ABSTRACT

Title of Document:	APPLICATION AND REFINEMENTS OF THE REPS THEORY FOR SAFETY CRITICAL SOFTWARE
	Ying Shi, Doctor of Philosophy, 2010
Directed By:	Professor Carol Smidts Professor Department of Mechanical Engineering

With the replacement of old analog control systems with software-based digital control systems, there is an urgent need for developing a method to quantitatively and accurately assess the reliability of safety critical software systems. This research focuses on proposing a systematic software metric-based reliability prediction method. The method starts with the measurement of a metric. Measurement results are then either directly linked to software defects through inspections and peer reviews or indirectly linked to software defects through empirical software engineering models. Three types of defect characteristics can be obtained, namely, 1) the number of defects remaining, 2) the number and the exact location of the defects found, and 3) the number and the exact location of defects found in an earlier version. Three models, Musa's exponential model, the PIE model and a mixed Musa-PIE model, are then used to link each of the three categories of defect characteristics with reliability respectively. In addition, the use of the PIE model requires mapping defects identified to an Extended Finite State Machine (EFSM) model. A procedure that can assist in the construction of the EFSM model and increase its repeatability is also provided.

This metric-based software reliability prediction method is then applied to a safety-critical software used in the nuclear industry using eleven software metrics. Reliability prediction results are compared with the real reliability assessed by using operational failure data. Experiences and lessons learned from the application are discussed. Based on the results and findings, four software metrics are recommended.

This dissertation then focuses on one of the four recommended metrics, Test Coverage. A reliability prediction model based on Test Coverage is discussed in detail and this model is further refined to be able to take into consideration more realistic conditions, such as imperfect debugging and the use of multiple testing phases.

APPLICATION AND REFINEMENTS OF THE REPS THEORY FOR SAFETY CRITICAL SOFTWARE

By

Ying Shi

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2010

Advisory Committee:
Professor Carol Smidts, Advisor, Co-Chair
Professor Balakumar Balachandran, Chair
Professor Ali Mosleh
Professor Mohammad Modarres
Associate Professor Jeffrey Herrmann
Professor Marvin Zelkowitz

© Copyright by
Ying Shi
2010

Dedication

To my parents.

Acknowledgements

First and foremost I would like to express my deepest gratitude to my advisor and friend, Dr. Carol S. Smidts. Her supervision, guidance, support and commitment to the highest standards inspired and motivated me throughout my graduate study at the University of Maryland and will remain with me for a lifetime. From the bottom of my heart, I thank her.

I want to thank the other members of my dissertation committee, Dr. Balakumar Balachandran, Dr. Ali Mosleh, Dr. Mohammad Modarres, Dr. Jeffrey Herrmann and Dr. Marvin Zelkowitz, for agreeing to serve on my committee and for their comments and ideas.

Many thanks go in particular to Dr. Ming Li and Dr. Wende Kong for their invaluable help and guidance in refining the methodologies presented in this dissertation. I also want to thank my previous lab members Mr. Jun Dai, Ms. Lulu Wang, Mr. Chao Hu and Mr. Anand Ladda for the collaboration and support they have provided for my research.

I thank my parents Mr. Xinming Shi and Mrs. Li Sun, who have always believed in me. Their love, support and encouragement propelled me to complete my Ph.D. I also want to thank my daughter, Katelyn Zhao, who has been a great joy of my life.

At last and foremost, I want to thank my husband Dr. Bryan M. Zhao for his love and persistent confidence in me. This dissertation would not have been possible without his understanding, patience and support.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1 Motivation

Over the past few decades, software has been used more and more widely in our daily lives: from personal computers, home appliances, telecommunications, automobiles to medical devices, nuclear power plants, space missions and many more. Compared to pure hardware based systems, software possesses a greater capability to solve complex problems. As a result, the size and complexity of software have dramatically increased in recent years and this trend will continue.

In the safety critical domain which is the focus of this research, software-based digital control systems are now replacing the old analog control systems. The performance of such new systems thus heavily depends on software for their correct operation. Like hardware failure, software failure can also lead to severe and even fatal consequences. For example, the failure of the Patriot mission defense system in 1991, the explosion of the Ariane 5 rocket in 1996 and the anomaly experienced in the Mars Exploration Rover in 2004 are all due to software failures. Therefore, to verify whether released software meets the users' requirements, one should first know how good it is, more specifically, how reliable it is. There is a great need for developing a method to accurately and quantitatively assess the reliability of safety critical software systems.

In the nuclear industry, the Nuclear Regulatory Commission (NRC) hasn't endorsed any particular quantitative software reliability method. With the acceptance of Probabilistic Risk Assessment (PRA) as a tool for quantify system risks, there is an urgent need of a quantitative method to assess the software failure rate. National Aeronautics and Space Administration (NASA) Headquarters specified general requirements for software reliability assurance [1] after having experienced an increased number of software-related mission failures. However, there is no clear single guideline at present on how to implement detailed requirements and practices across the NASA centers and, in particular, no critical guidance for robotic missions. The aircraft industry is also seeking new methods or guidance for assessing software failure rates since current available software reliability methods recommended in its current standards "do not provide results in which confidence can be placed to the level required for this purpose [2]."

1.2 Research Objective

As addressed in section 1.1, there is an urgent need for an acceptable software reliability quantification method in the safety critical industry. Many software reliability prediction models have been proposed in the research literature [3][4]. However, none of these models have found wide acceptance in the safety critical industry due to existing limitations. For example, software reliability growth models rely heavily on the collection of testing and operational data which is limited in the safety critical domain. These models can be only applied in the late stages of development, i.e. after testing, which provides little help in decision making in the early development stages.

Software metrics have been used in software engineering and their connection to reliability has been recognized. The objective of this research is to identify software metrics which could be used for quantitative software reliability prediction throughout the software development process. More specifically, a need exists for the definition of an approach which will use software metrics to predict software reliability, for a successful application of a set of software metrics to a safety critical application, for refinements of the models linking software metrics to reliability based on the experience gained from the effort, for recommendations for the practical and direct usage of such software metrics in the safety critical domain.

1.3 Related Research

Software reliability is defined by IEEE as the probability of failure free software[1] operation for a specified period of time in a specified environment [5]. It is one of the most important aspects of software quality.

Over the past 40 years, a multitude of software reliability growth models (SRGMs) have been proposed to assess the reliability of software systems. However, these models have not been recommended for use in the safety critical domain due to the following reasons. First, model parameters are typically estimated from failure data. For safety critical software, historical failure data is rare to nonexistent since these systems are designed to be ultra-reliable and their typical failure rates are less than 10^{-7} failures per hour. Similarly, failures are also rarely observed during testing. Even after thousands of years of testing [6], one would not be able to observe a sufficient number of failures to accurately estimate the reliability of such systems. Second, failures experienced in the testing may not represent those under the actual operational environment. Reliability predicted through statistical extrapolation of such testing failure data is also questionable. Third, SRGMs can be only applied in

[1] Failure free software does not guarantee the software is 100% correct. Defects could still exist in the software but are not exposed and converted to failures.

the late stages of development, i.e. after testing. This is generally too late and is not able to provide real time feedback to requirements and design development and therefore could not assist in decision making in the early development stages.

Other models use Bayesian Belief Network (BBN) as an analytical tool to assess reliability [7] by combining diverse sources of product and process information such as the number of latent defects, effectiveness of inspections and debug testing etc. However, approaches based on BBN use either published empirical data or subjective judgment for node probability quantification which is difficult to validate in numerical terms or even through qualitative relationships.

Several researchers have proposed software reliability models specifically designed to handle safety critical software. Schneidewind [8] developed an approach for reliability prediction which integrates software-safety criteria, risk analysis, reliability prediction and stopping rules for testing. Unfortunately, this model still relies on the collection and selection of space shuttle failure data. Miller proposed formulae which incorporate random testing results, input distribution and prior assumptions about the probability of failure for the cases when testing reveals no failures [9]. But how to partition the input space and how to estimate the prior failure probability distributions are the remaining problems. A quantitative model using statistics of the extremes which can analyze rare events was proposed by Kaufman et al [10]. Statistics of the extremes is a statistical approach which allows analysis of cases where there is few or no failure data without assuming any prior distribution. Kaufman's method has not been validated on real safety critical software. Thomason [11] extended this model by incorporating the statistics of the extremes approach to a

Markov Chain model to capture the stochastic behavior of the software system. However, the method was not validated either.

Other research efforts have focused on the development and/or study of metric-based software reliability prediction models which capture the characteristics of software products through software engineering measures. As addressed in [12], software metrics are essential not only to good software engineering practice, but also for the thorough understanding of software failure behavior and reliability prediction. Software characteristics, such as size and complexity can be used to predict the number or location of defects which themselves can be used to predict reliability [13]. Zhang and Pham [14] identified and ranked 32 factors which affect software reliability based on results of a survey of 13 participating companies. Lawrence Livermore National Laboratory [15] documents 78 software engineering measures[2]. IEEE [16] provides 39 measures which could be used to predict reliability of mission critical software systems. Forty software engineering measures from either the LLNL report or IEEE 982.1 were systematically ranked with respect to their ability to predict software reliability using expert opinion elicitation [17][18].

1.4 Approach

This research focuses on proposing a software metric-based reliability prediction method. The method starts with the measurement of a software metric. The measurement results are then linked to three different types of defect characteristics: 1) the number of defects remaining or 2) the number and the exact location of the remaining defects or 3) the number and the exact locations of defects in an earlier

[2] In this research, software engineering measures are represented by software metrics.

version. Three models, Musa's exponential model [3], the PIE [19] model and a mixed Musa-PIE model, are used to link the three categories of defect characteristics with reliability using the operational profile. Applying the PIE model requires mapping defects identified through inspection and/or testing to an Extended Finite State Machine (EFSM) model. A procedure is defined to construct the EFSM model in a repeatable fashion.

This software reliability prediction method is then applied to a safety-critical software used in the nuclear industry using eleven software metrics. Reliability prediction results are compared with the real reliability assessed using operational failure data. Experiences and lessons learned from the application are discussed.

Possible extensions to the existing models as well as procedures for repeatable measurement and prediction are proposed. The test coverage measure, which provides credible prediction results, is discussed in details and is refined to be able to take into consideration more realistic conditions, such as imperfect debugging and the use of multiple testing phases.

1.5 Contents

In Chapter 2, a systematic software metric-based reliability prediction method is presented. By using this method, measurement results are connected to defect information that is then combined with the operational profile for reliability prediction. An approach is presented for construction of the EFSM model which is a simple, convenient and effective method to model the software failure mechanism. The approach allows the mapping of the defects and the operational profile to the constructed EFSM model so that the execution of the updated EFSM model (UEFSM) can be used to abstractly represent the faulty execution of the real software.

An application of the proposed method based on eleven software metrics to a safety critical system is presented. Results show that the metric-based prediction method can be applied to safety-critical software for reliability assessment, and the prediction results from four metrics are close to the reality. Experiences and lessons learnt from the application are also discussed.

In Chapter 3, the author introduces a first refinement of the reliability prediction method based on the Test Coverage metric which is among the top metrics providing best reliability prediction. This refinement assesses the impact of newly introduced defects during the debugging process on reliability. The newly introduced defects could be located non-uniformly around the fault being fixed and they may possibly display different propagation characteristics than the faults being fixed. The refinement combines a fault taxonomy, code mutation and Bayesian statistics.

In Chapter 4, a second refinement of the reliability prediction method based on the Test Coverage metric is described. This refinement allows the description of software systems developed through multiple phases of functional testing. Multi-Phase functional testing is a common practice that is used in ultra-reliable software development to ensure that no known faults reside in the software to be delivered. This refinement is further extended to take advantage of auxiliary observations collected during the multi-phase testing and of the consequent process of analysis to refine the predictions made. This refinement also describes software systems where either the initial fault distribution is non-uniform with respect to location, or the repair/test and detection process favor certain locations.

In Chapter 5, conclusions and directions for future research are provided.

1.6 Summary of Contributions

The contributions of the dissertation can be summarized as follows:

1) The author identified the key components in the RePS theory, i.e. the M-D model, the D-R model and the Operational Profile. With the successful identification of these key components, software metrics are connected to software reliability through the RePS theory.
2) The author participated in the application of the eleven metric-based prediction methods to a safety critical system. All the measurements rules, reliability prediction results and lessons learned are documented in a NUREG report [20]. The author conducted measurements for two metrics, assessed reliability based on the measurements results of five metrics. The author also summarized the application results, summarized the application process for each metric-based RePS and identified the lessons learned.
3) The author proposed a method for operational profile quantification for safety critical software and successfully applied this method to the software system under study.
4) The author created a procedure for EFSM construction through which reliability prediction can be completed in a repeatable fashion.
5) The author developed an approach for assessing the impact of imperfect repair on reliability by predicting repair fault propagation rates. These rates can be

calculated as soon as primary faults are uncovered. The method also provides possible locations and types for newly introduced faults from imperfect repairs.

6) The author developed a method (based on test coverage) for defect and reliability prediction during late phases of the development life cycle which considers the fact that safety critical software typically undergoes a multi-phase test process.

7) Associated publications are:

   a. Application results of the RePS theory on a safety critical software were presented at the 5th American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Controls, and Human Machine Interface Technology (NPIC&HMIT) in 2006 (coauthored with Carol Smidts).

   b. "*Data Collection and Analysis for the Reliability Prediction and Estimation of a Safety Critical System*" was published in the proceedings of the Reliability Analysis of System Failure Data 2007 Workshop held by Microsoft Research (coauthored with Wende Kong and Carol Smidts) [21].

   c. "*Early Software Reliability Prediction Using Cause-Effect Graphing Analysis*" was published in the Annual Reliability and Maintainability Symposium (RAMS) in 2007 (coauthored with Wende Kong and Carol Smidts) [22] .

d. "*Lesson Learnt from the Application of Test Coverage RePS*", in the 6th American Nuclear Society International Topical Meeting on NPIC&HMIT in 2009 (coauthored with Man Cheol Kim and Carol Smidts) [23].

e. "*On the Use of Extended Finite State Machine Models for Software Fault Propagation and Software Reliability Estimation*" was published in the 6th American Nuclear Society International Topical Meeting on NPIC&HMIT in 2009 (coauthored with Ming Li and Carol Smidts) [24].

f. *"A Reliability Prediction Method for Safety Critical Systems Based on Test Coverage*" which describes the new Test Coverage RePS method was published at the 3rd International Conference on Reliability and Safety Engineering (INCRESE) in 2007 (coauthored with Wende Kong, Jun Dai and Carol Smidts) [25].

g. "*Predicting the Types and Locations of Faults Introduced During an Imperfect Repair Process and their Impact on Reliability*" was published in the International Journal of Systems Assurance Engineering and Management (IJSAEM) in March 2010, Springer Verlag (coauthored with Carol Smidts) [26].

h. "*A Test Coverage-Based Model for Predicting Software Fault Content and Location*" was published in Advanced Technologies for Software Reliability and Safety (ATSRS) in 2009 (coauthored with Carol Smidts) [27].

i. "*Predicting Residual Software Fault Content and their Location during Multi-Phase Functional Testing Using Test Coverage*" was submitted to the International Journal of Reliability and Safety (coauthored with Carol Smidts).

Chapter 2: Metric-based Software Reliability Prediction Approach and its Application

2.1 *Metric-based Software Reliability Prediction Approach*

Software fails due to defects introduced during the development process. As discussed in [17], software reliability is essentially determined by product characteristics and operational environment. The reliability of a software system is therefore determined by the defects residing in the software and the ways in which the software is operated. That is,

$$R_{SW} = f\{D, OP\} \tag{2.1}$$

Where:

R_{SW} is the reliability of the software,

D stands for the defects which are residing in the software and

OP is the operational profile.

Figure 2.1 displays the metric-based software reliability prediction system (RePS) introduced to bridge the gap between a software metric and reliability.

The construction of a RePS as shown in Figure 2.1 starts with the "Metric", which is also the "root" of a RePS. “Support metrics” are identified to help connect the root metric to “Software defects” (i.e. software defects information) through a “Metric-Defect Model” (M-D Model) if necessary. The "Defect-Reliability Model" (D-R Model) derives software reliability predictions (i.e. Reliability) using the “Software Defects” and the “Operational Profile”. “Support metrics” may be also required in D-R model to help the connection between "Software Defects" and "Reliability".

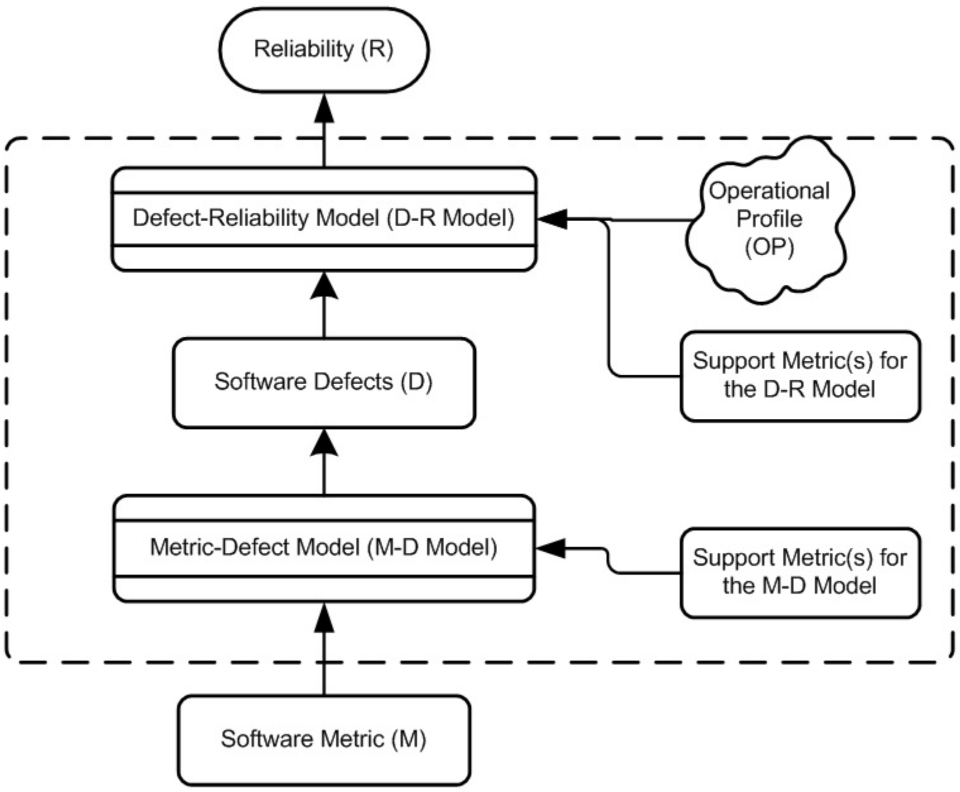

Reliability (R)
Defect-Reliability Model (D-R Model)
Operational Profile (OP)
Software Defects (D)
Support Metric(s) for the D-R Model
Metric-Defect Model (M-D Model)
Support Metric(s) for the M-D Model
Software Metric (M)

Figure 2.1 Metric-Based RePS

Software metrics may be direct measurements of defects characteristics (such as a number of defects obtained through software quality assurance activities, e.g. formal inspection, peer review etc). Software metrics can also be indirectly connected to the number of defects or other defect characteristics through empirical models. For example, Gaffney [28] established that the number of defects remaining in the software could be expressed empirically as a function of the number of line of codes (where "lines of code" is an example root measure in Figure 2.1).

Three categories of defect information can be derived from the M-D model:

1) The number of defects estimated only;
2) The number of defects found and the exact location of the defects;

3) The estimated number of defects in the current version and the exact location of the defects found in an earlier version of the software.

How each of the three categories of defects information is to be incorporated in an appropriate software reliability model for reliability quantification and how the OP should be constructed will be discussed in sections 2.2 through 2.6. Section 2.7 presents the results obtained from the application of the metric-based RePS method to a safety critical software used in the nuclear industry. Section 2.8 introduces the Test Coverage – based RePS which will be the object of further refinements in Chapters 3-4.

2.2 *D-R Model I: Reliability Prediction Model using only the Number of Defects*

When one only knows the number of defects, Musa's exponential model for reliability prediction can be applied. Musa [3] proposed the concept of fault exposure ratio *K* and its relation to *λ,* the failure rate and *N*, the number of defects remaining. That is

$$\lambda = \frac{K}{T_L} N \tag{2.2}$$

Thus, the probability of success at time *t* is obtained using:

$$R(t) = e^{-\lambda t} = e^{-K \times N \times t / T_L} \tag{2.3}$$

where

K	Fault exposure ratio, in failure/defect.
N	Number of defects estimated using specific software metrics (the root metric and the support metrics of a RePS) and the M-D model for this RePS (whose outcome is *N*).

T_L	Linear execution time of a system[3], in second
t	Execution time, in second.

Since a priori knowledge of the defects' location and their impact on failure probability is not known, the average K value given in [3], which is 4.2E-7 failure/defect, will be used.

It should be noted that if only the number of defects is known, the use of Musa's model is a natural choice since what can be obtained from the measurement is the number of defects at a specific time only. No detailed defect characteristics, such as the location of the defects, how and when the defects are revealed, are available. Musa's model provides a simple connection between the number of defects and reliability with the support of the empirical fault exposure ratio.

A more advanced model which could account for the impact of the defects' location on reliability is presented as follows.

2.3 *D-R Model II: Reliability Prediction Model using the Exact Locations of Defects*

When one knows the location of defects, the failure mechanism can be modeled explicitly using the Propagation, Infection and Execution (PIE) theory [19]. According to the PIE theory, a defect will lead to a failure if it meets the following constraints: first, it needs to be triggered (executed), then the execution of this defect should modify the state of the execution, and finally the abnormal state-change should propagate to the output of the software and manifest itself as an abnormal output, in other words, as a failure. The acronym PIE corresponds to the above three

[3] T_L is usually estimated as the ratio of the execution time and the software size on a single microprocessor basis.

program characteristics: the probability that a particular section of a program (termed "location") is executed (execution and noted as E), the probability that the execution of such section affects the data state (infection and noted I), the probability that such an infection of the data state has an effect on program output (propagation and noted P). Therefore the failure probability of the software (P_f) given that a specific location contains a defect is:

$$P_f = P \times I \times E \tag{2.4}$$

Where

P, I and E are evaluated for this particular defect and its location.

Reliability can be estimated using the PIE model. Indeed:

$$R_{SW} = 1 - P_f = 1 - \oint_{1}^{t/\tau} \oint_{i}^{N} P(i) \times I(i) \times E(i) \tag{2.5}$$

where

$P(i)$, $I(i)$, and $E(i)$ are the values of P, I and E for the i^{th} defect respectively.

τ is the average execution time per demand;

t is the execution time

t/τ is the average number of software executions;

$\oint_{1}^{t/\tau}$ is defined to account for the accumulated failure probability due to the t/τ iterations (average number of software executions).

In this thesis, we propose a simple, convenient, and effective method to solve equation (2.5) using an extended finite state machine model (EFSM) [29]. EFSMs describe a system's dynamic behavior using hierarchically arranged states and transitions. A state describes a condition of the system; and the transition can graphically describe the system's new state as the result of a triggering event. Detailed EFSM construction procedures will be presented in section 2.6.

It should be noted that by using equation (2.5), we assume that all the defects are independently located in different I/O (input/output) paths. If two or more defects are located in the same I/O path, using equation (2.5) is not accurate since it overestimates the probability of failure and therefore underestimates reliability. This limitation has been overcome by using EFSM since EFSM is developed to reflect the actual execution of the software and handles dependent defects. For example, the failure probability will not be double counted if two defects are located in the same I/O path. The integral notation used in equation (2.5) is defined to be able to eliminate the dependency issues automatically.

2.4 D-R Model III: Reliability Prediction Model using an Estimate of the Number of Defects in the Current Version and the Exact Locations of the Defects Found in an Earlier Version (mixed Musa-PIE model)

When the defect information available falls in the third category, using Model I alone overlooks the available defect content information found in previous versions. In this case, both Model I and Model II need to be used. More specifically, since the defect location in previous versions of the software is known, the PIE model can be used first to obtain a software-specific fault exposure ratio (*υK*) through the propagation of these known defects:

$$\nu K(t) = \frac{K}{T_L} t \tag{2.6}$$

υK(t) is an average value, and can be estimated analytically from the *N'* known defects in an early version of the software using the PIE theory and the inverse of Musa's model. That is:

$$\nu K(t) = -\frac{1}{N'} Ln\left[1 - \oint_{1}^{t/\tau}\left(\oint_{i}^{N'} P(i) \times I(i) \times E(i)\right)\right] \qquad (2.7)$$

This new calculated *υK* will be much more accurate than the average *K* used in Model I. Once the new fault exposure ratio is obtained, Model I is then used for reliability prediction knowing the number of defects remaining in the software. We thus name this model as the Combinational Model (Model III). The probability of failure simply becomes a function of the number of defects:

$$P_f = 1 - e^{-\nu k(t) \times N} \qquad (2.8)$$

2.5 Operational Profile

The operational profile (OP) is a quantitative characterization of the way in which a system will be used [30]. It associates a set of probabilities to the program input space and therefore describes the behavior of the system. The OP is traditionally evaluated by enumerating field inputs and evaluating their occurrence frequencies. Expert opinion can also be used to estimate the hardware components-related operational profile if field data is in limited availability. Musa's [30] recommended approach for identifying the environmental variables (i.e. those variables that might necessitate the program to respond in different ways) is to have several experienced system design engineers brainstorm a possible list. Sandfoss [31] suggests that estimation of occurrence probabilities could be based on numbers obtained from project documentation, engineering judgment, and system development experience.

2.6 Extended Finite State Machine (EFSM)

As specified in section 2.3, for D-R model II, the PIE concept [19] was introduced to describe the software failure mechanism if one knows the location of the defects. D-R model III, introduced in section 2.4, also requires the use of the PIE concept to propagate the known defects in an early version of the software. How to implement the PIE concept for reliability quantification is discussed in this section.

In the original assessment method, P, I and E are quantified statistically using mutation [19]. This method, however, is neither able to combine the operational profile, nor able to consider defects that do not appear in the source code, such as requirements or design errors (e.g. "missing functions"). Moreover, the large amount of mutants required hampers the practical implementation of the method to complex systems.

In this section, a simple, convenient and effective method to solve this problem using an Extended Finite State Machine (EFSM) model [29] is proposed. An EFSM describes a system's dynamic behavior using hierarchically arranged states and transitions. A state describes a condition of the system; and the transition visually describes the system's new state as a result of a triggering event. The operational profile of the software system is mapped into the model to analytically represent the probabilities of the system traversing each execution state. More specifically, an EFSM is a septuple (Σ, Γ, S, T, P, V, OP), where:

- Σ is the set of input variables of the software. These variables are crossing the boundary of the application.
- Γ is the set of output variables of the software. These variables are crossing the boundary of the application.
- S is a finite non empty set of states. A state usually corresponds to the real-world condition of the system.
- T is the set of transitions. An event causes a change of state and this change of state is represented by a transition from one state to another.
- P is the set of predicates, the truth value of the predicates is attached to the relevant transition,
- V is the set of variables defined and used within the boundary of the application, and
- OP is the set of probabilities of the input variables.

The method proposed for assessing software reliability based on EFSM proceeds in five stages:

1) Construct a high level EFSM based on the Software Requirement Specifications (SRS);
2) Identify, record and classify the defects;
3) Modify the high level EFSM by mapping the identified defects;
4) Map the operational profile of the software to the appropriate variables (or transitions);
5) Obtain the probability of failure by executing the modified EFSM.

