NUREG/CR-7042

A Large Scale Validation of a Methodology for Assessing Software Reliability

Office of Nuclear Regulatory Research

AVAILABILITY OF REFERENCE MATERIALS IN NRC PUBLICATIONS

NRC Reference Material

As of November 1999, you may electronically access NUREG-series publications and other NRC records at NRC's Public Electronic Reading Room at http://www.nrc.gov/reading-rm.html. Publicly released records include, to name a few, NUREG-series publications; *Federal Register* notices; applicant, licensee, and vendor documents and correspondence; NRC correspondence and internal memoranda; bulletins and information notices; inspection and investigative reports; licensee event reports; and Commission papers and their attachments.

NRC publications in the NUREG series, NRC regulations, and *Title 10, Energy*, in the Code of *Federal Regulations* may also be purchased from one of these two sources.

1. The Superintendent of Documents
   U.S. Government Printing Office
   Mail Stop SSOP
   Washington, DC 20402–0001
   Internet: bookstore.gpo.gov
   Telephone: 202-512-1800
   Fax: 202-512-2250
2. The National Technical Information Service
   Springfield, VA 22161–0002
   www.ntis.gov
   1–800–553–6847 or, locally, 703–605–6000

A single copy of each NRC draft report for comment is available free, to the extent of supply, upon written request as follows:

Address: U.S. Nuclear Regulatory Commission
Office of Administration
Publications Branch
Washington, DC 20555-0001
E-mail: DISTRIBUTION.RESOURCE@NRC.GOV
Facsimile: 301–415–2289

Some publications in the NUREG series that are posted at NRC's Web site address http://www.nrc.gov/reading-rm/doc-collections/nuregs are updated periodically and may differ from the last printed version. Although references to material found on a Web site bear the date the material was accessed, the material available on the date cited may subsequently be removed from the site.

Non-NRC Reference Material

Documents available from public and special technical libraries include all open literature items, such as books, journal articles, and transactions, *Federal Register* notices, Federal and State legislation, and congressional reports. Such documents as theses, dissertations, foreign reports and translations, and non-NRC conference proceedings may be purchased from their sponsoring organization.

Copies of industry codes and standards used in a substantive manner in the NRC regulatory process are maintained at—

The NRC Technical Library
Two White Flint North
11545 Rockville Pike
Rockville, MD 20852–2738

These standards are available in the library for reference use by the public. Codes and standards are usually copyrighted and may be purchased from the originating organization or, if they are American National Standards, from—

American National Standards Institute
11 West 42nd Street
New York, NY 10036–8002
www.ansi.org
212–642–4900

Legally binding regulatory requirements are stated only in laws; NRC regulations; licenses, including technical specifications; or orders, not in NUREG-series publications. The views expressed in contractor-prepared publications in this series are not necessarily those of the NRC.

The NUREG series comprises (1) technical and administrative reports and books prepared by the staff (NUREG–XXXX) or agency contractors (NUREG/CR–XXXX), (2) proceedings of conferences (NUREG/CP–XXXX), (3) reports resulting from international agreements (NUREG/IA–XXXX), (4) brochures (NUREG/BR–XXXX), and (5) compilations of legal decisions and orders of the Commission and Atomic and Safety Licensing Boards and of Directors' decisions under Section 2.206 of NRC's regulations (NUREG–0750).

U.S.NRC
United States Nuclear Regulatory Commission
Protecting People and the Environment

NUREG/CR-7042

A Large Scale Validation of a Methodology for Assessing Software Reliability

Manuscript Completed: November 2010
Date Published: July 2011

Prepared by
C. S. Smidts,
Y. Shi, M. Li, W. Kong, J. Dai

Reliability and Risk Laboratory
Nuclear Engineering Program
The Ohio State University
Columbus, Ohio

NRC Project Managers
S. Arndt, N. Carte, R. Shaffer, and M. Waterman

NRC Job Codes Y6591, N6878

Office of Nuclear Regulatory Research

ABSTRACT

This report summarizes the results of a research program initiated by the U.S. Nuclear Regulatory Commission at the University of Maryland[1] to validate a method for predicting software reliability. The method is termed the Reliability Prediction System (RePS). The RePS methodology was initially presented in NUREG/GR-0019, "Software Engineering Measures for Predicting Software Reliability in Safety Critical Digital Systems" and validated on a small control system application with a set of five RePSs in NUREG/CR-6848, "Validation of a Methodology for Assessing Software Quality." The current effort is a validation of the RePS methodology with respect to its ability to predict software quality (measured in this report and in NUREG/GR-0019 in terms of software reliability) and, to a lesser extent, its usability when applied to safety-critical applications.

The application under validation, herein defined as APP, is based on a safety-related digital module typical of what might be used in a nuclear power plant. The APP module contains both discrete and high-level analog input and output circuits. These circuits read input signals from a plant and send outputs that can be used to provide trips or actuations of system equipment, control a process, or provide alarms and indications. The transfer functions performed between the inputs and outputs are dependent on the software that is installed in the module.

The research described in this report provides evidence that twelve selected software engineering measures in the form of RePSs can be used (with different degrees of accuracy) to predict the reliability of software in safety-critical applications. These twelve measures are ranked based on their prediction ability. The rankings are then compared with those obtained through an expert opinion elicitation effort, as described in NUREG/GR-0019, and with those obtained through a small-scale validation, as described in NUREG/CR-6848.

[1] The research was initially performed at the University of Maryland and the report was completed at The Ohio State University.

FOREWORD

This report summarizes the results of a research program initiated by the U.S. Nuclear Regulatory Commission at the University of Maryland and documented in its present form by Ohio State University to validate a method for predicting software quality and reliability. The method is termed the Reliability Prediction System (RePS). The RePS methodology was initially presented in NUREG/GR-0019, "Software Engineering Measures for Predicting Software Reliability in Safety Critical Digital Systems" (ML003775310) and validated on a small control system application with a set of five RePSs in NUREG/CR-6848, "Validation of a Methodology for Assessing Software Quality" (ML042170285).

Since the initial study was limited to five measures and considered a small application, the study only partially validated the expert opinion rankings and RePS theory and thus was not yet conclusive. Validation on an application of larger size was required. The objective of the research described in this report was to perform a large-scale validation of the methodology proposed in NUREG/GR-0019 for twelve measures for all life-cycle phases and apply it to a nuclear safety application. The purpose of the validation was to determine the predictive ability and practical applicability of the methodology to nuclear industry safety systems. The validation results provide insights to guide NRC review and endorsement of IEEE standards such as IEEE Std 1061-1998, *IEEE Standard for a Software Quality Metrics Methodology*.

For this study new RePSs were developed for the measures Cyclomatic Complexity, Cause and Effect Graphing, Requirements Specification Change Requests, Fault-days Number, Capability and Maturity Model, Completeness, and Coverage Factor. In this current study, the mean time to failure (MTTF) measure was not applied and an alternative approach for assessing the failure rate was introduced.

A summary description of the twelve measures is provided, and the results of the RePS software reliability predictions are displayed and analyzed. These predictions are then validated by comparison to a software reliability estimate obtained from operational data and statistical inference. The comparison between the NUREG/GR-0019 ratings and the RePS prediction error is also made, and the efficacy of the proposed methodology for predicting software quality is determined.

The current regulatory review process does not use metrics to assess the potential reliability of digital instrumentation and control systems in quantitative terms. The goal of the research described in this report was to identify methods that could improve the regulatory review process by giving it a more objective technical basis. While some of the models in this report use generic industry data, experimental data, and subjective assessments, much of the modeling is based on direct measurements of the application under study and, as such, is purely objective in nature. Thus, the use of the proposed RePSs models (i.e., of the highly accurate RePSs) could potentially yield better results than what can be obtained from the current review process.

A correlation of the metrics in this report with current NRC regulatory review practice suggests some potential applicability of the metrics for use in licensing activities. The metrics described in

this report provide varying degrees of support to the software lifecycle phase characteristics endorsed by current NRC regulatory guidance; however, some metrics may prove to be too costly or time consuming to implement for the benefit derived. The ultimate feasibility of using these measures in the NRC regulatory process for digital safety systems is outside the scope of this report.

The report advances the study of software quality metrics for potential use in nuclear safety applications and concludes with follow-on activities needed to address issues that were identified in the report. The report provides a priority ranking for follow-up activities that may be needed if future decisions support developing products to be incorporated into the NRC regulatory review process.

TABLE OF CONTENTS

Figures

Tables

ACRONYMS

A/D	Analog to Digital
AA	Percentage of Assessment and Assimilation
AAF	Adaptation Adjustment Factor
AAM	Adaptation Adjustment Modifier
ABL	Address Bus Line
ACAP	Analyst Capability
ACAT	Application Category
ACEG	Actual Cause-Effect Graph
ANSI	American National Standards Institute
APEX	Application Experience
APP	Application
ARM	Automated Reliability Measurement
AT	Acceptance Testing
AVIM	Analog Voltage Isolation Process
BCEG	Benchmark Cause-Effect Graph
BDD	Binary Decision Diagram
BLOC	Bugs per Line Of Code
CBA IPI	Capability Maturity Model-Based Appraisal for Internal Process Improvement
C/R	Capture Recapture
CC	Cyclomatic Complexity
CEG	Cause-Effect Graphing
CF	Coverage Factor
CM	Percentage of Code Modified
CMM	Capability Maturity Model
COM	Completeness

CP	Communication Microprocessor
CPU	Central Processing Unit
CR	Code Review
CTMC	Continuous-Time Markov Chain
D/A	Digital to Analog
DD	Defect Density
DE	Design Phase
DET	Data Element Type
DF	Delta Flux
DM	Percentage of Design Modified
DR	Design Review Phase
DS	Data Set
DTMC	Discrete-Time Markov Chain
EEPROM	Electrical Erasable Programmable Read Only Memory
EFSM	Extended Finite State Machine
EI	External Input
EIF	External Interface File
EO	External Output
EQ	External Query
FDN	Fault-days Number
FP	Function Point
FPU	Float Point Unit
FR	Functional Requirement
FT	Function Testing
FTM	Fault-Tolerant Mechanism
FTR	File Type Reference
GUI	Graphical User Interface
HLEFSM	High Level Extended Finite State Machine

I&C	Instrumentation and Control
I/O	Input/Output
IC	Integrated Circuit
IgT	Integration Testing
ILF	Internal Logical File
IM	Implementation
IpT	Independent Testing
IU	Integer Unit
KLOC	Kilo Lines of Code
KPA	Key Process Area
LCOM	Lack of Cohesion in Methods
LLEFSM	Low Level Extended Finite State Machine
LLNL	Lawrence Livermore National Laboratory
LOC	Line of Code
LTEX	Language and Tool Experience
MIS	Management Information System
MTTF	Mean Time To Failure
NOC	Number Of Children
NR	Non-functional Requirement
N/A	Not Applicable
OBDD	Ordered Binary Decision Diagram
OO	Object Oriented
OP	Operational Profile
PACS	Personnel Access Control System
PCAP	Programmer Capability
PCI	Peripheral Component Interconnect
PCON	Personnel Continuity
PIE	Propagation, Infection, Execution

PIF	Performance Influencing Factor
PLEX	Platform Experience
PMAT	Process Maturity Factor
PROM	Programmable Read Only Memory
QuARS	Quality Analyzer for Requirements Specifications
RAM	Random Access Memory
RC	Run Commands
RCS	Reactor Coolant System
RePS	Reliability Prediction System
RET	Record Element Type
REVL	Requirements Evolution Factor
RMRP	Rule-based Model Refinement Process
ROBDD	Reduced Ordinary Binary Decision Diagram
ROM	Read-Only Memory
RPS	Reactor Protection System
RQ	Requirements Analysis
RR	Requirements Review
RSCR	Requirements Specifications Change Request
RT	Requirements Traceability
SBFI	Simulation-Based Fault Injection
SCED	Development Schedule Factor
SDD	Software Design Description
SITE	Development Site Factor
SLI	Success Likelihood Index
SLIM	Success Likelihood Methodology
SRM	Software Reliability Model
SRS	Software Requirements Specifications
STYLE	Management Style Factor

SU	Percentage of Software Understanding
SUT	Software Under Test
SW-CMM	Software Capability Maturity Model
SWIFI	Software-Implemented Fault Injection
TEAM	Team Cohesion Factor
TC	Test Coverage
TCAP	Tester Capability
TE	Testing
TOOL	Development Tools Factor
TSL	Test Script Language
UFPC	Unadjusted Function Point Count
UMD	University of Maryland
UNFM	Programmers Unfamiliarity with Software
V&V	Verification and Validation
VAF	Value Adjustment Factor
WMC	Weighted Method per Class

1. INTRODUCTION

1.1 Background

The current regulatory review process does not use metrics to assess the potential reliability of digital instrumentation and control systems in quantitative terms. The goal of the research described in this report was to identify methods that could improve the regulatory review process by giving it a more objective technical basis. While some of the models in this report use generic industry data, experimental data, and subjective assessments, much of the modeling is based on direct measurements of the application under study and, as such, is purely objective in nature. Thus, the use of the proposed RePSs models (i.e., of the highly accurate RePSs) could potentially yield better results than what can be obtained from the current review process.

As one of the most important characteristics of software quality [ISO, 2001], software reliability concerns itself with how well software functions meet the requirements of the customer [Musa, 1987]. Software reliability is defined [IEEE, 1991] as the probability of failure-free software operation for a specified period of time in a specified environment. Failures are the result of the triggering of software faults. Triggering of such faults occurs when the right external input conditions are met, i.e., the inputs direct the execution towards the location of a fault. In addition, the defective state of the application (or product) persists until the output results in a significant change of the output conditions when compared to the "correct" or "expected" output conditions. Whether or not the defective state persists depends on the logical structure of the application, on the types of operations encoded, etc. The input conditions are defined by the operational environment in which the application runs. Thus, software reliability is essentially determined by *product characteristics* and the *operational environment*. Product characteristics are further determined by *project characteristics* (the type of application, the project's functional size) and by *development characteristics* (the development team's skill level, the schedule, the tools, and development methods). These characteristics influence the likelihood of faults being introduced into the application/product.

All the above characteristics can be explicitly or implicitly reflected by software engineering measures. Therefore, one inference is that software engineering measures determine software reliability.

Software reliability, in this study, is defined to be the probability that the software-based digital system will successfully perform its intended safety function, for all conditions under which it is expected to respond, upon demand, and with no unintended functions that might affect system safety.

A study sponsored by the U.S. Nuclear Regulatory Commission (NRC) (NUREG/GR-0019, [Smidts, 2000]) systematically ranked 40 software engineering measures with respect to their ability to predict software reliability using expert opinion elicitation. These measures are listed in

the Lawrence Livermore National Laboratory (LLNL) report [LLNL, 1998] and IEEE Std. 982.1-1988 [IEEE 982.1, 1988]. Additional measures are identified in NUREG/GR-0019.

The concept of a Reliability Prediction System (RePS) was proposed in the NRC study to bridge the gap between measures and reliability (see Figure 1.1). A RePS is defined as "a complete set of software engineering measures from which software reliability can be predicted." Figure 1.1 shows the constitution of the RePS. The construction of a RePS starts with the "Measure," which is also the "root" of a RePS. Support measures are identified in order to connect the measure to reliability. The set of the "measure" and "support measures" constitutes a RePS. The "Model" between the measures and reliability is generally termed a Software Reliability Model (SRM).

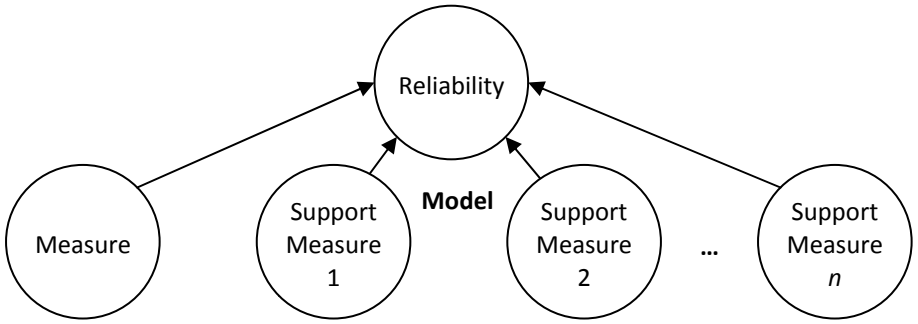


Figure 1.1 RePS Constitution

A small, experimental validation of the expert-opinion-based rankings was performed using six of the measures documented in NUREG/CR-6848 [Smidts, 2004]. These measures were "mean time to failure," "defect density" (DD), "test coverage" (TC), "requirements traceability" (RT), "function point" (FP) analysis and "bugs per line of code" (BLOC) (Gaffney estimate). The application used in the validation study, PACS (Personnel Access Control System) (see NUREG/CR-6848 [Smidts, 2004]), is a simplified version of an automated personnel entry access system controlling a gate used to provide privileged physical access to rooms and buildings. The application was developed industrially using the waterfall lifecycle and a Capability Maturity Model (CMM) level 4 software development process. The application contains approximately 800 lines of code and was developed in C++.

