Signals = [OFF_1, OFF_2, OFF_3]
_Trip_Signals = [Trip_1, Trip_2, Trip_3]
Soft_PID_Constants = [_PID_1, _PID_2, _PID_3]
Heater_Set_Points = [_SP_1, _SP_2, _SP_3]]
Average_Temperature_of_Vessel_1 = average (
validate_thermocouple_set (Invessel_1_TCs) ++
validate_thermocouple_set (Outvessel_1_TCs)
),
Average_Temperature_of_Vessel_2 = average (
validate_thermocouple_set (Invessel_2_TCs) ++
validate_thermocouple_set (Outvessel_2_TCs)
),
Average_Temperature_of_Vessel_3 = average (
validate_thermocouple_set (Invessel_3_TCs) ++
validate_thermocouple_set (Outvessel_3_TCs)
)
Control_Vessel_1_Heater = requires_heater_control (
ON_1, OFF_1,
Trip_1, Average_Temperature_of_Vessel_1
),
Control_Vessel_2_Heater = requires_heater_control (
ON_2, OFF_2,
Trip_2, Average_Temperature_of_Vessel_2
),
Control_Vessel_3_Heater = requires_heater_control (
ON_3, OFF_3,
Trip_3, Average_Temperature_of_Vessel_3
)
Heater_Controls = [
{1, Control_Vessel_1_Heater},
{2, Control_Vessel_2_Heater},
{3, Control_Vessel_3_Heater}
]
Result = [
{Heater_ID, Heater_Control, Heater_Set_Point, Soft_PID_Constant} ||
{
{Heater_ID, Heater_Control},
Heater_Set_Point,
Soft_PID_Constant
} <-
lists:zip3 (Heater_Controls, Heater_Set_Points, Soft_PID_Constants)
]
Result
End


average
List
length (List) =:= 0
YES
NO
Average = lists:sum(List) / length(List)
Average = 0
Average
End


validate_thermocouple_set
[Current_TCs = [_C1, _C2, _C3, _C4, _C5, _C6]
Previous_TCs = [_P1, _P2, _P3, _P4, _P5, _P6]]
Min = lists:min (Current_TCs),
Max = lists:max (Current_TCs)
Average_Temperature = (lists:sum (Current_TCs) - Max - Min) / 4.0
Valid_Thermocouples = [Current_Value ||
{Current_Value, Previous_Value} <- lists:zip (Current_TCs, Previous_TCs),
Current_Value > 0.0, Current_Value < 500.0,
abs (Current_Value - Previous_Value) < 5.0
]
Number_of_Thermocouples_Close_to_Average = length (
[
TC || TC <- Valid_Thermocouples,
abs (TC - Average_Temperature) < 10.0
]
)
Number_of_Thermocouples_Close_to_Average > 4
NO
YES
Valid_Thermocouples
[]
End


requires_heater_control
Heater_is_On_Command
Heater_is_Off_Command
Trip_Check_Back
Average_Temperature_of_Vessel
Result = (
Heater_is_On_Command
and
(not Heater_is_Off_Command)
and
(not Trip_Check_Back)
and
(Average_Temperature_of_Vessel < 593.0)
and
(not (Average_Temperature_of_Vessel < 603.0))
)
Result
End
all_are_true
List
length ([X || X <- List, X =:= true]) =:= length(List)
End
any_one_is_true
List
length ([X || X <- List, X =:= true]) > 0
End

B

List of mutant operators

Below is the list of mutant operators used in the present study:

#	Substring	Mutated to *(separated by ,)*	Remarks
1	<	!=, >, <=, >=, == , =	Relational operators
2	>	!=, <, <=, >=, ==, =	"
3	<=	!=, <, >, >=, == , =	"
4	>=	!=, <, <=, >, == , =	"
5	==	!=, <, >, <=, >= , =	"
6	!=	==, <, >, <=, >= , =	"
7	=	= -, == , = 0 *, = ~ , = !	Assignment operator
8	=	= 0; //, = NULL; //	"
9	+	-, *, /, %	Arithmetic operators
10	-	+, *, /, %	"
11	*	+, -, /, %	"
12	/	%, *, +, -	"
13	%	/, +, -, *	"
14	+ 1	- 1, + 0, + 2, - 2	Increment / Decrement
15	- 1	+ 1, + 0, + 2, - 2	"
16	+= 1	-= 2, -= 1, += 0, += 2	"
17	-= 1	-= 2, += 0, += 1, += 2	"
18	++	−−	"
19	−−	++	"
20	++;	−−;, +=2;, -=2; ,;	"
21	++)	−−), +=2), -=2)	"
22	−−;	++;, +=2;, -=2; ,;	"
23	−−)	++), +=2), -=2)	"
24	&	|, ∧	Bit-wise operators
25	|	&, ∧	"
26	∧	&, |	"
27	~	!, ~~	"
28	!	~, !!	"
29	>>	<<	"
30	<<	>>	"
31	<< 1	<< -1, << 0, << 2	"
32	>> 1	>> -1, >> 0, >> 2	"
33	&&	&, ||, && ! , |	Logical operators
34	||	|, &&, || !, &	"
35	true / false	false / true	"
36	if (	if (!, if (~, if (true ||, if (false &&	"
37	else	else if (false)	"
38	while (	while (!, while (~, while (false &&	"

Table B.1: *List of mutant operators used in mutation testing - I*

#	Substring	Mutated to *(seperated by ,)*	Remarks
39	break; / continue;	{ ; }	Replaces with an empty block
40	return	return 0; //, return 1; //	Modifies function's return value
41	return	return -1; //, return NULL; //	"
42	return	return -1 * , return 2*, return !	"
43	0x0	0x1, 0x5, 0xA, 0xF	Mutates bits in constants
44	0x1	0x0, 0x5, 0xA, 0xF	(0x0 = 0000
45	0x5	0x0, 0x1, 0xA, 0xF	0x5 = 0101
46	0xA	0x0, 0x1, 0x5, 0xF	0xA = 1010
47	0xF	0x0, 0x1, 0x5, 0xA	0xF = 1111)
48	0x00	0x55, 0xAA, 0xFF	"
49	0x55	0x00, 0xAA, 0xFF	"
50	0xAA	0x00, 0x55, 0xFF	"
51	0xFF	0x00, 0x55, 0xAA	"
52	[	[-1 +, [1 +, [0 *	Mutates array index
53	(	(!, (~, (-1 *, (2 *, (0 *, (1 +, (-1 +	Mutates first argument of a function
54	,	,~, , !, ,0*, ,-1*, ,2* ,1 +, ,-1 +	Mutates function arguments, array and structure initializations.
55	)	*0), *-1), *2), + 1), - 1)	Mutates last argument of a function
56	?	&& false ?, || true ?	Ternitary operator
57	unsigned / signed	signed / unsigned	Truncates data
58	int	short int, char	"
59	long	int, short int, char	"
60	float / double	int	"