As stated before, the failure probability can be assessed by calculating the product of the execution probability, the infection probability and the propagation probability. The first three steps of the proposed method are used to construct the EFSM model and identify the infected states. The execution probability can be determined through step 4) by mapping the operational profile to the EFSM. The overall failure probability can be obtained through execution of the EFSM in step 5). Generally speaking, the proposed approach is based on constructing and refining the EFSM model. Both construction and refinement steps are rule-based processes. Different rules for handling different requirement specifications and different types of defects are provided. Thus, the approach is actually a Rule-based Model Refinement Process (RMRP).

The advantages of this approach are: 1) it can avoid time and labor intensive mutation testing; 2) it can combine the operational profile which reflects the actual usage of the software system; 3) it allows assessment of the impact of requirements defects, e.g. "missing functions", on software reliability; 4) tools are available for executing the constructed EFSM model.

Each of the five steps is discussed in turn in the following subsections.

2.6.1 <u>Step 1: Construct of a High Level EFSM Based on the SRS</u>

This step is used to construct a high level EFSM (HLEFSM) based on the SRS. This step is independent of the defect identification process and corresponding results, i.e. the defects identified.

The HLEFSM can be systematically constructed by mapping each occurrence of a function specification to a transition. HLEFSM will be manually constructed based on the SRS. Figure 2.2 shows a typical prototype outline for SRS [32]. The general procedure to be followed for constructing a HLEFSM can be illustrated in Figure 2.3.

```
3.Specific Requirements
  3.1 Functional Requirements
    3.1.1 Functional Requirement 1
      3.1.1.1 Introduction
      3.1.1.2 Inputs
      3.1.1.3 Processing
      3.1.1.4 Outputs
    3.1.2 Functional Requirement 2
    ……
  3.2 External Interface Requirements
    3.2.1 User Interfaces
    3.2.2 Hardware Interfaces
    3.2.3 Software Interfaces
    3.2.4 Communications Interfaces
 3.3 Performance Requirements
 3.4 Design Constraints
    3.4.1 Standards Compliance
    3.4.2 Hardware Limitations
……
 3.5 Attributes
    3.5.1 Security
    3.5.2 Maintainability
……
  3.6 Other Requirements
    3.6.1 Data Base
    3.6.2 Operations
    3.6.3 Site Adaptation
```

Figure 2.2 Typical Prototype Outline of SRS

The general construction procedure includes:

a) Study the SRS and focus on the Functional Requirements section, here section 3.1. It should be noted that there exists several other SRS prototypes [32]. For those prototypes, one can still find a section similar to the Functional Requirements section which describes the functions of the software system.

b) Create an ENTRY state and an EXIT state for the entire application;

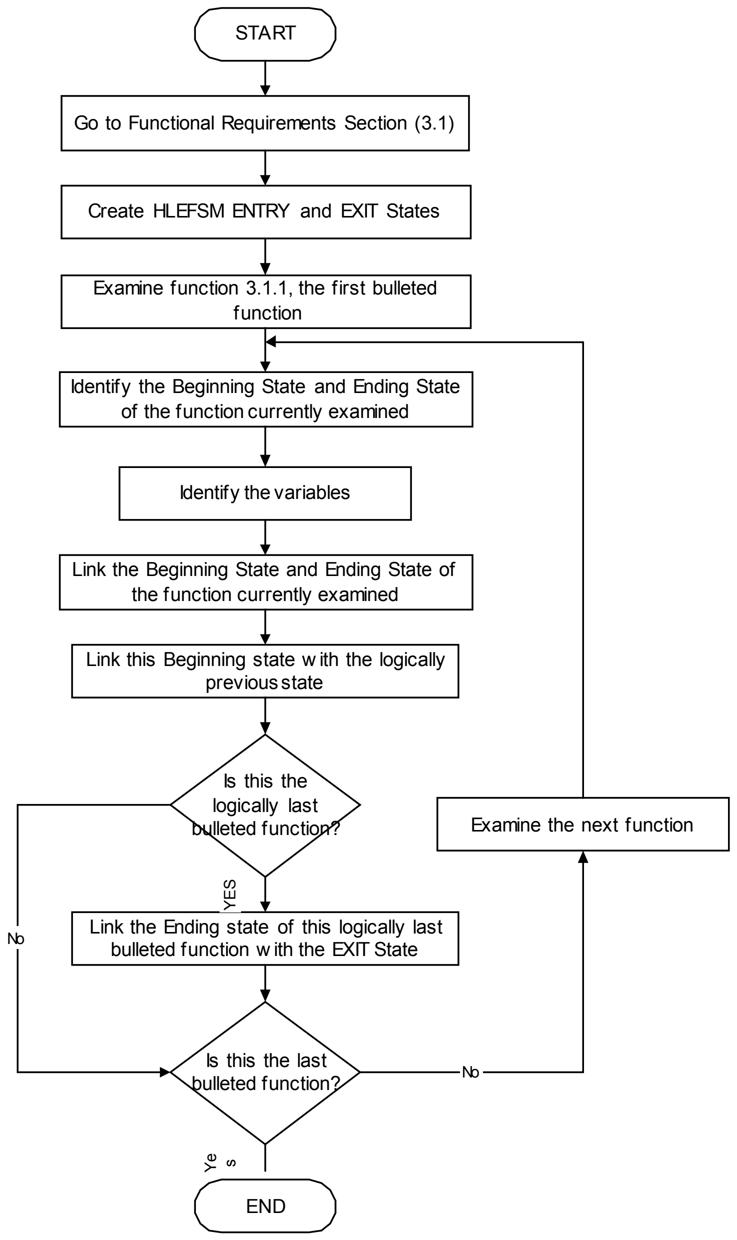

START
Go to Functional Requirements Section (3.1)
Create HLEFSM ENTRY and EXIT States
Examine function 3.1.1, the first bulleted function
Identify the Beginning State and Ending State of the function currently examined
Identify the variables
Link the Beginning State and Ending State of the function currently examined
Link this Beginning state with the logically previous state
Is this the logically last bulleted function?
YES
No
Link the Ending state of this logically last bulleted function with the EXIT State
Examine the next function
Is this the last bulleted function?
No
Yes
END

Figure 2.3 SRS-Based High Level EFSM Construction

c) Examine the first bulleted[4] function f_1 here denoted as 3.1.1;

d) Define the corresponding states of the function f_1 (normally it is the logically first function of the software system): the starting state $S_i(f_1) : S_i(f_1) \in S$ and the ending state $S_o(f_1)$: $S_o(f_1) \in S$ of the function f_1.

[4] A bulleted function is a function explicitly documented using a bullet in the SRS document for distinguishing it from other functions. It should not be a function within a paragraph which will certainly contain multiple functions.

e) Identify the following elements:

i. Specify the input variables $iv(f_1)$ of function f_1 based on section 3.1.1.2 “Input”: ‘iv’ could be part of Σ or V or a combination of Σ and V.

ii. Specify the predicates $p(f_1)$. Normally, the predicates can be found in section 3.1.1.1 “Introduction”.

iii. Specify the output variables $ov(f_1)$ of function f_1 based on section 3.1.1.4 “Output”: ‘ov’ could also be part of Γ or V or a combination of Γ and V.

iv. Specify the variables stored in $S_i(f_1)$, denoted as $V_{S_i(f_1)}$, and the variables stored in $S_o(f_1)$, denoted as $V_{S_o(f_1)}$, since a state is the condition of a finite state machine at a certain time and is represented by a set of variables and their potential values. It should be noted that not all of the variables stored in $S_i(f_1)$ will be used by function f_1, that is $V_{S_i(f_1)} \supset iv(f_1)$. The predicates also should be part of the variables stored in $S_i(f_1)$ and $V_{S_i(f_1)} \supset p(f_1)$. Those variables, denoted as $nu(f_1)$, which are neither used as the input variables nor used as the predicates of function f_1 will remain the same and be part of the variables stored in the output. Thus $V_{S_i(f_1)} = iv(f_1) \cup p(f_1) \cup nu(f_1)$ and $V_{S_o(f_1)} = ov(f_1) \cup nu(f_1)$.

f) Link the beginning state and the ending state of function f_1 by a transition, t_1: $t_1 \in T$ and t_1 is the set of the function f_1 and its associated predicates $p(f_1)$, $t_1 = \{p(f_1), f_1\}$, pointing from starting state $S_i(f_1)$ to the ending state $S_o(f_1)$.

g) For function f_1, link the starting state $S_i(f_1)$ to the ENTRY state. For function f_j, link the starting state $S_i(f_j)$ of to the ending state of the logically previous function f_{j-1}. The logical relationship between the functions should be specified in the "introduction" subsection of the description of the bulleted function. The variables stored in the starting state of function f_j, $V_{S_i(f_j)}$, should be the variables stored in the ending state of its logically previous function, $V_{S_o(f_{j-1})}$ plus some inputs from Σ. That is $V_{S_i(f_j)} = V_{S_o(f_{j-1})} \cup v_j$, where $v_j \subset \Sigma$.

h) Iterate step d) to step g) for the next function until all the bulleted functions are represented in the HLEFSM. It should be noted that the HLEFSM model should remain at a high level to minimize the construction effort. Only the bulleted functions, i.e. 3.1.1, 3.1.2 etc shown in Figure 2.2, should be represented in this HLEFSM model. There is no need at this point to further break down the bulleted functions and display their corresponding sub-functions.

i) Link the ending state of the logically last bulleted function to the EXIT state. Normally the logically last bulleted function will send out all required outputs and reset all variables to their initial values for the next round of processing.

Example 1: To better illustrate the above EFSM construction step, a paragraph excerpted from PACS (Personal Access Control System[5]) [33] SRS and its associated EFSM elements identifications are shown in Table 2.1.

Table 2.1 EFSM Construction Step 1 for Example 1

PACS SRS: Software will validate the entrant's card data (SSN and last name). If correct data, software will display "Enter PIN".	
Function 1	Function f_1: card validation function;
• Starting State of the function	$S_i(f_1)$:card is awaiting for validation;
• Ending state of the function	$S_o(f_1)$:card has been validated;
• Input variables	$iv(f_1)$ = {SSN, Last name};
• Output variables	$ov(f_1)$ = {card validation results};
• Predicates	N/A
• Variables stored in the starting state	In this case, the variables stored in $S_i(f_1)$ will all be used by function f_1. That is, $V_{S_i(f_1)} = iv(f_1)$
• Variables stored in the ending state	$V_{S_o(f_1)} = ov(f_1)$
Function 2	Function f_2: card validation results display function;
• Starting State of the function	$S_i(f_2)$:card validation results are awaiting to be displayed;
• Ending state of the function	$S_o(f_2)$:card validation results have been displayed;
• Input variables	$iv(f_2)$ = {card validation results};
• Output variables	$ov(f_2)$ = {"Enter PIN" displayed};
• Predicates	$p(f_2)$ ={card data = correct}.
• Variables stored in the starting state	$V_{S_i(f_2)} = iv(f_2) \cup p(f_2)$
• Variables stored in the ending state	$V_{S_o(f_2)} = ov(f_2)$

2.6.2 Step 2: Identify, Record and Classify the Defects

This step is used to identify defects through software inspection or testing. Software defects can be uncovered by using different inspection and testing techniques [34] [35]. All the defects identified through inspection or testing should be recorded properly for further references and examinations. Table 2.2 or similar table should be generated.

[5] PACS is a system which provides privileged physical access to rooms/buildings, etc. The user needs to swipe his card and enter a four digit PIN. The application verifies this against a database and if authorized, provides access to the room/building by opening the gate.

Table 2.2 Example Table for Recording Identified Defects

NO.	Defect Description	Defect Location	Defect Type	Variables/Functions Affected
1				
2				
……				

The possible instances or further description of each field are shown in Table 2.3. In the Defect Description column, the inspector should provide a general description of the defect using plain English sentences; in the Defect Location column, one should record where the defect originated, i.e. either in the SRS, SDD or Code. The module name or function name (associated to the location of the defect) should be provided as well. The specific defect type should be documented in the Defect Type column of the table. The exact affected variable/function should be specified in detail in the Variable/Functions Affected column of Table 2.2.

Table 2.3 Possible Instances or Further Description for Each Field in Table 2.2

Item	**Possible Instances of Each Field in** Table 2.2	
Defect Description	Plain English sentence.	

Defect Location	SRS;
	Software Design Documents (SDD);
	Code
		Function name (if the defect is in SRS);
		Module name (if the defect is in SDD or code)

Defect Type	Missing function; Extra function; Incorrect function; Ambiguous function;
	Missing input; Extra input; Input with incorrect/ambiguous value; Input with incorrect/ambiguous type; Input with incorrect/ambiguous range; Missing output; Extra output; Output with incorrect/ambiguous value; Output with incorrect/ambiguous type; Output with incorrect/ambiguous range;
	Missing predicate; Extra predicate; Incorrect/ambiguous predicate.	

Variables/Functions Affected	The exact name of the affected variables or functions given in the documents.	

Using the same PACS SRS described in step 1 as an example, the following table should be generated:

Table 2.4 Record of Identified Defects for Example 1

NO.	Defect Description	Defect Location	Defect Type	Variables/Functions Affected
1	This requirement specification does not specify the case where the data stored in the card is not correct.	PACS SRS: Card validation results display function	Missing predicate	*p(f$_2$)* ={card data = incorrect}

2.6.3 *Step 3: Modify the HLEFSM by Mapping the Identified Defects*

Once defects have been identified, they should be mapped into the HLEFSM and the infected states should be identified for later assessment of their final impacts. The defect mapping process ultimately modifies the HLEFSM. The modified EFSM obtained is therefore an octuple (Σ, Γ, S, T, P, V, OP, D) where D is the set of defects discovered through inspection.

The defect mapping procedures are shown in Figure 2.4. The following subsections will describe how to localize the defects in the HLEFSM and how to modify a HLEFSM and the low level EFSM (LLEFSM) obtained.

2.6.3.1 *Section a: Localize the defects in the HLEFSM:*

One needs to know the exact locations of the defects in order to modify the HLEFSM correctly. The localization of the defects is based on tracing among the development documents: SRS, SDD and code which have been inspected. Figure 2.5 illustrates the detailed tracing procedures.

2.6.3.2 *Section b: Modify the HLEFSM:*

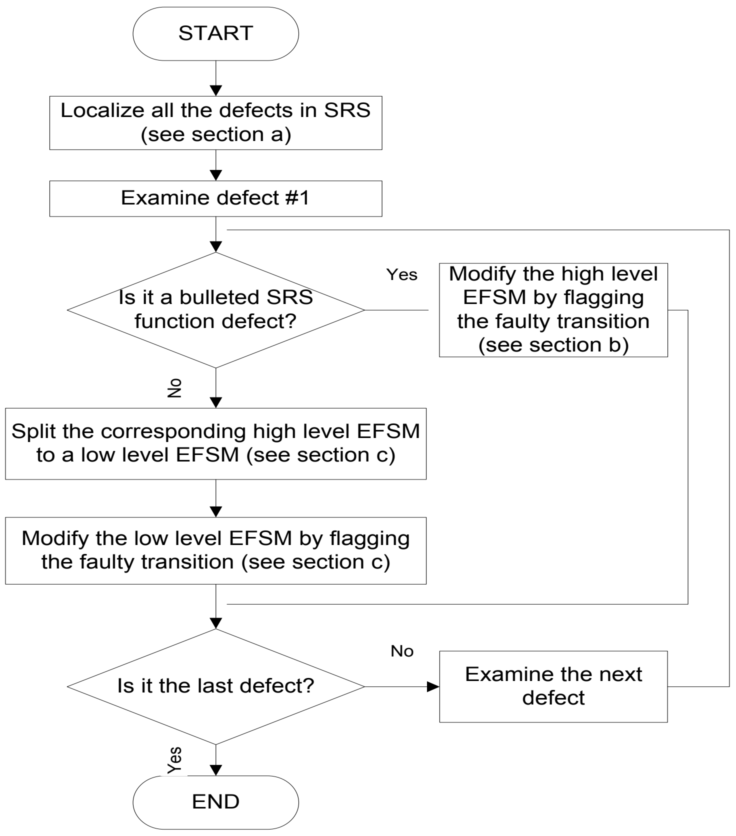

START
Localize all the defects in SRS (see section a)
Examine defect #1
Is it a bulleted SRS function defect?
Yes
Modify the high level EFSM by flagging the faulty transition (see section b)
No
Split the corresponding high level EFSM to a low level EFSM (see section c)
Modify the low level EFSM by flagging the faulty transition (see section c)
Is it the last defect?
No
Examine the next defect
Yes
END

Figure 2.4 General procedures for defect mapping

The infected state should be identified during the EFSM modification process. The process of definition and identification of the infected state is discussed next. If a defect found was directly related to a bulleted function, (i.e. the defect is a bulleted function level defect,) there is no need to split the HLEFSM. A new state or transition should be created or certain variables within the transitions should be flagged to reflect the infections. It should be mentioned that all the defects should be represented by a variable, i.e. variable *d*, and attached to the transitions. If *d* with the initial value of 0 is assigned to 1, it means there is a defect along with the transition. Thus, the attributes of the transition *t* have now changed from $t_i = \{p_i, f_i(iv)\}$ to $t_i = \{p_i, f_i(iv), d_i\}$.

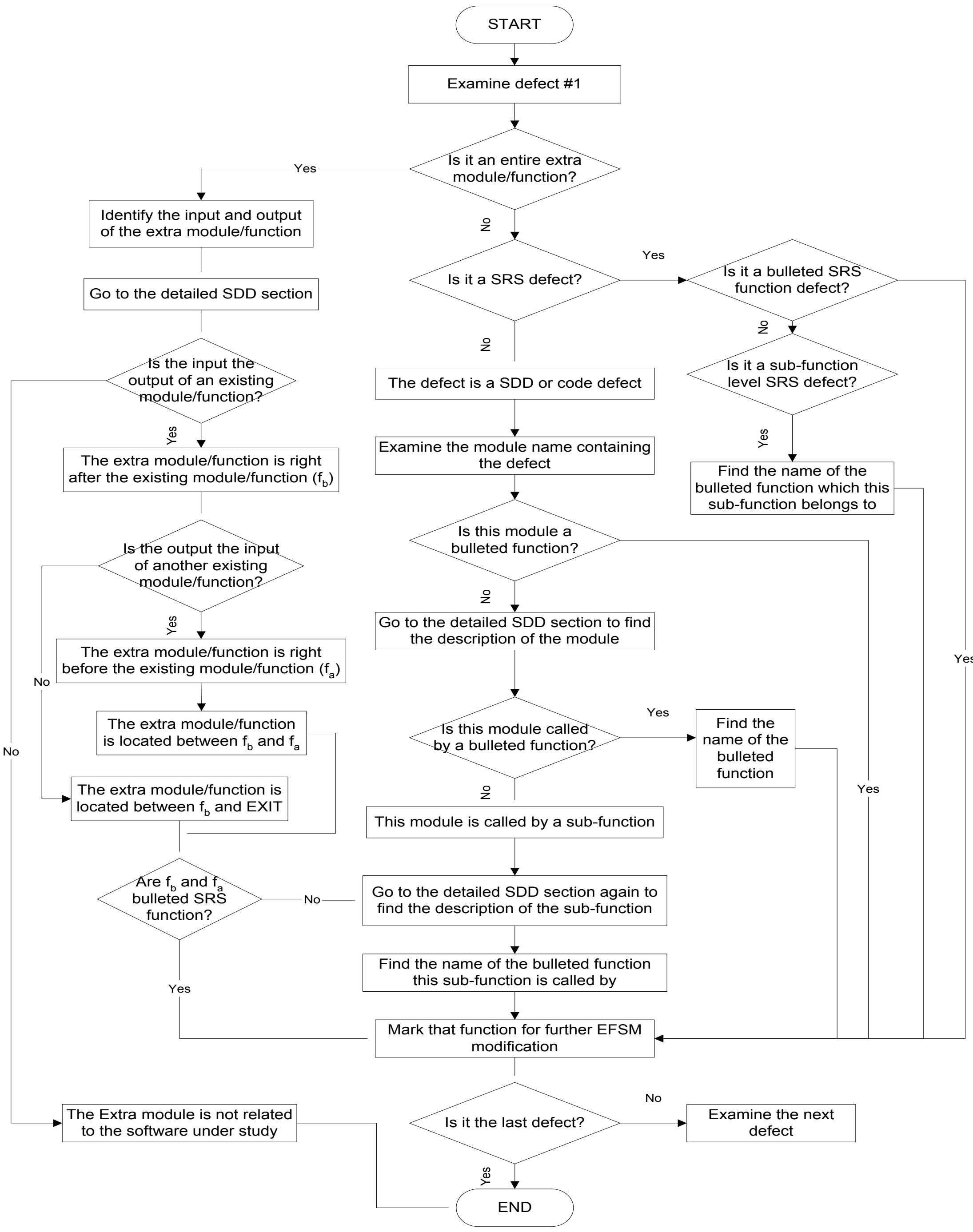

START
Examine defect #1
Is it an entire extra module/function?
Yes
No
Identify the input and output of the extra module/function
Go to the detailed SDD section
Is the input the output of an existing module/function?
Yes
No
The extra module/function is right after the existing module/function (f_b)
Is the output the input of another existing module/function?
Yes
No
The extra module/function is right before the existing module/function (f_a)
The extra module/function is located between f_b and f_a
The extra module/function is located between f_b and EXIT
Are f_b and f_a bulleted SRS function?
No
Yes
The Extra module is not related to the software under study
Is it a SRS defect?
Yes
No
Is it a bulleted SRS function defect?
Yes
No
Is it a sub-function level SRS defect?
Yes
Find the name of the bulleted function which this sub-function belongs to
The defect is a SDD or code defect
Examine the module name containing the defect
Is this module a bulleted function?
Yes
No
Go to the detailed SDD section to find the description of the module
Is this module called by a bulleted function?
Yes
No
Find the name of the bulleted function
This module is called by a sub-function
Go to the detailed SDD section again to find the description of the sub-function
Find the name of the bulleted function this sub-function is called by
Mark that function for further EFSM modification
Is it the last defect?
No
Yes
Examine the next defect
END

Figure 2.5 Flowchart for localizing the defects

Using the defect mapping procedures, the original and the modified EFSM for example 1 is shown below:

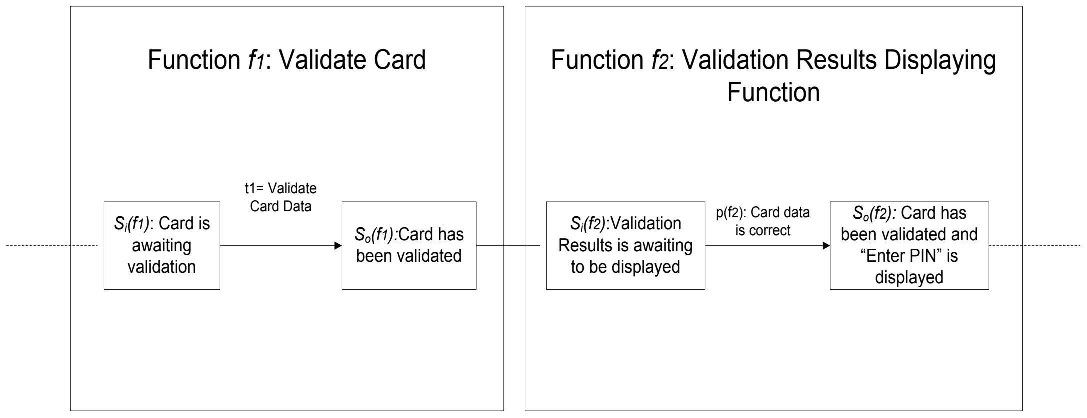

Function f_1: Validate Card
Function f_2: Validation Results Displaying Function
$S_i(f_1)$: Card is awaiting validation
t1= Validate Card Data
$S_o(f_1)$:Card has been validated
$S_i(f_2)$:Validation Results is awaiting to be displayed
p(f2): Card data is correct
$S_o(f_2)$: Card has been validated and "Enter PIN" is displayed

Figure 2.6 Original EFSM for Example 1

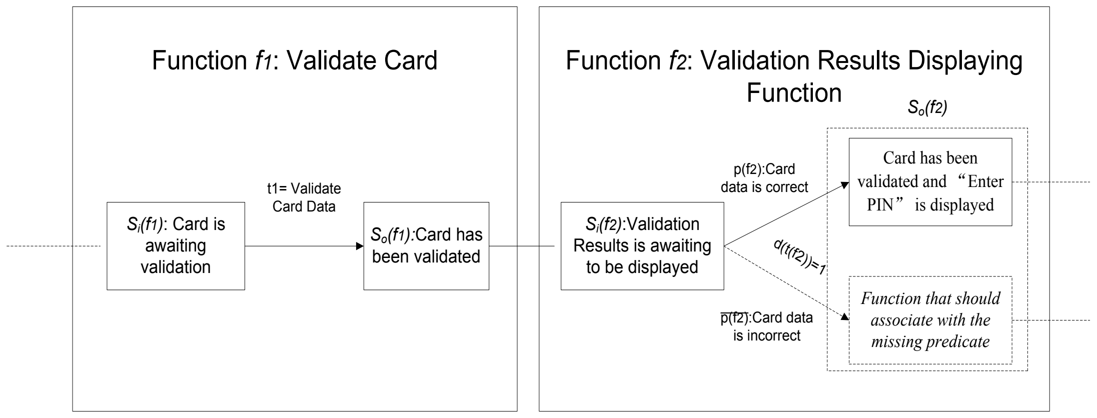

Function f_1: Validate Card
Function f_2: Validation Results Displaying Function
$S_o(f_2)$
$S_i(f_1)$: Card is awaiting validation
t1= Validate Card Data
$S_o(f_1)$:Card has been validated
$S_i(f_2)$:Validation Results is awaiting to be displayed
p(f2):Card data is correct
$d(t(f_2))=1$
$\overline{p(f2)}$:Card data is incorrect
Card has been validated and "Enter PIN" is displayed
Function that should associate with the missing predicate

Figure 2.7 Modified EFSM for Example 1

2.6.3.3 Section c: Split the HLEFSM to a LLEFSM and modify it:

If a defect was not directly related to a bulleted function, the HLEFSM model should be decomposed to a lower level of modeling. This is because a defect could be within a bulleted function while only part of the bulleted function is infected and will fail to perform adequately. Thus, one needs to break down the bulleted function to the level where the defect can be represented directly[6].

[6] A defect can be represented directly if the variable/function/subfunction which contains the defect is visible in the model since the level of detail in the model reaches the variable/function/subfunction.

The general procedures for the construction of the HLEFSM are still valid for the construction of the LLEFSM. However, special attention should be paid to the following issues:

1) Function f_i has a hierarchical structure, i.e. it is the parent function of its n sub-functions $f_{ij}, j = 1, 2 \ldots n$. These identified sub-functions are acting as child functions;

2) The I/O connections between the child functions can be easily determined by following steps c) to f) of the general construction procedures for the bulleted functions (step 1) but applying it now to the "Processing" section of the bulleted function. One should determine the interface between the child functions and their parent function by linking the beginning state $S_i(f_i)$ of the parent function with the beginning state of its first child function $S_i(f_{i1})$ and linking the ending state $S_o(f_{in})$ of the last child function with the ending state of its parent function $S_o(f_i)$ directly.

3) The input and output of the child functions may not be only in the "input" and "output" section of their parent function. The "processing" part also needs to be manually examined to identify the input and output of the child functions.

2.6.4 Step 4: Map the OP to the Appropriate Variables (or Transitions)

Generally, the operational profile is defined as $\{iv, OP(iv)\}$ in EFSM, where iv is the set of input variables and $OP(iv)$ is the set of probabilities of iv. As a very important attribute of the EFSM, OP should be predetermined and then mapped into the EFSM constructed through steps 1 to 3. If there is any predicate existing in the constructed EFSM, the probability of the execution of each branch needs to be determined since there are multiple subsequent states after the predicate.