Different software engineering measures were collected at different stages of the software development life-cycle (e.g., requirements, design, coding (implementation), and integration and test[2]) and hence different RePSs can be developed for different phases of the life-cycle. The small-scale validation study performed for the NRC demonstrated the University of Maryland (UMD) research team's ability to construct RePSs during the test phase (i.e., from measures collected during the test phase) and assessed the difference between reliability estimates produced by these RePSs and actual operational reliability. PACS reliability (p_s) was assessed by

[2] The four stages listed (requirements, design, coding, and integration and test) are key development stages in the "waterfall" software development lifecycle model which is widely used in software development. The waterfall model is the recommended lifecycle for safety-critical applications. Variations of the waterfall lifecycle development model exist as well as radically different life-cycles, e.g., "spiral" software-development model, "incremental" software-development model, etc. In such models, the four listed stages may not follow one another in sequence. However, these four stages are always the essential stages in each development model. Thus, the methodology proposed in this report is the basis that can be extended (with some required adjustments) to all the development models.

testing the code against an expected operational profile. In addition, six RePSs were established for the test phase. From these RePSs, the UMD research team obtained six reliability estimates that were compared with p_s. The prediction error defined as the relative difference between p_s and the estimated value was used to rank the measures. This ranking was found to be consistent with the rankings obtained through expert opinion elicitation.

Since the study was limited to six measures, and used what is considered to be a small application, the study only partially validated the expert opinion rankings and RePS theory—thus the study was not conclusive. Validation on an application of larger size was required in which more measures needed to be considered and their corresponding RePSs needed to be constructed. Additionally, the six RePSs already constructed were refined to provide better software reliability estimates. This was not done during the NUREG/CR-6848 [Smidts, 2004] study because the UMD research team were under the requirement to limit the construction of the RePSs to current state-of-the-art validation tools, techniques, methodologies, and published literature.

This report documents a large scale validation of the methodology. It is a continuation of research started in NUREG/GR-0019 [Smidts, 2000] and in NUREG/CR-6848 [Smidts, 2004].

1.2 Objective

The objective of this research was to perform a large-scale validation of the methodology proposed in NUREG/GR-0019 [Smidts, 2000] and apply it to a nuclear-safety application. This was done by applying the methodology to a set of twelve, pre-determined software engineering measures (including five of the six measures that served in the initial validation study described in NUREG/CR-6848 [Smidts, 2004]). RePSs are developed for these twelve measures for all life-cycle phases. In this research, the application of the RePSs to a nuclear power plant reactor safety-control system (Plant X) was limited to the testing phase because the post-mortem nature of the study did not allow reconstruction of the required state of the application throughout the development life-cycle. Such validation helps determine the predictive ability and practical applicability of the methodology to the nuclear power industry.

Also, the validation results could help NRC determine whether or not to endorse a standard set of metrics, such as those described in IEEE Std 1061-1998 (IEEE Standard for a Software Quality Metrics Methodology) [IEEE, 1998].

Chapters 2 to 18 present the details of the theory and its application to the safety-critical system selected. Chapter 19 summarizes the analyses of the results and presents lessons learned, as well as issues to be addressed to further the use of RePS models. Chapter 19 also provides a discussion of how this methodology can be applied to support regulatory reviews of software used in nuclear power plant DI&C systems.

Chapter 20 provides an extended discussion of the potential for increased efficiency and effectiveness of the methodology through automation.

1.3 References

[IEEE 982.1, 1988] "IEEE Standard Dictionary of Measures to Produce Reliable Software," IEEE Std 982.1-1988, 1988.

[IEEE, 1998] "IEEE Standard for a Software Quality Metrics Methodology," IEEE Std 1061-1998, 1998.

[IEEE, 1991] "Standard Glossary of Software Engineering Terminology," IEEE Std 729-1991, 1991.

[LLNL, 1998] J.D. Lawrence et al. "Assessment of Software Reliability Measurement Methods for Use in Probabilistic Risk Assessment," FESSP, Lawrence Livermore National Laboratory, 1998.

[ISO, 2001] ISO/IEC 9126-1:2001, "Software engineering - Product quality - Part 1: Quality model," ISO, 2001.

[Musa, 1987] J.D. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Applications*. New York: McGraw-Hill, 1987.

[Smidts, 2000] C. Smidts and M. Li, "Software Engineering Measures for Predicting Software Reliability in Safety Critical Digital Systems," NRC, Office of Nuclear Regulatory Research, Washington DC NUREG/GR-0019, 2000.

[Smidts, 2004] C. Smidts and M. Li, "Validation of a Methodology for Assessing Software Quality," NRC, Office of Nuclear Regulatory Research, Washington DC NUREG/CR-6848, 2004.

2. RESEARCH METHODOLOGY

2.1 Overview

The research methodology is described below. It consists of six main steps. These are

1. Selection of the Application (APP)
2. Measures/Families Selection
3. Measurement Formalization
4. Reliability Assessment
5. Construction of Reliability Prediction Systems
6. Measurement and Analysis

The above methodology was developed in NUREG/CR-6848 and is applied in this research. Each step is described below.

2.2 Selection of the Application

Software used by nuclear power plants typically belongs to a class of high-integrity, safety-critical, and real-time software systems. The system selected for this study should, to the extent possible, reproduce these same characteristics.

The UMD research team selected a typical Reactor Protection System (RPS) multi-dimensional trip function that uses a number of reactor variables. The function is designed to prevent power operation with reactor power greater than that defined by a function of reactor coolant system flow and reactor core neutron flux imbalance (i.e., flux in the top half of the reactor core minus flux in the bottom half of the reactor core).

The APP was modeled on a typical nuclear power industry APP protection system trip function. The APP contained both discrete and high-level analog input and output circuits. These circuits read input signals from the plant and sent outputs that could be used to provide trips or actuations of safety system equipment, control a process, or provide alarms and indications. The transfer functions performed between the inputs and outputs were dependent on the software installed in the module. The APP function was developed using the processes described in the following standards:

- ANSI/IEEE Standard 830 (1984): IEEE Guide to Software Requirements Specifications.
- ANSI/IEEE Standard 1016 (1987): IEEE Recommended Practice for Software Design Descriptions.

- NRC Regulatory Guide 1.152: Criteria for Programmable Digital Computer System Software in Safety-Related Systems of Nuclear Power Plants.
- ANSI/IEEE/ANS Standard 7-4.3.2 (1982): Application Criteria for Programmable Digital Computer Systems in Safety Systems of Nuclear Power Generating Stations.
- ANSI/IEEE Std 279-1971. "Criteria for Protection Systems for Nuclear Power Generating Stations."
- IEEE Std 603-1991. "IEEE Standard Criteria for Safety Systems for Nuclear Power Generating Stations."
- IEEE Std 730.1-1989. "IEEE Standard for Quality Assurance Plans."

It should be noted that the APP was designed to be safety related. As such, it would have been developed following NUREG-0800 Chapter 7, BTP-14. The APP documentation available to the research team did not explicitly reference BTP-14, however, it cites many of the references found in the NUREG-0800 Chapter 7 BTP-14[3].

The following documents were provided to the contractor by the system developer:

- APP Instruction Manual
- APP Module-Design Specification
- APP Design Requirements
- APP Module µp1 System [Software Requirements Specification] SRS
- APP Module µp1 System [Software Design Description] SDD
- APP Module µp1 System Software Code
- APP Module µp1 Flux/Delta Flux/Flow Application SRS
- APP Module µp1 Flux/Delta Flux/Flow Application SDD
- APP Module µp1 Flux/Delta Flux/Flow Application Software Code
- APP Module µp2 System SRS
- APP Module µp2 System SDD
- APP Module µp2 System Software Code
- APP Module µp2 Flux/Delta Flux/Flow Application SRS
- APP Module µp2 Flux/Delta Flux/Flow Application SDD
- APP Module µp2 Flux/Delta Flux/Flow Application Software Code
- APP Module Communication Processor SRS
- APP Module Communication Processor SDD
- APP Module Communication Processor Software Code APP CTC and SMC System SRS
- APP CTC and SMC System SDD
- APP CTC and SMC System Software Code
- APP Flux/Flow CTC App SRS
- APP Flux/Flow CTC App SDD
- APP Flux/Flow CTC App Software Code
- APP Module Software V&V Plan
- Final V&V Report for APP Module Software
- APP Test Plan for µp1
- APP Test Plan for µp2

[3] NUREG-0800 Chapter 7 BTP-14 cites 28 references among which 17 are not applicable to APP. Among the remaining 11 references, six are also references in the APP documentation.

- APP Test Plan for Communication Processor
- APP Test Summary Report for µp1
- APP Test Summary Report for µp2
- APP Test Summary Report for Communication Processor

2.3 Measures/Families Selection

In order to perform a validation of the ranking of measures defined in NUREG/GR-0019 [Smidts, 2000], two software engineering measures were selected from the high-ranked categories, six from the medium-ranked categories, and four from the low-ranked categories that were identified in NUREG/GR-0019 [Smidts, 2000]. This selection of 12 measures allowed a *partial* validation of the ranking.

The set of measures selected for this study is listed below.

1. Highly-ranked measures: Defect density (DD), Coverage factor (CF).
2. Medium ranked measures: Fault-days number (FDN), Cyclomatic complexity (CC), Requirement specification change request (RSCR), Test coverage (TC), Software capability maturity model (CMM), Requirements traceability (RT).
3. Low-ranked Measures: Function point analysis (FPA), Cause and effect graphing (CEG), Bugs per line of code (Gaffney) (BLOC), Completeness (COM).

A detailed discussion of the measures selection process follows in Chapter 3.

2.4 Measurement Formalization

For a measurement to be useful it must be repeatable. Experience with NUREG/GR-0019 [Smidts, 2000] has shown that no standard definition of the measures exists, or at least no standard definition that ensures repeatability of the measurement. To address these issues, the UMD team began by reviewing the definitions of the measures [IEEE 982.1, 1988] [IEEE 982.2, 1988] to define precise and rigorous measurement rules. This step was seen as necessary due to the inherent limitations of the IEEE standard dictionaries [IEEE 982.1, 1988] and [IEEE 982.2, 1988]. This set of measurement rules is documented in Chapters 6 to 17. The values of the selected measures were then obtained by applying these established rules to the APP system.

Note that IEEE revised IEEE Std. 982.1-1988 in 2005 (see [IEEE 982.1, 2005]). IEEE Std. 982.1-2005 includes minor modifications for two of the twelve measures (Defect Density and Test Coverage) used in this research and adds maintainability and availability measures that are not related to this research. The definitions of defect density and test coverage and the approaches for measuring them have not been modified. Therefore, the release of IEEE Std. 982.1-2005 should not have significant effect on the results presented in this research.

2.5 Reliability Assessment

The quality of APP is measured in terms of its reliability estimate. Reliability is defined here as the probability that the digital system will successfully perform its intended safety function (for the distribution of conditions to which it is expected to respond) upon demand and with no unintended functions that might affect system safety. The UMD team assessed APP reliability using operational data. The operational data and consequent analysis are documented in Chapter 18.

2.6 Reliability Prediction Systems

The measurements do not directly reflect reliability. NUREG/GR-0019 [Smidts, 2000] recognizes the Reliability Prediction System (RePS) as a way to bridge the gap between the measurement and the reliability. RePSs for the measures selected were identified and additional measurements were carried out as required. In particular, the UMD team developed an operational profile to support quantification. This operational profile is documented in Chapter 4. RePS construction is discussed in Chapter 5 and further elaborated in Chapters 6 to 17.

2.7 Assessment of Measure Predictive Ability

The next step was to assess the ability of each measure to predict reliability by comparing the reliability of the code with the predicted reliability. Discrepancies between these two values were then analyzed. This analysis is presented in Chapter 19.

2.8 References

[IEEE 982.1, 1988] "IEEE Standard Dictionary of Measures to Produce Reliable Software," IEEE Std. 982.1-1988, 1988.

[IEEE 982.1, 2005] "IEEE Standard Dictionary of Measures of the Software Aspects of Dependability," IEEE Std. 982.1-2005, 2005.

[IEEE 982.2, 1988] "IEEE Guide for the use of Standard Dictionary of Measures to Produce Reliable Software," IEEE Std. 982.2-1988, 1988.

[Smidts, 2000] C. Smidts and M. Li, "Software Engineering Measures for Predicting Software Reliability in Safety Critical Digital Systems," NRC, Office of Nuclear Regulatory Research, Washington DC NUREG/GR-0019, 2000.

[Smidts, 2004] C. Smidts and M. Li, "Validation of a Methodology for Assessing Software Quality," NRC, Office of Nuclear Regulatory Research, Washington DC NUREG/CR-6848, 2004.

3. SELECTION OF MEASURES

This chapter discusses the rationale for the selection of the measures used in the project. The final selection of the measures includes "Defect density," "Coverage factor," "Fault days number," "Cyclomatic complexity," "Requirement specification change request," "Test coverage," "Software capability maturity model," "Requirements traceability," "Function point analysis," "Cause and effect graphing," "Bugs per line of code" (Gaffney [Gaffney, 1984]), and "Completeness."

3.1 Criteria for Measure Selection

Measures for the validation project were selected based upon the following criteria:

1. Ranking levels
2. Measure applicability
3. Data availability
4. Coverage of different families

Each of the above criteria is described below.

3.2 Ranking Levels

This project was designed to validate the results presented in NUREG/GR-0019 [Smidts, 2000], "Software Engineering Measures for Predicting Software Reliability in Safety Critical Digital Systems." In that study, forty[4] measures were ranked based on their ability to predict software reliability in safety-critical digital systems. This study documented in NUREG/GR-0019 must be validated to confirm that highly ranked measures can accurately predict software reliability. High prediction quality means that the prediction is close to the actual software reliability value.

A complete validation could be performed by: 1) predicting software reliability from each of the pre-selected forty measures in NUREG/GR-0019; and then 2) comparing predicted reliability with actual reliability obtained through reliability testing. However, the limited schedule and budget of the current research constrained UMD's ability to perform such a brute-force experiment on all forty measures. An alternative method was proposed whereby: a) two measures were selected from the high-ranked measures, six from the medium-ranked measures,

[4]The initial study involved 30 measures. The experts then identified an additional 10 missing measures bringing the total number of measures involved in the study to 40.

and four from the low-ranked measures; b) the above experiment was performed on these twelve measures; and c) the results were extrapolated to the whole spectrum of measures.

The forty measures available during the testing phase were classified into high-ranked, medium-ranked, and low-ranked measures by comparing the predicted reliability with the actually reliability (rate) using the following thresholds[5]:

1. High-ranked measures: $0.75 \leq \text{rate} \leq 0.83$
2. Medium-ranked measures: $0.51 \leq \text{rate} < 0.75$
3. Low-ranked measures: $0.30 \leq \text{rate} < 0.51$

Table 3.1 lists the high-ranked measures, medium-ranked measures, and low-ranked measures.

Table 3.1 Measures Ranking Classification

Measure	Rate	Rank No.	Ranking Class
Failure rate	0.83	1	High
Code defect density	0.83	2	
Coverage factor	0.81	3	
Mean time to failure	0.79	4	
Cumulative failure profile	0.76	5	
Design defect density	0.75	6	
Fault density	0.75	7	
Fault-days number	0.72	8	Medium
Cyclomatic complexity	0.72	9	
Mutation score	0.71	10	
Minimal unit test case determination	0.7	11	
Modular test coverage	0.7	12	
Requirements specification change requests	0.69	13	
Test coverage	0.68	14	
Class coupling	0.66	15	
Class hierarchy nesting level	0.66	16	
Error distribution	0.66	17	
Number of children (NOC)	0.66	18	
Number of class methods	0.66	19	

[5] These thresholds are determined by the mean (μ) and standard deviation (σ) of the distribution of the rates of the measures. Rates define the degree to which measures can be used to predict software reliability. These rates are real numbers ranging from 0 to 1. Rates of 1 indicate measures deemed crucial to the prediction of software reliability. Rates of 0 correspond to measures that definitely should not be used. The rate of a measure is obtained by aggregating the experts' opinions. The intervals correspond to: $\mu+\sigma \leq$ rate $\leq$ upper limit, $\mu-\delta \leq$ rate $< \mu+\sigma$, lower limit $\leq$ rate $< \mu - \sigma$. Please refer to NUREG/GR-0019 for details.