Table B.2: *List of mutant operators used in mutation testing - II*

C

Data for PCA of mutant characteristics

The data for Principal Component Analysis (PCA) (*Figures 5.13 to 5.15 on pages 62–64*) of static analysis, dynamic analysis, and coverage impact of mutants for the all case studies is as follows:

Using Splint			
Killed mutants		Unkilled mutants	
Number of mutants	*Warnings*	*Number of mutants*	*Warnings*
456	0	187	1
244	1	105	0
55	2	11	2
14	99	2	3
4	3	2	4
1	4	1	23
		1	7

Using Clang			
Killed mutants		Unkilled mutants	
Number of mutants	*Warnings*	*Number of mutants*	*Warnings*
766	0	309	0
8	1		

Using Cppcheck			
Killed mutants		Unkilled mutants	
Number of mutants	*Warnings*	*Number of mutants*	*Warnings*
765	0	303	0
8	1	6	1
1	2		

Table C.1: *Static analysis of FSHS mutants*

Using Valgrind			
Killed mutants		Unkilled mutants	
Number of mutants	*Warnings*	*Number of mutants*	*Warnings*
748	0	309	0
9	1		
2	26		
2	3102981		
1	27		
1	172954		
1	3199649		
1	248067		
1	5715		
1	136773		
1	10000000		
1	735		
1	36174		
1	894604		
1	505		
(remaining) 2	(avg) 25366		

Using Coverage impact			
Killed mutants		Unkilled mutants	
Number of mutants	*Coverage impact*	*Number of mutants*	*Coverage impact*
142	0	289	0
96	35910576	4	4424377
11	152572	4	778293
9	174534	4	4851727
8	1049147	1	6354
6	360	1	500356
6	33116832	1	2308355
6	190715	1	778572
6	33601689	1	345648
5	60885	1	65
5	1009	1	609319
5	9336	1	778533
5	3068		
5	37313606		
5	33689274		
(remaining) 454	(avg) 1076774260		

Table C.2: *Dynamic analysis of FSHS mutants*

Using Splint			
Killed mutants		Unkilled mutants	
Number of mutants	*Warnings*	*Number of mutants*	*Warnings*
124	1	51	1
99	0	40	0
14	2	3	2
6	99	1	3
3	3	1	5
		1	9

Using Clang			
Killed mutants		Unkilled mutants	
Number of mutants	*Warnings*	*Number of mutants*	*Warnings*
243	0	97	0
3	1		

Using Cppcheck			
Killed mutants		Unkilled mutants	
Number of mutants	*Warnings*	*Number of mutants*	*Warnings*
225	0	93	0
21	1	4	1

Table C.3: *Static analysis of RSU mutants*

Using Valgrind			
Killed mutants		Unkilled mutants	
Number of mutants	*Warnings*	*Number of mutants*	*Warnings*
239	0	96	0
1	10000000	1	9837
1	19477		
1	3743		
1	4330		
1	9837		
1	1484		
1	11857		

Using Coverage impact			
Killed mutants		Unkilled mutants	
Number of mutants	*Coverage impact*	*Number of mutants*	*Coverage impact*
38	1142851	70	0
19	0	3	483771
7	85608	3	586432
6	464309	2	897
5	79215	2	169774
5	467171	2	50053
5	20391	1	847
4	397997	1	693
3	500530	1	29894
3	111234	1	104544
3	299450	1	770
3	352992	1	61116
3	464869	1	616
3	244381	1	34874
3	436231	1	80190
(remaining) 136	(avg) 150239		

Table C.4: *Dynamic analysis of RSU mutants*

Using Splint			
Killed mutants		Unkilled mutants	
Number of mutants	*Warnings*	*Number of mutants*	*Warnings*
1969	0	800	1
1393	1	346	0
221	2	19	2
67	99	10	3
11	3	5	4
9	4	4	5
4	6	3	12
1	12	2	6
		1	13
		1	18
		1	43
		1	9
		1	8
		1	68

Using Clang			
Killed mutants		Unkilled mutants	
Number of mutants	*Warnings*	*Number of mutants*	*Warnings*
3629	0	1195	0
46	1		

Using Cppcheck			
Killed mutants		Unkilled mutants	
Number of mutants	*Warnings*	*Number of mutants*	*Warnings*
3586	0	1190	0
59	1	5	1
18	2		
12	4		

Table C.5: *Static analysis of SGTLD mutants*

Using Valgrind			
Killed mutants		Unkilled mutants	
Number of mutants	*Warnings*	*Number of mutants*	*Warnings*
3204	0	1195	0
212	1		
33	2		
12	3		
9	2665		
6	6		
6	78		
5	121582		
3	121514		
3	121598		
3	121741		
3	121591		
3	121644		
3	170		
3	121658		
(remaining) 167	(avg) 171954		

Using Coverage impact			
Killed mutants		Unkilled mutants	
Number of mutants	*Coverage impact*	*Number of mutants*	*Coverage impact*
1814	588947843	978	0
178	0	14	24
15	22296	13	18
14	1195204	8	1139348
13	2021908	8	6
13	947086	6	495690
13	704944	6	9
13	9968	5	3
13	14120	5	12
13	6727632	5	48
12	74836	4	7563297
12	1160405	4	2390799
11	4530176	4	2692978
10	2219688	3	12051464
10	238777	3	2269850
(remaining) 1521	(avg) 3197421		

Table C.6: *Dynamic analysis of SGTLD mutants*

Using Splint			
Killed mutants		Unkilled mutants	
Number of mutants	*Warnings*	*Number of mutants*	*Warnings*
2213	0	1021	1
1666	1	514	0
389	2	67	2
26	99	10	3
6	6	3	7
5	3	2	16
5	5	2	5
4	4	1	4
4	7	1	6
2	11	1	9
1	26	1	8
1	14		
1	8		

Using Clang			
Killed mutants		Unkilled mutants	
Number of mutants	*Warnings*	*Number of mutants*	*Warnings*
4255	0	1623	0
57	1		
10	2		
1	3		

Using Cppcheck			
Killed mutants		Unkilled mutants	
Number of mutants	*Warnings*	*Number of mutants*	*Warnings*
4179	0	1597	0
109	1	17	1
32	2	9	2
3	3		

Table C.7: *Static analysis of CTMS mutants*

Using Valgrind			
Killed mutants		Unkilled mutants	
Number of mutants	*Warnings*	*Number of mutants*	*Warnings*
4235	0	1623	0
55	1		
6	3		
4	5		
3	10000000		
3	311002		
3	2		
3	1010		
2	48		
1	318		
1	310790		
1	1370314		
1	4		
1	1748606		
1	9		
(remaining) 3	(avg) 102742		

Using Coverage impact			
Killed mutants		Unkilled mutants	
Number of mutants	*Coverage impact*	*Number of mutants*	*Coverage impact*
3184	231549497	1413	0
294	0	8	932997
42	1555010	6	975969
15	315736	6	640389
13	66889311	6	273838
13	414350	5	196
13	624763	5	164428
11	12050786	5	15464
11	607208	4	374311
11	33030145	4	155867
11	12658250	4	248
11	422403	4	21054
11	3644278	4	28711
10	751240	4	618613
10	29647851	4	189
(remaining) 663	(avg) 7332209		