If the predicate is only a function of the input variables from set Σ, which are crossing the boundary of the application, the probability of execution of each branch is usually determined by analyzing the operational data or can be found in various databases.

If the predicate is a function of internal variables from set V, i.e. variables which are within the boundary of the application, the probability of execution of each branch can be calculated based on input variables from set Σ since the internal variables are actually functions of the input variables from set Σ. For instance, consider the case where a predicate is determined by the value of an internal variable y which is a function of variable x ($y = f(x)$). Variable x is from set Σ whose OP is known either by analyzing operational data or by searching in databases. Thus, the OP of variable y can be analytically calculated through function $y = f(x)$. If function f is a complex function, the input/output table as suggested in Garret [36] should be utilized to obtain the value of y based on which the execution probability of each branch can be determined.

It should be mentioned that the mapping process does not entail as much work as one might think because the constructed EFSM is a compact version of the actual application since only defect related sections are modeled in detail. Furthermore, for safety critical systems, the relationship between the internal variables and the variables crossing the boundary of the system is kept simple to reduce the calculation error.

2.6.5 *Step 5: Obtain the Failure Probability by Executing the Constructed EFSM*

Application of the procedure described in Steps 1 to Step 4 yields the execution probability and the infected state. As for the propagation probability, it is assumed to be equal to 1. If a low level defect is detected, experimental methods such as fault injection can be utilized to assess the exact propagation probability.

The failure probability can be obtained by executing the constructed EFSM. The execution of the EFSM can be implemented using an automatic tool such as TestMaster [37]. TestMaster is a test design tool that uses the EFSM notation to model a system. TestMaster and similar tools capture system dynamic internal and external behaviors by modeling a system through various states and transitions. A state in a TestMaster model usually corresponds to the real-world condition of the system. An event causes a change of state and is represented by a transition from one state to another. TestMaster allows models to capture the history of the system and enables requirements-based finite state machine notation. It also allows for the specification of the likelihood that events or transitions from a state will occur. Therefore, the operational profile can be easily integrated into the model. Thus, the probability of failure from unresolved known defects can be assessed by simply executing the constructed TestMaster model.

First, TestMaster will execute all the possible paths of the constructed EFSM model. The paths which contain defect(s) can be recognized by TestMaster automatically. Thus, the probability of execution of the i^{th} path with defect(s) p_{path_i} can be calculated. Then the probability of failure is:

$$p_f = \sum_{path_i} p_{path_i} \tag{2.9}$$

where:

p_f is the probability of failure and

p_{path_i} is the probability of execution of the i^{th} path with defect(s).

2.7 *Application of the Metric-based RePS Theory*

Six of the forty metrics identified in [18] were applied to a small scale software, PACS, an automated Personnel entry/exit Access Control System [38] [39]. By "applied", it is meant that the six metrics were used as root metrics and the corresponding RePSs were developed and used to predict the reliability of the system under study, i.e. PACS. The predictions obtained were then compared to PACS' operational reliability. PACS was developed industrially by one of the US leading defense contractors using the waterfall methodology and a CMM level 4 software development process. PACS counts around 800 lines of code and was developed in C++. The six selected metrics were "Mean time to failure (MTTF)", "Defect density (DD)", "Test coverage (TC)", "Requirements traceability (RT)", "Function point analysis (FP)" and "Bugs per line of code (BLOC)"[7].

In the research at the origin of this thesis, we applied eleven root metrics to a safety-critical software system used in the nuclear industry and assessed its reliability. The software selected, APP [8], is a real-time safety-critical system. It is a microprocessor-based digital implementation of one of the trip functions of a Reactor Protection System (RPS) used in the nuclear power industry. The software system is based on a number of modules which include a "system software" and an "application

[7] The units of these six metrics are: hours, defects per line of code, percentage, percentage, function point and defects respectively.

[8] Detailed APP information (development and testing documents) can not be provided since they are proprietary documents.

software". The "system" software monitors the status of the system hardware components through well defined diagnostics procedures and conducts the communications protocols. The "application" software reads input signals from the plant and sends outputs that can be used to provide trips or actuations of safety system equipment, control a process, or provide alarms and indications. The APP software was developed in ANSI C and is about 12,000 lines of executable code.

The eleven root metrics are "Bugs per line of code" (BLOC), "Cause & effect graphing" (CEG), "Software capability maturity model" (CMM), "Completeness" (COM), "Cyclomatic complexity" (CC), "Defect density" (DD), "Fault days number" (FDN), "Function point analysis" (FP), "Requirement specification change request" (RSCR), "Requirements traceability" (RT) and "Test coverage" (TC). Definitions of the eleven software metrics used for APP reliability prediction are presented in the Appendix A.

A summary description of the eleven metrics is provided in section 2.7.1. The generation of APP's OP is presented in 2.7.2. The reliability prediction results obtained using the eleven metric-based RePSs are displayed and analyzed in section 2.7.3. These predictions are validated by comparison to the "real" software reliability obtained from operational data & statistical inference. Further discussions about the measurement process for the eleven metrics used in this research are provided in section 2.7.4. The discussion includes an analysis of feasibility which takes into account the time, cost and other concerns such as special technology required to perform the measurements. Conclusions are presented in section 2.7.5.

2.7.1 <u>Summary of the Measures and RePSs</u>

Different software metrics can be collected at different stages of the software development life-cycle (i.e. requirements, design, code and test) and measurements will be based on applicable software development products, i.e software requirements specifications (SRS), software design documents (SDD), software code (SCODE) etc. For instance, the BLOC measurement process can not be conducted until code is developed, thus the earliest phase at which BLOC becomes "applicable" is the end of the implementation phase. Applicable phases (marked as "√") for each measure studied are summarized in Table 2.5. The earliest applicable phase (i.e. the phase for which measurement of a particular metric becomes meaningful) is marked with an additional symbol "*".

Table 2.5 Phases for which Metrics are Applicable

Metrics	**Applicable Phases**				
	Requirements (RE)	**Design (DE)**	**Implementation (IM)**	**Testing (TE)**	**Operation (OP)**
BLOC			√ *	√	√
CEG	√*	√	√	√	√
CMM	√*	√	√	√	√
COM	√*	√	√	√	√
CC			√*	√	√
DD			√*	√	√
FDN	√*	√	√	√	√
FP	√*	√	√	√	√
RSCR	√*	√	√	√	√
RT		√*	√	√	√
TC				√*	√

It should be pointed out that all measurements are performed during APP's operation phase. Focus on the operation phase is driven by the time elapsed since delivery of the APP system and the consequent unavailability of important historical information which could have characterized the software development process. For example, one can measure the FP count in the Requirements phase using an early version of the SRS. This would give us an estimate of reliability based on FP early in the development life-cycle. Unfortunately, these early versions of APP's SRS are no longer available. The only SRS version available today is the final version, i.e., the version which was delivered at the end of the testing phase.

As addressed in section 2.1, measurement results can be either directly linked to software defects through inspections and peer reviews or indirectly linked to software defects through empirical software engineering models. The M-D models for each of the eleven root metrics are specified in Table 2.6. Support metrics used for the M-D models are also identified in the right column of Table 2.6. Detailed descriptions of the M-D models for each metric are provided in Appendix B.

As shown in Table 2.7, the eleven metrics under study can be further divided into three groups corresponding to the three types of defect information defined in section 2.1. In the case of the first group of metrics, only the number of defects can be obtained. Their location is unknown. Metrics in the second group correspond to cases where actual defects are obtained through inspections or testing. Thus, the exact location of the defects identified and their number is known. The metrics in the third group have the combinational characteristics of the first two groups. The exact location of defects detected in an earlier version is known and only the total number of defects in the current version can be predicted. Support metrics are also identified for each D-R model.

Table 2.6 M-D Models for Each Root Metric

Root Metrics	M-D Model	Support Metrics for M-D Model
BLOC	Gaffney's empirical model [28]	• The total number of modules • The code size ratio of a particular module • Severity level of the failures of interest
CEG	Rule-based Inspection	-
CMM	Correlation model relates CMM level with number of defects remaining based on empirical data [40]	• The number of function points • Severity level of the failures of interest
COM	Rule-based Inspection	-
CC	Correlation model based on the Success Likelihood Index Method (SLIM) [41] [42] [43]	• The number of modules whose CC belong to a certain pre-defined level • The total lines of code in the application
DD	Rule-based Inspection	-
FDN	Development process defects records and phase – based defect prediction method [44]	• The phase within which a defect originated (determined for each defect) • The number of requested repairs that are fixed in a specific phase • The number of repairs requested in a specific phase • The number of function points • The length of each life cycle phase • Type[9] of software systems • Severity level of the failures of interest
FP	Correlation model relates FP with number of defects remaining based on empirical data[45]	• Severity level of the failures of interest • Type of software systems
RSCR	Correlation model based on the Success Likelihood Index Method (SLIM) [41][42][43]	• The size of the changed source code corresponding to RSCR • The total lines of code in the application
RT	Rule-based Inspection	-
TC	Testing records and Malaiya's model [46]	• The number of defects found by test cases documented in the testing results.

[9] The types of software systems are defined as End-user software, Management information system, Outsourced and contract software, Commercial software, System software and Military software. Detailed definition of each type is provided in [45].

Table 2.7 Three Groups of Metrics

Group	**I**					**II**				**III**	
Model	**Model I**					**Model II**				**Model III**	
Metrics	BLOC	CMM	CC	FP	RSCR	CEG	COM	DD	RT	FDN	TC
Support Metrics	T_L, τ					τ				τ	

2.7.2 *APP Operational Profile Generation*

The operational profile for APP is defined as a complete set of "plant inputs" and "infrastructure inputs". The set of "Plant inputs" consists of the reactor's delta flux parameters which are being monitored by sensors. The counterparts to "plant inputs" are "infrastructure inputs," which are used to determine the hardware (i.e. computer platform) and software health statuses. Thus, APP's OP is:

$$OP_{APP} = (OP_{PI}, OP_{II}) \tag{2.10}$$

Where:

OP_{APP} is the operational profile for APP;

OP_{PI} is a subset of the operational profile which corresponds to the plant inputs;

OP_{II} is a subset of the operational profile which corresponds to the infrastructure inputs.

Ideally the OP for plant inputs can be derived from the plant's operational data if this data set is complete. By "complete", it is meant that data corresponding to both normal and abnormal conditions should be present in the data set. In the case of APP, "normal data" corresponds to situations under which the reactor operates within the Barn-Shape given in Figure 2.8; "abnormal data" corresponds to situations such that the data is outside the Barn-Shape. If the power and delta flux pair falls out of the barn-shape, APP's application software trips; otherwise it does not.

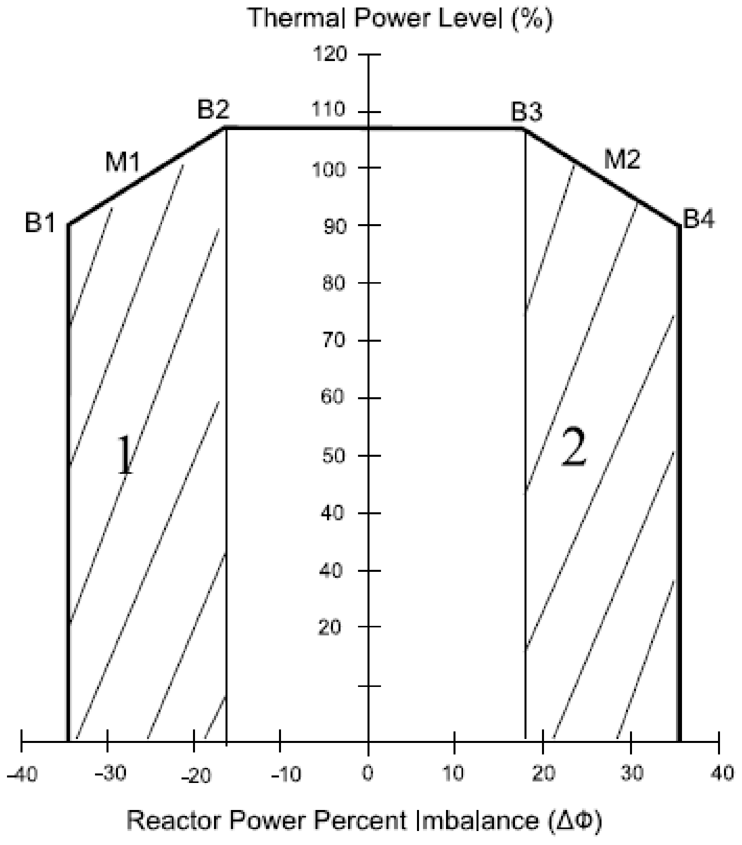


Figure 2.8 Flux/Flow Delta Flux Trip Condition (Barn Shape)[10]

We examined a data set which contains eleven years (1/1/96 to 1/1/06) worth of operational data collected (hour by hour) for one of the power plant's channel[11]. There are altogether 88,418 distinct data records. After eliminating 15,907 outage, missing and aberrant data, 72,511 data records are used for the assessment of OP_{PI}. The results are summarized in Table 2.8.

[10] *DF* is the flux imbalance, *P* is the thermal power, *TT* is the maximum thermal power, *B,1 B,2 B,3 B4 M1* and *M2* are setpoints (coefficients).
[11] The power plant control logic is comprised of three independent control units. Each unit contains 4 channels, each channel contains one APP safety module.

Table 2.8 APP's Operational Profile-Plant Inputs (OP_{PI})

Event	Condition[12]	Number of Data Records	Probability (per demand)
1	$DF < B_1$	2	9.8828E-10
2	$B_1 \leq DF \leq B_2$ and $P > (M_1)(DF) + [T_T - (M_1)(B_2)]$	0	5.1134E-10
3	$P > T_T$	7	3.4594E-9
4	$B_3 \leq DF \leq B_4$ and $P > (M_2)(DF) + [T_T - (M_2)(B_3)]$	0	2.4725E-10
5	$DF > B_4$	1	4.9414E-10
6	Normal	72,501	0.9999999943

The probability of occurrence of conditions 1, 3 and 5 can be estimated as the number of data records over the total number of operational data records. No data records fall within the domains delineated by conditions 2 and 4. A statistical extrapolation method was applied to generate estimates for the operational profile in these two regions. The data sets which could be used to perform the extrapolation are those in the shaded area in Figure 2.8. The number of data records in area 1 is forty five (45) and in area 2 is one (1). A normal distribution is proved to fit the 45 data records corresponding to condition 2[13]. For condition 4, obviously, the fact that there exists only 1 data record in area 2 is not sufficient to perform a valid statistical extrapolation. The maximum likelihood estimator, an unbiased estimator of the likelihood of occurrence of an event is given in [47] as $\hat{\lambda} = \frac{r}{T}$ if we assume "r" event occurrences are observed in "T" hours of operating time. A common solution to occurrence rate estimation when no event has been observed is to take one half as the numerator (r) [48]. Thus, the probability of occurrence of condition 4 can be roughly estimated as 0.5/72,511 = 6.9 x 10^{-6}/hour (2.4725E-10/demand).

[12] It should be noted that these conditions are mutually exclusive although they may not appear to be so since the software handles these events in sequence. For instance, the software will first check whether Event 1 is satisfied. If *DF* does not satisfy the condition, Event 2 is then triggered and the code checks whether *DF* and *P* satisfy the second condition.

[13] Use Shapiro-Wilk test.

The operational profile for the infrastructure inputs cannot be obtained from field data. This is simply because OP_{II} is a function of the health status (i.e. normal conditions and in failure conditions) of related hardware components. The failures of these hardware components are rare and hardly observed, sometimes even over their entire performance periods. For OP_{II} quantification, first, the hardware-related OP events and the hardware component involved in such OP events should be identified. Then a quantitative fault tree technique can be used to assess the probability of such events. This approach is further explained using the following example. The example refers to a PROM (Programmable Read-Only Memory) self-test carried out by APP's system software. The EFSM which models the test is given in FigureFigure 2.9.

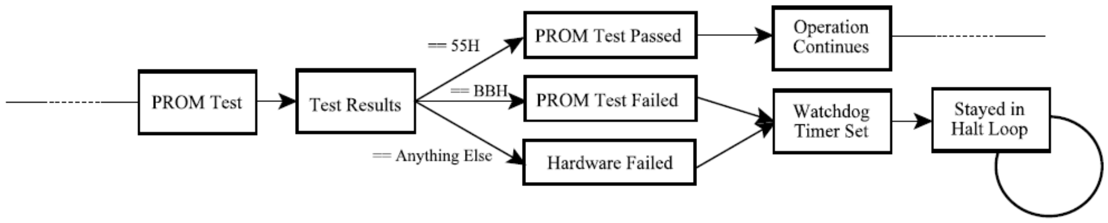


Figure 2.9 An Example EFSM Model for a PROM self-test function in the APP system

The events whose occurrence we are interested in are: Event 1 = "The "*Test Results == 55H*" ", Event 2 = "The "*Test Results == BBH*" ", and Event 3 = "The "*Test Results == Anything Else*" ". The probabilities of these events should be determined. The PROM test compares the checksum of the PROM with a predefined value. The value 55H will be written to a specific status address if the test passes or BBH if it fails. Any value other than 55H or BBH is not expected but may occur if the hardware components performing the writing operation (i.e. Random Access Memory, register, etc.) fail during the status writing operation.

The following fault tree is constructed to quantify Event 2 = "The "*Test Results == BBH*"".

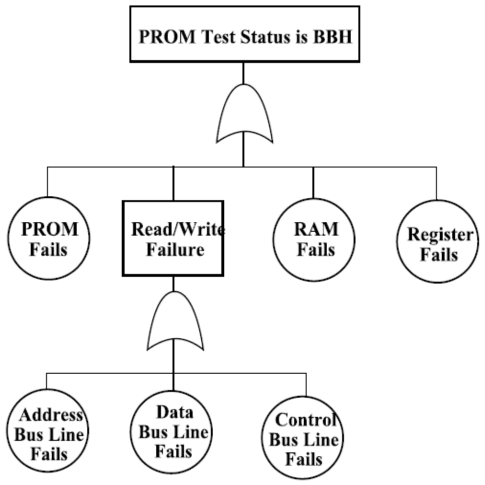


Figure 2.10 Fault Tree for Event 2: PROM Test Status Flag is BBH

The basic fault tree events need to be quantified. The ideal solution is to obtain all failure rate data of the specific hardware components from the manufacturers. This approach normally does not work due to the proprietary nature of such information. Some public databases, such as the RAC database [49], [50] and the Nuclear Regulation Commission (NRC) database [51] which provide generic failure rate data of a certain type of hardware component (i.e. the data can not be traced to a particular manufacturer or to the detailed specification of a particular hardware component), can be used for the probabilistic modeling of digital systems.

In this application, NRC database was used to obtain the failure rates of the basic event. The information required to quantify the fault tree illustrated in Figure 2.10 is listed in Table 2.9. A same approach was taken for event 3 probability quantification.

The complete infrastructure inputs-related OP for the PROM test in Figure 2.9 is shown in Table 2.10.

Table 2.9 Failure data information required to quantify Event 2

Events Hardware	Components	Failure Rate (failure/demand)	Resources	Results (failure/demand)
The probability of Event 2 = "PROM Test Status Flag is BBH"	PROM	$FR_1 = 9.32 \times 10^{-13}$	NUREG/CR-5750	$FR = \sum FR_i = 7.23 \times 10^{-11}$
	RAM	$FR_2 = 1.18 \times 10^{-11}$	NUREG/CR-5750	
	Read/Write	$FR_3 = 5.73 \times 10^{-11}$	NUREG/CR-5750	
	Register	$FR_4 = 2.19 \times 10^{-12}$	NUREG/CR-5750	

Table 2.10 APP OP-Infrastructure Example Results

No.	Event	OP (per demand)
1	PROM Test Status Flag is 55H	P1 = 1 - P2 - P3 = 0.9999999998564
2	PROM Test Status Flag is BBH	P2 = 7.23E-11
3	PROM Test Status Flag is neither 55H nor BBH	P3 = 7.13E-11

2.7.3 Application Results: Analysis

APP's actual reliability is assessed though analyzing the operational data. It is known that three copies of APP have been deployed in a nuclear power plant and have been functioning for a total of 281 months. APP operational failures have been documented in the Problem Records. Each record mainly consists of a detailed problem description and a corresponding set of corrective actions. Two out of the fourteen APP related records have been identified as being the result of APP software-related failures. Therefore, the failure rate (failure/demand) of the APP software can be estimated as: $\hat{\lambda} = \frac{r}{T} \times \tau = \frac{2}{281 \times 30 \times 24 \times 3600} \times 0.129 = 3.542 \times 10^{-10}$ per demand.

The artifacts used, the number of defects found using each metric and the probability of failure based on each metric's RePS are shown in Table 2.11. The prediction results are further compared with the actual assessment. The inaccuracy ratio (ρ) is defined to quantify the quality of the prediction:

$$\rho_{(RePS)} = \left| log \frac{P_f(RePS)}{P_f} \right| \qquad (2.11)$$

where

ρ_{RePS} is the inaccuracy ratio for a particular RePS;

P_f is the probability of actual failure per demand obtained from APP operational data;

$P_{f\ \mathrm{(RePS)}}$ is the probability of failure per demand predicted by the particular RePS.

This definition implies that the lower the value of $\rho_{(RePS)}$, the better the prediction. The last column of Table 2.11 provides the inaccuracy ratio for each of the eleven RePSs.

Table 2.11 Reliability Prediction Results

Group	Root **Metric**	**Required Documents**	**Number of Defects (Outcome of the M-D model)**	**Probability of Failure (per demand) (Outcome of the D-R model)**	**Inaccuracy Ratio**
I	BLOC	Code	14	8.43E-5	5.3764
	CMM	SRS, SDD, Code	26	1.144E-4	5.5091
	CC	Code	29	1.746E-4	5.6927
	FP	SRS	10	6.02E-5	5.2303
	RSCR	SRS, Code	12	7.22E-5	5.3095
II	CEG	SRS, Code	1	6.732E-13	2.7243
	COM	SRS, Code	1	6.683E-13	2.7211
	DD	SRS, SDD, Code	4	2.312E-10	0.1853
	RT	SRS, Code	5	3.280E-10	0.0334
III	FDN	SRS, SDD, Code	1	6.45E-11	0.7397
	TC	Code, Test Plan, Test Results	9	5.805E-10	0.2146

Generally speaking, reliability prediction based on RePSs constructed from the metrics in the first group is not good. This is because the defects' locations are unknown and Musa's exponential model is unable to model the exact software system structure. For instance, during normal operation, two microprocessors work redundantly for safety concerns. If either of the microprocessors calculates a trip condition, the APP system will send out a trip signal. However, it may be very difficult to take into account the actual structure of the system in Musa's exponential model since it is difficult to separate the number of defects per processor and envision what type of failure might occur.

In addition, Group I RePSs use an exponential reliability prediction model with a fault exposure ratio parameter set to $4.2\text{x}10^{-7}$. This parameter always dominates the results despite possible variations in the number of defects. This is evidenced by the small variation of the inaccuracy ratios observed for Group I RePSs. The value of K is not suitable for current safety critical systems. For instance, if one evaluates safety critical software reliability within a one-year period, the time t is roughly 3.15E7 seconds. For a real time system, the T_L is normally below 1 second. Further assuming there is only one fault remaining in the code, the reliability is calculated as: $R(t) = \mathrm{e}^{-\mathrm{K}\times\mathrm{N}\times\mathrm{t}/\mathrm{T_L}} = \mathrm{e}^{-4.2\times10^{-7}\times1\times3.15\times10^{7}/1} = 1.8\times10^{-6}$ where we assume that the T_L is less than 1 second. This tells us that software with only one fault remaining definitely fails at the end of one year. This conclusion contradicts what we have learnt from power plant field data.

DD in Group II enforces the inspection of all documents (SRS, SDD and code) for all possible types of faults. Application of this metric, however, requires more software engineering experience than that which is needed to implement measures like CEG, COM and RT whose inspection rules are relatively simple.

In the case of Test coverage, the fault exposure ratio, K, can be updated using the finite state machine models and defects found during testing. The result is shown in the following table.

Table 2.12 Fault Exposure Ratio Results

	Fault Exposure Ratio
Musa's K	4.2×10^{-7}
$\upsilon K(\tau)$	4.5×10^{-12}

It is clear that the actual fault exposure ratio for APP is far less than 4.2×10^{-7}. It is proved again that Musa's K is not suitable for safety critical systems.

2.7.4 Discussion about the Measurement Process

An estimate of the total time taken[14] for reliability prediction based on each of the eleven root metrics is provided in Table 2.13. The total time spent is further separated to account for the following five categories of effort:

1) Effort category 1 covers the time spent for tool acquisition, comparison between possible tools and training to become familiar with the identified tools;
2) Effort category 2 covers the time spent for the implementation of the M-D models: i.e. measurement of the root metric and of the support metrics;
3) Effort category 3 covers the time spent for the implementation of the D-R models: e.g. construction of the EFSM;
4) Effort category 4 covers time spent for documentation;
5) Effort category 5 covers other contributions.

[14] All the measurements except the measurement of FP were performed by graduate students. The total time taken was approximately recorded by the students.

The time spent in each effort category is also provided in Table 2.13.

Table 2.13 Total Time Spent for the Eleven RePSs

Group	Root Metric	Total Time Spent	Speed	Effort Category (in days)				
				1	2	3	4	5
I	BLOC	160 hrs (20 days)	Fast	4	10	2	4	-
	CMM	120 hrs (15 days)	Fast	-	4	8	3	-
	CC	360 hrs (45 days)	Medium	4	2	3	6	30
	FP	128 hrs (16 days)	Fast	-	7	5	4	-
	RSCR	360 hrs (45 days)	Medium	-	8	3	4	30
II	CEG	350 hrs (44 days)	Medium	10	26	5	3	-
	COM	512 hrs (64 days)	Medium	10	45	5	4	-
	DD	704 hrs (88 days)	Slow	10	69	5	4	-
	RT	640 hrs (80 days)	Slow	10	61	5	4	-
III	FDN	240 hrs (30 days)	Fast	-	16	8	6	-
	TC	904 hrs (113 days)	Slow	10	30	15	4	54

The speed is defined as follows:

1) Fast − the set of measurements and calculations can be finished within 300 hours;
2) Medium − the sets of measurements and calculations require at least 300 hours and no more than 600 hours; and
3) Slow − the sets of measurements and calculations require more than 600 hours.

Measurements and calculations related to BLOC, CMM, FDN, and FP RePSs can be completed quickly since there is no need to inspect the SRS, SDD, and Code. Measurements and calculations related to CEG, COM, CC, and RSCR require careful inspection of the SRS or the Code. Thus they need more time. Measurements related to DD and RT[15] require inspection of all the related documents. As a result, the RePSs measurement process for these two metrics is slow.

For CC and RSCR, additional effort (30 days for each) was spent developing new correlation models linking CC and RSCR measurements to number of software defects.

[15]The effort devoted to RT measurement could have been improved with current traceability tools.

For the measurement of test coverage, since no defects were uncovered by the last set of test cases, 20 out of the 30 days of measurement effort were devoted to exploring earlier test plans and corresponding test results. In addition, much time was spent (54 days) modifying the original test cases to adapt them to current simulation environments. In the absence of such compatibility problems, the measurements would have been completed significantly faster.

Some measurements are also quite costly. In Table 2.14, the required tools and corresponding cost for performing measurements related to the eleven RePSs are shown.