Table 3.1 Measures Ranking Classification (continued)

Measure	Rate	Rank No.	Ranking Class
Lack of cohesion in methods (LCOM)	0.65	20	Medium
Weighted method per class (WMC)	0.65	21	
Man hours per major defect detected	0.63	22	
Functional test coverage	0.62	23	
Reviews, inspections and walkthroughs	0.61	24	
Software capability maturity model	0.6	25	
Data flow complexity	0.59	26	
Requirements traceability	0.55	27	
System design complexity	0.53	28	
Number of faults remaining (error seeding)	0.51	29	
Number of key classes	0.51	30	
Function point analysis	0.5	31	Low
Mutation testing (error seeding)	0.5	32	
Requirements compliance	0.5	33	
Full function point	0.48	34	
Graph-theoretic static architecture complexity	0.46	35	
Feature point analysis	0.45	36	
Cause and effect graphing	0.44	37	
Bugs per line of code (Gaffney)	0.4	38	
Cohesion	0.33	39	
Completeness	0.36	40	

The measures “Defect density,” (which includes “Code defect density” [ranks No. 2] and “Design defect density” [No. 6]) and “Coverage factor” [No. 3] were chosen as high-ranked measures. The measures “Fault-days number” [No. 8], “Cyclomatic complexity” [No. 9], “Requirements specification change request” [No. 13], “Test coverage” [No. 14], “Software capability maturity model” [No. 25], and “Requirements traceability” [No. 27] were selected as the medium-ranked measures. The low-ranked measures included “Function point analysis” [No. 31], “Cause and effect graphing” [No. 37], “Bugs per line of code (Gaffney estimate)” [No. 38], and “Completeness” [No. 40].

3.3 Measure Applicability

Measure applicability is an important criterion by which measures were selected. Since the APP code was written in ANSI C and Assembly language, only non-Object Oriented (OO) measures could be considered as a part of the pool of measures.

3.4 Data Availability

Data availability is another criterion that limits the selection. None of the measures were directly available. However, base data from which software engineering measures could be calculated were mostly available in the testing phase and either totally or partially unavailable in other phases of the life-cycle (see Table 3.2).

3.5 Coverage of Different Families

As addressed in Section 2.3 of NUREG/GR-0019:

> Measures can be related to a small number of concepts such as for instance the concept of complexity, the concept of software failure or software fault. Although the number of these concepts is certainly limited, the number of software engineering measures certainly does not seem to be. Therefore a many-to-one relationship must exist between measures and primary concepts. These primary concepts are at the basis of groups of software engineering measures, which in this study are called *families*. Two measures are said to belong to the same family if, and only if, they measure the same quantity (or more precisely, concept) using alternate means of evaluation. For example, the family *Functional Size* contains measures "Function Point" and "Feature Point." Feature point analysis is a revised version of function point analysis appropriate for real-time embedded systems. Both measures are based on the same fundamental concepts. [Albrecht, 1979] [Jones, 1986] [Jones, 1991]

In this study, the attempt was made to select measures from as many families as possible so as to obtain a broad coverage of semantic concepts[6]. The twelve selected measures were chosen from the following families: "Fault detected per unit of size," "Fault-tolerant coverage factor," "Time taken to detect and remove faults," "Module structural complexity," "Requirements specification change requests," "Test coverage," "Software development maturity," "Requirements traceability," "Functional size," "Cause and effect graphing," "Estimate of faults remaining in code," and "Completeness" (see Table 3.2). This selection reflects a bias toward failure- and fault-related families as well as requirements-related families. This is due to a strong belief that software reliability is largely based upon faulty characteristics of the artifact and the quality of requirements used to build the artifact.

[6]The "semantic" concept was also termed as "family" which is defined as a set of software engineering measures that evaluate the same quantity.

3.6 Final Selection

Table 3.2 lists several characteristics of the pre-selected measures including: the family to which the measure pertains, the measure applicability, the availability of APP data, and the ranking class.

The final selection is thus as follows: "Defect density," "Coverage factor," "Fault days number," "Cyclomatic complexity," "Requirement specification change request," "Test coverage," "Software capability maturity model," "Requirements traceability," "Function point analysis," "Cause and effect graphing," "Bugs per line of code" (Gaffney estimate), and "Completeness." In Table 3.2, these measures are in boldface. The applicable life-cycle phases of each measure are provided in Table 3.3.

Table 3.2 Measure, Family, Measure Applicability, Data Availability, and Ranking Class

Measure	Family	Measure Applicability	Data Availability	Ranking Class
Failure rate	Failure rate	Applicable	Available	High
Mean time to failure	Failure rate	Applicable	Available	
Cumulative failure profile	Failure rate	Applicable	Not Available	
Coverage factor	**Fault-tolerant coverage factor**	**Applicable**	**Available**	
Code defect density	**Fault detected per unit of size**	**Applicable**	**Available**	
Design defect density	Fault detected per unit of size	Applicable	Available	
Fault density	Fault detected per unit of size	Applicable	Available	

Table 3.2 Measure, Family, Measure Applicability, Data Availability, and Ranking Class (continued)

Measure	Family	Measure Applicability	Data Availability	Ranking Class
Modular test coverage	Test coverage	Applicable	Available	Medium
Test coverage	**Test coverage**	**Applicable**	**Available**	
Fault-days number	**Time taken to detect and remove faults**	**Applicable**	**Available**	
Functional test coverage	Test coverage	Applicable	Available	
System design complexity	System architectural complexity	Applicable	Available	
Mutation score	Test adequacy	Applicable	Available	
Minimal unit test case determination	Module structural complexity	Applicable	Available	
Requirements specification change requests	**Requirements specification change requests**	**Applicable**	**Available**	
Error distribution	Error distribution	Applicable	Available	
Class coupling	Coupling	Not Applicable	-	
Class hierarchy nesting level	Class inheritance depth	Not Applicable	-	
Number of children (NOC)	Class inheritance breadth	Not Applicable	-	
Number of class methods	Class behavioral complexity	Not Applicable	-	
Lack of cohesion in methods (LCOM)	Cohesion	Not Applicable	-	
Weighted method per class (WMC)	Class structural complexity	Not Applicable	-	
Man hours per major defect detected	Time taken to detect and remove faults	Applicable	Available	
Reviews, inspections and walkthroughs	Reviews, inspections and walkthroughs	Applicable	Available	

Table 3.2 Measure, Family, Measure Applicability, Data Availability, and Ranking Class (continued)

Measure	Family	Measure Applicability	Data Availability	Ranking Class
Software capability maturity model	**Software development maturity**	**Applicable**	**Available**	Medium
Requirements traceability	**Requirements traceability**	**Applicable**	**Available**	
Number of key classes	Functional size	Not Applicable	-	
Number of faults remaining (error seeding)	Estimate faults remaining in code	Applicable	Available	
Cyclomatic complexity	**Module structural complexity**	**Applicable**	**Available**	Low
Data flow complexity	System architectural complexity	Applicable	Available	
Requirements compliance	Requirements compliance	Applicable	Available	
Mutation testing (error seeding)	Estimate faults remaining in code	Applicable	Available	
Cause and effect graphing	**Cause and effect graphing**	**Applicable**	**Available**	
Full function point	Functional size	Applicable	Available	
Function point analysis	**Functional size**	**Applicable**	**Available**	
Graph-theoretic static architecture complexity	System architectural complexity	Applicable	Available	
Feature point analysis	Functional size	Applicable	Available	
Cohesion	Cohesion	Applicable	Available	
Completeness	**Completeness**	**Applicable**	**Available**	

Table 3.3 Applicable Life-Cycle Phases of the Selected Measures

Family	Measures	Applicable Life Cycle Phases
Estimate of Faults Remaining per Unit of Size	BLOC	Coding, Testing, Operation
Cause and Effect Graphing	CEG	Requirements, Design, Coding, Testing, Operation
Software Development Maturity	CMM	Requirements, Design, Coding, Testing, Operation
Completeness	COM	Requirements, Design, Coding, Testing, Operation
Fault-Tolerant Coverage Factor	CF	Testing, Operation
Module Structural Complexity	CC	Design, Coding, Testing, Operation
Faults Detected per Unit of Size	DD	Testing, Operation
Time Taken to Detect and Remove Faults	FDN	Requirements, Design, Coding, Testing, Operation
Functional Size	FP	Requirements, Design, Coding, Testing, Operation
Requirements Specification Change Request	RSCR	Requirements, Design, Coding, Testing, Operation
Requirement Traceability	RT	Design, Coding, Testing, Operation
Test Coverage	TC	Testing, Operation

3.7 References

[Albrecht, 1979] A.J. Albrecht. "Measuring Application Development," in *Proc. SHARE-GUIDE*, 1979, pp. 83–92.

[Gaffney, 1984] J.E. Gaffney. "Estimating the Number of Faults in Code." *IEEE Transactions on Software Engineering*, vol. 10, pp. 459–64, 1984.

[Jones, 1986] C. Jones. *Programming Productivity*. McGraw-Hill, Inc., 1986.

[Jones, 1991] C. Jones. *Applied Software Measurement*. McGraw-Hill, Inc., 1991.

[Smidts, 2000] C. Smidts and M. Li, "Software Engineering Measures for Predicting Software Reliability in Safety Critical Digital Systems," NRC, Office of Nuclear Regulatory Research, Washington DC NUREG/GR-0019, 2000.

4. OPERATIONAL PROFILE

4.1 Introduction

The operational profile (OP) is a quantitative characterization of the way in which a system will be used [Musa, 1992]. It associates a set of probabilities to the program input space and therefore describes the behavior of the system. The determination of the OP for a system is crucial because OP can help guide managerial and engineering decisions throughout the entire software development life cycle [Musa, 1992]. For instance, OP can assist in the allocation of resources for development, help manage the reviews on the basis of expected use, and act as a guideline for software testing.

The OP of a system is also a major deciding factor in assessing its reliability. Software reliability is the ability of a software system or component to perform its intended functions under stated conditions for a specific period of time [IEEE, 1991]. The OP is used to measure software reliability by testing the software in a manner that represents actual use or it is used to quantify the propagation of defects (or unreliability) through extended finite state machine models [Li, 2004] [Smidts, 2004]. However, determining the OP of the system is a challenging part of software reliability assessment in general [Shukla, 2004].

The OP is traditionally evaluated by enumerating field inputs and evaluating their occurrence frequencies. Musa pioneered a five-step approach to develop the OP. His approach is based on collecting information on customers and users, identifying the system modes, determining the functional profile, and recording the input states and their associated occurrence probabilities experienced in field operation. Expert opinion, instead, is normally used to estimate the hardware components-related OP due to the lack of field data.

Musa's approach has been widely utilized and adapted in the literature to generate the OP. Some of these applications are summarized below:

- Chruscielski and Tian applied Musa's approach to a Lockheed Martin Tactical Aircraft System's cartridge support system [Chruscielski, 1997]. User surveys were used instead of field data.

- Elbaum and Narla [Elbaum, 2001] refined Musa's approach by addressing heterogeneous user groups. They discovered that a single OP only "averages" the usage and "obscures" the real information about the operational probabilities. They utilized clustering to identify groups of similar customers.

- Gittens, et al., proposed an extended OP model which is composed of the process profile, structural profile, and data profile. The process profile addresses the processes and associated frequencies. The structural profile accounts for the system structure, the configuration or

structure of the actual application, and the data profile covers the inputs to the application from different users [Gittens, 2004].

In this research, the probabilities for individual operations instead of end-to-end operations are considered. Musa's approach and other extended approaches all require either field data or historic usage data. These approaches use an assumption that field data or historic usage data cover the entire input domain. This assumption is not always true and these approaches are not always successful simply because some input data may not be available, especially for safety critical control systems.

There are at least two reasons why the entire input data spectrum is often unavailable. First, the system may not be widely used. Therefore, very little field and historic usage data can be obtained. Second, the field data may not cover the entire spectrum of the input domain because some conditions may be extremely rare. Further many inputs may not be visible. The derivation of a generic OP generation method for safety critical systems based on limited available data is presented in this chapter.

Since the different values of the environmental inputs will have major effects on processing, Musa's [Musa, 1992] recommended approach for identifying the environmental variables is to have several experienced system design engineers brainstorm a list of those variables that might necessitate the program to respond in different ways. Furthermore, Sandfoss [Sandfoss, 1997] suggests that estimation of occurrence probabilities could be based on numbers obtained from project documentation, engineering judgment, and system development experience. According to Gittens [Gittens, 2004], a specific OP should include all users and all operating conditions that can affect the system. In this research their approaches have been extended and a systematic method to identify those environmental variables and estimate all the environmental inputs has been generated.

This chapter is structured as follows. Section 4.2 describes the generic architecture of the safety critical system under study. The method for OP generation will be introduced in Section 4.3 along with a detailed example.

4.2 Generic Architecture of Reactive Systems

Reactive systems continuously react with their environment and must satisfy timing constraints to be able to take into account all the external events [Ouab, 1995]. Such reactive systems may be used to implement a safety critical application.

A typical reactive system is composed of components such as sensors, actuators, voters and controllers (software and hardware). Both sensors and actuators are used to implement the mechanisms that interact with the reactive system's environment. Sensors are used to acquire the plant input information[7]. Safety critical systems are designed to control and monitor these systems. The outputs from the different controllers are provided to the voter and the voting

[7] The term "plant" has a broad definition. Complex systems such as nuclear power plants, aircrafts, and military systems are considered "plants."

results are sent to the actuators, which are used to maintain interaction with the plant, i.e. perform corresponding actions. The voter can be a hardware component or an independent software-based system. If an accident condition is identified by the voter, a safety protection action will be initialized. For instance, in a nuclear power plant, if the reactor's power distribution parameter exceeds its allowable limits, the reactor protection system will issue a trip signal to shutdown the reactor and inform the operator.

It should be noted that the notion of sensors and actuators can be extended to human beings. Human beings may indeed provide inputs (as sensors) and receive output from the controller and then take further actions (as actuators).

4.3 APP Architecture

The APP application under study was a model of a nuclear reactor protection system that falls into the reactive system category and in addition is a safety critical system. Figure 4.1 depicts the architecture of the APP system. Three layers coexist: the application software layer, the system software layer, and the infrastructure layer.

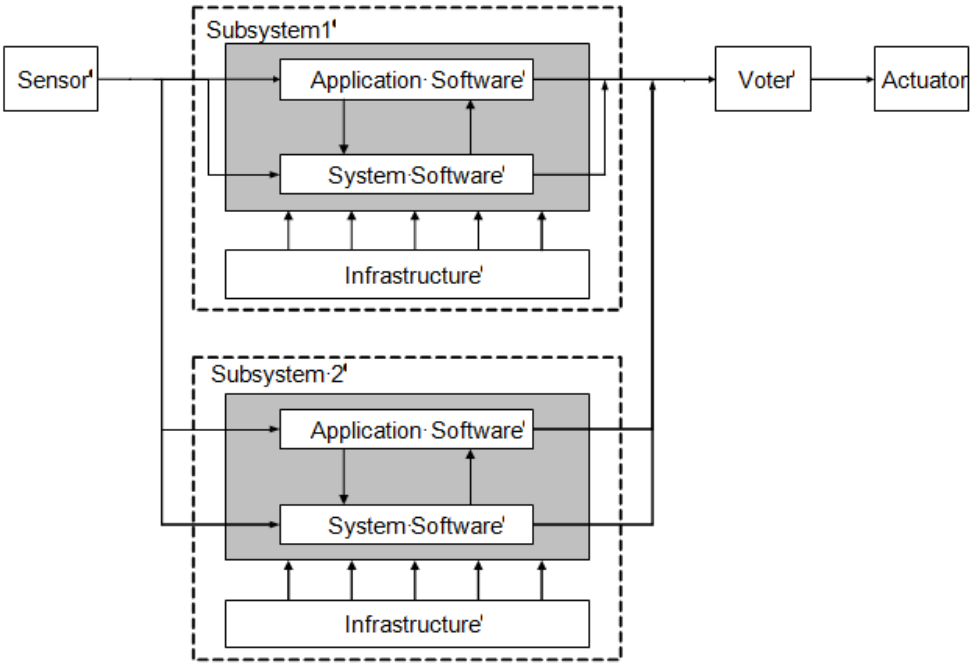


Figure 4.1 The APP Architecture

The top layer is an application software layer that contains the safety control algorithms, which implement the intended functionality. The APP application software receives the plant inputs, and then determines whether the reactor is operating normally. If this is not the case, a trip signal is issued to shutdown the reactor. Although this layer can independently perform its intended function, the features that monitor and assure the healthy functioning of this layer are not implemented in this layer. Such features include, but are not limited to, online diagnostics for critical hardware components such as memory, timely enforcement of each cycle to assure the

system can react in a real-time manner, etc. The APP system contains a layer that implements these features and is depicted in Figure 4.1 as the system software layer. This layer is also called the health-monitoring layer.