Table C.8: *Dynamic analysis of CTMS mutants*

Using Splint			
Killed mutants		Unkilled mutants	
Number of mutants	*Warnings*	*Number of mutants*	*Warnings*
108	1	97	1
91	0	25	0
8	2	3	2
2	99		

Using Clang			
Killed mutants		Unkilled mutants	
Number of mutants	*Warnings*	*Number of mutants*	*Warnings*
209	0	125	0

Using Cppcheck			
Killed mutants		Unkilled mutants	
Number of mutants	*Warnings*	*Number of mutants*	*Warnings*
209	0	125	0

Table C.9: *Static analysis of GES mutants*

Using Valgrind			
Killed mutants		Unkilled mutants	
Number of mutants	*Warnings*	*Number of mutants*	*Warnings*
209	0	125	0

Using Coverage impact			
Killed mutants		Unkilled mutants	
Number of mutants	*Coverage impact*	*Number of mutants*	*Coverage impact*
96	0	103	0
12	24	14	24
10	12	2	36
7	800	2	72
6	464	1	44
5	216	1	88
5	32	1	80
5	240	1	96
4	64		
4	112		
4	4		
4	504		
4	584		
4	893		
4	897		
(remaining) 35	(avg) 238		

Table C.10: *Dynamic analysis of GES mutants*

Using Splint			
Killed mutants		Unkilled mutants	
Number of mutants	*Warnings*	*Number of mutants*	*Warnings*
364	0	128	1
260	1	43	0
69	2	4	4
3	4	2	5
2	99	1	2
1	3		

Using Clang			
Killed mutants		Unkilled mutants	
Number of mutants	*Warnings*	*Number of mutants*	*Warnings*
698	0	178	0
1	2		

Using Cppcheck			
Killed mutants		Unkilled mutants	
Number of mutants	*Warnings*	*Number of mutants*	*Warnings*
697	0	178	0
2	1		

Table C.11: *Static analysis of SGDHR mutants*

Using Valgrind			
Killed mutants		Unkilled mutants	
Number of mutants	*Warnings*	*Number of mutants*	*Warnings*
689	0	178	0
10	1		

Using Coverage impact			
Killed mutants		Unkilled mutants	
Number of mutants	*Coverage impact*	*Number of mutants*	*Coverage impact*
188	0	174	0
10	7828390	1	3
6	38779	1	30
6	2766	1	348
6	296552	1	237244
6	36		
6	915630		
5	131562		
5	130977		
5	130407		
5	237965		
5	533088		
5	126633		
5	87015		
5	87108		
(remaining) 431	(avg) 141857		

Table C.12: *Dynamic analysis of SGDHR mutants*

References

[1] J. C. Knight, "Safety critical systems: challenges and directions," in *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pp. 547–550, IEEE, 2002.

[2] "Analysis of launch vehicle failure trends," tech. rep., Futron Corporation, August 2006.

[3] D. P. Murray and T. L. Hardy, "Developing safety-critical software requirements for commercial reusable launch vehicles," tech. rep., Federal Aviation Administration, Washington, DC, 2009.

[4] M. Lyu, *Handbook of Software Reliability Engineering*. McGraw-Hill, 1995.

[5] N. G. Leveson, "An investigation of the Therac-25 accidents," *IEEE Computer*, vol. 26, pp. 18–41, 1993.

[6] N. G. Leveson, "The role of software in spacecraft accidents," *AIAA Journal of Spacecraft and Rockets*, vol. 41, pp. 564–575, 2004.

[7] "Patriot missile defense - software problem led to system failure at dhahran, saudi arabia," Tech. Rep. IMTEC-92-26, US. Government Accountability Office, Feb 1992. http://www.gao.gov/assets/220/215614.pdf.

[8] B. Nuseibeh, "Ariane 5: Who dunnit?," *Software, IEEE*, vol. 14, pp. 15 –16, may-june 1997.

[9] D. Halperin, T. Heydt-Benjamin, B. Ransford, S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W. Maisel, "Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses," in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pp. 129 –142, may 2008.

[10] "Mars global surveyor (MGS) spacecraft loss of contact," tech. rep., NASA, April 2007.

[11] P. G. Neumann, "Some computer-related disasters and other egregious horrors," *Aerospace and Electronic Systems Magazine, IEEE*, vol. 1, pp. 18 –19, oct. 1986.

[12] S. Rogerson, "The chinook helicopter disaster," *IMIS Journal*, vol. 12, 2002. www.ccsr.cse.dmu.ac.uk/resources/general/ethicol/Ecv12no2.pdf.

[13] G. Slabodkin, "Software glitches leave navy smart ship dead in the water." http://gcn.com/articles/1998/07/13/software-glitches-leave-navy-smart-ship-dead-in-the-water.aspx, Jul 1998. Government Computer News.

[14] R. Charette, "Software problem blamed for woman's death in minnesota." http://spectrum.ieee.org/riskfactor/computing/it/software-problem-blamed-for-womans-death-in-minnesota, June 2010. IEEE Spectrum blog.

[15] ANSI/IEEE, "Standard glossary of software engineering terminology," 1991. STD-729-1991.

[16] A. P. Mathur, *Foundations of Software Testing*. Addison-Wesley Professional, 1st ed., 2008.

[17] J. D. Musa, *Software Reliability Engineering: More Reliable Software Faster and Cheaper 2nd Edition*. AuthorHouse, 2 ed., Sept. 2004.

[18] D. Hamlet, "Keeping the "engineering" in software engineering," in *Proceedings of the 10th International Software Quality Week*, May 1997.

[19] A. Goel, "Software reliability models: Assumptions, limitations, and applicability," *Software Engineering, IEEE Transactions on*, vol. SE-11, pp. 1411 – 1423, dec. 1985.

[20] A. Wood, "Software reliability growth models: Assumptions vs. reality," *Software Reliability Engineering, International Symposium on*, vol. 0, p. 136, 1997.

[21] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 675–689, 1999.

[22] C. Kai-Yuan, H. De-Bin, B. Cheng-Gang, H. Hu, and T. Jing, "Does software reliability growth behavior follow a non-homogeneous poisson process," *Information and Software Technology*, vol. 50, no. 12, pp. 1232–1247, 2008.

[23] C. Stringfellow and A. A. Andrews, "An empirical method for selecting software reliability growth models," *Empirical Software Engineering*, vol. 7, pp. 319–343, 2002.

[24] C. A. Asad, M. I. Ullah, and M. J.-U. Rehman, "An approach for software reliability model selection," in *Proceedings of the 28th Annual International Computer Software and Applications Conference - Volume 01*, COMPSAC '04, (Washington, DC, USA), pp. 534–539, IEEE Computer Society, 2004.