Table 2.14 Cost of Supporting Tools

Group	Root Metric	Required Tools	Tool Cost
I	BLOC	RSM Software	Free
	CMM	CMM Formal Assessment	$50,000[16]
	CC	RSM Software	Free
	FP	FP Inspection	$7,000
	RSCR	N/A	0
II	CEG	UMD Software 1 (CEGPT)	$750
	COM	TestMaster	$50,000
	DD	TestMaster	$50,000
	RT	TestMaster	$50,000
III	FDN	UMD Software 2 (FDNPT)	$750
	TC	TestMaster, Keil, IAR	$51,220

For three of these eleven RePSs, corresponding measurements have to be performed by experts. The following table presents the related information.

Table 2.15 Experts Required

Metric	Expert	Training
CMM	CMM Authorized Lead Appraiser and Development Team	SEI Formal Training
DD	Senior Software and System Engineer	10 Years Experience
FP	Function Point Analyzer and Development Team	Function Point Training

[16] If a company/organization is CMM certified, the cost related to CMM measurement will be zero.

2.7.5 Conclusions

A panel of experts was invited to review and provide comments on the methodology and results presented in this research. The following experts were contacted and invited to participate in the review.

1) Dave N. Card, Fellow, Software Productivity Consortium

2) J. Dennis Lawrence, Partner, Computer Dependability Associates, LLC

3) Michael R. Lyu, Professor, Chinese University of Hong Kong

4) Allen P. Nikora, Principal Member, Jet Propulsion Laboratory

As an integral part of their review of [20] and based on the results of this research, the experts recommended a subset of the measures and corresponding RePSs for use. The experts elected to recommend a measure if the inaccuracy ratio of its related RePS is less than 1. Thus they recommended RT, DD, TC and FDN.

FDN seems to be able to provide good prediction results. However, the accuracy of FDN measurement heavily relies on the documentation which tracks defects throughout the development life cycle. The quality of these documents is unknown. Thus, if any of the other three metrics (RT, DD and TC) is available, FDN will not be recommended for use.

As shown earlier in Table 2.13, DD, RT and TC are all very time-consuming. Rule-based inspections or peer reviews are required by DD and RT. On the contrary, one should be able to conduct a reliability prediction based on the TC metric efficiently. In the case of APP, the time required for the measurement and calculations related to TC was excessive. This was due to the fact that significant time was wasted while modifying the original APP source code so that it could be compiled successfully by current compilers. In addition, for the measurement of TC, time was spent modifying the original test cases for the current simulation environments. In the absence of compatibility problems, the measurements would have been completed much faster.

Based on our findings and the expert's recommendations, we conclude that the TC RePS is a good candidate for reliability prediction. In the next section, the TC-based RePS is described in details.

2.8 Test Coverage Based RePS

Test coverage is routinely used in industry to determine the level of completeness of the testing process. In IEEE [16], Test coverage (TC) is defined as the percentage of requirement primitives implemented multiplied by the percentage of primitives executed during a set of tests. A simple interpretation of test coverage can be expressed by the following formula:

$$TC\% = \left(\frac{IC}{RC}\right) \times \left(\frac{PRT}{TPP}\right) \times 100 \tag{2.12}$$

Where:

IC is the implemented capabilities;

RC is the required capabilities;

PPT is the tested program primitives and

TPP is the total program primitives.

In this research, it is assumed that all the requirements have been implemented and the tested program primitives are represented by the lines of code, the definition of test coverage then becomes:

$$C_1 = \frac{LOC_{Tested}}{LOC_{Total}} \times 100\% \tag{2.13}$$

Where:

C_1 is test coverage obtained through testing.

LOC_{Tested} is the number of lines of code that are being executed by the test data listed in the test plan.

LOC_{Total} is the total number of lines of code.

The existence of a relationship between Test Coverage and reliability has been confirmed by many researchers. Empirical studies have shown that defect detectability is correlated to test coverage [52] [53] [54]. Piwowarsky [55] predicts reliability based on the fact that the fault removal rate is a linear function of the code coverage. Malaiya introduces a logarithmic model [46] that relates testing effort to test coverage and then estimates reliability using Musa's [3] exponential model. Chen's model [56] reduces the overestimation of the reliability prediction by Software Reliability Growth models by using coverage information collected during testing to extract only effective data from a given operational profile. Gokhale et al. [57][58] propose a unified definition of TC and incorporate explicitly the time-varying TC functions into the Enhanced Non-homogeneous Poisson Process (ENHPP) framework. In their model, variation in the number of failures experienced is proportional to variation in coverage via a detection rate function which varies with time. Pham and Zhang [59] revise the ENHPP reliability model by proposing *S*-shaped TC functions and by considering imperfect repair while assuming repairs take place as soon as the failure is experienced. Cai and Lyu [60] further integrate time and TC measurements together and present a hybrid reliability prediction model.

Among these proposed models, Malaiya's model directly relates test coverage to defect coverage and uses Musa's exponential model (described in section 2.2) to predict reliability. Malaiya's TC-based model is discussed next.

Malaiya's model calculates the number of defects remaining *N* from the number of defects found N_0 and test coverage C_1. The defects in the software can be grouped into three categories with respect to the concept of test coverage: 1) Type 1 defects include known defects discovered by test cases. These are located in the code covered; 2) Type 2 includes unknown defects located in the code covered; 3) Type 3 includes unknown defects located in areas of the code which have not been covered (see Figure 2.11). The number of defects remaining, *N*, calculated using Malaiya's model includes both known defects (type 1) and unknown defects (type 2 and type 3). N_0 corresponds to the type 1 defects. *N* is obtained using the following equations:

$$N = N_0/C_0 \tag{2.14}$$

where: N_0 The number of defects found by test cases provided in the test plan.

C_0 The defect coverage, which is defined in [30] as the fraction of defects found by test cases given in the test plan.

and

$$C_0 = a_0\, ln\big(1 + a_1(exp(a_2C_1) - 1)\big) \tag{2.15}$$

where: a_0, a_1, a_2 Coefficients which can be estimated from field data.

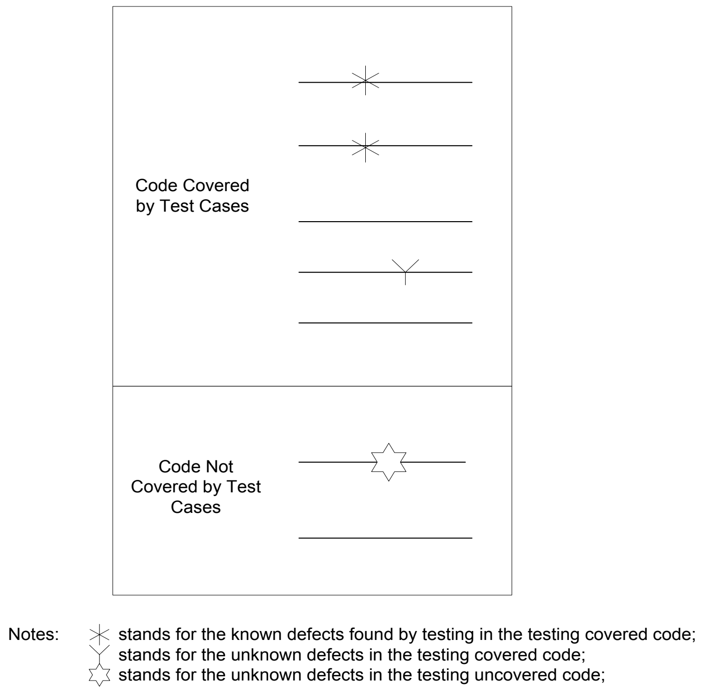

Code Covered by Test Cases
Code Not Covered by Test Cases
Notes:
stands for the known defects found by testing in the testing covered code;
stands for the unknown defects in the testing covered code;
stands for the unknown defects in the testing uncovered code;

Figure 2.11 Defect Categories

Once the total number of defects is obtained, D-R Model I described in section 2.2 can be applied as $R_{TC} = \mathrm{e}^{-\mathrm{K}\times\mathrm{N}\times\mathrm{t}/\mathrm{T_L}}$.

As we can see from equation (2.14), Malaiya's reliability prediction method is not valid if the number of defects found by test cases, N_0, is zero. It will lead to assessing N as equal to zero even when the test coverage is low which is meaningless. But for safety critical systems N_0 is typically zero since the last version of the code should be failure free [61]. Other existing approaches all rely on the assumption that the number of defects found by test cases is not zero; otherwise their approaches are not applicable.

Since the last version of a safety critical code should be failure free, there will definitely be multiple versions of this code. Consider Figure 2.12 displaying the possible multiple versions of the source code and test plan for a safety critical system *S*.

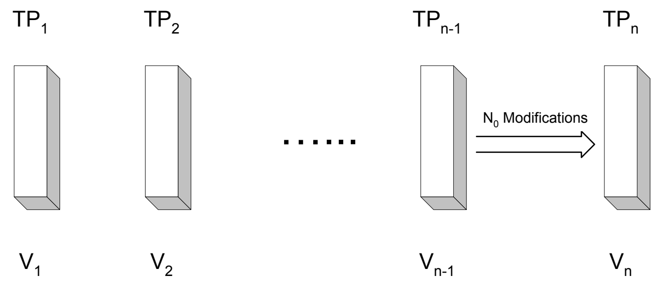


Figure 2.12 Multiple Versions of the Source Code and Test Plan

Let us assume that zero defects were found in version n, V_n, by test plan TP_n. While one can not as explained above use existing test coverage models for V_n , the models are applicable to earlier versions of the source code. However, the direct usage for reliability prediction based on TC of a previous version TP_{n-1} of the test plan and source code V_{n-1} is not accurate since the defects found in V_{n-1} by TP_{n-1} will undergo a repair process which will modify these defects and the affected code.

Direct usage of an earlier version of the source code to determine the number of defects remaining and that of the corresponding earlier version of the test plan to conduct the TC measurement and the reliability prediction introduces the following potential errors: 1) the prediction may be too conservative if the defects found are actually fixed; this is the most likely case; 2) the prediction may be overly optimistic if new defects are introduced during repair and not detected by the new test cases. These new defects could potentially be located on high probability paths and have the effect of drastically reducing reliability or on low probability paths and lead to more severe consequences than the original defects under repair. Two refinements for the TC RePS are presented in Chapter 3 and Chapter 4. They attempt to resolve issues associated to the use of test coverage (for reliability prediction) within a multi-phase functional testing process such as the one encountered in safety critical applications and the effects of non uniform repair.

Chapter 3: Predicting the Types and Locations of Faults Introduced During An Imperfect Repair Process and their Impact on Reliability

This chapter is a verbatim reproduction of the paper "Predicting The Types and Locations of Faults Introduced During An Imperfect Repair Process and their Impact on Reliability" published in the International Journal of Systems Assurance Engineering and Management, Vol 1, Issue 1, pp 33-40, March 2010, Springer Verlag.

Abstract

Imperfect debugging of software development faults (called primary faults) will lead to the creation of new software faults denoted secondary faults. Secondary faults are typically fewer in numbers than the initial primary faults and are introduced late in the testing phase. As such it is unlikely that they will be observed during testing and their failure characteristics are unlikely to be assessed accurately. This is an issue since they may possibly display different propagation characteristics than the primary faults that led to their creation. In particular their location will be distributed non-uniformly around the fault being fixed. The paper proposes a methodology to assess the impact of secondary faults on reliability based on predicting their possible types and locations. The methodology combines a fault taxonomy, code mutation and Bayesian statistics. The methodology is applied to portions of the application software code of a nuclear reactor protection system. The paper concludes with a

discussion on the integration of the results within existing Software Reliability Growth Models.

Keywords: *Software Reliability, Imperfect Debugging, Code Mutation*

3.1. *Introduction*

In early software reliability growth models (SRGMs), such as the Jelinski-Moranda (JM) [62] or the Goel-Okumoto (GO) model [63] it is assumed that once a fault is detected, it is removed instantaneously through repair and that no new fault is introduced. This assumption reduces the complexity of the models greatly. However, it is not valid in real projects and is therefore not a reasonable assumption. It is possible that either the fault was not fixed or the fault was fixed, but a new fault was introduced during the repair process. In addition, the repair process may not be immediate.

The assumption of perfect repair has been questioned by many researchers and some SRGMs have been proposed to remove this unrealistic assumption. Goel [64] updates the hazard function in his original GO model by introducing an imperfect debugging probability; Ohba and Chou [65] model repair as a Markov process and update the JM model and the Littlewood model [66] by introducing an imperfect repair rate. They also update the GO model using an error-reintroduction rate; Yamada et al [67] introduce a fault introduction rate to correct both the exponential fault-content function and the linear fault-content function; Zeephongsekul et al [68] further distinguish the original faults (primary faults) from the faults introduced by imperfect repair (secondary faults) and introduce different repair rates and detection rates for these two types of faults; Other researchers introduce time-dependent repair rates to reflect the learning process during the testing phase[69] [70]; Gokhale [71] identifies different repair policies and applies a rate-based simulation [4] technique to estimate the corresponding number of residual faults. Other researchers propose more realistic SRGMs by removing the assumption of instantaneous repair and incorporating repair time [72] [73].

These various models point to the fact that: 1) New faults due to imperfect debugging (secondary faults) are much fewer in number than the original faults. Depending on the repair rate, the number of secondary faults introduced might be 5 to 50 times less than the number of primary faults with an industry average of 14 [74] [75]. This phenomenon is most acute for safety critical systems and ultra reliable systems, developed using great care and which are characterized by high repair rates [75]; 2) Secondary faults are introduced late in the testing phase. Indeed one will need to first experience the primary faults [76], then repair these thereby introducing a significant time delay especially if resources are limited [71]; 3) Secondary faults may have different fault propagation characteristics [76]. Due to 1) and 2), we will most likely be limited to observations of primary faults and the secondary fault process may not be understood accurately. This may be an issue if the secondary fault introduced has a more significant impact than the primary fault being fixed, for instance, if it is located in a path with a high fault exposure rate[17] or if the defect resides on a path of execution where it will have more severe consequences. These facts argue for the separate study of secondary faults and their propagation characteristics.

[17] Fault exposure rate is taken loosely here to mean the probability per unit of time that this particular fault becomes a failure.

This paper presents a methodology for predicting the fault propagation characteristics of residual secondary faults (i.e. faults that have not been detected through testing).

The remainder of the paper is organized as follows: section 3.2 introduces a taxonomy of debugging/repair errors; section 3.3 proposes a technique to model debugging/repair errors and the corresponding secondary faults and assess whether these secondary faults would survive the testing process; section 3.4 describes a method for assessing the probability of existence of these residual secondary faults; section 3.5 presents a method to assess the impact of these faults on reliability and their fault propagation characteristics. An application of the proposed technique to a software system used in the nuclear industry is presented in section 3.6. Section 3.7 concludes the paper and discusses possible integration of the results into software reliability growth models.

3.2. Repair Error Taxonomy

To identify the types of secondary faults that can be created through an erroneous repair process, one should first have a well-defined repair error taxonomy which categorizes possible repair errors. Each imperfect code repair is due to a human error taking place during the repair process[18]. But what types of errors are possible? Many researchers have attempted to answer this question and have developed corresponding taxonomies (see for instance [77] [78]). These taxonomies were built to satisfy a wide range of underlying motivations (e.g., understanding root causes of programming mistakes, developing fault tolerance measures, or facilitating

[18] We assume the repair errors have similar characteristics as the programming errors.

testing [79]) which have influenced the classification scheme. In this paper, we classify errors with respect to their impact in terms of physical variations of the code since we are interested in the type and location of secondary faults. The taxonomy used is shown in Table 3.1. It consists of two levels of abstraction. To the left is the most abstract level of the taxonomy which derives from James' error taxonomy [80] and classifies errors in five broad categories from omission to blending (where blending is a mixture of the other four types of errors). To the right are fourteen classes of errors obtained by applying each of the five abstract error classes to three different logical groupings of the code: entities (*E*), logical lines of code (*L*) and clusters of lines of code denoted cluster of multi-logical lines of code (*M*). Entities, logical lines of code (LOC) and clusters of multi-logical lines of code correspond to various groupings of the code locations. An "entity" includes the primitive concepts of the language such as variables, constants, operators and syntactic connectives (e.g. braces). Entities form the base vocabulary of the programming language and as such correspond to the lowest grouping level. A "logical LOC" refers to a computer "instruction", but its specific definition is tied to specific computer languages. In C-like programming languages, a "logical LOC" is contained within two semicolons. Detailed logical LOC counting rules for a specific language can be found in the 2001 CMU/SEI technical report [81]. This grouping is used to express the lowest level functions[19] found in a computer program such as initialization, input, input data

[19] A "function" is a unit which performs a specific functionality. The 2nd level proposed in this paper is applicable to functional languages such as C where the primary concept of the language is that of function. For object oriented languages where the primary concept is that of object this second level of the taxonomy could be modified accordingly.

validation etc[20] [82]. A "cluster of multi-logical LOC" is a set of coupled logical LOCs which implements a meaningful system action. This corresponds to the next level of functional abstraction found in computer programs. The determination of clustered LOCs is not an easy task, and, preliminary rules for partitioning source code into clustered LOCs have been devised by adopting the rules used in natural language partitioning [83] [84].

An index (from Cl_1 to Cl_{14}) is assigned to each of these fourteen error classes. Note that the blend errors are not fully enumerated. Cl_{13} to Cl_{14} are two representative examples of the possible blend error classes.

Table 3.1 Hierarchical Error Taxonomy to Capture the Physical Manifestations of Repair Errors

1st Level Error Taxonomy	**2nd Level Error Taxonomy**
Omission	Omission of a multi- logical LOC cluster (Cl_1) Omission of a logical LOC (Cl_5) Omission of an entity in a logical LOC (Cl_9)
Addition	Addition of a multi- logical LOC cluster (Cl_2) Addition of a logical LOC (Cl_6) Addition of an entity in a logical LOC (Cl_{10})
Misordering	Misplacement of a multi- logical LOC cluster (Cl_3) Misplacement of a logical LOC (Cl_7) Misplacement of an entity within a logical LOC (Cl_{11})
Misformation	Using an incorrect multi- logical LOC cluster (Cl_4) Using an incorrect logical LOC (Cl_8) Using an incorrect entity within a logical LOC (Cl_{12})
Blend Error	Omission of an entity of a logical LOC and Misplacement of an entire logical LOC (Cl_{13}) Misplacement of an entire logical LOC and Misplacement of an entity of a logical LOC (Cl_{14}) ……

[20] If a logical LOC contains a nested function call, it is still considered as a logical LOC and the nested function call itself is considered as an entity of the line. For instance, "*a = f(b) +c;*" is considered as one LOC even if "*f(b)*" is implemented using multiple LOCs. "*f(b)*" is an entity of this LOC.

3.3. Modeling Repair Error Types and Determining Whether They Remain in the Program

The method which we will use to obtain the potential secondary fault types appearing as a result of repair errors and to assess whether they remain in the code after test uses code mutation. Code mutation is a fault-based technique which attempts to represent the behavior of code subject to programmer errors by injecting faults in the code and observing the modified code behavior under various inputs. The injection of faults in the program is obtained through application of mutant operators (MO) to the program. The resulting program is called a mutant program (MP). In order to mimic all types of programmer errors, errors are introduced into a program by creating many versions of the program, each of which contains one error [85].

Consider a particular code repair activity (see Figure 3.1), it can be either successful and lead to a perfectly modified code with probability "*1-r*", or, be unsuccessful and lead to an imperfectly modified code with probability "*r*". Systematic mutation of the portion of code being repaired will create a set of mutant programs within which the correct code should reside (in Figure 3.1, A3).

By limiting code mutation to the portion of code being repaired, we mimic the non-uniformity of secondary fault location. Secondary faults will indeed be restricted to the areas of the code being reworked. More specifically, secondary faults can be located within places either physically or logically close to the code repair areas. Areas physically close to code repair areas are easily identifiable. Areas logically close to the code repair areas can be determined by using techniques such as Program

Slicing (PS) or examining the Program Dependency Graph (PDG) which illustrates explicitly both the data and control dependence for each operation in a program.

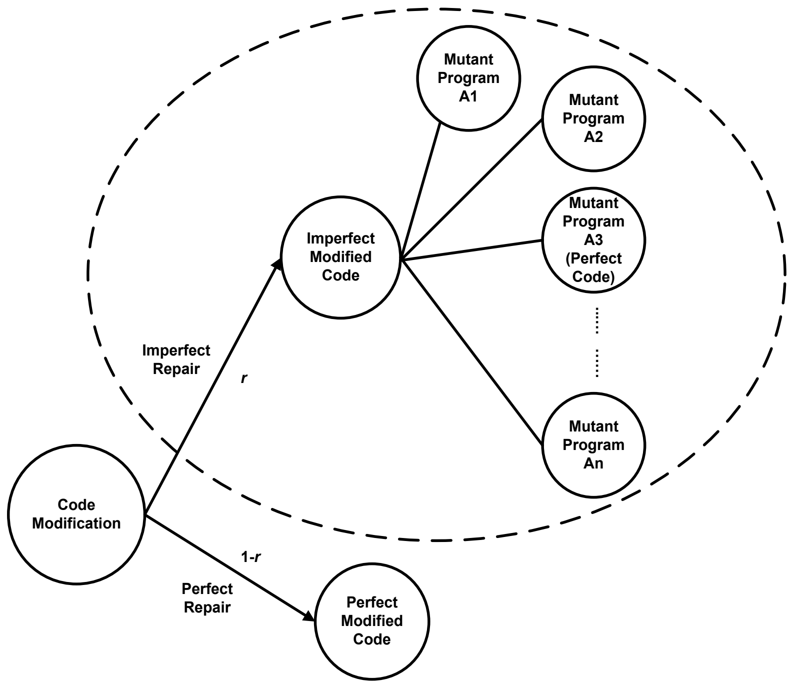


Figure 3.1 Code Repair and Its Mutants

During one specific attempt at repairing the code (i.e. one code repair), one or multiple repair errors can occur. The types of errors that might occur were defined in Table 3.1 of Section 3.2. If one assumes independence between errors, the probability of multiple errors occurring during one code repair activity is low. We will thus assume that only one error occurs during each repair. Therefore, all 2nd level error categories (i.e. the lowest abstraction level in Table 3.1) besides the "blend error" (which corresponds to a combination of multiple errors) should be modeled by applying corresponding mutant operators to reflect possible repair errors and obtain the related secondary faults.

For each mutant program generated, test cases are run to check whether they can distinguish the mutant program from the modified code. If they can, the mutant program is defined as "killed"; otherwise, the mutant program is defined as "live" (see Figure 3.2). Whenever the program is "killed", test cases distinguish the mutant program from the modified program. If there is indeed a defect in the modified code and the mutant is the correct program, test cases can detect it successfully. In case of a "live" program, test cases cannot distinguish differences between the mutant program and the modified program. If there is a defect in the modified code, the test cases cannot detect this fault. Therefore a live mutant program indicates potential existence of secondary faults in the modified code which are undetected by the suite of tests.

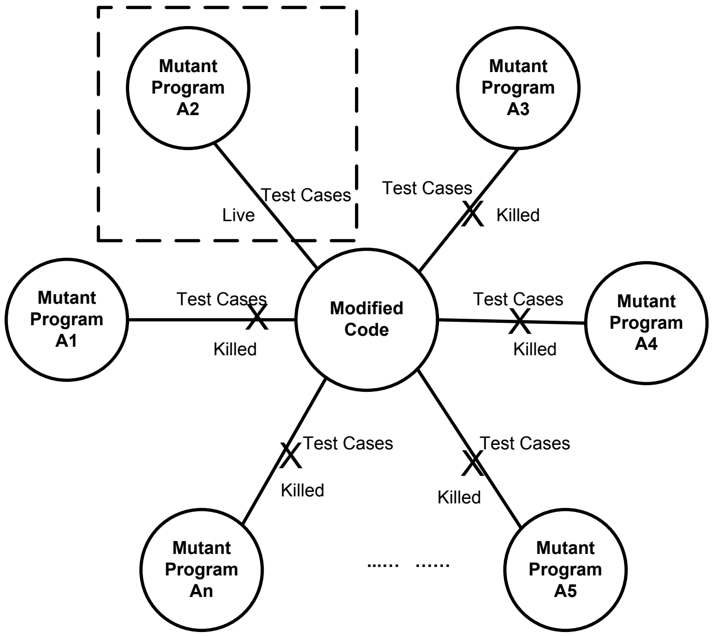


Figure 3.2 Test Cases and Mutants

3.4. Assessing the Probabilities of Different Types of Repair Errors

Using the methods described in Sections 3.2 and 3.3, one can identify the live mutant programs (LMP) which correspond to possible undetected secondary faults spawned by repair errors. These could lead to software failure. But initially before even assessing the impact these errors might have on reliability their probability of occurrence should be assessed.

The probability of occurrence of a mutant program, MP, is the probability that a repair error will lead to a version of the program under study whose physical form/embodiment is identical to that of the particular mutant program of interest.

The probabilities of occurrence of each repair error are not the same. Certain types of errors are more likely to happen depending on developers' programming characteristics. A programmer may tend to make certain types of errors over other types of error. The probabilities of occurrence of each repair error type can be determined using the following two sets of information: 1) the proportion of different types of repair errors. This reflects the developers' general preferences; 2) the total number of possible errors of each error type for a code segment under repair. There is a finite number of entities (*E*), logical lines of code (*L*) and multi-logical lines of code cluster (*M*) that repair errors can reside in. These two sets of information will be evaluated in turns.

Let us denote as θ_i the *normalized* proportion/percentages of errors that fall in the twelve error classes (Cl_1 to Cl_{12}) as given in Table 3.1[21]. We have: $\sum_{i=1}^{12} \theta_i = 1$.

The normalization process ensures independence of θ_i with code size and/or relative numbers of entities, lines of codes or clusters of multi-logical lines of code.

This normalized proportion can be derived from error data collected from the following three sources of evidence (S_1 to S_3) ordered by increasing degree of relevance: $S_{1:}$ error data, either from operation, testing or development, collected on other software systems; $S_{2:}$ error data collected during development of the software

[21] Blend errors are not considered.

under study (excludes data collected during testing); and $S_{3:}$ error data obtained during testing of the software under study.

Once collected, error data from any of these sources should be classified into the error taxonomy defined in section 3.2 (Table 3.1). Let us denote N_1e_T as the total number of errors from S_1. Among N_1e_T errors, there are N_1e_i (*i=1* to *12*) number of errors that fall in error classes Cl_1 to Cl_{12}. Similar notations can be applied to errors from sources S_2 and S_3.

To obtain the normalized proportion θ_i, the maximum number of possible occurrences of a particular type of error in the programs used to collect the error data discussed needs to be assessed. This number will be given by the number of occurrences of the semantic concept (violated by the error type) in the code[22]. For instance, for the error class "omission of an entire cluster of multi-LOC", the maximum number of opportunities is the total number of multi-LOC functions in the code. For the error class "omission of an entity of a LOC", the maximum number of opportunities is the total number of variables, constants, syntactic connectives such as parentheses and the nested functions in the code that served for data collection.

For a particular program P_j, the total number of opportunities for occurrence of an error in each error class (Cl_i), O_{ij}, is equal to the total number of semantic concepts underlying the class in the program. That is:

$$O_{ij} = \begin{cases} N_{Mj} & i = 1,...,4 \\ N_{Lj} & i = 5,...,8 \\ N_{Ej} & i = 9,...12 \end{cases} \tag{3.1}$$

where:

[22] This argument may not be true for those repair operations with iterated modifications. However, this case can be neglected since the possibility of occurrence of this situation is rare.

N_{Mj} is the total number of clusters of multi-logical LOC in program P_j,

N_{Lj} is the total number of logical LOCs in program P_j.

N_{Ej} is the total number of entities in program P_j.