The status of the system hardware components will be determined through well-defined diagnostics procedures. It is worth noting that the operating system for large-scale, safety critical control systems falls into this layer also. The system software layer also receives plant inputs to monitor the status of the sensors and to determine whether the inputs are in a normal range. If this layer detects anomalies, it will first maintain the entire control system in a fail-safe situation (for instance, shutdown the reactor in the nuclear industry) and then issue an alarm signal or its equivalent (for instance, trip signal in the nuclear industry). Communications are used to share information between these two layers.

The lowest layer is the infrastructure layer, which acts as the infrastructure of the system. It is obvious that the normal operation of the safety critical system relies on the successful operation of this layer. Failure of any hardware component may lead to the malfunction of the system. Such failures cannot be neglected in modern safety critical systems. The failure rates of hardware components have been reduced to the level of 10^{-7} failures per hour or less in light of contemporary manufacturing technologies [Poloski, 1998].

It should be pointed out that the division between layers may be somewhat arbitrary. Further the three layers are not independent. Application software, system software, and the infrastructure are required to work together to perform the system function. Failure modes between layers are interdependent. In the case of the APP, the possible impact of failures of the infrastructure layer on the application software is handled by the system software, which conducts the online monitoring of the infrastructure layer.

The counterparts to "plant inputs" are "infrastructure inputs," which include the hardware and software health statuses. The infrastructure input is an important component of the OP. This is because the input inevitably influences the way in which the system software executes. The infrastructure inputs are normally invisible and typically are not included in the OP. The customers generally are not aware of these infrastructure inputs [Musa, 1992].

4.4 Generating the Operational Profile

After studying the general architecture of reactive systems, one can conclude that an OP for such systems should address operating conditions for each subsystem and the operating conditions for the voter if it is an independent software-based system. As for each subsystem, both the operating conditions for the application software and the system software should be considered. That is,

$$O = (O_{S_1}, O_{S_2}, \ldots, O_{S_n}, O_v) \tag{4.1}$$

where

O_{S_1} OP for subsystem 1

O_{S_2} OP for subsystem 2

O_{S_n} OP for subsystem *n*

O_v OP for the voter if it is a software-based system

The OP for each sub-system with the exception of the voter is discussed in this section.

According to Musa, the system modes (subsystem modes) need to be determined before generating the OP. A system mode is a set of functions or operations that are grouped together for convenience for analyzing the system's operational behavior. A system can switch between system modes so that only one system mode is in effect at a given time, or different system modes can exist simultaneously, sharing the same computer resources [Musa, 1992]. After determining the system modes, the OP must be generated for each mode. Thus the general complete OP for a subsystem with multiple operational modes is:

$$O_{S_i} = (O_{S_iM_1}, \ldots, O_{S_iM_n}) \tag{4.2}$$

where

O_{S_i} OP for subsystem *i*

$O_{S_iM_1}$ OP for the first system mode of subsystem *i*

$O_{S_iM_n}$ OP for the *n*-th system mode of subsystem *i*

In general, an OP of a software system is the complete set of all the input probabilities in a given operational mode. Therefore, there is a high level system input which is used to determine the system mode. This type of input can be expressed as the probability of the system modes.

Based on the discussion in the previous section, the OP for a subsystem in a specific operational mode, $O_{S_iM_j}$, is a pair of two elements: the element denoted as $O_{S_ipi_j}$ that represents the OP for the plant inputs, and the element denoted as $O_{S_iii_j}$ that represents the OP for the infrastructure inputs. Therefore the complete set of an OP in operational mode *j* can be expressed as:

$$O_{S_iM_j} = (O_{S_ipi_j}, O_{S_iii_j}) \tag{4.3}$$

The construction of these two elements is discussed in turn in the following subsections.

There are two subsystems, µp1 and µp2, in the APP system used to implement the trip function. These two subsystems work independently. Each subsystem receives inputs from sensors and conducts its own internal calculations. Whether or not to send out a trip signal depends on the calculation results. The APP voter is a hardware component. There is a communication processor (CP) which handles communications between the two subsystems and other equipment outside the APP system. CP is only required during the power-up sequence, calibration, and tuning modes. CP only uses the infrastructure inputs as do µp1 and µp2 but is not related to the "plant inputs." Thus, the OP for the APP system should include the operational conditions for these three subsystems:

$$O_{APP} = (O_{\mu p1}, O_{\mu p2}, O_{CP}) \tag{4.4}$$

where

O_{APP} OP for the APP system
$O_{\mu p1}$ OP for µp1
$O_{\mu p2}$ OP for µp2
O_{CP} OP for CP

The APP possesses four distinct operational modes: Power On, Normal, Calibration, and Tuning [APP, 1].

1. The "Power On Mode" includes the initialization function and the self test procedures for each microprocessor in the APP. The system will not be put into action until it is successfully powered on.
2. The "Normal Mode" is the main working mode for the APP. In this mode the APP monitors the nuclear power plant operating conditions.
3. The "Calibration Mode" is chosen if there is a need to perform an input or output calibration.
4. The "Tuning Mode" is chosen if there is a need to reload the parameters used for the application algorithm.

There is a switch on the APP front panel that is used to force the APP to switch from one mode to another. The probability of each system mode is shown in Table 4.1 and the composition of the OP for each operational mode is also shown.

Table 4.1 Composition of the Operational Profile for the APP Operational Modes

Operational Mode	Operational Profile				Probability
	$O_{\mu p1iij}$	$O_{\mu p2iij}$	O_{CPiij}	$O_{\mu p1pij}, O_{\mu p2pij}$	
Power On	µp1 Infrastructure Inputs	µp2 Infrastructure Inputs	CP Infrastructure Inputs	∅	1.004×10^{-6}
Normal	µp1 Infrastructure Inputs	µp2 Infrastructure Inputs	∅	Plant-Specific Inputs	0.992
Calibration	µp1 Infrastructure Inputs	µp2 Infrastructure Inputs	CP Infrastructure Inputs	Fixed Plant Inputs	0.004
Tuning	µp1 Infrastructure Inputs	µp2 Infrastructure Inputs	CP Infrastructure Inputs	Fixed Plant Inputs	0.004

As shown in Table 4.1, in the power-on mode, the APP is not ready to receive inputs from the nuclear power plant system. UMD conducted experiments and the results revealed the average "power-on" duration to be around 20 s. This included the initialization procedures and "power-

on" self tests for all three microprocessors. UMD also understood that plant outages required the APP module to be shut down. Outage data will be shown later in Table 4.9. This data was obtained from a nuclear power plant that had been using a similar APP module in which there had been 19 outages in 12 years. Thus, the probability of the APP being in this mode can be estimated as

$$\frac{\left(\frac{20\text{ s}}{\text{power on}}\right)(19\text{ power ons})}{(12\text{ yr})(3600\text{ s/hr})(24\text{ hr/dy})(365\text{ dy/yr})} = 1.004 \times 10^{-6}$$

During the normal mode, the APP system implements a reactor protection (or trip) function that evaluated core power distribution [USNRC, 1995]. The trip function is used to prevent operation when reactor power is greater than that defined by a function of the reactor coolant system (RCS) flow rate and when the indicated power imbalance exceeds safety limits. A reactor trip will be issued by the APP system if the total power (flux) or power distribution exceeds a pre-determined safety boundary. This function is implemented by the APP system application software. The APP system software is used to diagnose whether its hardware components are in healthy condition. In the case of the APP, the two subsets of OPs are:

1. OP for APP infrastructure inputs. The infrastructure inputs of the APP consist of the statuses of all hardware components identified through the procedures predefined in the system software.
2. OP for APP plant inputs. The inputs to the APP include four analog inputs. The application software obtains these inputs from the plant and conducts the calculation based on the predefined algorithms. The system software also reads these inputs to verify whether the input components function normally. The actuator functions according to the output of the application and system software against the inputs. The four inputs are the measured reactor power in the top half of the reactor core as represented by neutron flux monitoring instrumentation (F_U), the measured reactor power in the bottom half of the reactor core (F_L), and reactor coolant flow rates represented by pressure differential measurement instruments in the RCS hot leg loop A (DP_A) and the RCS hot leg loop B (DP_B). The plant inputs OP consists of the probability distribution of these four inputs.

Per discussion with a APP system expert the calibration and tuning are performed every two weeks and the calibration and tuning required approximately 2 hours to perform. Thus, the probability that the APP is in a calibration or tuning mode can be estimated as

$$\frac{\left(26\frac{\text{cal}}{\text{yr}}\right)\left(2\frac{\text{hr}}{\text{cal}}\right)}{(24\text{ hr/dy})(365\text{ dy/yr})} = 0.006$$

Also, a functional test is performed every 45 days (or 8 tests/year) requiring 2 hr/test. Thus, the probability that APP is in a functional test mode per year can be estimated as:

$$\frac{\left(8\frac{\text{tests}}{\text{yr}}\right)\left(2\frac{\text{hr}}{\text{test}}\right)}{(24\text{ hr/dy})(365\text{ dy/yr})} = 0.002$$

While in these modes, the APP would be bypassed and would not receive any actual plant inputs. Instead, fixed plant inputs would be used to perform the calibration and tuning functions. These fixed inputs, however, act as parameters and do not need to be determined as part of the OP. The sum of the calibration and tuning probability and the functional test probability are divided between the calibration and tuning probabilities in Table 4.1 above (i.e., 0.004 for each mode).

The following subsections provide a general discussion of the OP construction. Further, a construction of the infrastructure inputs OP is discussed followed by the plant inputs OP. An application of this approach to the APP system is then illustrated.

4.4.1 A Guided Operational Profile Construction

The concept of OP has been used in automated software reliability-engineered testing and software reliability assessment studies [Li, 2003] [Li, 2004] [Widmaier, 2000].

Musa [Musa, 1992] pioneered a five-step approach to develop the OP. As shown in Figure 4.2, his approach is based on collecting information on customers and users, identifying the system modes, determining the functional profile, and recording the input states and their associated occurrence probabilities experienced in field operation. The Musa approach is user and customer centric and is most relevant for applications with a large user and customer group. In the case of the APP, the number of customers and users was limited and focus was mostly on physical system parameters and infrastructure parameters rather than on functions which may depend on the type of user or consumer. Furthermore, portions of the data space that represented the most significant portions of the OP may not have been encountered in the field (such as hardware failure modes, or physical input conditions that trigger trip conditions) and corresponding data may not exist. The approach used to generate the APP OP is discussed in this section.

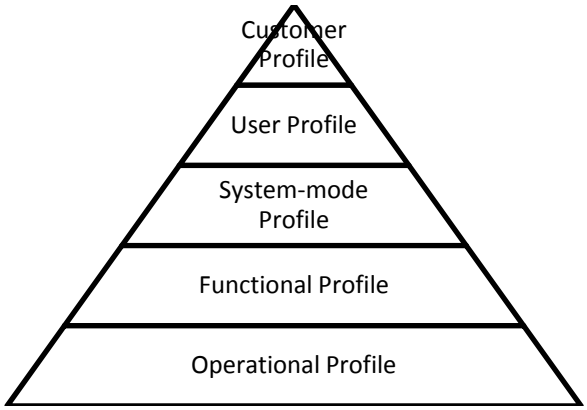


Figure 4.2 Musa's Five-Step Approach for OP Development

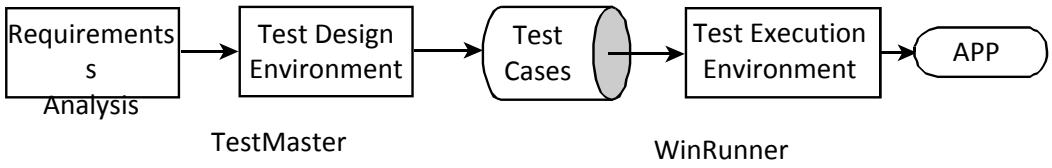


Figure 4.3 Test Environment

The automated software reliability-engineered testing process involves developing a test oracle represented by an Extended Finite State Machine (EFSM) model using a tool named TestMaster [TestMaster, 2000] [TestMaster, 2004]. The EFSM model is constructed based on the software requirements specification[8]. The TestMaster tool is used to build the EFSM model and execute this model to generate test scripts in accordance with the OP. The test scripts are then executed on the software under test (SUT) using WinRunner [WinRunner, 2001] as a test harness. The results of the tests (numbers of failures and trials) are recorded and used to calculate reliability.

TestMaster is a test design tool that uses the EFSM notation to model a system [TestMaster, 2000]. TestMaster captures 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, 2004]. TestMaster allows models to capture the history of the system and enables requirements-based extended finite state machine notation. It also allows for the specification of the likelihood that events or transitions from a state will occur. Therefore, the OP can be easily integrated in the model.

Figure 4.4 depicts an example EFSM that models the PROM (Programmable Read Only Memory) test function in the APP system.

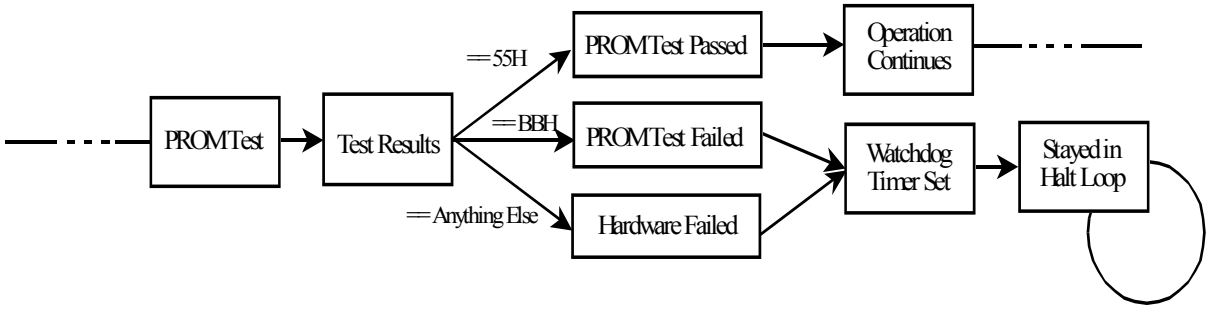


Figure 4.4 An Example EFSM Model for the APP system

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 fails during the status writing operation.

[8] Please refer to Chapter 5 for a more detailed discussion on EFSM and Appendix A for EFSM construction procedures.

After completing the model, software tests are created automatically with a test script generator. A test is defined as a path from the entry to the exit state. The test generator develops tests by identifying a path through the diagram from the entry to the exit state. The path is a sequence of events and actions that traverses the diagram, defining an actual-use scenario. As for the above example, the ordered state series {PROM Test, Test Results, PROM Test Passed, Operation Continues} (denoted as P_1), {PROM Test, Test Results, PROM Test Failed, Watchdog Timer Set, Stayed in Halt Loop}(denoted as P_2), and {PROM Test, Test Results, Hardware Failed, Watchdog Timer Set, Stayed in Halt Loop} (denoted as P_3) are possible paths.

TestMaster implements several test strategies such as Full Cover, Transition Cover, and Profile Cover. The strategy used to generate test cases is Profile Cover. Profile Cover generates a pre-specified number of N test cases in accordance with the likelihood of each path. In TestMaster, the likelihood of a path is the product of the likelihoods of transitions that traverse this path. Only likelihoods for the three conditional transitions count:

$T_{1,1} == 55H$ occurs
$T_{1,2}: == BBH$ occurs
$T_{1,3}$: Anything else occurs, as shown in Figure 4.4

This is because likelihoods for other transitions are 0.0. Therefore, we have:

$$\Pr(P_1) = \Pr(T_{1,1})$$
$$\Pr(P_2) = \Pr(T_{1,2})$$
$$\Pr(P_3) = \Pr(T_{1,3})$$

As such, we define the OP for the example in Figure 4.4, O, as:

$$\{\{T_{1,1}, \Pr(T_{1,1})\}, \{T_{1,2}, \Pr(T_{1,2})\}, \{T_{1,3}, \Pr(T_{1,3})\}\}$$

It should be noted that:

$$\Pr(T_{1,1}) + \Pr(T_{1,2}) + \Pr(T_{1,3}) = 1$$

The OP is generally defined as:

$$O: \{T, \Pr(T)\}$$

Where O is the set of OP, T is the set of occurrences of the multiple transitions (multiple options after one state), and $\Pr(t)$ is the set of probabilities of T. In other words:

$$T = \{T_{i,j}\}$$

Where *i* is the index for the occurrence and *j* is the index for the transitions within each occurrence.

$$\sum_j \Pr(T_{i,j}) = 1$$

holds for the *i*-th occurrence.