[25] A. Beckhaus, L. M. Karg, and G. Hanselmann, "Applicability of software reliability growth modeling in the quality assurance phase of a large business software vendor," *Computer Software and Applications Conference, Annual International*, vol. 1, pp. 209–215, 2009.

[26] AERB, "Safety related instrumentation and control for pressurised heavy water reactor based nuclear power plants," January 2003. AERB/NPP-PHWR/SG/D-20.

[27] "Software for computer based systems important to safety in nuclear power plants," 2000. Safety standards series No. NS-G-1.1.

[28] "Computer based systems of pressurised heavy water reactors." http://www.aerb.gov.in/t/publications/codesguides/sg-d-25.pdf, January 2010. Guide no. AERB/NPP-PHWR/SG/D-25.

[29] B. G. Blair, "Nukes: A lesson from russia." http://www.cdi.org/nuclear/blair071101.html, July 2001. The Washington Post, Wednesday, July 11, 2001, Page A19.

[30] K. Poulsen, "Slammer worm crashed Ohio nuke plant network." http://www.securityfocus.com/news/6767, August 2003. Securityfocus.

[31] B. Krebs, "Cyber incident blamed for nuclear power plant shutdown," June 2008. The Washington Post, June 5, 2008.

[32] N. Falliere, L. O. Murchu, and E. Chien, "W32.stuxnet dossier," tech. rep., Symantec, February 2011. Version 1.4.

[33] M. Hecht and H. Hecht, "Digital systems software requirements guidelines," tech. rep., Nuclear Regulatory Commission, Washington, DC, June 2001. Vol.2, Failure Descriptions, Contract RES-00-037.

[34] IAEA, "Implementing digital instrumentation and control systems in the modernization of nuclear power plants," Tech. Rep. NP-T-1.4, IAEA, 2009.

[35] IEC-61508-5, "Functional safety of electrical, electronic, programmable electronic safety-related systems, part 5: Examples of methods for the determination of safety integrity levels," tech. rep., International Electrotechnical Commission, 1998.

[36] R. W. Butler and G. B. Finelli, "The infeasibility of quantifying the reliability of life-critical real-time software," *IEEE Transactions on Software Engineering*, vol. 19, pp. 3–12, 1993.

[37] B. Littlewood, "The problems of assessing software reliability ...when you really need to depend on it," in *in Proceedings of SCSS-2000*, Springer-Verlag, 2000.

[38] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, pp. 649–678, 2011.

[39] B. Littlewood, "Software reliability modelling: achievements and limitations," in *CompEuro'91. Advanced Computer Technology, Reliable Systems and Applications. 5th Annual European Computer Conference. Proceedings.*, pp. 336–344, IEEE, 1991.

[40] T.-L. Chu, M. Yue, G. Martinez-Guridi, and J. Lehner, "Review of quantitative software reliability methods," 2010. Brookhaven National Laboratory Letter report, Digital system software PRA, JCN N-6725.

[41] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," in *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, pp. 3–9, 1978.

[42] S. Brilliant, J. Knight, and N. Leveson, "Analysis of faults in an n-version software experiment," *Software Engineering, IEEE Transactions on*, vol. 16, no. 2, pp. 238–247, 1990.

[43] T.L.Chu, G. M. Guridi, M. Yue, J. Lehner, and P. Samanta, "Traditional probabilistic risk assessment methods for digital systems." www.nrc.gov/reading-rm/doc-collections/nuregs/contract/cr6962/cr6962.pdf, May 2008.

[44] Courtois, A. Geens, M. Jarvinen, and P. Suvanto, "Licensing of safety critical software for nuclear reactors common position of seven european nuclear regulators and authorised technical support organisations," 2010. Revision 2010.

[45] D. C. Stidolph and J. Whitehead, "Managerial issues for the consideration and use of formal methods," in *In Stefania Gnesi, Keijiro Araki, and Dino Mandrioli (eds.), FME 2003, International Symposium of Formal Methods Europe*, pp. 8–14, 2003.

[46] B. Meyer, "On formalism in specifications," *IEEE Softw.*, vol. 2, pp. 6–26, Jan. 1985.

[47] S. Lauesen and O. Vinter, "Preventing requirement defects: An experiment in process improvement," *Requir. Eng.*, vol. 6, no. 1, pp. 37–50, 2001.

[48] C. Schwaber, "The root of the problem: Poor requirements," tech. rep., Forrester Research. IT View Research Document.

[49] N. Mike, J. Clark, and M. A. Spurlock, "Curing the software requirements and cost estimating blues," Nov-Dec 1999. The Defense Acquisition University Program Manager Magazine.

[50] R. R. Lutz, "Analyzing software requirements errors in safety-critical, embedded systems," in *Proceedings of the IEEE International Symposium on Requirements Engineering*, pp. 126–133, 1993.

[51] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "The ASTRÉE analyzer," *Programming Languages and Systems*, pp. 140–140, 2005.

[52] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: Practice and experience," *ACM Comput. Surv*, p. 2009.

[53] J. Jacky, "Specifying a safety-critical control system in z," *IEEE Trans. Softw. Eng.*, vol. 21, pp. 99–106, Feb. 1995.

[54] J. Yoo, E. Jee, and S. S. Cha, "Formal modeling and verification of safety-critical software," *IEEE Softw.*, vol. 26, pp. 42–49, May 2009.

[55] D. Craigen, S. Gerhart, and T. Ralston, "Case study: Darlington nuclear generating station," *IEEE Softw.*, vol. 11, pp. 30–39, 28, Jan. 1994.

[56] C. B. Jones, *Systematic software development using VDM (2nd ed.)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1990.

[57] J. M. Spivey, *Understanding Z: a specification language and its formal semantics*. New York, NY, USA: Cambridge University Press, 1988.

[58] J.-R. Abrial, *The B-book: assigning programs to meanings*. New York, NY, USA: Cambridge University Press, 1996.

[59] P. G. Larsen, J. Fitzgerald, and T. Brookes, "Applying formal specification in industry," *IEEE Softw.*, vol. 13, pp. 48–56, May 1996.

[60] P. Behm, P. Benoit, A. Faivre, and J. M. Meynadier, "METEOR : A successful application of B in a large project," in *Proceedings of FM'99: World Congress on Formal Methods* (J. M. Wing, J. Woodcock, and J. Davies, eds.), no. 1709 in Lecture Notes in Computer Science (Springer-Verlag), pp. 369–387, Springer-Verlag, Sept. 1999.

[61] H. Baumeister and D. Bert, "Algebraic specification in CASL," in *Software specification Methods: An Overview Using a Case Study* (M. Frappier and H. Habrias, eds.), ch. 15, ISTE Publishing Company, April 2006.

[62] V. S. Alagar, K. Periyasamy, and K. Periyasamy, *Specification of Software Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1st ed., 1998.

[63] E. Dürr and J. van Katwijk, "VDM++: a formal specification language for object-oriented designs," in *Proceedings of the seventh international conference on Technology of object-oriented languages and systems*, TOOLS 7, (Hertfordshire, UK, UK), pp. 63–77, Prentice Hall International (UK) Ltd., 1992.