If we have observed Ne_{ij} ($i=1$ to 12) number of errors in error class Cl_i for a particular program P_j from one of the data sources (S_1 to S_3), and if m such programs are available, the normalized proportion of errors in error class Cl_i is:

$$\theta_i = \begin{cases} \dfrac{\sum_{j=1}^{m} \frac{Ne_{ij}}{N_{Mj}}}{\sum_{i=1}^{4}\sum_{j=1}^{m} \frac{Ne_{ij}}{N_{Mj}} + \sum_{i=5}^{8}\sum_{j=1}^{m} \frac{Ne_{ij}}{N_{Lj}} + \sum_{i=9}^{12}\sum_{j=1}^{m} \frac{Ne_{ij}}{N_{Ej}}} & i = 1,\ldots,4 \\ \dfrac{\sum_{j=1}^{m} \frac{Ne_{ij}}{N_{Lj}}}{\sum_{i=1}^{4}\sum_{j=1}^{m} \frac{Ne_{ij}}{N_{Mj}} + \sum_{i=5}^{8}\sum_{j=1}^{m} \frac{Ne_{ij}}{N_{Lj}} + \sum_{i=9}^{12}\sum_{j=1}^{m} \frac{Ne_{ij}}{N_{Ej}}} & i = 5,\ldots,8 \\ \dfrac{\sum_{j=1}^{m} \frac{Ne_{ij}}{N_{Ej}}}{\sum_{i=1}^{4}\sum_{j=1}^{m} \frac{Ne_{ij}}{N_{Mj}} + \sum_{i=5}^{8}\sum_{j=1}^{m} \frac{Ne_{ij}}{N_{Lj}} + \sum_{i=9}^{12}\sum_{j=1}^{m} \frac{Ne_{ij}}{N_{Ej}}} & i = 9,\ldots,12 \end{cases} \qquad (3.2)$$

Using Equation (3.2), one can easily transform the direct counts of errors into the normalized proportion, θ_i, which accurately represents the error proportion distribution for the error data from a particular source. A Bayesian updating approach is applied to incorporate both historical/heritage repair data S_1 (e.g. [86]) and project specific repair data (S_2 and S_3). ζ_i^s ($i=1,\ldots,12$ and $s=1,2,3$) are denoted to represent the proportion of the i^{th} class of error in the s^{th} error data source. The Bayesian framework is selected over classical statistics because it can handle insufficient data; it can incorporate subjective data with objective data; and it can ease the parameter updating process if conjugate distributions can be used. In this paper, we use the

Dirichlet-Multinomial prior-likelihood conjugate pair to update the parameters of the Dirichlet distribution as more data on proportions becomes available. The formulas used to update the parameters of the prior and posterior Dirichlet distributions are shown in Table 3.2.

Table 3.2 Parameters of the Prior and Posterior Distribution[23]

Initial Prior (Unifor m Prior)	**Evidence from S_1**	$\rightarrow$	**Scaled Posterior (Prior For Next Update)**	**Evidence from S_2**	$\rightarrow$	**Scaled Posterior (Prior For Next Update)**	**Evidence from S_3**	$\rightarrow$	**Posterior**
$\gamma_i = 1$	ζ_i^1		$\gamma_i' = 1 + W_1 * N_3 e_T * \zeta_i^1$	ζ_i^2		$\gamma_i'' = \gamma_i' + W_2 * N_3 e_T * \zeta_i^2$	ζ_i^3		$\gamma_i''' = \gamma_i'' + W_3 * N_3 e_T * \zeta_i^3$

The average proportion for each error class (which corresponds to the expected value of the Dirichlet distribution) is obtained using the parameters γ_i, calculated with the formulas provided in Table 3.2:

$$\theta_i = E(\vartheta_i) = \frac{\gamma_i'''}{\sum_{i=1}^{12} \gamma_i'''} \tag{3.3}$$

Having obtained the average normalized proportion θ_i, we move onto calculating the *exact* proportion for a specific piece of modified code segment (*CS)*. The *exact* proportion of each error class can be calculated with the known total number of opportunities. This exact proportion, denoted as φ_{CS_i}, is no longer independent of code segment size and/or respective number of entities, lines of code or clusters of multi-lines of LOCs in CS. The *exact* proportion of errors of type "i" is given by:

[23] γ is a parameter of the Dirichlet distribution and v is the variable. W_1 W_2 and W_3 are weights assigned to sources of data S_1, S_2 and S_3 used to update the Dirichlet distribution. The weighting scheme selected ensures that the importance/weight of sources of data S_1 and S_2 does not exceed the importance of error data related to S_3.

$$\varphi_{CS_i} = \begin{cases} \dfrac{\theta_i \cdot N_M^{CS}}{\sum_{i=1}^{4}\theta_i \cdot N_M^{CS} + \sum_{i=5}^{8}\theta_i \cdot N_L^{CS} + \sum_{i=9}^{12}\theta_i \cdot N_E^{CS}} & i = 1,\ldots 4 \\ \dfrac{\theta_i \cdot N_L^{CS}}{\sum_{i=1}^{4}\theta_i \cdot N_M^{CS} + \sum_{i=5}^{8}\theta_i \cdot N_L^{CS} + \sum_{i=9}^{12}\theta_i \cdot N_E^{CS}} & i = 5,\ldots 8 \\ \dfrac{\theta_i \cdot N_E^{CS}}{\sum_{i=1}^{4}\theta_i \cdot N_M^{CS} + \sum_{i=5}^{8}\theta_i \cdot N_L^{CS} + \sum_{i=9}^{12}\theta_i \cdot N_E^{CS}} & i = 9,\ldots 12 \end{cases} \tag{3.4}$$

where:

φ_{CS_i} is the exact proportion of the i^{th} class of error for code repair *CS*;

N_M^{CS}, N_L^{CS} and N_E^{CS} are the total number of multi-logical LOC clusters, logical LOCs and entities of *CS* respectively;

θ_i is the average normalized proportion/percentages of errors that fall in the twelve error classes and $\sum_{i=1}^{12}\theta_i = 1$.

Thus, the probability of occurrence of the k^{th} live mutant program (corresponding to an error in the i^{th} class of error) of a *CS*, L_{CS_k}, during repair can be determined by the proportion of the class of errors (φ_{CS_i}) and the total number of possible MPs of this error class (N_{MPi}) that can be generated for the modified *CS*. That is:

$$L_{CS_k} = \frac{\varphi_{CS_i}}{N_{MPi}} \tag{3.5}$$

Here we assume uniform distribution between mutants which is a reasonable initial assumption since no evidence to the contrary is currently available.

The total number of possible MPs (N_{MPi}) can be readily obtained by multiplying the total number of opportunities (the total number of semantic concepts in *CS*) and the total number of substitutions defined by the mutation rules for each opportunity[24]. For instance, for the error class (Cl_{12}) "using an incorrect entity within a statement" class error, the total number of opportunities for using an incorrect variable is the total number of variables (N_v) in the modified code. The number of possible substitutions would be the number of other variables (N_v') defined in the program. Thus, the total number of MPs which can be generated for "using an incorrect variable" would be N_v*N_v'.

3.5. Effect of Remaining Faults on Reliability

Having defined in Section 3.2 a taxonomy of possible repair errors, in Section 3.3 a method to replicate the form of these errors in code modifications and to determine whether they will remain in the code after test (i.e. identify LMP), having introduced a quantification framework in Section 3.4 which allows us to estimate the conditional probability of existence of a particular live mutant program LMP given a repair error, one should now determine the effect of this LMP on reliability. The LMP as discussed earlier corresponds to the possible existence of an undetected secondary fault spawned through repair errors.

The impact of the secondary faults largely depends on their location, their type, the architecture of the system and the operational profile. The failure probability contribution due to each live mutant program can be estimated by applying the PIE [19] concept and using a Finite State Machine (FSM) model of the software system.

[24] For error class "addition" and "misplacement", multiple possible code injection locations, e.g. "before" or "after" or "within" places which are either physically or logically close to the code repair areas, are also counted when determining the total number of possible MPs.

Application of this technique allows us to identify the propagation characteristics of each potential remaining secondary faults spawned by repair errors.

Once the FSM model is built and executed, the fault propagation rate of secondary faults created during repair of a code segment CS can be obtained using the following equation:

$$b(CS) = \sum_{k=1}^{n_{CS}} (P(f_{CS}|CS_k) \cdot L_{CS_k})/\tau_{CS_k} \tag{3.6}$$

where

$b(CS)$	The fault propagation rate of secondary faults created during repair of a code segment *CS*.
$P(f_{CS}\|CS_k)$	The probability of failure due to a specific live mutant program L_{CS_k} of code segment *CS*. This probability is obtained by execution of the Finite State Machine for the particular mutant program;
L_{CS_k}	The probability of occurrence of a specific k^{th} live mutant program of code segment *CS*;
n_{CS}	The total number of live mutant programs for code segment *CS*.
τ_{CS_k}	The time required to execute a specific k^{th} live mutant program of code segment *CS* under the operational profile.

3.6. Application

The methodology is now applied to an ultra-reliable software of about 12,000 lines of executable code. The software studied, APP, is a real-time microprocessor-based digital implementation of one of the trip functions of a Reactor Protection System (RPS) used in the nuclear power industry. The software is based on a number of modules which include a “system software” and an “application software”. The “system” software monitors the status of the system hardware components through well defined diagnostics procedures and conducts the communications protocols. The “application” software reads input signals from the plant and sends outputs that can be used to provide trips or actuations of safety system equipment, control a process, or provide alarms and indications. The APP software was developed in ANSI C language. In the following, results of the application of the proposed approach to the “application” software of APP are described.

We apply the approach defined to three code repairs undertaken towards the end of APP’s testing phase. By applying the code mutation technique to the three code repairs, a total of 1,969 MPs were generated among which there are 27 LMPs (see Table 3.3 for mutants generated for the first code modification CS_1). The probability of occurrence of the 27 identified LMPs was determined (see Table 3.4 and

Table 3.5 for determination of the normalized and exact proportions respectively and Table 3.6 for the probability of occurrence of LMPs of CS_1).

The probability of failure corresponding to these 27 LMPs was obtained by mapping each of them into an FSM representing APP’s functionality. Among the 27 identified LMPs, 15 LMPs of CS1 and 8 LMPs of CS2 had the same propagation characteristics. Their contribution to the probability of failure of the application was obtained as 7.08×10^{-10} per demand (the average duration of a demand is .082

seconds). The remaining 4 LMPs of CS3 had a zero failure contribution since they had no impact on the trip function[25].

The fault propagation rate for each code repair can be obtained as:

$b(CS1) = \sum_{k=1}^{15} \frac{P(f_{CS1}|CS1_k)\cdot L_{CS1_k}}{\tau_{CS1_k}} = \frac{15\times(7.08\times10^{-10})\times(6.3\times10^{-4})}{0.082} = 8.16 \times 10^{-11}$ per second and

$b(CS2) = \sum_{k=16}^{23} \frac{P(f_{CS2}|CS2_k)\cdot L_{CS2_k}}{\tau_{CS2_k}} = \frac{8\times(7.08\times10^{-10})\times(5.9\times10^{-4})}{0.082} = 4.07 \times 10^{-11}$ per second.

Table 3.3 Mutant Programs Generated for the Code Modification *CS₁*

Error Class	Total MPs	LMPs
Cl_1	3	0
Cl_2	3	0
Cl_3	66	0
Cl_4	0	0
Cl_5	4	0
Cl_6	30	0
Cl_7	20	0
Cl_8	4	0
Cl_9	2	0
Cl_{10}	2	0
Cl_{11}	0	0
Cl_{12}	438	15
Total	**572**	**15**

Table 3.4 Error Data from sources S_1, S_2 and S_3 and corresponding updated value of the average Normalized Proportion θ_{12} for Error Class Cl_{12}.

Error Class	N_1e_{12}	ζ^1_{12}	N_2e_{12}	ζ^2_{12}	N_3e_{12}	ζ^3_{12}	$\boldsymbol{\theta_{12}}$
Cl_{12}	18	0.017	27	0.054	3	0.060	0.081

[25] They have an impact on the display function, i.e. these failures could result in an incorrect display. However, this function is not safety related. In this study, only type II failures are considered. This type of failure occurs when the system sends out a signal to trip the reactor while it should not.

Table 3.5 The Exact Error Class Proportions φ_{CS_i} for APP's Code Modification CS_1 derived from the Normalized Proportions

Error Class	**Normalized Proportions**	**CS1**	
	θ	**Number of Opportunities**	$\boldsymbol{\varphi_{CS_1}}$
Cl_1	0.298	$N_M^{CS1} = 2$	0.063
Cl_2	0.059		0.013
Cl_3	0.079		0.017
Cl_4	0.069		0.015
Cl_5	0.109	$N_L^{CS1} = 4$	0.046
Cl_6	0.053		0.023
Cl_7	0.05		0.021
Cl_8	0.053		0.022
Cl_9	0.05	$N_E^{CS1} = 32$	0.172
Cl_{10}	0.049		0.166
Cl_{11}	0.049		0.166
Cl_{12}	0.081		0.276

Table 3.6 Determination of the Probability of Occurrence of APP's Live MPs

Code Modifications	**Error Class**	**Number of Live MPs**	**Probability of Occurrence**
CS1	Cl_{12}	15	$L_{CS_k} = \frac{\varphi_{CS_i}}{N_{MPi}} = \frac{0.276}{438} = 6.3 \times 10^{-4}, k = 1, \dots, 15,$

3.7. Conclusions: Integrating the Results of the Proposed Method into the SRGMs

The method presented in Sections 3.2 to 3.5 allows us to predict the fault propagation rate for residual secondary faults due to repair errors as they would appear in operation. These rates can be calculated as soon as primary faults are uncovered. The method also provides possible locations and types for secondary faults. Potential locations, types and rates can help further inform the testing processes and can help increase learning as well as detection. The rates can also be used in conjunction with the number of residual secondary faults predicted by SRGMs to predict the contribution of secondary faults to reliability in operation.

When the testing environment reflects the operational environment, fault detection rates and fault propagation rates calculated by our method are identical and can be used to obtain a better assessment of the residual number of secondary faults. Further research will focus on how to better integrate our research with existing SRGMs.

Chapter 4: Predicting Residual Software Fault Content and their Location during Multi-Phase Functional Testing Using Test Coverage

This chapter is a verbatim reproduction of the paper "Predicting Residual Software Fault Content and their Location during Multi-Phase Functional Testing Using Test Coverage" submitted to the International Journal of Reliability and Safety.

Abstract

Multi-Phase functional testing is a common practice which is used in ultra-reliable software development to ensure that no known faults reside in the software to be delivered. In this paper, we present a new test coverage-based model which allows the description of software systems developed through multiple phases of functional testing. This model is further extended: 1) to take advantage of auxiliary observations collected during the multi-phase testing and consequent analysis process to refine the predictions made; 2) to describe software systems where either the initial fault distribution is non-uniform with respect to location, or the repair and test and detection process favor certain locations.

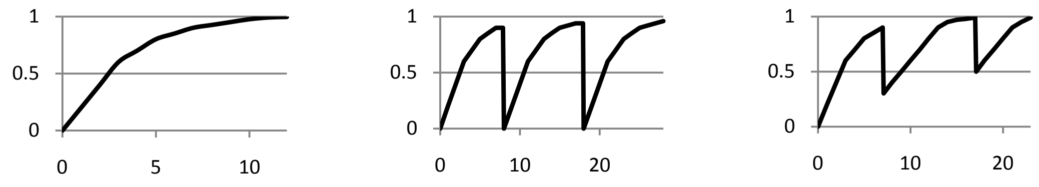
Keyword: Test Coverage, Multi-phase Testing, Imperfect Repair, Defect Location Prediction, Recursive Bayesian estimation

4.1. Introduction

Test coverage is an important measure used in software testing to reflect the degree to which the software has been tested. The relationship between test coverage

(TC) and defect coverage (i.e. percentage of defects identified through test) has been highlighted by many and number of research efforts have been devoted to linking test coverage to the number of faults remaining and number of failures experienced. To cite only a few, Vouk[52] directly relates the number of detected faults and test coverage through a Weibull function. Piwowarsky et al. [55] predicts reliability based on the fact that the fault removal rate is a linear function of the code coverage. Malaiya et al. introduces a logarithmic model [46] that relates testing effort to TC and then estimates reliability using Musa's exponential model. Malaiya et al. [87] also develop a logarithmic-exponential model which differs from his earlier model by considering the linear relations between defect coverage and TC once a certain TC level is achieved. Gokhale et al. [57][58] propose a unified definition of TC and incorporate explicitly the time-varying TC functions into the Enhanced Non-homogeneous Poisson Process (ENHPP) framework. In their model, variation in the number of failures experienced is proportional to variation in coverage via a detection rate function which varies with time. Pham and Zhang [59] revise the ENHPP reliability model by proposing *S*-shaped TC functions and by considering imperfect repair while assuming repairs take place as soon as the failure is experienced. Cai and Lyu [60] further integrate time and TC measurements together and present a hybrid reliability prediction model.

Existing models assume that functional testing is a continuous single phase process where the software is run through a single predefined series of tests. For such cases the coverage function is a monotonically non-decreasing function of time as described in Figure 4.1a. While this assumption is appropriate for a large class of software development efforts, it fails to represent development efforts where functional testing is organized as a multi-phase process. In such case the software will undergo several series of functional tests and the coverage function will increase monotonically by phase while experiencing discontinuities between phases (see Figure 4.1b and Figure 4.1c). Furthermore, repairs are not attempted as soon as failures are experienced but are deferred to the end of each phase. This process will in particular be found in the case of ultra-reliable systems where one needs to ensure that the software will pass through an entire series of tests without experiencing faults (See Section B.3.1.12.4 of [61]; Section 5.4.2 of [88]; [20]). This leads to the existence of at least two phases: one with faults, and one without faults.



(a) (b) (c)

Figure 4.1 Coverage is a continuous monotonic non-decreasing function of testing time (a); Coverage function for Multiple Phases of Functional Testing (b) (c)

In addition, these models make the assumption that faults are distributed uniformly in the code. There is no evidence that this might be true in practice (see for instance a recent study by [89]). While the uniformity assumption may not be critical for most software systems and as such is a very useful assumption, it needs to be carefully examined for ultra-reliable systems which are more sensitive to the location of faults. Location is indeed an important contributing factor in the severity of faults.

This paper proposes analytical expressions for the number of remaining faults and the fault location distribution which can be used for reliability prediction or to adjust testing efforts. Section 4.2 lists key notations used in the paper. In Section 4.3 we derive expressions for failures experienced and faults remaining for a software system undergoing multiple phases of functional test before being declared ready for fielding and operation. In Section 4.4 we extended the model presented in section II by incorporating auxiliary testing observations for key model parameters estimation. In Section 4.5, we account for the non-uniformity of the distribution of faults on fault sites. We conclude with possible applications of the extensions presented (see Section 4.6).

4.2. Notations

ã	Initial number of faults which exist in the code at the beginning of the first test phase
$\tilde{a}^{P(0)}$	Initial number of faults predicted at time $t = 0$
$m(t)$	Number of faults experienced by time *t*
$\Delta m_j(t)$	Number of faults experienced between t_{i-1} and t where $t_{i-1} \leq t < t_i$
$\Delta M_i^{P(0)}$	Number of failures we expect to experience during testing phase *i* predicted at time $t = 0$
ΔM_i^E	Total number of failures observed at the end of testing phase *i*
$\Delta m_{i\_new}^E(t_i^*)$	Observed number of new faults introduced during the repair process which takes place at the end of phase *i*
$\Delta m_{i\_nr}^E(t_i^*)$	Observed number of faults which were not repaired during the repair

	process which took place at the end of phase *i*
$nfr_{i(0)}$	Number of faults remaining in the code at the end of testing phase *i*, predictions made at time $t = 0$
$nfr_{i(..)}$	The number of faults remaining in the code at the end of testing phase *i*, predictions made at time $t = t_i^*, t_i^+$, ...
r	Repair rate
$r^{P(0)}$	Repair rate predicted at time $t = 0$
$r^{P(..)}$	Repair rate predicted at time $t = 0, t_i^*, t_i^+$, ...
γ	New fault introduction rate given that a repair fault has occurred
$\gamma^{P(0)}$	New fault introduction rate (given that a repair fault has occurred) predicted at time $t = 0$
$\gamma^{P(..)}$	New fault introduction rate predicted (given that a repair fault has occurred) at time $t = 0, t_i^*, t_i^+$, ...
K	Fault detection probability
K_i	Fault detection probability during phase *i*
$K_i^{P(..)}$	Fault detection probability during phase *i* predicted at time $t = 0, t_i^*, t_i^+$
$c_i(t)$	Coverage function over interval of time t_{i-1} and t_i where $t_{i-1} \leq t < t_i$
$c_U(i)$	Upper bound for the coverage in phase *i*
$c_L(i)$	Lower bound for the coverage in phase *i*
$C_i^E(t)$	Actual coverage function for phase *i*
$C_U^{P(0)}(i)$	Predicted (at time $t = 0$) upper bound for the coverage in phase *i*

$C_L^{P(0)}(i)$	Predicted (at time $t = 0$) lower bound for the coverage in phase *i*
k_D	Number of failed repair attempts during the development process
n_D	Number of repair attempts during the development process
v_D	Number of newly introduced faults due to imperfect repair
$c_i(L,t)$	Probability that a location is covered by time *t* in phase *i*.
$f_i(L,t)$	Probability that a fault resides in location L at time t during phase *i*
$K_i(L,t)$	Probability that a fault residing in location L at time t during phase *i* is detected when the potential fault site is covered
$k_i(L,L')$	Conditional probability that given that a fault is introduced or is moved due to a repair at location *L*, it moves to *L'*

4.3. *Number of Failures Experienced and Faults Remaining In the Case of Multiple Functional Test Phases*

For software systems such as ultra-reliable systems, a software component before being considered ready for release will need to undergo multiple phases of functional test. In phase 1, a first test plan will be used which contains a first set of functional tests. Failures are uncovered as testing progresses. Corresponding fixes are made at the end of the test phase. The modified code then undergoes another set of functional tests extract from a second test plan and so forth and so on. There may of course be some overlap between consecutive test plans. In such case the evolution of coverage with time will cease to be a continuous monotonic non-decreasing function of time as assumed in the models of section 4.1 and instead will take the form given in Figure 4.1or Figure 4.1c. Let us then try to express $m(t)$, the number of faults experienced by time *t*. We will denote by t_{i-1} and t_i respectively the beginning and end of phase "i"; by $\Delta m_j(t)$ the number of faults experienced between t_{i-1} and t where $t_{i-1} \leq t < t_i$; by $\tilde{a}$, the initial number of faults which exist in the code at the onset of functional testing, i.e. at the beginning of the first test phase.

Our assumptions are as follows: 1) Faults are uniformly distributed over all potential fault sites; 2) When a potential fault-site is covered, any fault present at that site is detected with probability K(t); 3) Repairs take place at the end of the phase and new faults may be introduced through repair errors. The repair rate is r and the probability of fault introduction given that a repair error has occurred is γ; 4) Coverage is a continuous monotonic non-decreasing function of testing time per phase as displayed in Figure 4.1b or Figure 4.1c.

Under those assumptions we obtain the following set of equations:

$$\frac{d\Delta m_1(t)}{dt} = \tilde{a}\, K_1(t)\, \frac{dc_1(t)}{dt} \tag{4.1}$$

for $0 \leq t < t_1$ where t_1 is the end of the first phase, $\Delta m_1(0) = 0$, and $c_1(t)$ is defined over $0 \leq t < t_1$ and is the coverage function over that interval of time.

$$\frac{d\Delta m_2(t)}{dt} = (\tilde{a} - \Delta m_1(t_1) \times r + \Delta m_1(t_1) \times (1 - r) \times \gamma) K_2(t) \frac{dc_2(t)}{dt} \tag{4.2}$$

for $t_1 \leq t < t_2$ where t_2 is the end of the second phase, $\Delta m_2(t_1) = 0$, and $c_2(t)$ is defined over $t_1 \leq t < t_2$ and is the coverage function over that interval of time.

$$\begin{aligned} \frac{d\Delta m_3(t)}{dt} = (\, \tilde{a} - \Delta m_1(t_1) \times r + \Delta m_1(t_1) \times (1 - r) \times \gamma) - \Delta m_2(t_2) \times r \\ + \Delta m_2(t_2) \times (1 - r) \times \gamma)\, K_3(t)\, \frac{dc_3(t)}{dt} \end{aligned} \tag{4.3}$$

for $t_2 \leq t < t_3$ where t_3 is the end of the third phase, $\Delta m_3(t_2) = 0$, and $c_3(t)$ is defined over $t_2 \leq t < t_3$ and is the coverage function over that interval of time.

We rewrite $\frac{d\Delta m_3(t)}{dt}$ as:

$$\frac{d\square m_3(t)}{dt} = \left(\tilde{a} - (\Delta m_1(t_1) + \Delta m_2(t_2)) \times r + (\Delta m_1(t_1) + \Delta m_2(t_2)) \times (1-r) \times \gamma\right) K_3(t)\,\frac{dc_3(t)}{dt} \tag{4.4}$$

So more generally we have:

$$\frac{d\Delta m_i(t)}{dt} = \left(\tilde{a} - r \times \sum_{k=1}^{i-1} (\Delta m_k(t_k) + (1-r) \times \gamma \times (\Delta m_k(t_k)))\right) K_i(t)\,\frac{dc_i(t)}{dt} \tag{4.5}$$

for $t_{i-1} \leq t < t_i$ where t_i is the end of the ith phase, $\Delta m_i(t_{i-1}) = 0$, and $c_i(t)$ is defined over $t_{i-1} \leq t < t_i$ and is the coverage function over that interval of time.

If the different K's are constant per phase, equation (4.5) can be integrated over each phase and we will obtain after integration:

$$\Delta m_i(t) = \left(\tilde{a} - r \times (\sum_{k=1}^{i-1} \Delta m_k(t_k)) + (1-r) \times \gamma \times \left(\sum_{k=1}^{i-1} \Delta m_k(t_k)\right)\right) K_i \int_{c_L(i)}^{c_U(i)} \frac{dc_i(t)}{dt} dt \tag{4.6}$$

which is thus

$$\Delta m_i(t) = \left(\tilde{a} - r \times (\sum_{k=1}^{i-1} \Delta m_k(t_k)) + (1-r) \times \gamma \times \left(\sum_{k=1}^{i-1} \Delta m_k(t_k)\right)\right) K_i \times (c_U(i) - c_L(i)) \tag{4.7}$$

where $c_U(i)$ and $c_L(i)$ are respectively the upper and lower bounds of coverage for phase i.

Example- Let us consider example software *S*. *S* is undergoing three functional test phases. Let us also assume that the upper and lower coverage values for each functional test phase, as well as the number of defects found in each phase is given in Table 4.1.

Table 4.1 Multiple Phase Test Profile for Software *S*

Phases	1	2	3
$C_L(i)$	0	0	0
$C_U(i)$	.5	.7	.95
Number of faults found during Phase "i" Functional Test	5	2	0

This profile is representative of ultra-reliable systems which typically will achieve high levels of test coverage at the end of functional testing and are also characterized by no-defects found during the last functional test phase. Let us also assume that r =.9 and γ=.25. Under those conditions the set of equations (4.7) becomes:

$$\tilde{a}K_1 = .1$$

$$(\tilde{a} - 4.375) \times K_2 = 0.028571$$

$$(\tilde{a} - 6.125) \times K_3 = 0$$

The set of equations contains four unknowns and as such can not be solved without the help of an additional equation. In particular one could use early prediction models to compute a value for ã as suggested in Gokhale [57]. The issue with using early prediction models is of course the large uncertainty in the estimate which can lead us to either overestimate or underestimate the number of faults. Note also that the last equation leads to a situation where one can possibly make two different conclusions. One is that $\tilde{a}$ = 6.125 (which we can interpret conservatively as being $\tilde{a}$ = 7) or K_3 = 0 (i.e. the tests are not able to trigger failures and correspondingly reveal faults).