This OP definition is different from Musa's in the sense that the point of interest is transitions instead of each individual input. It is worth noting that the condition for a transition may be the combination of multiple inputs. This issue will be addressed later in this chapter.

The other OP application is the determination of the software unreliability (probability of failure) from the defects using an EFSM. In this study, the defects are propagated by using an EFSM. This method proceeds in three stages:

1. Construction of an EFSM representing a user's requirements and embedding a user's profile information. The OP is represented as the set of probabilities of the transitions.
2. Mapping of the defects to this model and the actual tagging of the states and transitions.
3. Executing the model to evaluate the impact of the defects identified by the TestMaster test generator using Full Coverage.

The Full Coverage generates all paths and then paths with tagged defects are identified and their associated probabilities extracted. The sum of these probabilities is the failure probability per demand.

Some conditions in the EFSM are determined by multiple input variables. The determination of the likelihoods of these conditions from the input profile (contains likelihood for individual input) can be very complicated, especially if the individual inputs are statistically dependent. Not all likelihoods for individual inputs are required, especially in the software-reliability propagation study—only the likelihoods of the paths that traverse the defects are required. By using our method, one can improve the OP generation efficiency by simply not considering the non-defect-related transitions.

In summary, the OP is defined in both applications as the occurrence probability of *transitions* rather than the occurrence probability of *inputs*. Identification and exploration of the multiple transitions, termed "OP events" throughout this chapter, guide the construction of such OPs. It is worth noting that this method is within Musa's OP framework. The high-level principles are applicable to this study. The procedures for constructing the OP are discussed in detail in the following subsections.

4.4.2 Method for Identifying Infrastructure Inputs Related to the OP

As shown in the generic reactive system architecture, the normal operation of such a system heavily depends on the infrastructure inputs. In order to obtain a complete OP, each of the infrastructure inputs should be identified. The infrastructure inputs usually cannot be obtained from the field. This is simply because the failures of these hardware components are rare and hardly observed, sometimes even over their entire performance periods.

The six-step method discussed below was used to define the OP for the infrastructure inputs:

1. Collect required documents;
2. Construct the EFSM;
3. Identify the hardware-related OP events;
4. Identify the hardware components related to OP events identified in Step 3;
5. Model the OP events identified in Step 3 using fault trees;
6. Quantify the fault trees established in Step 5.

These six steps are explained in turn.

Step 1: Collect Required Documents:

The required documents are:

1. Requirements specifications for the system
2. Requirements specification for the application software
3. Requirements specification for the system software
4. Basic failure rate information

The requirements specifications documents clearly define the software functionality and the software-hardware interaction. These documents are used to construct the EFSM, to identify hardware related OP events, and to construct the fault tree. Failure rate databases were used to quantify the fault trees in Step 6.

Step 2: Construct the EFSM. The EFSM was constructed based on the requirements specifications. Figure 4.4 depicts an example EFSM based on the requirements given in Figure 4.5. A discussion on EFSM construction is presented in Chapter 5 and Appendix A. Please refer to [Savage, 1997] for an in-depth explanation.

"A code Checksum shall be performed in the 'Power-Up Self Tests' and 'On-Line Diagnostics' operations. This is done by adding all of the programmed address locations in PROM and comparing the final value to a preprogrammed checksum value. A code checksum is a calculated number that represents a summation of all of the code bytes. The code checksum shall be stored at the end of the PROM.

The test shall start by reading the program memory data bytes and summing all of the values. This process shall continue until all of the code memory locations have been read and a checksum has been generated.

The calculated value shall be compared to the reference checksum stored in RAM. If the values match, the algorithm shall update the status byte in the status table with the value 55H and increment the status counter by one count. If the checksums don't match, then BBH shall be written instead to the status byte and the status counter shall not be incremented."

Figure 4.5 Excerpt from the APP SRS

Step 3: Identify the hardware-related OP events. This can be done by scrutinizing all OP events to see if any transition condition relates to the hardware status. For instance, the OP event in Figure 4.3 is the occurrence of the multiple transitions after the state Test Results. All three transitions were hardware related and hence this OP event is hardware-related. In general, most system software OP events are hardware-related.

The example in Figure 4.4 is used to illustrate how to identify the hardware-related OP events. This EFSM is constructed based on the fragment of SRS in Figure 4.5. As a general rule, most hardware components and the application algorithms should be examined in most safety critical systems. These components include but are not limited to, RAM (Random Access Memory), ROM (Read-Only Memory), PROM, EEPROM (Electrical Erasable PROM), Data Bus Line and Address Bus Line, input and output devices (for instance, A/D (analog to digital), D/A (digital to analog) converters), etc. The software components are the calculation algorithm, the input reading algorithm and so on.

From Figure 4.4, the hardware-related OP events are identified and presented in Table 4.2.

Table 4.2 Identified Hardware-Related OP Events for PROM Diagnostics in the APP system

No.	OP Events
1	The probability of 55H being written into APP status table
2	The probability of BBH being written into APP status table
3	The probability of neither 55H nor BBH being written into APP status table

Step 4: Identify hardware components related to OP events identified in Step 3

The hardware components that contribute to the OP events in Table 4.2 can be either explicitly identified from the SRS in Figure 4.5, (for instance, the hardware component PROM is easily identified); or from background knowledge about the workings of the control system (for instance, the check-sum operation involves read/write activities), and the RAM that contains the intermediate results of the checksum. In principle, normally the components under examination plus the components involved in the process should be considered. The hardware components for each OP event listed in Table 4.2 were examined in turn. Table 4.3 summarizes the findings for the OP Event 1 and Table 4.4 for the OP events 2 and 3.

Table 4.3 Hardware Components Related to OP Event 1

No.	Requirements	Basic Components
1	This is done by adding all of the programmed address locations in PROM and comparing the final value to a preprogrammed checksum value. A code checksum is a calculated number that represents a summation of all of the code bytes. The code checksum shall be stored at the end of the PROM.	PROM
		RAM
		Components Involved in Read/Write Operation
		Register
2	The test shall start by reading the program memory data bytes and summing all of the values. This process shall continue until all of the code memory locations have been read and a checksum has been generated.	Components Involved in Read/Write Operation
		Register
3	The calculated value shall be compared to the reference checksum stored in RAM.	RAM
		Components Involved in Read/Write Operation
		Register
4	If the values match, the algorithm shall update the status byte in the APP status table with the value 55H and increment the status counter by one count.	RAM
		Components Involved in Read/Write Operation
		Register
5	If the checksums don't match, then BBH shall be written instead to the status byte and the status counter shall not be incremented.	RAM
		Components Involved in Read/Write Operation
		Register

Table 4.4 Basic Components for Events 2 and 3

Event No.	Event	Basic Components
2	The probabilities of BBH is written into the APP status table	PROM
		RAM
		Components Involved in Read/Write Operation
		Register
3	The probabilities of neither 55H nor BBH is written into the APP status table	RAM
		Components Involved in Read/Write Operation
		Register

Step 5: Model the OP events identified in Step 3 using fault trees

Fault tree analysis is a mature technique widely used in the reliability and risk analysis fields. This technique is restricted only to the identification of the system elements and events that lead to one particular undesired failure. The undesired failure event appears as the top event, and this is linked to more basic fault events by logic gates. In this study the fault tree is used to model the OP events.

For example, the fault trees for events 2 and 3 are shown in Figure 4.6 and Figure 4.7 respectively. The PROM test result is BBH if any of the following four events occur: PROM fails, the RAM that contains the intermediate checksum results fails, the R/W operation fails (due to control bus, data bus, or address bus failures), or the Central Processing Unit (CPU) fails.

Step 6: Quantify fault trees established in Step 5

The basic events presented in Figure 4.6 and Figure 4.7, such as RAM fails, and PROM fails, need to be quantified. The ideal solution is to obtain failure rate information from the hardware manufacturer. This approach normally does not work due to the proprietary nature of such information. Some public databases, such as the RAC database [RAC, 1995], MIL-HDBK-217 [MIL, 1995] and the Nuclear Regulation Commission (NRC) database [Poloski, 1998, NUREG/CR-5750] can be used for the probabilistic modeling of digital systems. The use of such databases may lead to sacrificing the precision of the data. The failure rate for a specific component may not be found but information for similar hardware may be available. For instance, the specific RAM used in APP cannot be found in those databases. A failure rate for a general RAM device is used instead. Table 4.5 shows the failure rate of the APP hardware components.

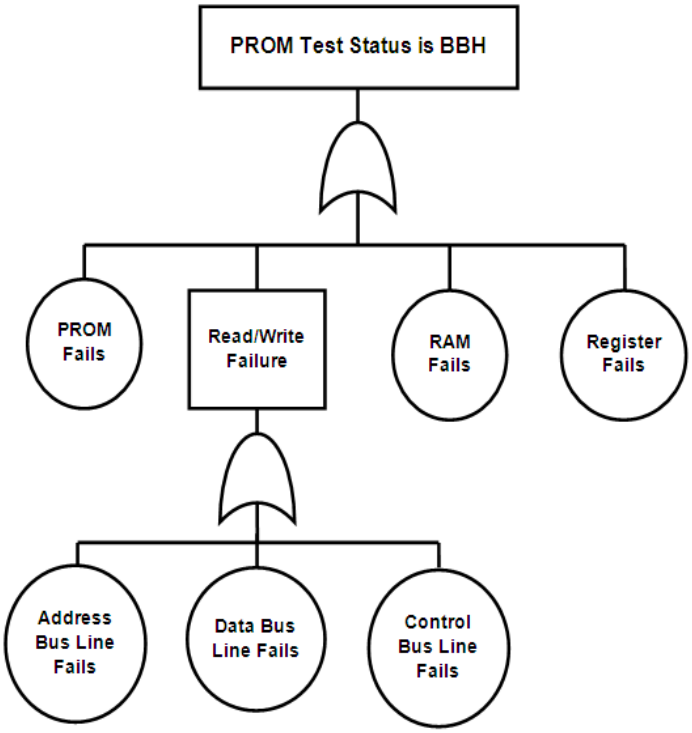


Figure 4.6 Fault Tree for Event 2

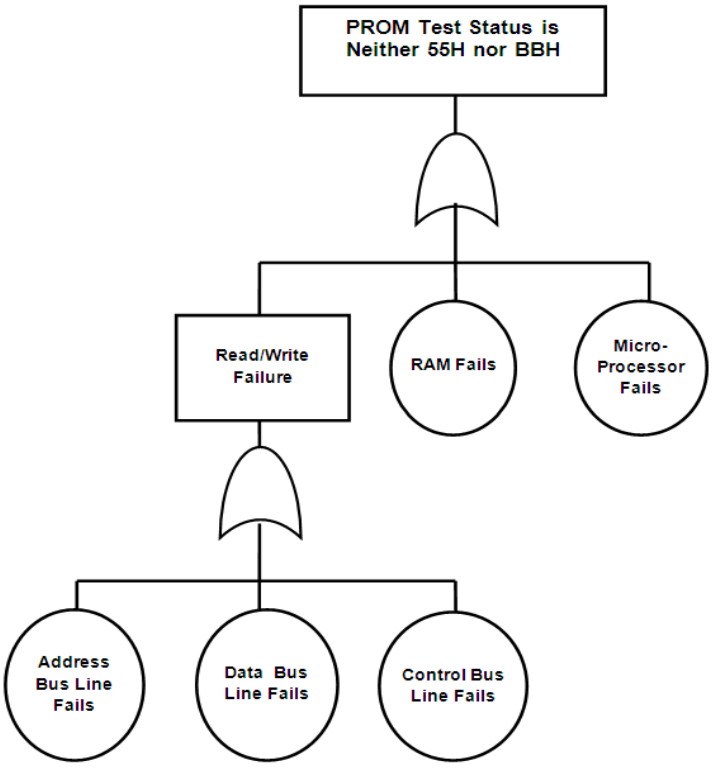


Figure 4.7 Fault Tree for Event 3

Table 4.5 Failure Rate for APP Hardware Components

Hardware Components	Description	Sub-Components Failure Rate (failure/hour)	Components Failure Rate (failure/hour)
RAM	8K byte	3.3E-7	3.3E-7
DPM	Dual Port RAM	1.7E-8	1.7E-8
PROM	64K byte	2.6E-8	2.6E-8
EEPROM	64K byte	2.4E-9	2.4E-9
CPU register	N/A	6.1E-8	6.1E-8
Latch	N/A	1.2E-8	1.2E-8
Address bus line	Line Bus Driver	4.6E-7	5.22E-7
	Line Bus Receiver	6.2E-8	
Data bus line	Line Bus Driver	4.6E-7	5.22E-7
	Line Bus Receiver	6.2E-8	
Control line	Line Bus Driver	4.6E-7	5.22E-7
	Line Bus Receiver	6.2E-8	
MUX	For analog input	3.3E-8	3.3E-8

The results of this step are summarized in Table 4.6 and Table 4.7.

Table 4.6 OP Events Quantification Results

Events	Hardware Components	Failure Rate (failure/hour)	Resources	Results
The probability of BBH is written into the APP status table.	PROM	λ_1 = 2.6E-8	NUREG/CR-5750	7.13×10^{-5}/demand
	RAM	λ_2 = 3.3E-7	NUREG/CR-5750	
	Components Involved in Read/Write Operation	λ_3 = 1.6E-6	NUREG/CR-5750	
	Microprocessor	λ_4 = 3.3E-8	NUREG/CR-5750	
The probability of neither 55H nor BBH is written into the APP status table.	RAM	λ_2 = 3.3E-7	NUREG/CR-5750	7.03×10^{-5}/demand
	Components Involved in Read/Write Operation	λ_3 = 1.6E-6	NUREG/CR-5750	
	Microprocessor	λ_4 = 3.3E-8	NUREG/CR-5750	

Table 4.7 Operational Profile for APP PROM Diagnostics Test

No.	Event	Operational Profile (per demand)
1	PROM Test Status Flag is 55H	$P_1 = 1 - P_2 - P_3 = 0.9998584$
2	PROM Test Status Flag is BBH	$P_2 = 7.13\times10^{-5}$
3	PROM Test Status Flag is neither 55H nor BBH	$P_3 = 7.03\times10^{-5}$

It should be noted that simply using the failure rate data from the databases is based on the assumption that the infrastructure inputs related to hardware components have not been replaced. If any hardware component has been replaced, the classical renewal theory should be applied to obtain a more accurate OP. For the case of the APP system, as will be stated later in Chapter 18, some hardware components such as EEPROM, AVIM (Analog Voltage Isolation Process) and 5V DC regulator had been replaced. Thus, the renewal theory should be incorporated to the OP estimation.

For instance, the EEPROM of the APP module used in a power plant unit had been replaced by a new EEPROM. The old EEPROM had been deployed for 77,040 hours and the new EEPROM was deployed for 18,000 hours. The failure rate information given in the databases is an average value (2.4×10^{-9} failure/hour). In this particular study, the estimation of reliability is on a per demand basis. If one neglects the occurrence of this replacement and assumes the cycle time for one calculation is 0.129 s, the probability of failure per demand is:

$$\Pr(EEPROM) = \frac{2.4 \times 10^{-9} failure/hr}{3600\ s/hr} \times 0.129 \frac{s}{demand} = 8.6 \times 10^{-14} failure/demand$$

If the replacement is taken into account, the average failure rate throughout the entire deployment period can be roughly estimated as:

$$\lambda_{ave} = -\frac{\ln\left(e^{-\lambda t_2}\right)}{t_1 + t_2} = \frac{2.4 \times \frac{10^{-9} failure}{hr} \times 18{,}000 hr}{(77{,}040\ hr + 18{,}000\ hr)} = 5.61 \times 10^{-10} failure/hour$$

Therefore the probability of failure per demand of the EEPROM can be updated to:

$$\Pr(EEPROM)' = \frac{5.61 \times 10^{-10} failure/hr}{3600\ s/hr} \times 0.129\ s/demand$$
$$= 2.01 \times 10^{-14} failure/demand$$

4.4.3 Estimating the Plant Inputs Based on Plant Operational Data

The plant inputs are gathered from the field through sensors. 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 both normal and abnormal data are available. In the case of the APP, "normal data" corresponds to situations under which the reactor operates within the power distribution envelope

shown in Figure 4.8; "abnormal data" corresponds to situations under which the data is outside the power distribution envelope. The truth, however, is that abnormal conditions are extremely rare.