[64] *The Object-Z specification language*. Kluwer Academic Publishers, 2000.

[65] "Object constraint language specification." http://www.omg.org/spec/OCL/2.3.1, January 2012.

[66] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of JML: a behavioral interface specification language for Java," *SIGSOFT Softw. Eng. Notes*, vol. 31, pp. 1–38, May 2006.

[67] M. Barnett, Leino, and W. Schulte, *The Spec# Programming System: An Overview*, vol. 3362/2005 of *Lecture Notes in Computer Science*, ch. 3, pp. 49–69. Berlin / Heidelberg: Springer, Jan. 2005.

[68] "jmlc, a tool to compile JML annotated java files with runtime assertion checks." http://www.eecs.ucf.edu/~leavens/JML2/docs/man/jmlc.html.

[69] B. Meyer, *Object-Oriented Software Construction*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1st ed., 1988.

[70] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A prototype verification system," in *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction*, CADE-11, (London, UK, UK), pp. 748–752, Springer-Verlag, 1992.

[71] S. Owre, J. M. Rushby, N. Shankar, and D. W. J. Stringer-Calvert, "PVS: An experience report," in *Proceedings of the International Workshop on Current Trends in Applied Formal Method: Applied Formal Methods*, FM-Trends 98, (London, UK, UK), pp. 338–345, Springer-Verlag, 1999.

[72] S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton, "Experiences using lightweight formal methods for requirements modeling," *IEEE Trans. Softw. Eng.*, vol. 24, pp. 4–14, Jan. 1998.

[73] J. Crow and B. Di Vito, "Formalizing space shuttle software requirements: four case studies," *ACM Trans. Softw. Eng. Methodol.*, vol. 7, pp. 296–332, July 1998.

[74] G. L. Steele, Jr., *Common LISP: the language (2nd ed.)*. Newton, MA, USA: Digital Press, 1990.

[75] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler, "A history of haskell: being lazy with class," in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, (New York, NY, USA), pp. 12–1–12–55, ACM, 2007.

[76] H. Sondergaard and P. Sestoft, "Referential transparency, definiteness and unfoldability," *Acta Informatica*, vol. 27, pp. 505–517, 1990. 10.1007/BF00277387.

[77] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, (New York, NY, USA), pp. 207–220, ACM, 2009.

[78] M. Fowler, *Domain-specific languages*. Addison-Wesley Professional, 2010.

[79] A. Hall, "Realising the benefits of formal methods," *Formal Methods and Software Engineering*, pp. 1–4, 2005.

[80] J. Barnes, *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[81] R. Kemmerer, "Testing formal specifications to detect design errors," *Software Engineering, IEEE Transactions on*, no. 1, pp. 32–43, 1985.

[82] N. Halbwachs, F. Lagnier, and C. Ratel, "Programming and verifying real-time systems by means of the synchronous data-flow language lustre," *IEEE Trans. Softw. Eng.*, vol. 18, pp. 785–793, Sept. 1992.

[83] J. M. Wing, *Hints to Specifiers*, pp. 57–77. Academic Press, 1996.

[84] C. DeJong, M. Gibble, J. Knight, and L. Nakano, "Formal specifications: A systematic evaluation," tech. rep., Department of Computer Science, University of Virginia, Charlottesville, VA, USA, June 1997. Technical Report CS-97-09.

[85] J. C. Knight, C. L. DeJong, M. S. Gibble, and L. G. Nakano, "Why are formal methods not used more widely?," pp. 1–12, September 1997. The Fourth NASA Langley Formal Methods Workshop, NASA Conference Publication 3356.

[86] F. X. Dormoy, "Scade 6: a model based solution for safety critical software development," in *In Embedded Real-Time Systems Conference*, 2008.

[87] M. Güdemann, F. Ortmeier, and W. Reif, "Using deductive cause-consequence analysis (DCCA) with SCADE," *Computer Safety, Reliability, and Security*, pp. 465–478, 2007.

[88] P. Caspi, C. Mazuet, R. Salem, and D. Weber, "Formal design of distributed control systems with lustre," *Computer Safety, Reliability and Security*, pp. 687–687, 1999.

[89] J. Botaschanjan, L. Kof, C. Kühnel, and M. Spichkova, "Towards verified automotive software," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 1–6, ACM, 2005.

[90] A. Lawrence and M. Seisenberger, "Verification of railway interlockings in SCADE," in *AVOCS*, vol. 10, pp. 112–114, 2011.

[91] H. Wang, S. Liu, and C. Gao, "Study on model-based safety verification of automatic train protection system," in *Computational Intelligence and Industrial Applications, 2009. PACIIA 2009. Asia-Pacific Conference on*, vol. 1, pp. 467–470, Nov.

[92] G. Berry, "Synchronous design and verification of critical embedded systems using SCADE and esterel," in *Proceedings of the 12th international conference on Formal methods for industrial critical systems*, pp. 2–2, Springer-Verlag, 2007.

[93] E. M. Clarke, "The birth of model checking," *25 Years of Model Checking*, pp. 1–26, 2008.

[94] E. A. Emerson, "The beginning of model checking: A personal perspective," *25 Years of Model Checking*, pp. 27–45, 2008.

[95] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.-J. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Information and computation*, vol. 98, no. 2, pp. 142–170, 1992.

[96] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using sat procedures instead of BDDs," in *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, DAC '99, (New York, NY, USA), pp. 317–320, ACM, 1999.

[97] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in computers*, vol. 58, pp. 117–148, 2003.

[98] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Vardi, "Benefits of bounded model checking at an industrial setting," in *Computer Aided Verification*, pp. 436–453, Springer, 2001.

[99] R. Jhala and R. Majumdar, "Software model checking," *ACM Computing Surveys (CSUR)*, vol. 41, no. 4, p. 21, 2009.

[100] E. A. Strunk, M. A. Aiello, and J. C. Knight, "A survey of tools for model checking and model-based development," tech. rep., 2006. Technical Report, CS-2006-17.

[101] P. R. Gluck and G. J. Holzmann, "Using SPIN model checking for flight software verification," in *Aerospace Conference Proceedings, 2002. IEEE*, vol. 1, pp. 1–105, IEEE, 2002.

[102] D. Angeletti, E. Giunchiglia, M. Narizzano, A. Puddu, and S. Sabina, "Using bounded model checking for coverage analysis of safety-critical software in an industrial setting," *Journal of Automated Reasoning*, vol. 45, no. 4, pp. 397–414, 2010.

[103] G. Brat, K. Havelund, S. Park, and W. Visser, "Java PathFinder - second generation of a Java model checker," in *In Proceedings of the Workshop on Advances in Verification*, Citeseer, 2000.

[104] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An opensource tool for symbolic model checking," in *Computer Aided Verification*, pp. 241–268, Springer, 2002.

[105] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas, "PVS: Combining specification, proof checking, and model checking," in *Computer Aided Verification*, pp. 411–414, Springer, 1996.