In the phase-based functional test expression defined by equation (4.7), one should also note that:

$$\tilde{a} - r \times \left(\sum_{k=1}^{i-1} \Delta m_k(t_k)\right) + (1 - r) \times \gamma \times \left(\sum_{k=1}^{i-1} \Delta m_k(t_k)\right) \tag{4.8}$$

is the number of faults remaining after *i-1* phases of functional testing. For system *S* and *i - 1= n*, this number is given by $\tilde{a}$ - 6.125.

4.4. Extensions to Account for Auxiliary Observations and Continuously Refine the Predicted Fault Count

For a multi-phase testing process with *n* phases, equations (4.7) and (4.8) provide the number of failures experienced in each phase and the number of faults remaining at the end of a phase respectively. If predictions for the quantities, $\tilde{a}$, *r*, γ, K_i, $c_U(i)$, $c_L(i)$ are available from the onset of the testing process, one can derive predictions for the number of failures experienced and number of faults found for all phases of testing (i=1 to n) at the onset of phase 1. These early predictions may or may not be accurate. However, as the testing progresses information becomes available which will allow us to correct our predictions. The purpose of this section is to show which auxiliary observations become available during the multi-phase testing process and how these can be used to refine our predictions. It should be noted that an "observation" is defined as information that one can collect through one's experience. Mixing of model predictions and observations is referred to as model-data fusion (MDF). Model-data fusion approaches allow the use of observations which are representatives of the "real world" to update the initial prediction models which may be subjective and incomplete. The updated models should therefore have stronger prediction ability.

Current MDF research proposes four basic strategies to integrate available observations in the prediction process: sequential-intermittent assimilation (SIA), sequential-continuous assimilation (SCA), non sequential-intermittent assimilation (NSIA) and non sequential-continuous assimilation (NSCA) [90]. In order to continuously refine the predicted number of faults over multiple testing phases, sequential-intermittent assimilation (SIA) which considers observations made in a past period of time until the time of analysis, is applicable. The typical SIA framework is depicted in Figure 4.2.

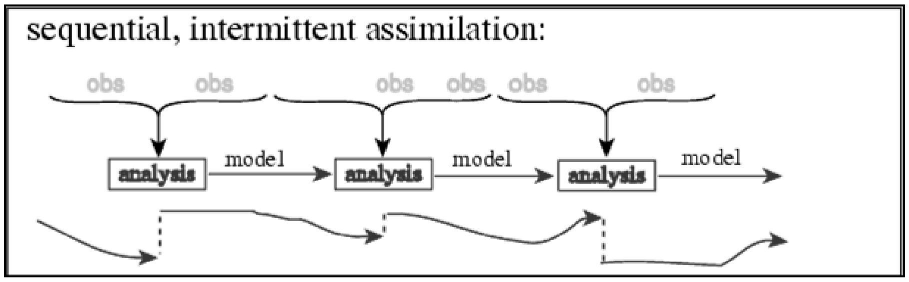


Figure 4.2 Sequential-intermittent Assimilation Framework, excerpted from [90]

Figure 4.3 depicts for a multi-phase testing process which follows the SIA framework illustrated in Figure 4.2: 1) the observations available during each testing phase, 2) the types of predictions one can make and 3) the time at which such predictions can be made as well as the parameters used for these predictions.

While the framework and corresponding derivations are applicable and can be extended to any number of functional testing phases, n, the discussion in this paper is limited to a multi-phase testing process where n is equal to two.

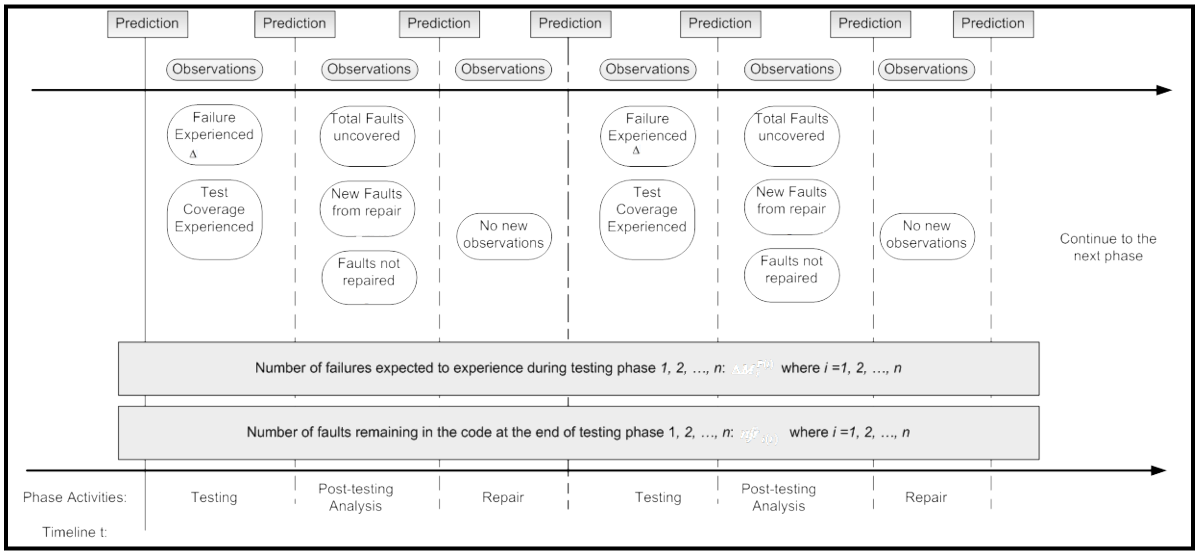

Prediction
Observations
Failure Experienced Δ
Test Coverage Experienced
Total Faults uncovered
New Faults from repair
Faults not repaired
No new observations
Continue to the next phase
Number of failures expected to experience during testing phase 1, 2, …, n: where i =1, 2, …, n
Number of faults remaining in the code at the end of testing phase 1, 2, …, n: where i =1, 2, …, n
Phase Activities:
Testing
Post-testing Analysis
Repair
Timeline t:

Figure 4.3 Predictions and Observations made throughout the testing phases

At the beginning of phase 1, i.e. $t = 0$, the only observations available pertain to the software development process. These can help us derive predictions for ã [57][20][75] as well as a first set of predictions for r [74] and γ [75]. These predictions are denoted as: $\tilde{a}^{P(0)}$, $r^{P(0)}$ and $\gamma^{P(0)}$ respectively where the superscript $P(0)$ denotes a prediction made at time "t = 0". From these predictions and equations (4.7) and (4.8) one derives the first set of predictions in Table 4.2 (i.e. predictions at t=0). In Table 4.2, $\Delta M_{1(2)}^{P(0)}$ stand for the number of failures we expect to experience during testing phase 1(2) predicted at time t=0 using equation (4.7), $nfr_{1(0)}$ (respectively $nfr_{2(0)}$) stand for predictions made at time t=0 of the number of faults remaining in the code at the end of testing phase 1 (respectively testing phase 2) obtained using equation (4.9). During phase 1 testing, i.e. $t \in (0, t_1)$, one observes 1) software failures through testing and the total number of failures observed is ΔM_1^E; 2) the actual test coverage $C_1^E(t)$ from which one can derive the upper and lower bounds of coverage achieved at the end of phase 1, $C_U^E(1)$ and $C_L^E(1)$. From those two

)

observations one derives the second set of predictions in Table 4.2 (i.e. predictions at $t=t_1$). Values of ΔM_1^E, $C_U^E(1)$, and $C_L^E(1)$ used in combination with predictions of the number of faults remaining at the end of a phase allow updating of K_1 and K_2 as will be explained in section 4.4.2.

One should note at this point that our MDF strategy further consists in updating parameters (such as r, γ, K_1, K_2) when observations are available using a Bayesian framework (see the following sections 4.4.1 and 4.4.2. for a discussion of the Bayesian updating methodology) and replacing unknowns such as $\Delta M_{1(2)}^{P(..)}$ by their observed value $\Delta M_{1(2)}^{E}$ when they become available (see Table 4.2). Choices such as these further characterize the MDF strategy followed beyond it being of the type SIA. For example, one could consider that observations may not truly reflect "reality" due to an imperfect "understanding" of the situation at hand and modify the MDF strategy accordingly by combining observations such as ΔM_1^E and predictions such as $\Delta M_{1(2)}^{P(..)}$ through a weighted scheme instead of replacing $\Delta M_{1(2)}^{P(..)}$ by ΔM_1^E when the latter becomes available.

During the following post-testing analysis (PTA) period where $t \in (t_1, t_1^*)$, one will identify the faults which correspond to the failures experienced in the phase 1 testing. Thus, the observation during PTA is the total number of uncovered faults $\Delta m_1^E(t_1^*)$. These faults were the result of software development errors. Amongst these faults, some may be the result of repairs which took place during the development process and were not carried out properly. These correspond to the additional observations: $\Delta m_{1\_nr}^E(t_1^*)$ are faults which were not repaired although they should have and $\Delta m^{E}_{1\_new}(t_1^*)$ are new faults introduced during the repair process. This information can be used to update the rates $r^{P(0)}$ and $\gamma^{P(0)}$ and leads to the third update of our predictions in Table 4.2 (i.e. predictions at $t = t_1*$). A repair process is conducted after faults have been identified. One will not observe the effect of such repair until the next testing phase. Therefore, no new observation is available during phase 1 repair (which occurs for $t \in (t_1^+, t_1^*)$). As such predictions remain identical (see Table 4.2, predictions at $t=t_1^*$ and $t= t_1^+$ are identical).

During the phase 2 testing, i.e. $t \in (t_1^+, t_2)$, one observes the total number of software failures ΔM_2^E and test coverage $C_2^E(t)$. The predictions are once again updated taking this information into consideration (see Table 4.2, predictions at $t=t_2$). During the next PTA (which occurs for $t \in (t_2, t_2^*)$), two new observations besides $\Delta m_2^E(t_2^*)$ are available and can be used to update the predictions. These observations are: 1) the number of new faults introduced into the software due to bad repairs, $\Delta m^{E}_{2\_new}(t_2^*)$, 2) the number of faults that have not been repaired, $\Delta m^{E}_{2\_nr}(t_2^*)$. This information can be used to update the rates $r^{P(t_1^*)}$ and $\gamma^{P(t_1^*)}$.

Table 4.2 provides the set of high level equations used to update the unknowns $\Delta M_{1(2)}$, $nfr_{1(0)}$ and $nfr_{2(0)}$. These equations involve parameters such as r, γ, K_1 and K_2 who are also updated periodically as observations pertinent to these quantities become available. We next examine a possible Bayesian updating framework for these parameters.

Table 4.2 Predictions made at different instants of time of a multi-phase testing process

Time of Prediction	**Prediction Equation**
$t = 0$	$\Delta M_1^{P(0)} = \Delta m_1^{P(0)}(\tilde{a}^{P(0)}, K_1^{P(0)}, r^{P(0)}, \gamma^{P(0)}, c_U^{P(0)}(1), c_L^{P(0)}(1))$ $\Delta M_2^{P(0)} = \Delta m_2^{P(0)}(\tilde{a}^{P(0)}, K_2^{P(0)}, r^{P(0)}, \gamma^{P(0)}, c_U^{P(0)}(2), c_L^{P(0)}(2), \Delta m_1^{P(0)}))$ $nfr_{1(0)} = \big(\tilde{a}^{P(0)} + (-r^{P(0)} + \big(1 - r^{P(0)}\big) \times \gamma^{P(0)})$ $\times \Delta m_1^{P(0)}(\tilde{a}^{P(0)}, K_1^{P(0)}, \square^{P(0)}, \gamma^{P(0)}, c_U^{P(0)}(1), c_L^{P(0)}(1)\big)$ $nfr_{2(0)} = \Bigg(\tilde{a}^{P(0)} + (-r^{P(0)} + \big(1 - r^{P(0)}\big) \times \gamma^{P(0)})$ $\times \begin{pmatrix} \Delta m_1^{P(0)}(\tilde{a}^{P(0)}, K_1^{P(0)}, r^{P(0)}, \gamma^{P(0)}, c_U^{P(0)}(1), c_L^{P(0)}(1)) \\ +\Delta m_2^{P(0)}(\tilde{a}^{P(0)}, K_2^{P(0)}, r^{P(0)}, \gamma^{P(0)}, c_U^{P(0)}(2), c_L^{P(0)}(2), \Delta m_1^{P(0)}) \end{pmatrix}\Bigg)$
$t = t_1$	ΔM_1^E $\Delta M_2^{P(t_1)} = \Delta m_2^{P(t_1)}(\tilde{a}^{P(0)}, K_2^{P(t_1)}, r^{P(0)}, \gamma^{P(0)}, c_U^{P(0)}(2), c_L^{P(0)}(2), \Delta m_1^{P(t_1)}))$ $nfr_{1(t_1)} = \big(\tilde{a}^{P(0)} + (-r^{P(0)} + \big(1 - r^{P(0)}\big) \times \gamma^{P(0)}) \times \Delta m_1^{P(t_1)}(\tilde{a}^{P(0)}, K_1^{P(t_1)}, r^{P(0)}, \gamma^{P(0)}, c_U^E(1), c_L^E(1)\big)$ $nfr_{2(t_1)} = \Bigg(\tilde{a}^{P(0)} + (-r^{P(0)} + \big(1 - r^{P(0)}\big) \times \gamma^{P(0)})$ $\times \begin{pmatrix} \Delta m_1^{P(t_1)}(\tilde{a}^{P(0)}, K_1^{P(t_1)}, r^{P(0)}, \gamma^{P(0)}, c_U^E(1), c_L^E(1)) \\ +\Delta m_2^{P(t_1)}(\tilde{a}^{P(0)}, K_2^{P(t_1)}, r^{P(0)}, \gamma^{P(0)}, c_U^{P(0)}(2), c_L^{P(0)}(2), \Delta m_1^{P(t_1)}) \end{pmatrix}\Bigg)$
$t = t_1^*$	ΔM_1^E $\Delta M_2^{P(t_1^*)} = \Delta m_2^{P(t_1^*)}(\tilde{a}^{P(0)}, K_2^{P(t_1^*)}, r^{P(t_1^*)}, \gamma^{P(t_1^*)}, c_U^{P(0)}(2), c_L^{P(0)}(2), \Delta m_1^E))$ $nfr_{1(t_1^*)} = \big(\tilde{a}^{P(0)} + (-r^{P(t_1^*)} + \big(1 - r^{P(t_1^*)}\big) \times \gamma^{P(t_1^*)}) \times \Delta m_1^E\big)$ $nfr_{2(t_1^*)} = \Big(\tilde{a}^{P(0)} + (-r^{P(t_1^*)} + \big(1 - r^{P(t_1^*)}\big) \times \gamma^{P(t_1^*)})$ $\times \Big(\Delta m_1^E + \Delta m_2^{P(t_1^*)}(\tilde{a}^{P(0)}, K_2^{P(t_1^*)}, r^{P(t_1^*)}, \gamma^{P(t_1^*)}, c_U^{P(0)}(2), c_L^{P(0)}(2), \Delta m_1^E)\Big)\Big)$
$t = t_1^+$	Identical to $t = t_1^*$
$t = t_2$	ΔM_1^E ΔM_2^E $nfr_{1(t_2)} = \big(\tilde{a}^{P(0)} + (-r^{P(t_1^*)} + \big(1 - r^{P(t_1^*)}\big) \times \gamma^{P(t_1^*)}) \times \Delta m_1^E\big)$ $nfr_{2(t_2)} = \Big(\tilde{a}^{P(0)} + (-r^{P(t_1^*)} + \big(1 - r^{P(t_1^*)}\big) \times \gamma^{P(t_1^*)})$ $\times \Big(\Delta m_1^E + \Delta m_2^{P(t_2)}(\tilde{a}^{P(0)}, K_2^{P(t_2)}, r^{P(t_1^*)}, \gamma^{P(t_1^*)}, c_U^E(2), c_L^E(2), \Delta m_1^E)\Big)\Big)$
$t = t_2^*$	ΔM_1^E ΔM_2^E $nfr_{1(t_2^*)} = \big(\tilde{a}^{P(0)} + (-r^{P(t_2^*)} + \big(1 - r^{P(t_2^*)}\big) \times \gamma^{P(t_2^*)}) \times \Delta m_1^E\big)$ $nfr_{2(t_2^*)} = \big(\tilde{a}^{P(0)} + \big(-r^{P(t_2^*)} + \big(1 - r^{P(t_2^*)}\big) \times \gamma^{(t_2^*)}\big) \times (\Delta m_1^E + \Delta m_2^E))$
$t = t_2^+$	Identical to $t = t_2^*$

4.4.1. *Updating of the Repair and Fault Introduction Rates*

Let us examine how updating of the repair and fault introduction rate may be performed using a Bayesian updating approach.

A Beta distribution $Beta(r;\alpha,\beta)=\frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}r^{\alpha-1}(1-r)^{\beta-1}$ is selected as prior distribution for the repair rate (r). The Beta distribution is an appropriate choice for the representation of quantities which take values on the interval [0, 1] like the repair rate and allows modeling of a large range of behaviors through the distribution's parameters α and β.

It will be initially assumed that no prior information exists on the repair rate, a situation which is represented by a uniform distribution, a Beta distribution with $\alpha = 1$ and $\beta = 1$. Different sources of evidence can then be used to update the prior: 1) Evidence on other software systems than the one currently tested (such as for instance Capers Jones' compendium of industry data which shows that the percentage of perfect code modifications (r_j) can vary from 40% to 99% [74] dependent upon the degree of formality of reviewing techniques and other factors), 2) Evidence related to development of the software system and which precedes the multi-phase testing if such evidence is available. These two sources of evidence can be used to define $r^{P(0)}$. They should be cast in terms of repair attempts and corresponding repair successes/failures experienced in order to allow updating of the initial prior, i.e. the uniform distribution. More specifically, the repair data needs to take the form "(k, n)" where n is the total number of repairs made and k is the number of imperfect repairs as a result of the repairs undertaken.

The likelihood function used in the Bayesian updating process should characterize the repair process and gives the likelihood of observing the evidence given the prior. The repair process can be seen as a succession of independent repair attempts. Each repair attempt can typically be described using the Binomial distribution $B(k, n/\square) = C_k^n r^{n-k}(1-r)^k$ which is the probability distribution of the number of "unsuccessful trials (here imperfect repair)" in n independent Bernoulli trials (n repairs), with the same probability of "failure", "*1-r*".

The posterior distribution obtained after each of the updating processes is also a Beta distribution with parameters α' and β' based on the fact that Beta, Binomial are conjugate pairs.

Since the number of repair attempts at the origin of Jones' data is unknown and this source of evidence should not be given more importance than the data collected on the system under study, the scaling method proposed in [91] is utilized. Firstly, this scaling method assigns the same number of observations (here repair attempts) to each source of evidence. Then, appropriate weights (w_{1r} for Jones' data and w_{2r} for auxiliary observations during development) are assigned to these two sources of evidence. Here, the number of observations is taken as n_D which is the number of repair attempts during the development process, k_D the number of failed repair attempts during the development process, and w_{2r} is larger than w_{1r} since development process data pertinent to the software under study should be given more importance than generic data related to industry averages.

Once testing is initiated further observations become available such as $\Delta m^{E}_{1\_nr}(t_1^*)$, $\Delta m^{E}_{1_{new}}(t_1^*)$, $\Delta m^{E}_{2\_nr}(t_2^*)$, $\Delta m^{E}_{2_{new}}(t_2^*)$ and Δm_1^E. The first four observations correspond to repair errors which can be traced either to development or to repair attempts at the end of phase 1 and can be lumped with the k_D failed repair attempts. As for Δm_1^E, it corresponds to new repair attempts at the end of phase 1 and can be lumped with the n_D repair attempts. Note that this particular choice, assumes that the repair process taking place during development and at the end of the testing process retains similar characteristics. Note also that the scaling is revised as the number of opportunities for repair errors grows from n_D to $n_D + \Delta m_1^E$. Table 4.3 describes the multiple updates discussed.

Table 4.3 Parameters of the Prior and Posterior Distributions for repair rate $r^{P(\dots)}$

	Initial Prior	**Evidence from Capers Jones**	**Posterior**
$1-r^{P(0)}$	Uniform Distribution	The percentage of perfect code modifications, r, from industrial data averages (i.e. Capers Jones) and k_D out of n_D repairs during the development process of the current software are imperfect	$a(0) = 1 + w_{1r} * (1-r_j) * n_D + w_{2r}\, k_D$ $\beta(0) = 1 + w_{1r} * r * n_D + w_{2r}(n_D - k_D)$
$1 - r^{P(t_1^*)}$	Beta Distribution for $1- r^{P(0)}$	$\Delta m_{1\_nr}^E(t_1^*)$ and $\Delta m_{1_{new}}^E(t_1^*)$	$a(t_1^*) = 1 + w_{1r} * (1- r_j) * n_D + w_{2r} *(k_D + \Delta m_{1_{new}}^E(t_1^*) + \Delta m_{1_{nr}}^E(t_1^*))$ $\beta(t_1^*) = 1 + w_1 * r_j * n_D + w_2 (n_D - k_D + \Delta m_{1_{new}}^E(t_1^*) + \Delta m_{1\_nr}^E(t_1^*)))$
$1 - r^{P(t_2^*)}$	Beta Distribution for $1- r^{P(t_1^*)}$	$\Delta m_{2\_nr}^E(t_2^*)$, $\Delta m_{2_{new}}^E(t_2^*)$ and $\Delta m_{\mathbf{1}}^{\boldsymbol{E}}$	$a(t_2^*) = 1 + w_{1r} * (1- r_j) *(n_D + \Delta m_{\mathbf{1}}^{\boldsymbol{E}}) + w_{2r} *(k_D + \Delta m_{1_{new}}^E(t_1^*) + \Delta m_{1\_nr}^E(t_1^*) + m_{2_{new}}^E(t_2^*) + \Delta m_{2\_nr}^E(t_2^*))$ $\beta(t_2^*) = 1 + w_{1r} * r_j *(n_D + \Delta m_{\mathbf{1}}^{\boldsymbol{E}})) + w_{2r}((n_D + \Delta m_{\mathbf{1}}^{\boldsymbol{E}}) - (k_D + \Delta m_{1_{new}}^E(t_1^*) + \Delta m_{1\_nr}^E(t_1^*) + \Delta m_{2_{new}}^E(t_2^*) + \Delta m_{2\_nr}^E(t_2^*)))$

The same Bayesian updating approach can be applied to the new fault introduction rate γ. Industry averages data on the fault introduction rate such as the one found in [75] and denoted γ_j can serve to initially update the uniform distribution.

This evidence is given a weight w_{1j}. Repair data collected during the development process "(v_D, k_D)" where v_D number of newly introduced faults due to imperfect repair as a result of a repair process will serve as second source of evidence and is assigned a weight $w_{2\gamma}$. Finally further evidence collected during the multi-phase functional testing process is also used. This evidence is also assigned weight $w_{2\gamma}$ (as it is assumed that repairs taking place during development process and at the end of each testing phase retain same characteristics. This assumption can be easily modified if need by adding supplementary weights). The evidence in question is: $\Delta m^{E}_{1\_nr}(t^*_1)$, $\Delta m^{E}_{1_{new}}(t^*_1)$, $\Delta m^{E}_{2_{new}}(t^*_2)$, $\Delta m^{E}_{2\_nr}(t^*_2)$. Table 4.4 shows the different updates for the parameters of the prior and posterior distributions for γ.

Table 4.4 Parameters of the Prior and Posterior Distributions for the new fault introduction rate $\gamma^{P(...)}$

	Initial Prior	**Evidence**	**Posterior**
$\gamma^{P(0)}$	Uniform Distribution	γ from Capers Jones and v_D out of k_D failed repairs during the development process led to the introduction of new faults	$a(0) = 1 + w_{1\gamma} * \gamma_j * k_D + w_{2\gamma} * v_D$ $\beta(0) = 1 + w_{1\gamma} *(1- \gamma_j)* k_D + w_{2\gamma} *(k_D - v_D)$
$\gamma^{P(t^*_1)}$	Beta Distribution for $\gamma^{P(0)}$	$\Delta m^{E}_{1\_nr}(t^*_1)$ and $\Delta m^{E}_{1_{new}}(t^*_1)$	$a(t_1^*) = 1 + w_{1\gamma} * \gamma_j *(k_D + \Delta m^{E}_{1_{new}}(t^*_1) + \Delta m^{E}_{1\_nr}(t^*_1)) + w_{2\gamma} *(v_D + \Delta m^{E}_{1_{new}}(t^*_1))$ $\beta(t_1^*) = 1 + w_{1\gamma} *(1- \gamma_j) *(k_D + \Delta m^{E}_{1_{new}}(t^*_1) + \Delta m^{E}_{1\_nr}(t^*_1)) + w_{2\gamma} *((k_D + \Delta m^{E}_{1_{new}}(t^*_1) + \Delta m^{E}_{1\_nr}(t^*_1)) - (v_D + \Delta m^{E}_{1_{new}}(t^*_1)))$
$\gamma^{P(t^*_2)}$	Beta Distribution for $\gamma^{P(t_1^*)}$	$\Delta m^{E}_{2_{new}}(t^*_2)$ and $\Delta m^{E}_{2\_nr}(t^*_2)$	$a(t_2^*) = 1 + w_{1\gamma} * \gamma_j *(k_D + \Delta m^{E}_{1_{new}}(t^*_1) + \Delta m^{E}_{1\_nr}(t^*_1) + \Delta m^{E}_{2_{new}}(t^*_2) + \Delta m^{E}_{2\_nr}(t^*_2)) + w_{2\gamma} *(v_D + \Delta m^{E}_{1_{new}}(t^*_1) + \Delta m^{E}_{2_{new}}(t^*_2))$ $\beta(t_2^*) = 1 + w_{1\gamma} *(1- \gamma_j) *(k_D + \Delta m^{E}_{1_{new}}(t^*_1) + \Delta m^{E}_{1\_nr}(t^*_1) + \Delta m^{E}_{2_{new}}(t^*_2) + \Delta m^{E}_{2\_nr}(t^*_2)) + w_{2\gamma} *((k_D + \Delta m^{E}_{1_{new}}(t^*_1) + \Delta m^{E}_{1\_nr}(t^*_1) + \Delta m^{E}_{2_{new}}(t^*_2) + \Delta m^{E}_{2\_nr}(t^*_2)) - (v_D + \Delta m^{E}_{1_{new}}(t^*_1) + \Delta m^{E}_{2_{new}}(t^*_2)))$

4.4.2. *Updating of the Fault Propagation Constant*

A fault propagation constant K_i (K_1 and K_2) is used within the multi-phase testing process to model the probability that when faults are covered they will lead to failures. This probability depends on the faults remaining in the application and on the test cases used. Values for K_i can be obtained experimentally using techniques such as fault seeding and determining a value of the propagation constants for the test cases under consideration or using the evidence obtained from the remaining faults and failures identified.

In

Table 4.2, we are concerned with: $K_1^{P(0)}$, $K_1^{P(t_1)}$, $K_2^{P(0)}$, $K_2^{P(t_1)}$, $K_2^{P(t_1^*)}$ and $K_2^{P(t_2)}$. Since the K values are probabilities, they can also be represented using a Beta prior and the likelihood function can be initially assumed Binomial. Each set of tests (within a phase) which covers a fault attempts to trigger a failure and this experiment can be seen (in first approximation) as independent of the triggering of another failure by another covered fault.