The following steps describe the general procedure used to estimate the OP of plant inputs based on plant operational data.

1. Construct the EFSM for the application software. The algorithm used in the application software is given in Figure 4.9. In other words, if the power and power distribution (as represented by neutron flux measurements) is outside the power distribution envelope, the application software trips; otherwise it does not. The notions in Figure 4.9 are: *DF* is the measured neutron flux imbalance, *P* is the reactor thermal power, T_T is the maximum reactor thermal power, B_1, B_2, B_3, B_4, M_1 and M_2 are setpoints (coefficients). The corresponding EFSM is shown in Figure 4.8.

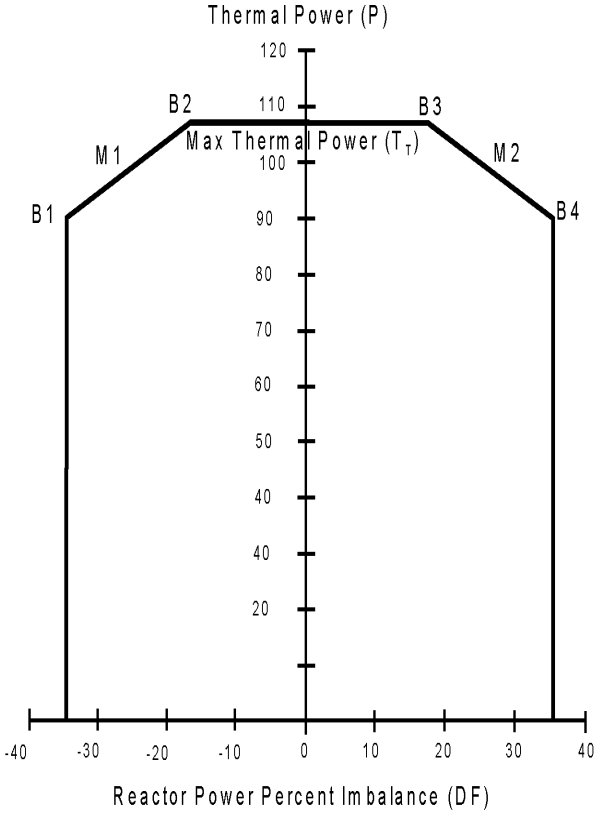


Figure 4.8 Barn Shape of the Power Distribution Trip Condition

2. Identify the OP events. The OP events and the associated conditions are defined in Figure 4.9. The results are presented in Table 4.8.

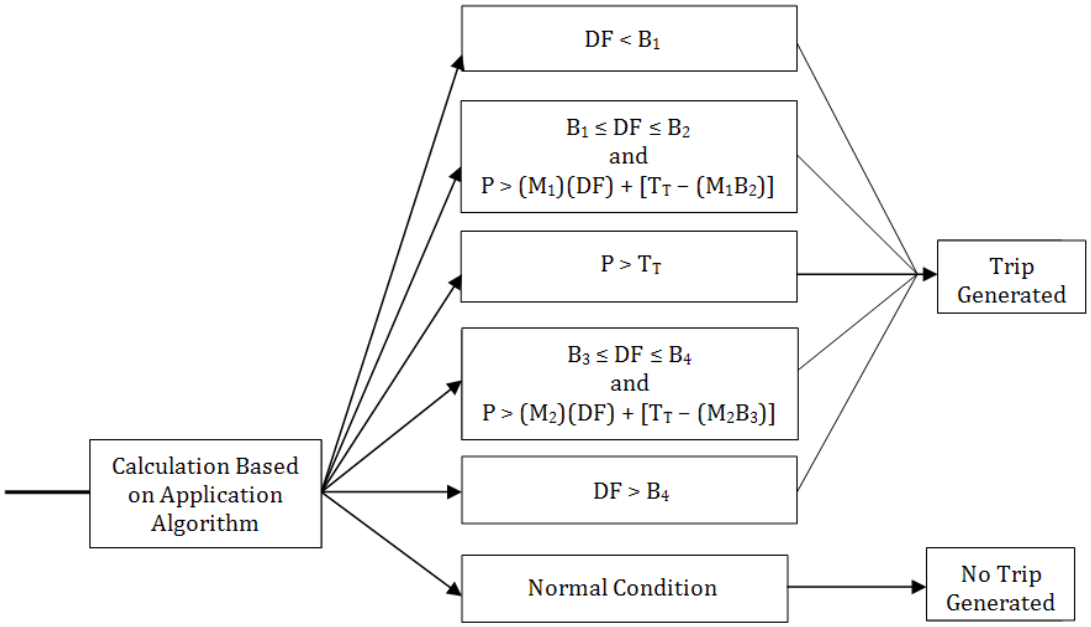


Figure 4.9 EFSM for APP Application Software

Table 4.8 APP Application Software Algorithm

Event	Condition
1	$DF < B_1$
2	$B_1 \leq DF \leq B_2$ and $P > (M_1)(DF) + [T_T - (M_1)(B_2)]$
3	$P > T_T$
4	$B_3 \leq DF \leq B_4$ and $P > (M_2)(DF) + [T_T - (M_2)(B_3)]$
5	$DF > B_4$
6	Normal condition

3. Derive the data sets representing individual OP event's conditions from the normal field operation data.

UMD obtained a data set that contained eleven-years of normal operational data (hour by hour) from a nuclear power plant. There were 88,418 distinct data records. Each record included the total reactor coolant flow, the neutron detector flux difference, and other critical plant parameters. An example of such records is shown in Figure 4.10.

1		NI 5 PR	RPS CH A TOTAL	RC LOOP A	RC LOOP B	NI 5	CORE THERMAL	INCORE
2		FLUX	RCS FLOW	FLOW 1	FLOW 1	DETECTOR	POWER BEST	IMBALANCE
3		%	KLB/HR	KLB/HR	KLB/HR	FLUX DIFF %	% FP	% FP
4	01-Jan-96 00:00:00	99.4332962	136655.6719	68831.1094	67725.85938	-4.119585991	100.0942612	-3.7107456
5	01-Jan-96 01:00:00	99.39963531	136894.75	68856.8047	67569.375	-4.06170845	99.78861237	-3.699759
6	01-Jan-96 02:00:00	99.84185028	137266.8906	69044.8594	67422.30469	-3.657032967	100.012558	-3.4214301
7	01-Jan-96 03:00:00	99.37675476	136541.1094	68365.0938	67565.50781	-3.931015015	99.88789368	-3.60271
8	01-Jan-96 04:00:00	99.28988647	136831.0313	69575.6016	67560.46875	-3.917248249	100.2363892	-3.5972166
9	01-Jan-96 05:00:00	99.29319	136203.5469	68932.3828	67769.39844	-3.925071001	99.91703796	-3.5990479
10	01-Jan-96 06:00:00	99.56278992	136854.8906	69435.7109	67980.07031	-3.5662148	99.71530151	-3.4049501
11	01-Jan-96 07:00:00	99.47690582	136464.7656	68899.6016	66734.82813	-3.512936831	100.1186981	-3.33903
12	01-Jan-96 08:00:00	99.39264679	136480.8906	68850.3672	67628.83594	-3.506470919	100.3689346	-3.3079011
13	01-Jan-96 09:00:00	99.36070251	136956.2656	68781.6875	67354.95313	-3.530076027	99.77853394	-3.304239
14	01-Jan-96 10:00:00	99.33335114	136453.5781	69308.0391	67434.90625	-3.530076027	100.046608	-3.2950835
15	01-Jan-96 11:00:00	99.63507843	136483.75	69248.3125	67804.10938	-3.467787743	99.76218414	-3.2145145
16	01-Jan-96 12:00:00	99.46977234	136752.9219	69405.1328	67724.05469	-3.906979561	100.1316299	-3.5715811
17	01-Jan-96 13:00:00	99.51477814	136789.8438	69338.5078	67492.17969	-3.970458031	100.2498245	-3.6521499
18	01-Jan-96 14:00:00	99.47581482	136703.7188	69191.4531	67609.33594	-4.12518692	100.0655746	-3.7235634
19	01-Jan-96 15:00:00	99.47998047	136802.5781	69593.3906	67285.9375	-4.19931078	99.86380005	-3.8151188
20	01-Jan-96 16:00:00	99.43876648	136523.8594	69205.2578	68017.96094	-4.310226917	99.90866852	-3.8792078

where:

NI 5 PR FLUX	the current flux percentage
RPS CH A TOTAL RCS FLOW	the total reactor coolant system flow of reactor protection system channel A
RC LOOP A FLOW	the flow of reactor coolant loop A;
RC LOOP B FLOW	the flow of reactor coolant loop B;
NI 5 DETECTOR FLUX DIFF	the detector flux difference;
CORE THERMAL POWER BEST	current thermal power percentage;
INCORE IMBALANCE	the indicator of core delta flux.

Figure 4.10 Example of Plant Operational Data

After a careful study of the data set, UMD identified three classes of data that could not be treated as normal operational data. The three classes are described in turn:

1) Outage Data.

Data recorded during outages cannot be considered an integral part of the normal operational data set. Indeed, data recorded during these time periods is out-of-range and basically meaningless. The plant owner provided UMD with outage start date and end date information for the power plant, as shown in Table 4.9. There are 15,094 records falling within these time intervals.

Table 4.9 Outage Information for Plant

From	To
4/27/95 3:59 AM	5/10/95 10:30 PM
11/2/95 1:00 AM	12/10/95 4:58 AM
2/28/96 9:02 AM	3/1/96 1:59 PM
10/4/96 12:33 AM	2/12/97 8:54 PM
3/28/97 2:42 PM	4/11/97 4:12 PM
6/13/97 4:30 PM	7/3/97 2:52 PM
9/18/97 3:41 AM	12/24/97 11:55 PM
12/28/97 3:55 PM	12/31/97 11:59 PM
1/1/98 12:01 AM	2/11/98 3:04 AM
2/15/98 3:47 AM	2/19/98 12:38 AM
8/8/98 9:11 AM	8/25/98 8:46 PM
5/21/99 1:18 AM	7/3/99 5:00 PM
2/17/00 3:35 PM	3/2/00 2:10 AM
11/24/00 1:10 AM	1/9/01 11:48 PM
3/23/02 4:48 PM	4/26/02 11:46 AM
9/20/03 2:11 PM	12/13/03 2:00 AM
12/18/03 8:00 AM	1/1/04 2:00 PM
4/9/05 9:27 AM	5/11/05 9:20 AM
10/7/06 12:00 AM	11/30/06 12:00 AM

2) Missing Data

Some operational data was missing from the data set. This data typically was labeled: “bad input,” “shut down,” or “under range.” The plant APP system expert stated that these records likely corresponded to data recorded during maintenance or test activities. Therefore, the data cannot be considered an integral part of the normal operational data either. The number of data records affected was 792.

3) Aberrant Data

There were 21 strange records either with a negative reactor coolant flow value or an extremely large reactor flow value (of the order of 10^{26} which far exceeds the normal values that are

typically of the order of 10^5). This data was suspicious, so UMD eliminated this data from consideration[9].

The total number of operational data points, with each data point representing the equivalent of one hour of operating history is:

$$88{,}418 - 15{,}094 - 792 - 21 = 72{,}511 \text{ hours}$$

The number of data points falling within each domain (OP event) was then counted and is reported in Table 4.10.

Table 4.10 Number of Trip Data Sets Falling within Each Domain

Event	Condition	Number of Data Sets
1	$DF < B_1$	2
2	$B_1 \leq DF \leq B_2$ and $P > (M_1)(DF) + [T_T - (M_1)(B_2)]$	0
3	$P > T_T$	7
4	$B_3 \leq DF \leq B_4$ and $P > (M_2)(DF) + [T_T - (M_2)(B_3)]$	0
5	$DF > B_4$	1
6	Normal condition	72,501

It is clear that for conditions 1, 3 and 5, the probability of occurrence of the condition can be estimated as the number of data points over the total number of operational data points. Therefore, the probability of occurrence of conditions 1, 3 and 5 is respectively:

$$\frac{2 \text{ trips}}{72{,}511 \text{ hr}} = 2.758 \times 10^{-5} \text{ trip/hr}$$

$$\frac{7 \text{ trips}}{72{,}511 \text{ hr}} = 9.654 \times 10^{-5} \text{ trip/hr}$$

$$\frac{1 \text{ trip}}{72{,}511 \text{ hr}} = 1.379 \times 10^{-5} \text{ trip/hr}$$

[9] It would seem that one of the reasons that the records may show these "strange records" might be a failure of the hardware or software in the system or failure of systems that provide inputs to the system. These "strange records" may also reflect additional maintenance/outage data.

Because there are no data points (events) within the domains of conditions 2 and 4, one could conclude that the probabilities of occurrence of these two conditions are zero. However, to obtain a more accurate estimation, a statistical extrapolation method can be applied. The data sets that can be used for the extrapolation are those in area 1 and area 2 in Figure 4.11. The number of data points in area 1 is forty five and in area 2 is one.

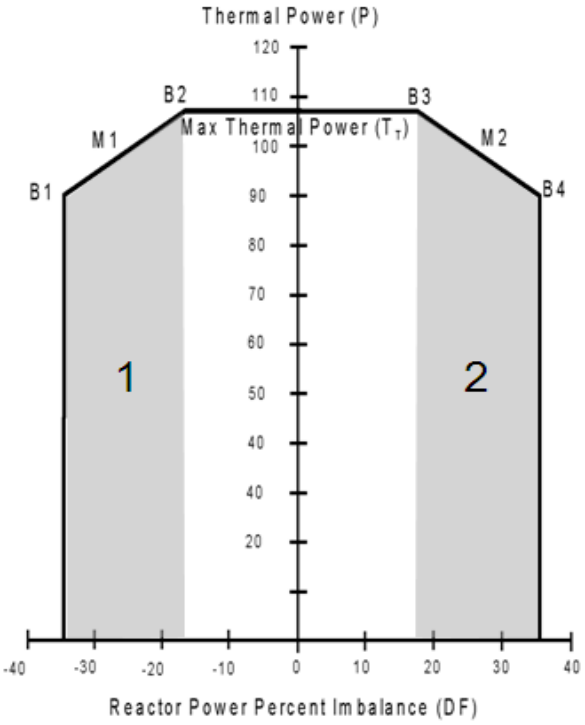


Figure 4.11 Data used for Statistical Extrapolation

The Shapiro-Wilk test is applied to test the normality of the 45 data points in Area 1. This test evaluates the null hypothesis H_0 (i.e. data set $(x_1, x_2, \dots, x_n)$ comes from a normally distributed population) using the test statistics:

$$W = \frac{\left(\sum_{i=1}^{n} a_i x_{(i)}\right)^2}{\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{4.5}$$

where coefficients a_i are functions of the expected value and covariance matrix of the order statistics of random variables from the standard normal distribution and would be fixed for a given sample size and $(x_1, x_2, \dots, x_n)$ are ordered sample values.

The guiding principle of the test is to construct a regression of ordered sample values on corresponding expected normal order statistics, which should be linear for a data set from a normally distributed population. W represents the linear fit of the regression, i.e., the closer W is to a value of 1, the more evidence exists that $(x_1, x_2, \dots, x_n)$ are normally distributed.

In the test, the p-value illustrates the probability of obtaining a particular value of the test statistics or a more extreme value of this statistic under the null hypothesis. As shown in Table 4.11, the possibility of observing $W = 0.969387$ or smaller is 27.51% (larger than 10%). This result, which includes both the value of p and that of W, offers sufficient evidence that the null hypothesis is reasonable. Consequently, the hypothesis that the data points come from a normal distribution cannot be rejected.

Table 4.11 Tests for Normality Results

Tests for Normality				
Test	Statistic		P Value	
Shapiro-Wilk	W	0.969387	$\Pr < W$	0.2751

For this distribution, the mean of the data points is 30.32 and the standard deviation is 15.29. The extrapolation result is:

$$\phi\left(\frac{0 - 30.32}{15.29}\right) = 0.023$$

where ϕ is the cdf of the standard deviation.