[106] L. Hoffman, "Talking model-checking technology," *Communications of the ACM*, vol. 51, no. 7, pp. 110–112, 2008.

[107] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, H. Zheng, *et al.*, "Bandera: Extracting finite-state models from java source code," in *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pp. 439–448, IEEE, 2000.

[108] MISRA, "Guidelines for the use of the C language in critical systems." http://www.misra-c.com, October 2004.

[109] MISRA, "Guidelines for the use of the C++ language in critical systems." http://www.misra-cpp.com, June 2008.

[110] JSF, "Joint strike fighter air vehicle C++ coding standards for the system development and demonstration program." http://www.research.att.com/~bs/JSF-AV-rules.pdf, December 2005. Doc. No. 2RDU00001 Rev C.

[111] B. A. Hamilton, "Software security assessment tools review," tech. rep., Naval Ordnance Safety and Security Activity, 2009.

[112] "ISO/DIS 26262-8:2009. Draft International Standard Road vehicles - Functional safety - Part 8: Supporting processes," 2009.

[113] "Software consideration in airborne systems and equipment certification, rtca-requirements and technical concepts for aviation," 1992.

[114] "EN 50128: Railway applications " communications, signalling and processing systems " software for railway control and protection systems," 2000.

[115] S. Brown, "Overview of IEC 61508," *Nuclear Engineer*, vol. 42, pp. 39–44, Mar-Apr 2001.

[116] W. Cullyer and N. Storey, "Tools and techniques for the testing of safety-critical software," *Computing Control Engineering Journal*, vol. 5, pp. 239 –244, oct 1994.

[117] L. Hatton, "Safer language subsets: an overview and a case history, MISRA C," *Information and Software Technology*, vol. 46, no. 7, pp. 465 – 472, 2004.

[118] C. Kaner, "What is a good test case," *Relation*, vol. 10, no. 1.100, p. 5569, 2003. http://www.kaner.com/pdfs/GoodTest.pdf.

[119] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson, "A practical tutorial on modified condition/ decision coverage," tech. rep., NASA, 2001.

[120] M. A. Hennell, M. R. Woodward, and D. Hedley, "On program analysis," *Inf. Process. Lett.*, pp. 136–140, 1976.

[121] M. Woodward, D. Hedley, and M. Hennell, "Experience with path analysis and testing of programs," *IEEE Transactions on Software Engineering*, vol. 6, pp. 278–286, 1980.

[122] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *Software Engineering, IEEE Transactions on*, vol. SE-10, pp. 438 –444, july 1984.

[123] P. S. Loo and W. K. Tsai, "Random testing revisited," *Information and Software Technology*, vol. 30, no. 7, pp. 402–417, 1988.

[124] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 1 ed., November 2006.

[125] T. Y. Chen, H. Leung, and I. K. Mak, "Adaptive Random Testing," *Advances in Computer Science - ASIAN 2004*, pp. 320–329, 2004.

[126] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," *SIGPLAN Not.*, vol. 40, pp. 213–223, June 2005.

[127] C. Pacheco, *Directed Random Testing*. Ph.D., MIT Department of Electrical Engineering and Computer Science, Cambridge, Massachusetts, June 2009.

[128] M. Mitchell, "An introduction to genetic algorithms," *Cambridge, Massachusetts London, England, Fifth printing*, 1999.

[129] R. P. Pargas, M. J. Harrold, and R. R. Peck, "Test-data generation using genetic algorithms," *Software Testing, Verification and Reliability*, vol. 9, no. 4, pp. 263–282, 1999.

[130] M. Pei, E. Goodman, Z. Gao, and K. Zhong, "Automated software test data generation using a genetic algorithm," *Michigan State University, Tech. Rep*, 1994.

[131] J. A. Jones and M. J. Harrold, "Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage," *IEEE Trans. Softw. Eng.*, vol. 29, pp. 195–209, Mar. 2003.

[132] D. Jeffrey and N. Gupta, "Improving Fault Detection Capability by Selectively Retaining Test Cases during Test Suite Reduction," *Software Engineering, IEEE Transactions on*, vol. 33, no. 2, pp. 108–123, 2007.

[133] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing Test Cases For Regression Testing," *Software Engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[134] A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environment," in *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, (New York, NY, USA), pp. 97–106, ACM, 2002.

[135] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," *Softw: Pract. Exper.*, vol. 28, no. 4, pp. 347–369, 1998.

[136] W. E. Wong, J. R. Horgan, A. P. Mathur, and A. Pasquini, "Test set size minimization and fault detection effectiveness: a case study in a space application," pp. 522–528, Aug. 1997.

[137] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, 2010.

[138] B. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[139] B. Miller, D. Koski, C. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, *Fuzz revisited: A re-examination of the reliability of UNIX utilities and services*. University of Wisconsin-Madison, Computer Sciences Department, 1995.

[140] E. W. Dijkstra, "The humble programmer," *Commun. ACM*, vol. 15, pp. 859–866, Oct. 1972.

[141] E. J. Weyuker, "On Testing Non-Testable Programs," *The Computer Journal*, vol. 25, pp. 465–470, Nov. 1982.

[142] D. Hoffman, "A taxonomy for test oracles." http://www.softwarequalitymethods.com/Papers/OracleTax.pdf, March 1998.

[143] A. Rajan, M. W. Whalen, and M. P. Heimdahl, "The effect of program and model structure on mc/dc test adequacy coverage," in *Proceedings of the 30th international conference on Software engineering*, ICSE '08, (New York, NY, USA), pp. 161–170, ACM, 2008.

[144] M. Staats, G. Gay, M. W. Whalen, and M. P. E. Heimdahl, "On the danger of coverage directed test case generation," in *FASE*, pp. 409–424, 2012.

[145] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and Billions of Constraints: Whitebox Fuzz Testing in Production," tech. rep., Tech. rep., Microsoft Research, 2012.

[146] J. Neystadt, "Automated Penetration Testing with White-Box Fuzzing," *MSDN Library*, 2008.

[147] R. DeMillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[148] R. Hamlet, "Testing programs with the aid of a compiler," *Software Engineering, IEEE Transactions on*, no. 4, pp. 279–290, 1977.

[149] T. A. Budd and D. Angluin, "Two Notions of Correctness and Their Relation to Testing," *Acta Informatica*, vol. 18, pp. 31–45, 1982.

[150] F. Baldwin, Douglas ; Sayward, "Heuristics for determining equivalence of program mutations.," tech. rep., Georgia inst of Tech, Atlanta school of information and computer Science, 1979.

[151] J. Pan, "Using constraints to detect equivalent mutants," Master's thesis, George Mason University, 1994.

[152] D. Schuler and A. Zeller, "(Un-)Covering Equivalent Mutants," in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pp. 45 –54, april 2010.

[153] A. Offutt and J. Pan, "Detecting equivalent mutants and the feasible path problem," in *Computer Assurance, 1996. COMPASS '96, 'Systems Integrity. Software Safety. Process Security'. Proceedings of the Eleventh Annual Conference on*, pp. 224 –236, jun 1996.