Initially, if we exclude an experimental a priori determination of K_1, no evidence is available and the prior is a Uniform distribution. Evidence becomes available when the application is tested in phase 1 under the form of ΔM_1^E. These failures are the direct result of triggering in average (and as a first approximation) ($\tilde{a}^{P(0)} * (c_U^E(1) - c_L^E(1))$ possible covered faults. Hence the update proposed in Table 4.5 for $K_1^{P(t_1)}$. The weight assigned to this evidence is w_{1K1}.

A similar reasoning is applied for K_2. At time t=0, if we again preclude a possible experimental determination of K_2, no evidence is available and the prior is a

Uniform distribution. Evidence on K_1 (ΔM_1^E which becomes available at the end of testing phase 1) is our first source of evidence. It provides indirect information on the fault propagation characteristics of the application under test. Hence we use it to update K_2 and obtain $K_2^{P(t_1)}$. The weight assigned to this evidence is w_{1K2}.

Table 4.5 Parameters of the Prior and Posterior Distributions for $K_1^{P(..)}$

Variable	**Initial Prior**	**Evidence**	**Posterior**
$K_1^{P(0)}$	Uniform	None	$a(0) = 1$ $\beta(0) = 1$
$K_1^{P(t_1)}$	$K_1^{P(0)}$	ΔM_1^E, $c_U^E(1)$, $c_L^E(1)$	$a(t_1) = 1 + w_{1k1}\ \Delta M_1^E$ $\beta(t_1) = 1 + w_{1k1}$* (ã$^{P(0)}$*$(c_U^E(1) - c_L^E(1))$-ΔM_1^E)

The scaling factor applied is $nfr_{1(t_1)} * (c_U^{P(0)}(2) - c_L^{P(0)}(2))$ since it represents the maximum number of failures which we will experience during phase 2 testing. At t= t_1*, nfr1 is updated and as such a new estimate of K_2 is produced , $K_2^{P(t_1^*)}$. During phase 2 testing, new evidence in the form of ΔM_2^E failures becomes available. The actual coverage $(c_U^E(2), c_L^E(2))$ is also known. K_2 is updated with this new evidence which is this time directly pertinent to the test cases K_2 characterizes and given a weight of w_{2K2} (where w_{2K2} is superior to w_{2K1}). Updates for K_2 are described in Table 4.6.

Parameters α() and β() in Table 4.5 and Table 4.6 are the parameters of the respective Beta distributions. It should be noted that since there is only one set of observations available for updating K_1, weight w_{1K1} should be equal to 1.

Table 4.6 Parameters of the Prior and Posterior Distributions for $K_2^{P(..)}$

Variable	Initial Prior	Evidence	Posterior
$K_2^{P(0)}$	Uniform Distribution	None	$\alpha(0) = 1$ $\beta(0) = 1$
$K_2^{P(t_1)}$	$K_2^{P(0)}$	$\Delta M_1^E, c_U^E(1), c_L^E(1)$	$a(t_1) = 1 + w_{1k2} * \Delta M_1^E * nfr1\ (t_1) * (c_U^{P(0)}(2) - c_L^{P(0)}(2))$ $\beta(t_1) = 1 + w_{1k2} * (\tilde{a}^{P(0)} * (c_U^E(1) - c_L^E(1)) - \Delta M_1^E))) * nfr\ 1\ (t_1) * (c_U^{P(0)}(2) - c_L^{P(0)}(2))$
$K_2^{P(t_1^*)}$ and $K_2^{P(t_1^+)}$	$K_2^{P(t_1)}$	Δm_1^E	$a(t_1^*) = 1 + w_{1k2} * \Delta M_1^E * nfr\ 1\ (t_1^*) * (c_U^{P(0)}(2) - c_L^{P(0)}(2))$ $\beta(t_1^*) = 1 + w_{1k2} * (\tilde{a}^{P(0)} * (c_U^E(1) - c_L^E(1)) - \Delta M_1^E))) * nfr\ 1\ (t_1^*) * (c_U^{P(0)}(2) - c_L^{P(0)}(2))$
$K_2^{P(t_2)}$	$K_2^{P(t_1^+)}$	$\Delta M_2^E, c_U^E(2), c_L^{\square}(2)$	$a(t_2) = 1 + (w_{1k2} * \Delta M_1^E + w_2 * \Delta M_2^E) * nfr\ 1\ (t_2) * (c_U^E(2) - c_L^E(2))$ $\beta(t_2) = 1 + (w_{1k2} * (\tilde{a}^{P(0)} * (c_U^E(1) - c_L^E(1)) - \Delta M_1^E)) + w_{2k2} * nfr\ 1\ (t_2) * (c_U^E(2) - c_L^E(2)) - \Delta M_2^E))) * nfr\ 1\ (t_2) * (c_U^E(2) - c_L^E(2))$

Example-

The equations developed in paragraphs 4.4.1 and 4.4.2 were applied to a hypothetical software system *SY*. The industry average data, weights, predicted values of test coverage made using a description of planned tests in the test plan and example observations are provided respectively in Table 4.7 - Table 4.10. Three cases are considered. The results obtained are given in Figure 4.4.

Table 4.7 Industry Average Data Used in the Analysis of Software System *SY*

Case #	r_j	γ_j
1-3	.9	.1

Table 4.8 Weights Used in the Analysis of Software System *SY*

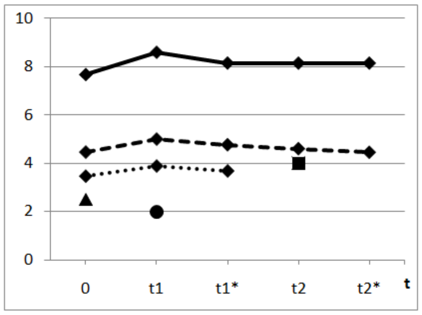
Case #	w_{1r}	w_{2r}	$w_{1\gamma}$	$w_{2\gamma}$	w_{1K1}	w_{1K2}	w_{2K2}
1-3	.25	.75	.25	.75	1	.25	.75

Table 4.9 Predictions Made for the Upper and Lower Bounds of Test Coverage Based on Information Available in the Test Plan

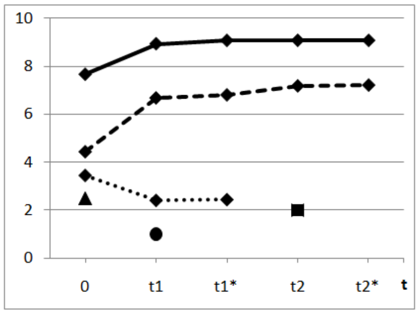
Case #	$c_L^{P(0)}(1)$	$c_U^{P(0)}(1)$	$c_L^{P(0)}(2)$	$c_U^{P(0)}(2)$
1-3	0	.5	0	.9

Table 4.10 Observations Used in the Analysis of Software System *SY*

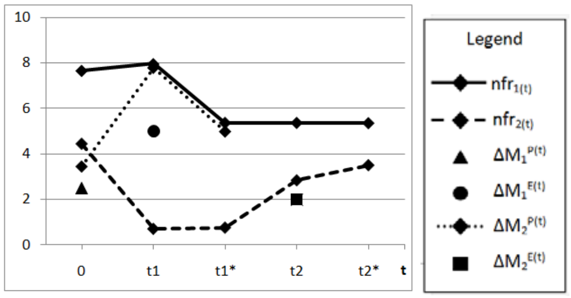
Case	$t \in (-\infty, 0)$			$t \in (0, t_1)$			$t \in (t_1, t_1^*)$			$t \in (t_1^+, t_2)$			$t \in (t_2, t_2^*)$		
#	n_d	k_d	v_d	ΔM_1^E	$c_L^E(1)$	$c_U^E(1)$	Δm_1^E	$\Delta m_{1\_nr}^E(t_1^*)$	$\Delta m_{1_{new}}^E(t_1^*)$	ΔM_2^E	$c_L^E(2)$	$c_U^E(2)$	Δm_2^E	$\Delta m_{2\_nr}^E(t_2^*)$	$\Delta m_{2_{new}}^E(t_2^*)$
1	100	3	0	2	0	0.4	2	1	0	4	0	0.8	4	0	0
2	100	3	0	1	0	0.4	1	1	0	2	0	0.8	2	0	0
3	100	3	0	5	0	0.4	5	1	0	2	0	0.8	2	0	0



(a) Case 1



(b) Case 2



(c) Case 3

Figure 4.4 $\Delta M_1^{P(t)}, \Delta M_2^{P(t)}, \Delta M_1^E, \Delta M_2^E, nfr\ 1(t)$and $nfr\ 2\ (t)$ for SY

The figures clearly demonstrate the impact of observations on the number of faults remaining in phases 1 and 2. Case 1 is a case where predicted behavior and observations match closely (i.e.ΔM_1^E and ΔM_1^P are close and so are ΔM_2^E and ΔM_2^P). Only slight corrections are brought to the parameters and the predictions at time 0 are close to the predictions at the end of phase 2. On the other hand Case 2 and Case 3 present situations that further and further deviate from the initial predictions. The figure shows how these deviations are accounted for and corrected.

4.5. Extensions in the Case of a Non-Uniform Distribution of Faults

We will now focus on another extension of interest which considers the fact that faults may not be distributed uniformly over the different fault locations. This is of particular importance in ultra-reliable systems because the location of a fault is a determinant factor in the severity of the associated failure. As such it is necessary if possible to locate the position of the remaining faults. We will establish the equations providing the number of failures experienced and the number of faults remaining for multiple test phases.

Our assumptions have now become: 1) Faults are not uniformly distributed over all potential fault sites. The probability that a fault resides in location L at time t during phase "i" is given by $f_i(L,t)$; 2) When a potential fault-site is covered, any fault present at that site is detected with probability $K_i(L,t)$ where "L" is the fault location and "i" is the functional test phase; 3) Repairs take place at the end of each phase. Repair activities are subject to errors. The probability of a perfect repair which eliminates the original fault and does not introduce any new faults is r_i for functional test phase "i". Different repair errors are considered: a) the fault is not corrected and remains in its initial location L, no new fault is introduced; b) a fault moves from its original location L to a new location L'; c) a fault remains in its original location L and a new fault is introduced in a new location L'. To express these different cases we introduce: γ_i the conditional probability that a new fault is introduced during functional test phase "*i*"; m_i the conditional probability that a fault changes location; $k_i(L,L')$ the conditional probability that given that a fault is introduced or is moved due to a repair at location *L*, it moves to *L'*; 4) Coverage is a continuous monotonic non-decreasing function of testing time per phase.

Let us introduce additional notations for the coverage function. We define by $c_i(L,t)$ the probability that a location is covered by time *t* in phase "*i*". $C_i(t)$ is the program coverage at time t during phase "*i*" and is given by:

$$\mathrm{C_i(t)} = \sum_{\mathrm{S}} \mathrm{c_i(L,t)} \qquad \text{for } \mathrm{t_{i-1}} \leq \mathrm{t} < t_i \tag{4.10}$$

where *S* is the set of software locations for software system *S*. From there, one obtains the number of failures experienced since the beginning of phase "*i*" as:

$$\Delta\mathrm{m_i}(\mathrm{t}+dt) = \Delta\mathrm{m_i(t)} + \sum_{\mathrm{S}} \left(\mathrm{c_i}(\mathrm{L,t}+dt) - \mathrm{c_i(L,t)}\right) \times \tilde{\mathrm{a}} \times \mathrm{f_i(L,t)} \times \mathrm{K_i(L,t)} \tag{4.11}$$

for $t_{i-1} \leq t < t_i$ and where ã as before is the number of faults in the code at time $t_0 = 0$. Dividing by *dt* and taking the limit for *dt* going to zero, we obtain:

$$\frac{d\Delta m_i(t)}{dt} = \sum_{\mathrm{S}} \frac{\partial \mathrm{c_i(L,t)}}{\partial \mathrm{t}} \times \tilde{\mathrm{a}} \times \mathrm{f_i(L,t)} \times \mathrm{K_i(L,t)} \tag{4.12}$$

for $t_{i-1} \leq t < t_i$. The number of failures experienced during a phase "i" is then given by:

$$\int_{t_{i-1}}^{t_i} \frac{d\Delta m_i(t)}{dt} dt = \int_{t_{i-1}}^{t_i} \sum_{\mathrm{S}} \frac{\partial \mathrm{c_i(L,t)}}{\partial \mathrm{t}} \times \tilde{\mathrm{a}} \times \mathrm{f_i(L,t)} \times \mathrm{K_i(L,t)}\, dt \tag{4.13}$$

The number of failures experienced due to a particular location over all *n* phases is:

$$\sum_{i=1}^{n} \int_{t_{i-1}}^{t_i} \frac{d\Delta m_i(L,t)}{dt} dt = \sum_{i=1}^{n} \int_{t_{i-1}}^{t_i} \frac{\partial \mathrm{c_i(L,t)}}{\partial \mathrm{t}} \times \tilde{\mathrm{a}} \times \mathrm{f_i(L,t)} \times \mathrm{K_i(L,t)}\, dt \tag{4.14}$$

The number of faults in location *L* changes as a function of detection and repair. During the phase, the fault count does not change but faults are uncovered. At the end of the phase, faults uncovered due to failures experienced are fixed. Let us denote by t_i+ the time at the end of phase "*i*" where repair is attempted, the probability that a fault still exists at time t_i+ in location *L* is given by the following equation:

$$\begin{aligned} f_i(L, t_i+) = {} & f_{i-1}(L, t_{i-1}+) - r_i \times \int_{t_{i-1}}^{t_i} \frac{\delta c_i(L,t)}{\delta t} \times f_{i-1}(L, t_{i-1}+) \times K_i(L,t)dt \\ & - (1 - r_i) \times (1 - \gamma_i) \\ & \times m_i \int_{t_{i-1}}^{t_i} \frac{\delta c_i(L,t)}{\delta t} \times f_{i-1}(L, t_{i-1}+) \times K_i(L,t)dt + (1 - r_i) \times \gamma_i \\ & \times \sum_{L' \neq L} \left[\int_{t_{i-1}}^{t_i} \frac{\delta c_i(L',t)}{\delta t} \times f_{i-1}(L', t_{i-1}+) \times K_i(L',t) \times k_i(L', L)dt \right] \\ & + (1 - r_i) \times (1 - \gamma_i) \times m_i \\ & \times \sum_{L' \neq L} \left[\int_{t_{i-1}}^{t_i} \frac{\delta c_i(L',t)}{\delta t} \times f_{i-1}(L', t_{i-1}+) \times K_i(L',t) \times k_i(L', L)dt \right] \end{aligned} \tag{4.15}$$

The first term corresponds to faults that were in *L* at the beginning of the phase, the second term corresponds to successful repair of faults detected through failures experienced, the third term corresponds to the attempted repair of a fault in *L* where the fault will be moved to some unknown location *L'*, the fourth term corresponds to the attempted repair of a fault at location *L'* resulting in a new fault introduced in location *L*, and the fifth term to unsuccessful repairs in location *L'* that resulted in the moving of the fault in *L*. The number of faults remaining is given by:

$$A(t_i+) = \sum_{S} \tilde{a}\, f_i(L, t_i+) \quad \text{with } A(t_0+) = \tilde{a}. \tag{4.16}$$

Example- Let us consider an example software *SX*. *SX* is undergoing two functional test phases. Let us also assume that the upper and lower coverage values for each functional test phase are given in Table 4.11. The case study considers 100,000 different locations. These could be different modules, lines of code, etc dependent upon the level of abstraction selected. Values of the additional parameters are given in Table 4.12 to Table 4.14. In this example we assume that $K_i(L,t)$ is dependent upon the phase and the location but not upon time.

Table 4.11 Multi-Phase Test Profile for Software *SX*

Phases	1	2
$C_L(i)$	0	0
$C_U(i)$	.67	1

Table 4.12 Detection probabilities per location for the two phases for Software *SX*

Case #	(a)	(b)
$K_1(L)$	0 for L= [12000, 21999] U [32000, 41999] U [82000, 94999] .5 for all other values of L	0 for L= [33000, 54999] U [82000, 92999] .5 for all other values of L
$K_2(L)$	.5 for all L in [0, 100000]	.5 for all L in [0,100000]

Table 4.13 $k_i(L',L)$ for the two phases (i = 1,2)

Case #	**For L'=1**	**For L'=2 to 99,999**	**For L'=100,000**
(a)	$k_i(L',L)=1$ if L=2 $k_i(L',L)=0$ otherwise	$k_i(L',L)=.5$ if L'=L-1 or L'=L+1 $k_i(L',L)=0$ otherwise	$k_i(L',L)=1$ if L=99999 $k_i(L',L)=0$ otherwise
(b)	$k_i(L',L)=1$ if L=2 $k_i(L',L)=0$ otherwise	$k_i(L',L)=1$ if L'=L-1 $k_i(L',L)=0$ otherwise	$k_i(L',L)=1$ if L=99999 $k_i(L',L)=0$ otherwise

Table 4.14 Remaining Parameters for *SX*

Case #	**(a)**				**(b)**				
	$r_1=r_2$	$\gamma_1=\gamma_2$	$m_1=m_2$	$\tilde{a}$	$r_1=r_2$	γ_1	γ_2	$m_1=m_2$	$\tilde{a}$
	.9	.25	.2	3	.9	.4	.3	.2	3

Results are given in Figure 4.5. The example shows the likely location of remaining defects and their distribution as well as the potential number of faults remaining.

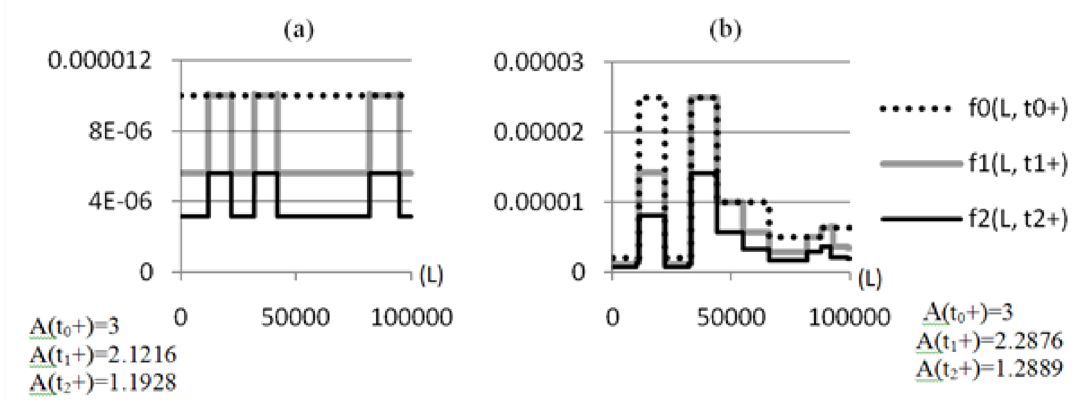


Figure 4.5 Fault Location Distributions for *SX*

4.6. Conclusions

This paper presents a new test coverage-based model which allows the description of ultra reliable software systems developed through multiple phases of functional testing. The paper establishes the equations governing the number of failures experienced and the number of faults remaining as a function of the multi-phase test coverage function.

This model is further extended: 1) to take advantage of auxiliary observations collected during the multi-phase testing and analysis process to refine the predictions made; 2) to describe software systems where either the initial fault distribution is non-uniform with respect to location, or the repair and test and detection process favor certain locations.

The first extension is based on a model-data fusion paradigm where parameters and unknowns are updated sequentially as and when information becomes available.

The example discussed in section 4.4 demonstrates that the model-data fusion framework proposed can be used to progressively and efficiently correct and refine the residual fault counts prediction if initial test-coverage based predictions deviate significantly from actual observations. The second extension of the model deals with potential fault location predictions. For ultra-reliable systems fault location information is of importance because the location of a fault determines how frequently it will be executed in operation and whether it will propagate. As such location is primary in determining fault propagation characteristics and whether the fault will have a large impact or not. The fault location distribution could be used in combination with mutation or modeling approaches to determine the fault propagation characteristics of the software in operation before it runs in the field. Such information is important for ultra-reliable systems which cannot be allowed to fail in the field and for which we will not be able to collect field data (failures in operation). Location information can also be used to refine testing and target it towards high impact high likelihood faults if those exist. In this paper the framework was applied to synthetic examples whose characteristics were chosen to be representative of real case studies. Application of the framework to a system with 100,000 locations shows that its computational complexity is limited, that results obtained for diverse sets of parameters display foreseeable trends, and that the tool developed can easily be expanded to handle more complex systems. Through simulations such as those presented in section 4.5, one can observe the effect of different test strategies, of initial fault distributions, of repair and new fault introduction rates, of the number of functional test phases and determine how these influence the final fault distributions. The knowledge gained can be used to optimize testing and improve reliability. Application of the models developed to an actual case study is not discussed in this paper and will be the object of future extensions. This will in particular entail selection of adequate approaches for parameter estimation. The parameters should be identifiable using a combination of methods and tools such as: code coverage tools for *c(L)*; early prediction methods [20] for $f_0(L)$; limited mutation for transfers of faults *k(L',L)* [25]; field data for repair rates, fault introduction rates and *K(L')* updated using information related to faults observed during the different phases.

Chapter 5: Summary and Future Research

5.1 Summary

This research focused on proposing a systematic software metric-based reliability prediction method. The method started with the measurement of a metric. Measurement results are then linked to three different types of defect characteristics: 1) the number of defects remaining or 2) the number and the exact locations of the defects found or 3) the number and the exact locations of defects found in an earlier version. Three models, Musa's exponential model, the PIE model and a mixed Musa-PIE model, are used to link the three categories of defect characteristics with reliability using the operational profile.

In order to implement the PIE model, an approach for construction of the EFSM model is presented. The EFSM is used in most of the top ranked software metrics studied in this thesis to identify the fault propagation rates of faults. The approach allows the mapping of the defects and the operational profile to the constructed EFSM model so that the execution of the updated EFSM model can be used to abstractly represent the faulty execution of the real software.

This software reliability prediction method is then applied to a safety-critical software used in the nuclear industry using eleven metrics. Reliability prediction results are compared with the real reliability assessed using operational failure data. Results show that reliability prediction based on DD, FDN, RT and TC are close to the operational reliability. Experiences and lessons learned from the application are discussed.

Possible extensions to the existing models as well as procedures for repeatable measurement and prediction are proposed.

The RePS built upon the test coverage measure provides credible prediction results and is refined to be able to take into consideration more realistic conditions, such as imperfect and non-uniformly distributed debugging and the use of multiple testing phases. More specifically, Chapter 3 introduces a first refinement for TC RePS. This refinement assesses the impact of newly introduced defects during the debugging process on reliability. The newly introduced defects could be located non-uniformly around the fault being fixed and they may possibly display different propagation characteristics than the faults being fixed. Chapter 4 describes a second refinement for TC RePS. This refinement allows the description of software systems developed through multiple phases of functional testing and takes advantage of auxiliary observations collected during the multi-phase testing and of the consequent process of analysis to refine the predictions made. This refinement also describes software systems where either the initial fault distribution is non-uniform with respect to location, or the repair/test and detection process favor certain locations.

5.2 Areas for Future Research

This section discusses the follow-on issues raised as a consequence of performing this study. The issues are listed as follows:

5.2.1 Defect Density Robustness

Defect density RePS is one of the best RePSs. The key step in this measurement is to identify defects in the products of each software development phase. That is, to reveal defects in the SRS, SDD and the code. The quality of results obtained using this RePS is a function of the inspector's detection efficiency. More specifically, the question is "What is the relationship between the ability of on inspector to detect a defect and the fault exposure probability of this defect?" Restated: "Is an inspector more likely to detect a defect with high exposure probability (probability of observing the failure is high) than with low exposure probability (probability of observing the failure is low) or reversely? Or is his/her detection ability independent of the fault exposure probability of that defect?" If the inspector mostly detects defects that have a small probability of occurrence then reliability assessments may be of low quality. If the inspector on the other hand detects defects that have a high likelihood of occurrence, then reliability estimation may be precise even if the defect detection efficiency is low.

5.2.2 Issues with Repeatability and Scalability

As has been clearly shown in section 2.7.4, the measurement process can be extremely time-consuming, error prone and highly dependent on the qualification of the inspectors involved. A considerable amount of time may be spent in manually "parsing" the natural language SRS, SDD or even the code and the number and type of defects found may depend heavily on the inspectors. Two solutions to these problems are possible: 1) Training and certification of inspectors; 2) Automation of the measurement process. Automation can help improve the repeatability of the measurement process while assisting the analyst and thereby increasing review speed. Future research should examine each of these avenues and how they should be implemented.

5.2.3 Issues with Common Cause Failures (CCF)

At this point, none of the RePSs considered include a measurement of common cause. This may lead to an underestimation of the probability of failure at the software system level since we currently assume independence between the versions. This underestimation may be of several orders of magnitude. For metrics such as CC, FP, BLOC and RSCR, a CCF correction factor will need to be investigated. This factor would represent the fraction of CCF which will be observed. For metrics such as DD and RT, the EFSM propagation technique will need to be modified to account for similar defects in multiple versions.

5.2.4 Issues with Uncertainty

Software reliability prediction is subject to uncertainty. The sources of uncertainties in software reliability prediction can generally be divided into two main categories: measurement uncertainty and model uncertainty. Measurement uncertainty can arise from inaccuracies in the methods and tools used to assess a quantity, from the artifact being measured, from the operator, and from other sources. Model uncertainty can stem from simplifications, assumptions and approximations, or from uncertainties in the values assumed by the model parameters. Further research is needed in the area of identifying the uncertainty components in a measurement process, estimating the total uncertainty, and reducing the degree of uncertainty.

5.2.5 Combining Measures

A future research project could determine how to down-select to a smaller number of measures that can be combined and yield a more accurate reliability estimation than that produced by any one measure taken in isolation.

Appendix A. Eleven Software Metrics used in this Study

1) ***Bugs per line of code (BLOC)***

The goal of this measure is to give a crude estimate of the number of faults in a program module per line of code.

2) ***Cause & effect graphing (CEG)***

A CEG is a formal translation of natural-language specification into its input conditions and expected outputs. The graph depicts a combinatorial logic network. CEG aids in identifying requirements that are incomplete and ambiguous. This measure explores the inputs and expected outputs of a program and identifies the ambiguities. Once these ambiguities are eliminated, the specifications are considered complete and consistent. The measure is computed as follows:

$$CE\% = 100 \times \left(1 - \frac{A_{existing}}{A_{tot}}\right)$$

Where:

$A_{existing}$ is the number of ambiguities in a program remaining to be eliminated and

A_{tot} is the total number of ambiguities identified.

3) ***Software capability maturity model (CMM)***

CMM is a framework that describes the key elements of an effective software process. The goal of this measure is to describe the principles and practices underlying software-process maturity and to help software organizations improve the maturity of their software processes [16]:

$$CMM = i \quad i \in \{1, 2, 3, 4, 5\}$$

4) ***Completeness (COM)***

The COM measure determines the completeness of the software requirements specifications (SRS). This measure provides a systematic guideline to identify the incompleteness defects in SRS. COM is the weighted sum of ten derivatives, D_1 through D_{10} [16] [92]:

$$COM = \sum_{i=1}^{10} w_i D_i$$

Where:

COM is the completeness measure,

w_i is the weight of the ith derived measure,

D_i is the i^{th} derived measure calculated from the primitive measures B_i ($i = 1,\ldots,18$).

5) ***Cyclomatic complexity (CC)***

This measure determines the structural complexity of a coded module. The Cyclomatic Complexity (CC) of a module is the number of linearly independent paths through a module. The cyclomatic complexity for the i^{th} module is originally defined by McCabe [93] [94] as:

$$CC_i = E_i - N_i + 1$$

Where:

CC_i : is the cyclomatic complexity measure of the ith module,

E_i : is the number of edges of the i^{th} module (program flows between nodes)

N_i : is the number of nodes of the i^{th} module (sequential groups of program statements).