Therefore, the probability of occurrence of condition 2 is a conditional probability calculated as:

$$\left(\frac{45 \text{ data points}}{72{,}511 \text{ hr}}\right)(0.023) = 1.427 \times 10^{-5} \text{ data points/hr}$$

For condition 4, obviously, the fact that there exists only 1 data point in area 2 is not sufficient to perform a valid statistical extrapolation. Traditionally, the maximum likelihood and unbiased estimate of the failure rate $\hat{\lambda}$ is given in Equation 4.6 [Ireson, 1966] if we assume r failures are observed in T hours of operating time:

$$\hat{\lambda} = \frac{r}{T} \tag{4.6}$$

A common solution to failure rate estimation when no failure event has been observed is to take one half as the numerator (r) in Equation 4.6 [Welker, 1974]. Thus, the probability of the occurrence of condition 4 can be roughly estimated as 0.5 data points/72,511 hr = 6.9×10^{-6} data point/hr.

The analysis presented above yields the OP for the APP application software summarized in Table 4.12.

Table 4.12 Operational Profile for APP Application Software

Event	Condition	Probability	Probability
1	$DF < B_1$	2.758×10^{-5}	9.8828×10^{-10}/demand
2	$B_1 \leq DF \leq B_2$ and $P > (M_1)(DF) + [T_T - (M_1)(B_2)]$	1.427×10^{-5}	5.1134×10^{-10}/demand
3	$P > T_T$	9.654×10^{-5}	3.4594×10^{-9}/demand
4	$B_3 \leq DF \leq B_4$ and $P > (M_2)(DF) + [T_T - (M_2)(B_3)]$	6.9×10^{-6}	2.4725×10^{-10}/demand
5	$DF > B_4$	1.379×10^{-5}	4.9414×10^{-10}/demand
6	Normal condition	0.99984	0.9999999943/demand

4.5 References

[APP, 1] *APP Instruction Manual*.

[Chruscielski, 1997] K. Chruscielski and J. Tian. "An operational profile for the Cartridge Support Software," in *Proc. The Eighth International Symposium On Software Reliability Engineering*, 1997, pp. 203–212.

[Elbaum, 2001] S. Elbaum and S. Narla. "A Methodology for Operational Profile Refinement," in *Proc. Reliability and Maintainability Symposium*, 2001, pp. 142–149.

[Gittens, 2004] M. Gittens, H. Lutfiyya and M. Bauer. "An Extended Operational Profile Model," in *Proc. 15th International Symposium on Software Reliability Engineering*, 2004, pp. 314–325.

[IEEE, 1991] "IEEE Standard Glossary of Software Engineering Terminology," IEEE Std. 610.12-1990, 1991.

[Ireson, 1966] W.G. Ireson. *Reliability Handbook*. New York, NY: McGraw Hill, Inc., 1966.

[Kumamoto, 1996] H. Kumamoto and E.J. Henley. *Probabilistic Risk Assessment and Management for Engineers and Scientists*. IEEE Press, 1996.

[Li, 2003] B. Li et al. "Integrating Software into PRA," in *Proc. 14th International Symposium on Software Reliability Engineering*, 2003, pp. 457.

[Li, 2004] M. Li et al. "Validation of a Methodology for Assessing Software Reliability," in *Proc. 15th IEEE International Symposium of Software Reliability Engineering*, 2004, pp. 66–76.

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

[Musa, 1992] J. Musa. "The Operational Profile in Software Reliability Engineering: an Overview," in *Proc. 3rd International Symposium on Software Reliability Engineering*, 1992.

[Ouab, 1995] F. Ouabdesselam and I. Parissis. "Constructing Operational Profiles for Synchronous Critical Software," in *Proc. 6th International Symposium on Software Reliability Engineering*, 1995.

[Poloski, 1998] J.P. Poloski et al. "Rates of Initiating Events at U.S. Nuclear Power Plants: 1987-1995," NRC NUREG/CR-5750, 1998.

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

[Sandfoss, 1997] R.V. Sandfoss and S.A. Meyer. "Input Requirements Needed to Produce an Operational Profile for a New Telecommunications System," in *Proc. 8th International Symposium on Software Reliability Engineering*, 1997.

[Savage, 1997] P. Savage, S. Walters and M. Stephenson. "Automated Test Methodology for Operational Flight Programs," in *Proc. IEEE Aerospace Conference,* 1997.

[Shukla, 2004] R. Shukla, D. Carrington and P. Strooper. "Systematic Operational Profile Development for Software Components," in *Proc. 11th Asia-Pacific Software Engineering Conference*, 2004.

[Smidts, 2004] C. Smidts and M. Li, "Validation of a Methodology for Assessing Software Quality," NRC, Office of Nuclear Regulatory Research,

Washington DC NUREG/CR-6848, 2004.

[TestMaster, 2004] "TestMaster Reference Guide," Teradyne Software & System Test, Nashua, NH, 2000.

[TestMaster, 2004] "TestMaster User's Manual," Teradyne Software & Systems Test, Nashua, NH, 2004.

[USNRC, 1995] "Use of Probabilistic Risk Assessment Methods in Nuclear Regulatory Activities," USNRC, vol. 60, 1995.

[Welker, 1974] E.L. Welker and M. Lipow. "Estimating the Exponential Failure Rate from Data with No Failure Events," in *Proc. Annual Reliability and Maintainability Conference*, 1974.

[Widmaier, 2000] J.C. Widmaier, C. Smidts and X. Huang. "Producing more Reliable Software: Mature Software Engineering Process vs. State-of-the-Art Technology," presented at International Conference on Software Engineering, 2000.

[WinRunner, 2001] "WinRunner Test Script Language Reference Guide," Mercury Interactive Corp., Sunnyvale, CA, 2001.

5. RELIABILITY ESTIMATION CONSIDERATIONS

This chapter establishes a basis for estimating software reliability from the number of defects remaining in the software. The concept of fault exposure ratio, K, introduced by Musa [Musa, 1987] is revisited. A new concept, entitled "new K" (vK), is proposed to replace Musa's fault-exposure ratio. This vK is based on an analytical analysis of fault-propagation phenomena and, as such, eliminates the effort of estimating some parameters (such as linear execution time) using Musa's method.

5.1 Estimation of Reliability Based on Remaining Known Defects

Generally, software fails due to defects introduced during the development process. A defect leads to a failure if the following occurs: 1) the defect is triggered (executed), 2) such execution modifies the computational state, and 3) the abnormal state propagates to the output and manifests itself as an abnormal output, i.e., a failure [Voas, 1992] [Li, 2004].

The "Propagation, Infection, Execution" (PIE) concept [Voas, 1992] is borrowed to describe this failure mechanism. The acronym *PIE* corresponds to the three program characteristics above: the probability that a particular section of a program (termed "location") is executed (termed "execution" and denoted as *E*), the probability that the execution of such section affects the data state (termed "infection" and denoted as *I*), and the probability that such an infection of the data state has an effect on program output (termed "propagation" and denoted as *P*). Thus the failure probability (unreliability) p_f is given in Equation 5.1:

$$p_f = \int_i P(i) \times I(i) \times E(i) \tag{5.1}$$

where

$P(i)$ The propagation probability for the *i*-th defect
$I(i)$ The infection probability for the *i*-th defect
$E(i)$ The execution probability for the *i*-th defect

In the original *PIE* method, *P*, *I*, and *E* are statistically quantified using mutation [Voas, 1992]. However, this method is unable to combine the OP and unable to consider defects that do not appear in the source code (e.g., requirements errors like missing functions). In addition, the large amount of required mutants hinders the practical implementation of this method.

In this study, a simple, convenient, and effective method to solve this problem is proposed using an extended finite state machine model (EFSM) [Wang 1993]. 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.

The method proceeds in three stages:

1. Construction of an EFSM representing the user's requirements and embedding the user's OP information;
2. Mapping of the defects to this model and actual tagging of the states and transitions;
3. Execution of the model to evaluate the impact of the defects.

Assume a defective or faulty transition d (the transition that, when executed, leads to at least a faulty state in the system), and $P_d = (p_{d1}, p_{d2}, \ldots p_{dm})$ is the set of m input/output paths in the EFSM that traverse this defect. An input/output path is defined as a path in the EFSM model that starts from the start state at the very top level (denoted as top level 0) and ends with the final or exit state at level 0 and is the set of all the transitions along the path. Let $(p_{d(g)}) = (d_{g1}, d_{g2}, \ldots d_{gn})$ be the g-th input/output path consisting of n transitions and $Pr(p_{d(g)})$ be the probability of traversing the g-th path. The probability of failure caused by this defect can then be determined by:

$$p_f^* = \sum_{g=1}^{m} P_r(p_{d(g)}) \tag{5.2}$$

where:

$$P_r(p_{d(g)}) = \prod_{q=1}^{n(g)} \delta_{d(g)}(q) \tag{5.3}$$

and:

$\delta_{d(g)}(q)$	Conditional probability that the q-th transition is traversed in the g-th path
q	Transition index
g	Path index
$n(g)$	Number of transitions in the g-th path

Equations 5.2 and 5.3 also hold true if there are multiple defects M. In this case, these M defects first need to be mapped and tagged into the EFSM. P_d then becomes the set of paths encompassing M defects. The parameter m in (5.2) is then replaced with $m(M)$, number of input/output paths containing at least one of the M defects. This feature solves a critical problem in the software engineering literature: the interaction among multiple defects and its effect on the fault propagation process.

It should be noted that Equations 5.2 and 5.3 are based on an assumption that P and I are equal to 1. If this assumption does not hold, the EFSM model must be modified (refined to a lower level of modeling) in such a manner that P and I are equal to 1.

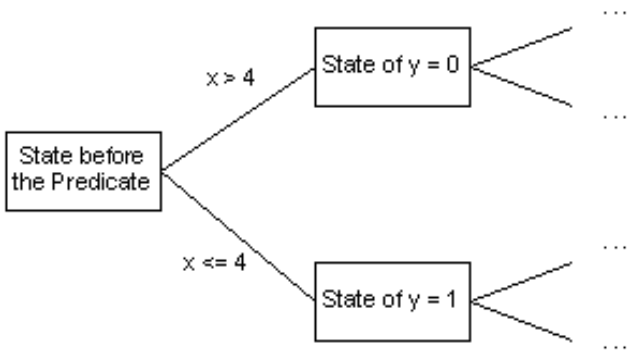
A defect does not infect and/or propagate if its execution is not triggered. For example, a correctly implemented code segment (written in C syntax) and its associated EFSM are given in Figure 5.1.

Assume that the real implementation contains a defect in the predicate: the threshold is 6 instead of 4. We deduce that if $x > 6$, then this defect will not infect the software state because y will be 0. Similarly, if $x \leq 4$, this defect will not infect the software execution since y will be 1. Only if $4 < x \leq 6$ will this defect infect the system by assigning y the value 1 instead of 0. Thus the condition under which the defect does not infect the software is $x > 6$ and $x \leq 4$. The faulty code and the decomposed EFSM are given in Figure 5.2. The bold branch indicates the defect

and its corresponding state and transition in the EFSM. By following the same principle and consideration, the non-propagation condition is identified and the EFSM is decomposed in a manner that assures *I* equals 1. The process of decomposition of the EFSM may also be stopped whenever conservative estimates of probabilities of failure become acceptable.

Correct Code **EFSM**

```
…
if (x > 4)
   y = 0;
else
   y = 1;
…
```



Correct Code and Its EFSM

Faulty Code **EFSM**

```
…
if (x > 6)
   y = 0;
else
   y = 1;
…
```

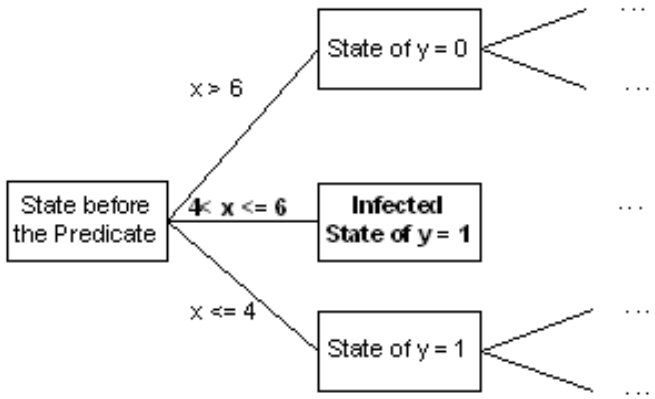


Figure 5.1 Faulty Code and Its EFSM

The following measures utilize EFSM to propagate the defects found during the measurement process: Completeness (Chapter 9), Defect Density (Chapter 12), Requirements Traceability (Chapter 16), and Test Coverage (Chapter 17).

5.2 Reliability Estimation from the Unknown Defects

The method described in Section 5.1 is limited to known defects, i.e., defects found by inspection and testing. The known defects discovered by inspection relate to the measures Completeness,

Defect Density, and Requirement Traceability. The Test Coverage measure also uses this method to obtain the fault exposure ratio for a specific application through propagation of defects found by testing. Unknown defects that may remain in the application will contribute to application failure, and not accounting for these defects will result in an overestimation of reliability. Therefore, to improve this method, one needs to: 1) estimate the number of unknown defects remaining in the application and 2) investigate the unknown defects' contribution to the probability of failure. In this section the number of defects remaining (both known and unknown) is obtained by means described in the following chapters and then used to analytically estimate the reliability.

5.2.1 Reliability Estimation from the Number of Defects Remaining

Once the number of defects remaining is determined, the software reliability of the system can be estimated using Equations 5.4 and 5.5. Musa [Musa, 1987] proposed the concept of fault exposure ratio K and its relation to λ (the failure rate) and N (the number of defects remaining—including both known and unknown unresolved defects).

$$\lambda = \frac{K}{T_L} N \tag{5.4}$$

Then, the software reliability at time t is[10]:

$$R(t) = e^{-\frac{K}{T_L} N t} \tag{5.5}$$

Where:

K	Fault exposure ratio, the average value is 4.2×10^{-7} [Musa, 1987]
T_L	Linear execution time, s.
N	Number of defects
t	Execution time, s

T_L is the linear execution time, defined as the execution time of the software if each statement executes only once.

As seen from Equation 5.4, the failure rate λ is constant if no change is made to the software. The failure rate λ will vary during software development phases (such as testing) as faults are being introduced, detected, and/or removed (thus N and K will change). It will also vary as the code is modified structurally (thus K or T_L will change). On the other hand, λ will not vary during operation when the code is frozen.

The value of K has become obsolete for modern safety-critical systems. For example, if one evaluates safety critical software reliability within a one-year period using Equation 5.5, the time t is roughly 3.15×10^7 seconds. For a real-time system, T_L is normally less than one second (e.g.,

[10] "If a program has been released and no changes will be made, the failure intensity of this program is a constant. For the basic execution time model and the logarithmic Poisson model, the failure process is then described by a simple homogeneous Poisson process with failure intensity as a parameter. The number of failures in a given time period follows a Poisson distribution. The failure intervals thus follow an exponential distribution [Musa, 1998]." It should be pointed out that although there are quite a few software reliability models available, Musa's basic execution time model (used here) is one of the two models (with Musa's logarithmic Poisson model) that have been fully validated and confirmed through many practical applications [Dale, 1982] [Derriennic, 1995] [Farr, 1996] [Jones, 1991] [Musa, 1975] [Malaiya, 1992].

0.129 s). Furthermore, assuming only one fault remains in the code, the reliability is calculated as:

$$R(t) = e^{-\frac{K}{T_L}Nt} = e^{-\frac{4.2\times10^{-7}}{1}\times1\times3.15\times10^{7}} = e^{-13.23} = 1.8\times10^{-6}$$

This implies that software with only one fault remaining almost definitely fails at the end of one year. This conclusion contradicts existing power plant field data.

To address this contradiction between theory and evidence, the concept of vK (new K) is proposed to simplify Equation 5.5:

$$vK = \frac{K}{T_L}t \tag{5.6}$$

where t is the execution time. The execution time is either the time-per-demand or the length of a year. The latter is normally used in the nuclear industry.

Both K and vK will vary as a function of life-cycle phases because the structural properties of the code and the number of defects in the code changes.