[154] B. J. Gruen, D. Schuler, and A. Zeller, "The impact of equivalent mutants," in *Mutation '09: Proceedings of the 3rd International Workshop on Mutation Analysis*, pp. 192–199, April 2009.

[155] A. Wood, "Software reliability growth models," *Tandem Computers Inc., Tech. Rep*, pp. 96–1, 1996.

[156] S. Yamada, M. Ohba, and S. Osaki, "S-shaped software reliability growth models and their applications," *Reliability, IEEE Transactions on*, vol. 33, no. 4, pp. 289–292, 1984.

[157] Z. Jelinski and P. B. Moranda, "Software reliability research," *Statistical computer performance evaluation*, pp. 465–484, 1972.

[158] M. Ohba, "Software reliability analysis models," *IBM Journal of research and Development*, vol. 28, no. 4, pp. 428–443, 1984.

[159] H.-J. Shyur, "A stochastic software reliability model with imperfect-debugging and change-point," *Journal of Systems and Software*, vol. 66, no. 2, pp. 135 – 141, 2003.

[160] J. Pearl, "Bayesian networks," tech. rep., Department of Statistics Papers, Department of Statistics, UCLA, UC Los Angeles, August 2011.

[161] A. Helminen, *Reliability estimation of safety-critical software-based systems using Bayesian networks*. Radiation and Nuclear Safety Authority, 2001.

[162] J. D. Lawrence, "Conceptual software reliability prediction models for nuclear power plant safety systems," tech. rep., Lawrence Livermore National Laboratory, 2000.

[163] B. A. Gran, "Assessment of programmable systems using bayesian belief nets," *Safety Science*, vol. 40, no. 9, pp. 797 – 812, 2002.

[164] G. Dahll and B. A. Gran, "The use of bayesian belief nets in safety assessment of software based systems," *International Journal of General System*, vol. 29, no. 2, pp. 205–229, 2000.

[165] H. seop Eom, G. yong Park, S. cheol Jang, H. S. Son, and H. G. Kang, "V&V-based remaining fault estimation model for safety-critical software of a nuclear power plant," *Annals of Nuclear Energy*, vol. 51, no. 0, pp. 38 – 49, 2013.

[166] K. Goseva-Popstojanova, A. P. Mathur, and K. S. Trivedi, "Comparison of architecture-based software reliability models," in *Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on*, pp. 22–31, IEEE, 2001.

[167] S. S. Gokhale, "Architecture-based software reliability analysis: Overview and limitations," *Dependable and Secure Computing, IEEE Transactions on*, vol. 4, no. 1, pp. 32–40, 2007.

[168] Y. Zhang, *Reliability quantification of nuclear safety-related software*. PhD thesis, Massachusetts Institute of Technology, 2004.

[169] H. Pham, *Handbook of reliability engineering*. Springer London etc., 2003.

[170] N. Fuqua, "The applicability of markov analysis methods to reliability, maintainability, and safety," *Reliability Anal. Center START Sheet*, vol. 10, no. 2, p. 8, 2003.

[171] W.-L. Wang, D. Pan, and M.-H. Chen, "Architecture-based software reliability modeling," *Journal of Systems and Software*, vol. 79, no. 1, pp. 132 – 146, 2006.

[172] J. A. Whittaker, K. Rekab, and M. G. Thomason, "A markov chain model for predicting the reliability of multi-build software," *Information and Software Technology*, vol. 42, no. 12, pp. 889–894, 2000.

[173] R. C. Cheung, "A user-oriented software reliability model," *Software Engineering, IEEE Transactions on*, no. 2, pp. 118–125, 1980.

[174] K. Goševa-Popstojanova and K. S. Trivedi, "Architecture-based approach to reliability assessment of software systems," *Performance Evaluation*, vol. 45, no. 2, pp. 179–204, 2001.

[175] S. Chetal, V. Balasubramaniyan, P. Chellapandi, P. Mohanakrishnan, P. Puthiyavinayagam, C. Pillai, S. Raghupathy, T. Shanmugham, and C. S. Pillai, "The design of the prototype fast breeder reactor," *Nuclear Engineering and Design*, vol. 236, no. 7-8, pp. 852 – 860, 2006.

[176] P. Swaminathan, *Modeling of instrumentation and Control system of prototype fast Breeder reactor*. PhD thesis, Sathyabama university, December 2008.

[177] IGCAR, "System requirement specification for FSIF, NFF, FSTC, TCC, FSEP, and FSPF," tech. rep., Indira Gandhi Centre for Atomic Research, 2011. PFBR/63510/SP/1001 Rev-B.

[178] IGCAR, "System requirements specification for reactor startup authorization logic," tech. rep., Indira Gandhi Centre for Atomic Research, 2010. PFBR/66710/SP/1002/Rev.C.

[179] IGCAR, "System requirements specifications for I&C of steam generator tube leak detection circuit," tech. rep., Indira Gandhi Centre for Atomic Research, 2006. PFBR/63370/SP/1003 Rev-D.

[180] IGCAR, "System requirement specification for RTC based core temperature monitoring system," tech. rep., Indira Gandhi Centre for Atomic Research, 2009. PFBR/63110/SP/1007/R-E.

[181] IGCAR, "System requirement specifications for I&C of Radioactive Gaseous Effluent Circuit," tech. rep., Indira Gandhi Centre for Atomic Research, 2011. PFBR/63720/SP/1003/Rev-B.

[182] IGCAR, "System requirement specifications for I&C of common sodium purification circuits for safety grade decay heat removal system," tech. rep., Indira Gandhi Centre for Atomic Research, 2008. PFBR/63420/SP/1003/Rev.D.

[183] D. L. Parnas, "Software aging," in *Proceedings of the 16th international conference on Software engineering*, pp. 279–287, IEEE Computer Society Press, 1994.

[184] S. Mitkin, "Drakon : The human revolution in understanding programs." http://drakon-editor.sourceforge.net/DRAKON.pdf, October 2011.

[185] F. Cesarini and S. Thompson, *ERLANG Programming*. O'Reilly Media, Inc., 1st ed., 2009.

[186] U. Wiger, G. Ask, and K. Boortz, "World-class product certification using erlang," *ACM SIGPLAN Notices*, vol. 37, no. 12, pp. 25–34, 2002.

[187] D. Palmer, "musasim : m68k simulator with GDB server based on Musashi." http://code.google.com/p/musasim/.

[188] S. I. Sambasivan, "Real time computers for instrumentation and control of PFBR," tech. rep., Electronics & Instrumentation Division Electronics & Instrumentation Group, IGCAR. http://www.igcar.gov.in/benchmark/Engg/21-engg.pdf.

[189] G. Hills, "Safety-critical products: INTEGRITY®-178B RTOS." http://www.ghs.com/products/safety_critical/integrity-do-178b.html.

[190] "Python multiprocessing module." http://docs.python.org/library/multiprocessing.html.