6) ***Defect density (DD)***

Defect density is defined as the number of defects remaining divided by the number of lines of code in the software. The defects are discovered by independent inspection. Defect Density is given as:

$$DD = \frac{\sum_{i=1}^{I} D_i}{KS□OC}$$

Where:

D_i : is the number of unique defects detected during the design and code inspection and still remain in the code.

KSLOC: is the number of source lines of code (LOC) in thousands.

7) ***Fault days number (FDN)***

This measure represents the number of days that faults remain in the software system from introduction to removal. The fault day measure evaluates the number of days between the time a fault is introduced into a system and until the point the fault is detected and removed [18] [95], such that:

$$FD_i = fout_i - fin_i \text{ and } FD = \sum_{i=1}^{I} FD_i$$

Where:

FD : Fault-days for the total system;

FD_i : Fault-days for the ith fault;

fin_i : Date at which the ith fault was introduced into the system;

$fout_i$: Date at which the ith fault was removed from the system;

8) ***Function point analysis (FP)***

Function Point is a measure designed to determine the functional size of the software. The Function Point Counting Practices Manual is the definitive description of the Function Pointing Counting Standard. The latest version is Release 4.2, which was published in 2004 [96].

9) ***Requirement specification change request (RSCR)***

RSCR is defined as the number of change requests that are made to the requirements specification. This measure indicates the stability and/or growth of the functional requirements. The requested changes are counted from the first release of the requirements specification document to the time when the product begins its operational life.

$$RSCR = \Sigma(\textit{requested changes to the requirements specification})$$

Where:

The summation is taken all requirements change requests initiated during the software development life cycle.

10) ***Requirements traceability (RT)***

According to IEEE [16], the requirements traceability measure aids in identifying requirements that are either missing from, or in addition to, the original requirements. Requirements traceability is defined as:

$$RT = \frac{R_1}{R_2} \times 100\%$$

Where:

RT is the value of the measure requirements traceability,

R_1 is the number of requirements met by architecture, and

R_2 is the number of original requirements.

11) ***Test coverage (TC)***

As in IEEE [16], Test coverage (TC) is the percentage of requirement primitives implemented multiplied by the percentage of primitives executed during a set of tests. A simple interpretation of test coverage can be expressed by the following formula:

$$TC\% = \left(\frac{IC}{RC}\right) \times \left(\frac{PRT}{TPP}\right) \times 100$$

Where:

IC is the implemented capabilities;

RC is the required capabilities;

PPT is the tested program primitives and

TPP is the total program primitives.

Appendix B. The M-D Models for each of the Eleven RePSs

1) ***Bugs per Line of code (BLOC)***

Gaffney [28] established that the number of defects remaining in the software (N_G) could be expressed empirically as a function of the number of line of codes:

$$N_G = \sum_{i=1}^{M} \left(4.2 + 0.0015\sqrt[3]{S_i{}^4}\right)$$

Where:

i is the module index,

M is the number of modules, and

S_i is the number of lines of code for the ith module.

The next step is the partitioning of the defects based on their criticality. Using Table B.1 from [45] for US Averages percentages for delivered defects by severity level and logarithmic interpolation, the percentages of delivered defects by severity level can be obtained.

Table B.1 Percentages for Delivered Defects by Severity Level

	Severity 1 (critical)	**Severity 2 (significant)**	**Severity 3 (minor)**	**Severity 4 (cosmetic)**
Percentage of delivered defects	0.0185	0.1206	0.3783	0.4826

So the number of delivered defects of interest (N) can be obtained as:

$$N = N_G \times SL$$

Where:

SL is the percentage of defects introduced at the severity level of interest with the value of 0.1391 (0.0185+0.1206).

2) ***Cause & effect graphing (CEG)***

Defects are uncovered during the inspection of the SRS using measurement rules for CEG. All the defects identified through inspection along with their descriptions, their locations and their types are recorded. Detailed measurements rules are provided in [20].

3) ***Software capability maturity model (CMM)***

Historical industry data collected by Software Productivity Research Inc [40] links the CMM level to the number of defects per function points. Table B.2 presents this data.

Table B.2 CMM Levels and Average Number of Defects per Function Point

CMM level	Average Defects/Function Point
Defects for SEI CMM level 1	0.75
Defects for SEI CMM level 2	0.44
Defects for SEI CMM level 3	0.27
Defects for SEI CMM level 4	0.14
Defects for SEI CMM level 5	0.05

Using Table B.1 from [45] for US Averages percentages for delivered defects by severity level and logarithmic interpolation, the percentages of delivered defects by severity level can be obtained.

Therefore, the number of defects based on the measurement of CMM can be expressed as:

$$N = D_{CMM} \times SL$$

Where:

D_{CMM} is the total number of defects for a certain CMM level;

SL is the percentage of defects introduced at the severity level of interest with the value of 0.1391 (0.0185+0.1206).

4) ***Completeness (COM)***

Defects are uncovered during the inspection of the SRS using measurement rules for COM. All the defects identified through inspection along with their descriptions, their locations and their types are recorded. Detailed measurements rules are provided in [20].

5) ***Cyclomatic complexity (CC)***

An empirical correlation was derived to relate cyclomatic complexity and number of defects. The correlation is based on an experimental data set composed of system software applications developed by graduate students The empirical correlation was established by following the form proposed in [97]:

$$SLI_{CC} = 1 - \sum_{i=1}^{9} f_i \times p_i\%$$

Where:

SLI_{CC}: The SLI[26] value of the cyclomatic complexity factor

f_i: Failure likelihood f_i used for SLI_1 calculations [97]

p_i: The percentage of modules whose cyclomatic complexity belong to the ith level, $i = 1, 2, ..., 9$.

The number of defects predicted based on CC measurement results is:

[26] SLI stands for Success Likelihood Index which is used to represent the likelihood of an error occurring in a particular situation depends on the combined effects of a relatively small number of performance influencing factors (PIFs). SLI was used as an index which quantifies whether a particular environment will increase the human error probability or decrease it (with respect to a "normal situation") [44]. SLI_{CC} is related to the likelihood that developers will err (i.e. introduce fault in the software product and/or fail to remove them) because of the cyclomatic complexity of the modules.

$$N = 0.036 \times SIZE \times (20)^{1-2SLI_{cc}}$$

where

SIZE : the size of the delivered source code in terms of LOC (Line of Code)

6) ***Defect density (DD)***

Defects are uncovered during the inspection of the SRS, SDD and code using measurement rules for DD. All the defects identified through inspection along with their descriptions, their locations and their types are recorded. Detailed measurements rules are provided in [20].

7) ***Fault days number (FDN)***

Based on the cumulative characteristic of the FDN metric and by using the concepts introduced in [44], one can show that FDN is related to $\mu_U(t)$ by the following equation:

$$\frac{d(FDN)}{dt} = \mu_U(t)$$

Where:

$\mu_U(t)$: expected fault count at time t;

This equation shows the direct relationship between the measured real FDN and the corresponding fault count. The number of faults can be obtained using this equation once FDN is known (i.e. measured).. However, the real FDN can not be obtained experimentally since not all the faults can be discovered during the inspection. One can only obtain the apparent FDN, FDN_A which corresponds to faults identified through the inspection process and removed through repair. One can relate FDN_A to FDN by:

$$\frac{d(FDN_A)}{dt} = \gamma(t; \vartheta\mu_H, z_a, \mu_R) \cdot \frac{d(FDN)}{dt}$$

Where:

$\gamma(t; \vartheta\mu_H, z_a, \mu_R,)$ is a function of $\vartheta\mu_H$, z_a, μ_R, relates *FDN_A* to *FDN;*

$\vartheta\ \mu_H$: estimate of fault introduction rate;

Z_a: intensity function of per-fault detection;

μ_R: expected change in fault count due to each repair.

Therefore, one can still obtain the fault count based on the measured apparent FDN as shown by:

$$\mu_U(t) = \frac{d(FDN_A)}{dt} \cdot \frac{1}{\gamma(t; \vartheta\mu_H, z_a, \mu_R)}$$

8) Function point analysis (FP):

Jones' empirical industry data [45] links the FP to the number of defects per function point for different categories of applications. Table B.3 (Table 3.46 in [45]) provides the average numbers for delivered defects per function point for different types of software systems. Logarithmic interpolation can then be used for number of defects quantification.

Table B.3 Averages for Delivered Defects per Function Point (Extracted From [45])

Function Points	End user	MIS	Outsource	Commercial	Systems	Military	Average
1	0.05	0	0	0	0	0	0.01
10	0.25	0.1	0.02	0.05	0.02	0.03	0.07
100	1.05	0.4	0.18	0.2	0.1	0.22	0.39
1000	0	0.85	0.59	0.4	0.36	0.47	0.56
10000	0	1.5	0.83	0.6	0.49	0.68	0.84
100000	0	2.54	1.3	0.9	0.8	0.94	1.33
Average	0.45	0.9	0.48	0.36	0.29	0.39	0.53

Therefore, the number of defects based on the measurement of FP can be expressed as:

$$N = D_{FP} \times SL$$

Where:

D_{FP} is the total number of defects obtained from Jones' data for a specific type of software system;

SL is the percentage of defects introduced at the severity level of interest with the value of 0.1391 (0.0185+0.1206).

9) ***Requirement specification change request (RSCR)***

To link requirements specification change requests to the reliability of a software system, a derived measure called REVL was used. The measure is:

$$REVL = \frac{SIZE_{changed\ due\ to\ RSCR}}{SIZE_{delivered}} \times 100\%$$

where

$REVL$: measure of requirements evolution and volatility;

$SIZE_{\text{changed due to RSCR}}$: size of changed source code corresponding to RSCR, in Kilo Line of Code (KLOC);

$SIZE_{\text{delivered}}$: size of the delivered source code, in *KLOC*.

SLI for the REVL, denoted by SLI_{RSCR}, is estimated using the value of REVL, as shown in Table B.4.

Table B.4 Rating Scale and SLI Estimation for REVL

REVL	5%	20%	35%	50%	65%	80%
Rating Levels	Very low	Low	Nominal	High	Very High	Extra High
SLI_{RSCR}	1	0.75	0.5	0.34	0.16	0

The number of defects predicted based on RSCR measurement results is:

$$N = 0.036 \times SIZE_{delivered} \times (20)^{1-2SLI_{RSCR}}$$

10) ***Requirements traceability (RT)***

Defects are uncovered during the inspection of the SRS, SDD and code using measurements rules for RT. All the defects identified through inspection along with their descriptions, their locations and their types are recorded. Detailed measurements rules are provided in [20].

11) ***Test coverage (TC)***

Malaiya et al. investigated the relationship between defect coverage, C_0, and statement coverage, C_1. In [46], the following relationship was proposed:

$$C_0 = a_0 ln(1 + a_1 e^{a_2 C_1 - 1})$$

where

a_0, a_1, a_2 are coefficients, and

C_1 is the statement coverage.

The number of defects remaining in the software N is:

$$N = \frac{N_0}{C_0}$$

where

N_0 is the number of defects found by test cases provided in the test plan,

C_0 is the defect coverage.

Bibliography

[1] NASA, "Standard for Software Assurance," NASA STD 8739.8, 2004.

[2] RTCA, "Software Considerations in Airborne Systems and Equipment Certification," RTCA RTCA/DO-178, 1992.

[3] J. D. Musa, *Software Reliability: Measurement, Prediction, Application*. New York, USA: McGraw-Hill, 1990.

[4] M. R. Lyu, Ed., *Handbook of Software Reliability Engineering*. New York: McGraw-Hill, 1996.

[5] IEEE, "IEEE standard glossary of software engineering terminology 610.12-1990," IEEE Computer Society, 1990.

[6] 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, p. 10, Jan. 1993.

[7] N. Fenton, et al., "Assessing dependability of safety critical systems using diverse evidence," *IEEE Proceedings Software Engineering*, vol. 145, no. 1, pp. 35-39, 1998.

[8] N. Schneidewind, "Reliability Modeling for Safety-critical Software," *IEEE Transactions on Reliability*, vol. 46, 1997.

[9] K. W. Miller, et al., "Estimating the Probability of Failure When Testing Reveals No Failures," *IEEE Transactions on Software Engineering*, vol. 18, no. 1, pp.

33-43, Jan. 1992.

[10] L. M. Kaufman, J. B. Dugan, and B. W. Johnson, "Using Statistics of the Extremes for Software Reliability Analysis of Safety Critical Systems," in *Proceedings of The Ninth International Symposium on Software Reliability Engineering*, Paderborn, Germany, 1998, p. 355.

[11] M. G. Thomason and J. A. Whittaker, "Properties of Rare Fail-States and Extreme Values of TTF in a Markov Chain Model for Software Reliabilit," University of Tennessee, 1999.

[12] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, 2nd ed. New York: Int'l Thomson Computer Press, 1997.

[13] J. C. Munson and T. Khoshgoftaar, "The Detection of Fault-Prone Program," *IEEE Transactions on Software Engineering*, vol. 18, no. 5, pp. 423-433, 1992.

[14] X. Zhang and H. Pham, "An Analysis of Factors Affecting Software Reliability," *The J. Systems and Software*, vol. 50, p. 14, 2000.

[15] J. D. Lawrence, A. S. W. L. Person, and G. L. Johnson, "Assessment of Software Reliability Measurement Methods for Use in Probabilistics Risk Assessment," Lawrence Livermore Nat'l Laboratory FESSP, 1998.

[16] IEEE, "IEEE Standard Dictionary of Measures to Produce Reliable Software 982.1," IEEE Computer Society 982.1, 1988.

[17] M. Li and C. Smidts, "A Ranking of Software Engineering Measures Based on Expert Opinion," *IEEE Transactions on Software Engineering*, vol. 29, no. 9, pp. 811-824, Sep. 2003.

[18] C. Smidts and M. Li, "Software Engineering Measures for Predicting Software Reliability in Safety Critical Digital Systems," USNRC NUREG/GR-0019, 2000.

[19] J. M. Voas, "PIE: A Dynamic Failure-Based Technique," *IEEE Transactions on Software Engineering*, vol. 18, pp. 717-727, 1992.

[20] C. S. Smidts, M. Li, Y. Shi, W. Kong, and J. Dai, "A Large Scale Validation of A Methodology for Assessing Software Quality (under revision)," University of Maryland-College Park, 2009.

[21] Y. Shi, W. Kong, and C. Smidts, "Data Collection and Analysis for the Reliability Prediction and Estimation of a Safety Critical System," in *Reliability Analysis of System Failure Data*, Cambridge, UK, 2007.

[22] W. Kong, Y. Shi, and C. Smidts, "Early Software Reliability Prediction Using Cause-Effect Graphing Analysis," in *Annual Reliability and Maintainability Symposium (RAMS)*, Orlando, Florida, 2007.

[23] Y. Shi, M. C. Kim, and C. Smidts, "Lesson Learnt from the Application of Test Coverage RePS," in *the 6th American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Controls, and Human Machine Interface Technology*, Knoxville, Tennessee, 2009.

[24] Y. Shi, M. Li, and C. Smidts, "On the Use of Extended Finite State Machine Models for Software Fault Propagation and Software Reliability Estimation," in *the 6th American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Controls, and Human Machine Interface Technology*,

Knoxville, Tennessee, 2009.

[25] Y. Shi, W. Kong, J. Dai, and C. Smidts, "A Reliability Prediction Method for Safety Critical Systems Based on Test Coverage," in *in The 3rd International Conference on Reliability and Safety Engineering*, Kharagpur, India, 2007.

[26] Y. Shi and C. Smidts, "Predicting the Types and Locations of Faults Introduced During an Imperfect Repair Process and their Impact on Reliability," *International Journal of System Assurance Engineering and Management*, vol. 1, no. 1, Mar. 2010.

[27] C. Smidts and Y. Shi, "A Test Coverage-Based Model for Predicting Software Fault Content and Location," in *Advanced Technologies for Software Reliability and Safety*, Jeju Island, Korea, 2009.

[28] J. E. Gaffney, "Estimating the Number of Faults in Code," *IEEE Transactions on Software Engineering*, vol. 10, pp. 459-464, 1984.

[29] C. J. Wang and M. T. Liu, "Generating Test Cases for EFSM with Given Fault Models," in *12th Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE INFOCOM '93)*, San Francisco, CA, 1993.

[30] J. Musa, "The operational profile in software reliability engineering: an overview," in *Third International Symposium on Software Reliability Engineering*, 1992.

[31] R. V. Sandfoss and S. A. Meyer, "Input Requirements needed to Produce an Operational Profile for a New Telecommunications System," in *The Eighth International Symposium On Software Reliability Engineering*, 1997.

[32] "IEEE recommended practice for software requirements specifications," IEEE IEEE Std 830, 1998.

[33] "PACS Requirements Specification," West Virginia University, 1998.

[34] M. E. Fagan, "Design and Code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182-211, 1976.

[35] B. Beizer, *Software Testing Techniques*, 2nd ed. Van Nostrand Reinhold, 1990.

[36] C. Garrett, S. Guarro, and G. Apostolakis, "Dynamic Flowgraph Methodology for Assessing the Dependability of Embedded Software Systems," *IEEE Transactions on Systems, Man and Cybernetics*, 1995.

[37] "TestMaster Users' Manual," Teradyne Software & Systems Test, 2000.

[38] M. Li, et al., "Validation of a Methodology for Assessing Software Reliability," in *Proceeding of The 15th International Symposium on Software Reliability Engineering*, Saint-Malo, France, 2004, pp. 66-76.

[39] C. S. Smidts and M. Li, "Preliminary Validation of a Methodology for Assessing Software Quality," U.S. Nuclear Regulatory Commission NUREG/CR-6848, 2004.

[40] C. Jones, "Measuring Global Software Quality," Software Productivity Research, 1995.

[41] D. E. Embrey, "The Use of Performance Shaping Factors and Quantified Expert Judgement in the Evaluation of Human Reliability: An Initial Appraisal," U.S. Nuclear Regulatory Commission NUREG/CR-2986, 1983.

[42] E. M. Dougherty and J. R. Fragola, *Human Reliability Analysis: A System*

Engineering Approach with Nuclear Power Plant Applications. John Wiley & Sons, 1988.

[43] J. Reason, *Human Error*. Cambridge University Press, 1990.

[44] M. A. Stutzke and C. S. Smidts, "A Stochastic Model of Fault Introduction and Removal During Software Development," *IEEE Transactions on Reliability Engineering*, vol. 50, no. 2, 2001.

[45] C. Jones, *Applied Software Measurement: Assuring Productivity and Quality*, 2nd ed. New York: McGraw-Hill, 1996.

[46] Y. K. Malaiya, N. Li, J. M. Bieman, R. Karcich, and B. Skibbe, "The relationship between test coverage and reliability," in *Proceeding of The 5th International Symposium on Software Reliability Engineering*, Los Alamitos, 1994, pp. 186-195.

[47] W. G. Ireson, *Reliability Handbook*. McGraw Hill, 1966.

[48] E. L. Welker and M. Lipow, "Estimating the exponential failure rate from data with no failure events," in *the 1974 Annual Reliability and Maintainability Conference*, New York, 1974.

[49] "Electronic Parts Reliability Data," Reliability Analysis Center EPRD-95, 1995.

[50] "Reliability Prediction of Electronic Equipment," Department of Defense Military Handbook 217FN2, 1995.

[51] J. P. Poloski, D.G.Marksberry, C. L. Atwood, and W.J.Galyean, "Rates of Initiating Events at U.S. Nuclear Power Plants: 1987-1995," Nuclear Regulatory Commission NUREG/CR-5750, 1998.

[52] M. A. Vouk, "Using reliability models during testing with non-operational profile," in *in Second Bellcore/Purdue Symposium on Issues in software reliability estimation*, 1993, pp. 103-110.

[53] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set size and block coverage on fault detection effectiveness," in *The Fifth IEEE International Symposium on Software Reliability Engineering*, Monterey, CA, 1994, pp. 230-238.

[54] F. D. Frate, P. Garg, A. Mathur, and A. Pasquini, "On the correlation between code coverage and software reliability," in *the Sixth IEEE International Symposium on Software Reliability Engineering*, Toulouse, France, 1995, pp. 124-132.

[55] P. Piwowarski, M. Ohba, and J. Caruso, "Coverage measurement experience during function test," in *Proceeding of The 15th International Conference on Software Engineering*, Baltimore, MD, 1993, pp. 287-301.

[56] M. H. Chen, M. R. Lyu, and W. E. Wong, "An Empirical Study of the Correlation between Code Coverage and Reliability Estimation," in *The 3rd International Software Metrics Symposium*, Los Alamitos, CA, 1996, pp. 133-141.

[57] S. S. Gokhale and K. S. Trivedi, "A Time/Structure Based Software Reliability Model," *Annals of Software Engineering*, vol. 8, pp. 85--121, 1999.

[58] S. S. Gokhale, T. Philip, P. N. Marinos, and K. S. Trivedi, "Unification of finite failure non-homogeneous Poisson process models through test coverage," in

Proceedings of the International Symposium on Software Reliability Engineering, White Plains, NY, 1996, pp. 299-307.

[59] H. Pham and X. Zhang, "NHPP software reliability and cost models with testing coverage," *European Journal of Operational Research*, vol. 145, no. 2, pp. 443-454, Mar. 2003.

[60] X. Cai and M. R. Lyu, "Software reliability modeling with test coverage: Experimentation and measurement with A fault-tolerant software project," in *Proceedings of the 18th IEEE International Symposium on Software Reliability Engineering*, Trollhättan, Sweden, 2007, pp. 17-26.

[61] USNRC, "Guidance on Software Reviews for Digital I &C Systems," NUREG 0800 BTP 7-14, 2007.

[62] Z. Jelinski and P. B. Moranda, "Software reliability research," *Statistical Computer Performance Evaluation*, 1972.

[63] A. L. Goel and T. Okumoto, "Time-dependent error detection rate model for software reliability and other performances measures," *IEEE Transactions on Reliability*, vol. 28, pp. 206-211, 1979.

[64] A. L. Goel, "Software reliability models: assumptions, limitations and applicability," *IEEE Transactions on Reliability*, vol. SE-II, pp. 1411-1423, 1985.

[65] M. Ohba and X. Chou, "Does Imperfect Debugging Affect Software Reliability Growth?," in *The 11th International Conference on Software Engineering*, Pittsburg, PA, 1989, pp. 237-244.

[66] B. Littlewood, "Stochastic reliability-growth: A model for fault removal in computer programs and hardware designs," *IEEE Transactions on Reliability*, vol. 30, no. 4, pp. 313-320, 1981.

[67] S. Yamada, K. Tokuno, and S. Osaki, "Imperfect debugging models with fault introduction rate for software reliability assessment," *International Journal of Systems Science*, vol. 23, no. 12, 1992.

[68] P. Zeephongsekul, G. Xia, and S. Kumar, "Software-Reliability Growth Model: primary-Failures Generate Secondary-Faults Under Imperfect Debugging," *IEEE Transactions on Reliability*, vol. 43, no. 3, pp. 408-413, Sep. 1994.

[69] P.K.Kapur and S. Younes, "Modelling an Imperfect Debugging Phenomenon in Software Reliability," *Microelectron Reliability*, vol. 36, no. 5, pp. 645-650, 1996.

[70] H. Pham, L. Nordman, and X. Zhang, "A General Imperfect-Software-Debugging Model with S-Shaped Fault-Detection Rate," *IEEE Transactions on Reliability*, vol. 48, no. 2, pp. 169-175, Jun. 1999.

[71] S. S. Gokhale, "Software Failure Rate and Reliability Incorporating Repair Policies," in *Proceeding of the 10th International Symposium on Software Metrics*, 2004.

[72] S. S. Gokhale, T. Philip, and P. N. Marinos, "Non-homogeneous Markov software reliability model with imperfect repair," in *Proceedings -IEEE International Computer Performance and Dependability Symposium*, 1996, pp. 262-270.

[73] Y. Levendel, "Reliability Analysis of Large Software Systems: Defect Data Modeling," *IEEE Transactions on Reliability*, vol. 16, no. 2, pp. 141-152, Feb. 1990.

[74] C. Jones, "Software defect-removal efficiency," *Computer*, vol. 29, pp. 94-95, 1996.

[75] C. Jones, "Software Engineering: The State of the Art in 2008," Software Productivity Research LLC, 2008.

[76] P. Zeephongsekul, "Reliability Growth of a Software Model under Imperfec Debugging and Generation of Errors," *Microelectron Reliability* , vol. 36, no. 10, pp. 1475-1482, 1996.

[77] D. Knuth, "The error of Tex," *Software: Practice and Experience*, vol. 19, pp. 607-685, 1989.

[78] W. L. Johnson, E. Soloway, B. Cutler, and S. Draper, "Bug Catalogue I," Yale University, 1983.

[79] J. Ploski, "Research Issues in Software Fault Categorization," *ACM Sigsoft Software Engineering Notes*, vol. 32, p. 8, 2007.

[80] C. James, *Errors in Language Learning and Use: exploring error analysis*. London, UK: Longman: Pearson Education, 1998.

[81] R. E. Park, "Software Size Measurement: A Framework for Counting Source Statements," Software Engineering Institute, Camegie-Mellon University, 2001.

[82] L. Wu, "Application of Functional Modeling to Software Reliability in Materials and Nuclear Engineering," Department of Mechanical Engineering, University of

Maryland-College Park, 1997.

[83] R. C. Schank and C. J. Rieger, "Inference and the Computer Understanding of Natural Language," *Artificial Intelligence*, vol. 5, pp. 373-412, 1974.

[84] J. Allen, *Natural Language Understanding*. Redwood City, CA, USA: Benjamin/Cummings Publishing Company , 1987.

[85] A. J. Offutt, " practical system for mutation testing: help for the common programmer," in *Proceeding of the International Test Conference*, Altoona, PA, 1994, pp. 824-830.

[86] J. Spohrer, "Bug Catalogue: II~IV," Yale University, 1985.

[87] Y. K. Malaiya, M. N. Li, J. M. Bieman, and R. Karcich, "Software reliability growth with test coverage," *IEEE Transactions on Reliability*, vol. 51, no. 4, pp. 420-426, Dec. 2002.

[88] IEEE, "Systems engineering — Application and management of the systems engineering process," IEEE STD 1220, 2005.

[89] A. G. Koru, D. Zhang, K. E. Eman, and H. Liu, "An Investigation into the Functional Form of the Size-Defect Relationship for Software Modules," *IEEE Transactions on Software Engineering*, vol. 35, no. 2, pp. 293-304, Mar. 2009.

[90] F. Bouttier and P. Courtier, "Data assimilation concepts and methods," European Centre for Medium-Range Weather Forecasts Lecture Notes, 1999.

[91] C. S. Smidts and D. Sova, "An Architectural Model for Software Reliability Quantification: Source of Data," *Reliability Engineering and System Safety*, vol. 64, pp. 279-290, 1999.

[92] G. E. Murine, "On Validating Software Quality Metrics," in *4th Annual IEEE Conference on Software Quality*, Phoenix, Arizona, 1985.

[93] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, 1976.

[94] T. J. McCabe, "Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric," National Bureau of Standards Special Publication 500-99, 1982.

[95] D. S. Herrmann, *Software Safety and Reliability: Techniques, Approaches, and Standards of Key Industrial Sectors*, First Edition ed. Wiley-IEEE Computer Society , 2000.

[96] "Function Point Counting Practices Manual (Release 4.2)," International Function Point Users Group, 2004.

[97] R. M. Chapman and D. Solomon, "Software Metrics as Error Predictors," NASA, 2002.

[98] M. C. Thompson, D. J. Richardson, and L. A. Clarke, "An Information Flow Model of Fault Detection," in *International Symposium on Software Testing and Analysis*, Cambridge, MA, 1993.