[191] M. Lutz and D. Ascher, *Learning python*. O'Reilly Media, Incorporated, 2003.

[192] C. Lattner and V. Adve, "The LLVM Compiler Framework and Infrastructure Tutorial," pp. 15–16, 2005.

[193] "Cppcheck : A tool for static C/C++ static code analysis." http://sourceforge.net/apps/mediawiki/cppcheck.

[194] D. Evans, "Splint - secure programming lint," tech. rep., 2002. University of Virginia.

[195] D. Kirkland, "Bogosec: Source code security quality calculator." http://public.dhe.ibm.com/software/dw/linux/l-bogosec.pdf, 2006.

[196] D. A. Wheeler, "FlawFinder man page." http://www.dwheeeler.com/flawfinder, may 2004.

[197] "RATS: Rough auditing tool for security." http://www.securesoftware.com.

[198] R. Braakman and C. Schwarz, "Lintian - debian package checker." http://lintian.debian.org/.

[199] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, (New York, NY, USA), pp. 89–100, ACM, 2007.

[200] B. Perens, "Electric fence malloc debugger." http://perens.com/FreeSoftware/ElectricFence/.

[201] K. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson, "A practical tutorial on modified condition/decision coverage," 2001.

[202] M. Haahr, "True random number service." http://www.random.org.

[203] B. Hendrickx and B. Vis, *Energiya-Buran: the Soviet space shuttle*. Praxis, 2007.

[204] G. Goebel, *The Space Shuttle Program*. March 2011. http://vectorsite.net/tashutl.html.

[205] A. Zak, "Buran - the soviet 'space shuttle'," November 2008. http://news.bbc.co.uk/2/hi/science/nature/7738489.stm.

[206] T. Addis and J. Addis, *Drawing Programs: The Theory and Practice of Schematic Functional Programming*. Springer, 2010.

[207] S. Mitkin, "DRAKON-Erlang: Visual functional programming," 2012. http://drakon-editor.sourceforge.net/drakon-erlang/intro.html.

[208] T. Lindahl and K. Sagonas, "Detecting software defects in telecom applications through lightweight static analysis: A war story," in *Programming Languages and Systems: Proceedings of the Second Asian Symposium (APLAS'04), volume 3302 of LNCS*, pp. 91–106, Springer, 2004.

[209] T. Lindahl and K. Sagonas, "Practical type inference based on success typings," in *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, (New York, NY, USA), pp. 167–178, ACM Press, 2006.

[210] "Dialyzer manpage." http://www.erlang.org/doc/man/dialyzer.html.

[211] I. T. Jolliffe, *Principal component analysis*. Springer verlag, 2002.

[212] Y. Jia and M. Harman, "Higher order mutation testing," *Inf. Softw. Technol.*, vol. 51, pp. 1379–1393, Oct. 2009.

[213] "LDRA Testbed ®." Liverpool Data Research Associates, http://www.ldra.com/testbed.asp.

[214] T. Hoare, "The verifying compiler: A grand challenge for computing research," in *Compiler Construction*, pp. 262–272, Springer, 2003.

[215] W. von Hagen, *The Definitive Guide to GCC*. Apress, 2nd ed. ed., aug 2006.

[216] S. C. Johnson, "A tour through the portable C compiler," *Unix programmers manual*, vol. 2, 1979.

[217] A. Magnusson and P. A. Jonsson, "Portable C Compiler homepage." http://pcc.ludd.ltu.se/.

[218] M. Kalos and P. Whitlock, *Monte Carlo methods*. Wiley-Blackwell, 2008.

[219] A. Wald and J. Wolfowitz, "Tolerance limits for a normal distribution," *The Annals of Mathematical Statistics*, vol. 17, no. 2, pp. 208–215, 1946.

[220] Y.-M. Chou and R. W. Mee, "Determination of sample sizes for setting β-content tolerance limits controlling both tails of the normal distribution," *Statistics & probability letters*, vol. 2, no. 5, pp. 311–314, 1984.

[221] S. S. Wilks, "Determination of sample sizes for setting tolerance limits," *The Annals of Mathematical Statistics*, vol. 12, no. 1, pp. 91–96, 1941.

[222] P. N. Somerville, "Tables for obtaining non-parametric tolerance limits," *The Annals of Mathematical Statistics*, vol. 29, no. 2, pp. 599–601, 1958.

[223] M. N. Li, Y. K. Malaiya, and J. Denton, "Estimating the number of defects: a simple and intuitive approach," in *Proc. 7th Int'l Symposium on Software Reliability Engineering (ISSRE)*, pp. 307–315, 1998.

[224] N. Fenton, M. Neil, W. Marsh, P. Hearty, D. Marquez, P. Krause, and R. Mishra, "Predicting software defects in varying development lifecycles using bayesian nets," *Information and Software Technology*, vol. 49, no. 1, pp. 32–43, 2007.

[225] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pp. 284–292, IEEE, 2005.

[226] P. Knab, M. Pinzger, and A. Bernstein, "Predicting defect densities in source code files with decision tree learners," in *Proceedings of the 2006 international workshop on Mining software repositories*, pp. 119–125, ACM, 2006.

[227] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proceedings of the 30th international conference on Software engineering*, pp. 531–540, ACM, 2008.

[228] N. G. Leveson, "Software safety," tech. rep., SEI Joint Program Office, July 1987. SEI-CM-6-1(Preliminary).

[229] S. Kishore, A. A. Kumar, S. Chandramouli, B. Nashine, K. Rajan, P. Kalyanasundaram, and S. Chetal, "An experimental study on impingement wastage of mod 9cr 1mo steel due to sodium water reaction," *Nuclear Engineering and Design*, vol. 243, no. 0, pp. 49 – 55, 2012.

[230] B. Raj and P. Kumar, "Safety adequacy of Indian Fast Breeder Reactor." http://www.bhavini.nic.in/attachments/pressrelease/3.11.11rejoinder.pdf.

Figure citations

1. **Figure 1.1 on page 3**

   `www.developergeeks.com/article/60/software-reliability-engineering`

   Author: Brad Stewart (used with permission).

2. **Figure 1.2 on page 4**

   `http://web.cecs.pdx.edu/~hamlet/pnsqcintro.pdf`

   Author: Dick Hamlet (used with permission).

3. **Figure 3.1 on page 32**

   `http://en.wikipedia.org/wiki/File:Nuclear_fission.svg`

   (a free image in public domain).

4. **Figure 3.2 on page 33**

   `en.wikipedia.org/wiki/File:Sodium-Cooled_Fast_Reactor_Schemata.svg`

   (a free image in public domain).

5. **Figure 3.5 on page 36**

   Reference: [**229**]

   Authors: S. Kishore *et al.* (used with permission).

6. **Figure 3.8 on page 39**

   Reference: [**230**]

   Authors: Baldev Raj and Prabhat Kumar (used with permission).

7. **Figure 5.4 on page 50**

   `http://commons.wikimedia.org/wiki/File:Crossover_genes.svg`

   (a free image in public domain).