Then the probability of failure (unreliability) simply becomes a function of the number of defects (assume the failure rate is very small for safety critical systems):

$$p_f = 1 - e^{-vKN} \approx vK \times N \tag{5.7}$$

It is worth noting that vK is an average value and can be analytically estimated from the known defects remaining in the software using the EFSM technique. The precision of the estimation statistically depends on the number of defects propagated and the time over which the defects existed in the life cycle. Defect locations also influence the value of vK. In this study it is assumed the average value of vK obtained from the known defects represents the value of the vK of the unknown defects. This is an assumption which needs further study to validate it or to find a way to improve it. Some studies [Lait, 1998] have demonstrated that different defect detection techniques may reveal different types of defects. It is reasonable to assume that the combination of different defect detection techniques may reveal the majority of defects and thus increase the validity of this assumption.

The remaining defects are classified into two groups: known and unknown. Let us assume that the number of known remaining defects is N_1, and the number of unknown defects is N_2. Thus, the total number of remaining defects is:

$$N = N_1 + N_2 \tag{5.8}$$

The unreliability contributed from N_1 and N_2 are:

$$p_{f1} = vK \times N_1 \tag{5.9}$$

and

$$p_{f2} = vK \times N_2 \tag{5.10}$$

respectively.

The N_1 defects can be mapped into the EFSM and thus p_{f1} can be propagated. The average vK is obtained from Equation 5.9 as:

$$vK = \frac{p_{f1}}{N_1} \tag{5.11}$$

Equation 5.7 can then be written as:

$$p_f = p_{f1} + \frac{p_{f1}}{N_1} N_2 \tag{5.12}$$

where N_1 is the number of known but unresolved defects and N_2 is the number of unknown and unresolved defects. Note that when no known, unresolved defects exist, one can still apply the technique using the last known and resolved defects and obtain a conservative estimation of p_f.

5.3 References

[Dale, 1982] C.J. Dale. "Software Reliability Evaluation Methods," Report ST26750. British Aerospace, 1982.

[Derriennic, 1995] H. Derriennic and G.L. Gall. "Use of Failure-Intensity Models in the Software-Validation Phase for Telecommunications." *IEEE Transactions on Reliability*, vol. 44, pp. 658–665, 1995.

[Farr, 1996] W. Farr. "Software Reliability Modeling Survey," in *Handbook of Software Reliability Engineering*, M. Lyu, Ed. New York, NY: McGraw-Hill, 1996.

[Jones, 1991] W.D. Jones. "Reliability Models for Very Large Software Systems in Industry," in *Proc. 2nd International Symposium on Software Reliability Engineering*, 1991, pp. 35–42.

[Lait, 1998] O. Laitenberger. "Studying the Effects of Code Inspection and Structural Testing on Software Quality," in *Proc. 9th International Symposium on Software Reliability Engineering*, 1998.

[Li, 2004] M. Li et al. "Validation of a Methodology for Assessing Software Reliability," in *Proc. 15th IEEE International Symposium of Software Reliability Engineering*, 2004, pp. 66–76.

[Malaiya, 1992] Y.K. Malaiya, N. Karunanithi and P. Verma. "Predictability of Software Reliability Models." *IEEE Transactions on Reliability*, vol. R-41, pp. 539–546, 1992.

[Musa, 1975] J.D. Musa. "A Theory of Software Reliability and its Application." *IEEE Transactions on Software Engineering*, vol. SE-1, pp. 312–327, 1975.

[Musa, 1987] J.D. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Applications*. New York: McGraw-Hill, 1987.

[Voas, 1992] J.M. Voas. "PIE: A Dynamic Failure-Based Technique," *IEEE Transactions on Software Engineering*, vol. 18, pp. 717–27, 1992.

[Wang, 1993] C.J. Wang and M.T. Liu. "Generating Test Cases for EFSM with Given Fault Models," in *Proc. 12th Annual Joint Conference of the IEEE Computer and Communications Societies*, 1993.

6. BUGS PER LINE OF CODE

The goal of this measure is estimate the number of faults in a program module per line of code. This measure is simplistic and ignores many aspects of the software and its development, so it is not likely to be very accurate.

This measure can only be applied when source code is available. As listed in Table 3.3, the applicable life cycle phases for the BLOC measure are Coding, Testing, and Operation.

6.1 Definition

Gaffney [Gaffney, 1984] established that the total number of defects in the software (*F*) could be empirically expressed as a function of the number of lines of code. That is:

$$F = \sum_{i=1}^{N_m}\left(4.2 + 0.00155\sqrt[3]{S_i^4}\right) \tag{6.1}$$

where

i	The module index
N_m	The number of modules
S_i	The number of lines of code for the *i*-th module

Gaffney justified the power factor of 4/3 in [Gaffney, 1984] based on Halstead's formula [Halstead, 1977]. The coefficients of 4.2 and 0.0015 were estimated based on the Akiyama assembly code data [Halstead, 1977] [Gaffney, 1984]. The experts engaged in the NUREG/GR-0019 study [Smidts, 2000] concluded that these coefficients are meaningful for modern programming languages such as C or C++, but did not express confidence in this measure's ability to predict reliability and therefore ranked it very low. It is obvious that size is not the only factor that influences reliability. However, at this point, no validated model exists that includes additional factors (such as the developers' skill) in the BLOC model. As illustrated in Figure 1.1, such additional factors, if identified and validated, can be easily incorporated in to the RePS model and can be used as support measures. Since the current RePS from BLOC only considers size, its prediction ability is limited.

6.2 Measurement Rules

The BLOC definition identified two primitives in Equation 6.1: the module and the Lines of Code (LOC) for each module. The module index, however, is used to numerate the module and is not considered a primitive. The counting rules for the two primitives are described in turn in Section 6.2.1 and 6.2.2.

The counting rules have been customized to the specific language (C language) used in the APP development process. The software on the safety microprocessor 1 (μp1) and communication microprocessor (CP) were developed using the Archimedes C-51 compiler, version 4.23; the

software on safety microprocessor 2 (μp2) was developed using the Softools compiler, version 1.60f. Due to the obsolescence of these tools, the software was ported to the Keil PK51 Professional Developer's Kit and IAR EWZ80, version 4.06a-2, respectively. The major modifications are the replacements of some obsolete keywords with their equivalents in the new compilers. Consequently, the porting does not change the results.

6.2.1 Module

A module is defined as "an independent piece of code with a well-defined interface to the rest of the product" in [Schach, 1993]. IEEE [IEEE, 1990] defines module in the following two ways: "A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading," or "A logically separable part of a program." Gaffney [Gaffney, 1984], however, did not provide a clear definition but mentions a module as a "functional group."

The existence of multiple definitions of the module concept and the lack of consensus make its measurement problematic.

In the previous validation study [Smidts, 2004], the system under study was implemented using the C++ language. The researchers thus defined a class as a module since a class is a functional group, an independent piece of code with a well-defined interface to the rest of the product, and a logically separable part of a program.

In this study, the definition of a module needs to be modified because the system under study was coded using the C language. The individual file rather than the function is considered as a module due to the dependency among functions in a file introduced by global variables.

The APP software is composed of two types of user-defined files: the source file (.c file) and the header file (.h file). The .c file contains the major software implementation while the .h file mainly contains the declaration of (global) variables, the function prototypes (function declarations), and macros or inline functions. A header file cannot be considered individually to be a module because it does not provide any functionality. Rather, a .c file together with the .h files included in it becomes an independent piece of code with a well-defined interface to the rest of the product. As such, a module is defined as a .c file together with all the user-defined .h files it includes.

The counting rule for the module is to enumerate all .c files in the APP software package. The user-defined .h files need to be identified per .c file to facilitate the LOC counting.

6.2.2 LOC

The C language used in the APP software development is a super set of the ANSI C language. "Super set" means that additional features such as keywords are added into the standard ANSI C language to reflect the characteristics of the embedded system development. It is worth noting that development environments (C compilers) differ in terms of keywords.

The LOC counting is heavily language- and keyword-dependent. Because two C compilers were used in the APP development, the LOC measurement rules needed to encompass the difference. Because only a limited number of features are added into the two compilers, the most efficient way to conduct the measurement was to measure according to the ANSI C standard for the first round, and then identify all added features and count them separately in the second round.

The following counting rules reflect this idea: the general ANSI C counting rules are introduced and are followed by the individual compiler.

6.2.2.1 LOC Counting Rules for ANSI C

Rule 0: Logical statement in a module (a .c file plus all user defined .h files it includes) counts. Each statement counts 1. A statement normally ends with ";". Exceptions are specified below.

Rule 1: Statements that count
The "while" statement: starts with the keyword "while" and ends with the finish of the condition ")"
The "if" statement: starts with the keyword "while" and ends with the finish of the condition ")"
The "else" keyword followed by "if" statement
The definition of a function: ends with ")"
The "switch" statement: ends with ")"
The "case" statement: ends with ":"
The "default" statement: ends with ":"
The "for" statement: ends with ")"
Other statements: end with ";"

Rule 2: Statements that do not count
Blank lines
Comment: starts with /* and ends with */
Preprocessor directive: starts with # and ends with a hard return
The beginning of a statement block, the left bracket "{"
The end of a statement block, the right bracket "}"
The "else" keyword itself
Other statements that cannot be classified by Rule 1 and Rule 2. These must be part of a statement that spans multiple physical lines.

6.2.2.2 LOC Counting Rules for Keil C

The Keil C compiler introduces the keywords in Table 6.1 in addition to the ANSI C standard. These new keywords are part of the statement and do not impact the counting rules described in the previous section. However, part of the functions in the µp1 software was implemented using assembly language. This section describes the counting rules for the C51 family assembly code.

Table 6.1 Additional Keywords in Keil Environment

_at_	**alien**	**bdata**
bit	**code**	**compact**
data	**far**	**idata**
interrupt	**large**	**pdata**
priority	**reentrant**	**sbit**
sfr	**sfr16**	**small**
task	**using**	**xdata**

Rule 0: Physical statement counts. Each statement counts 1.

Rule 1: Statements that count
Instructive statement: starts with a valid instruction, including "MOV," "MOVX," "JMP," "INC," "DJNZ," "CJNE," "RET" and more (summarized in Table 6.2).

Rule 2: Statements that do not count:
Blank line
Comment: starts with ";"
Label statement; ends with ":"
Preprocessor directive: starts with the keywords "NAME," "PUBLIC," "EXTRN," "DPR_START_ADDR," "SCODE," "RSEG," "END," "_ _ERROR_ _," "EVEN," "EXTERN," "LABEL," "ORG," "PUBLIC," "SEGMENT," "SET."
Other statements that cannot be classified by Rule 1 and Rule 2.

Table 6.2 C51 Assembly Instructions

BIT	**BSEG**	**CODE**
CSEG	DATA	DB
DBIT	DD	DS
DSB	DSD	DSEG
DSW	DW	IDATA
ISEG	LIT	PROC
ENDP	sfr	sfr16
sbit	USING	XDATA
XSEG		

6.2.2.3 LOC Counting Rules for IAR C

IAR C Compiler introduces the following keywords in addition to the ANSI C standard: “sfr,” “no_init,” “interrupt,” “monitor,” “using,” “_C_task.” These new keywords are part of the statement (modifier) and do not impact the counting rules described in the previous section.

The original µp1 software implementation contains pieces of embedded assembly code. This feature is not supported by IAR C Compiler. These pieces were rewritten to implement the same functionality.

Unlike the µp1 software, the µp2 software does not contain functions implemented in assembler. The counting rules for the assembly code were not developed.

6.2.2.4 Considerations for General Use

Most of the above counting rules, especially the rules for ANSI C, are generic to any C code. Although the rules for the two compilers are specific, the principle of counting the instructions is generic also. In conclusion this set of rules can be easily customized to any embedded software developed using C and assembly languages.

6.3 Measurement Results

It should be noted that the definition of F (the total number of defects in the software) includes an assumption that smaller modules are less-fault prone. Thus, the result of F might be highly dependent on the definition of module. The higher the level module definition used, the less F calculated by Equation 6.1. For the APP system, there can be two levels of definition of module:

1. Each “.c” and “.h” files (i.e., “SF1PROG” along with its header files is a module),
2. Each function or subroutine (i.e., the “Main function” of “SF1PROG” is a module).

Table 6.3 lists modules (according to the definition level 1), the corresponding number of lines of code for the source code and header files, and the corresponding value of F_i. The number of defects, F, per module is also shown. The total number of defects remaining in the APP source code is approximately equal to 115 (rounded up to an integer).

Similarly, Table 6.4 lists the measurement results according to the second level module definition. The total number of defects remaining in the APP source code is approximately 530.

One header file may appear multiple times in different modules. Since each file is included in a module individually, a header file’s defect contribution to one module is independent of its contribution to other modules. As such, one header file counts separately in different modules. It should also be mentioned that the header files are those developed by the APP development team and do not include standard library header files. There are sufficient reasons to believe that those

standard header files have higher reliability than modules assessed by Equation 6.1 due to their large usage in a number of applications and the consequent thorough testing they have undergone. Consequently, those files are not considered in this research.

Table 6.3 Bugs Per Line of Code Results (By Definition Level 1)

	Module	LOC	Header Files' LOC	Total LOC	F_i	F
µp1	SF1APP	226	254	480	9.84	53.54
	SF1CALTN	245	163	408	8.74	
	SF1FUNCT	285	163	448	9.34	
	SF1PROG	234	254	488	9.96	
	SF1TEST1	159	163	322	7.51	
	SF1TEST2	205	163	368	8.15	
µp2	APP1	206	0	206	6.02	31.66
	CAL_TUNE	318	0	318	7.46	
	MAIN	379	0	379	8.31	
	ON_LINE	44	0	44	4.43	
	POWER_ON	154	0	154	5.44	
CP	COMMONLI	76	114	190	5.84	29.25
	COMMPOW	241	114	355	7.97	
	COMMPROC	120	114	234	6.36	
	COMMSER	317	114	431	9.08	
Total		3,209	1,616	4,825	114.45	114.45

Table 6.4 Bugs Per Line of Code Results (By Definition Level 2)

		LOC	F_i	Total
µp1	SF1APP	480	24.55	243.25
	SF1CALTN	408	32.03	
	SF1FUNCT	448	70.05	
	SF1PROG	488	49.23	
	SF1TEST1	322	31.47	
	SF1TEST2	368	35.92	

Table 6.4 Bugs Per Line of Code Results (By Definition Level 2) (continued)

		LOC	F_i	Total
µp2	APP1	206	18.03	128.26
	CAL_TUNE	318	19.23	
	MAIN	379	52.37	
	ON_LINE	44	8.56	
	POWER_ON	154	30.07	
CP	COMMONLI	190	22.18	158.46
	COMMPOW	355	44.11	
	COMMPROC	234	35.09	
	COMMSER	431	57.08	
Total		4825	529.97	529.97 (530)

There are two main concerns regarding the results:

1. It is believed that definition 1 is not appropriate. As Gaffney specified, a module is a "functional group." But according to the inspection of the APP system, the modules shown in Table 6.5 are not all arranged by functionalities. For example, SF1APP is a special function used to decide whether or not to generate a trip signal while SF1PROC includes the initialization function and a high-level main program for the first safety microprocessor. So from this point of view, the level 2 module definition is more appropriate in the case of APP.
2. There are two issues with the coefficients used in Equation 6.1. First, those coefficients were determined about 20 years ago and have not been updated since then. No updating information could be obtained. Second, as stated before, the counting rules may be the same for both C code and assembly code, while the coefficients in Equation 6.1 for these two types of code may not be the same. This topic, however, is out of the scope of this research.

Once the total number of defects in the software using Gaffney's equation have been obtained, the number of remaining defects can be derived by subtracting the number of defects found during the development process (by inspection and testing). That is, $N = F - N_{found}$.

The number of defects found by inspection and testing is presented in Table 6.5.

Table 6.5 Number of Defects Found by Inspection and Testing during the Development Process

	Number of Defects Found
μp1 SRS Inspection	60
μp2 SRS Inspection	65
CP SRS Inspection	55
μp1 SDD Inspection	65
μp2 SDD Inspection	110
CP SDD Inspection	40
μp1 code Inspection	7
μp2 code Inspection	11
CP code Inspection	15
Testing	7
TOTAL	**435**

Thus, the total number of remaining defects is:

$$N = F - N_{found} = 530 - 435 = 95$$

The next step is to partition the defects based on their criticality. According to [Jones, 1996], defects are divided into four categories according to their severity level:

- Severity 1: Critical problem (software does not operate at all)
- Severity 2: Significant problem (major feature disabled or incorrect)
- Severity 3: Minor problem (some inconvenience for the users)
- Severity 4: Cosmetic problem (spelling errors in messages; no effect on operations)

Only defects of Severity 1 and Severity 2, called "critical defects" and "significant defects," respectively, should be considered for estimating software reliability. Defects with Severity 3 and 4, called "minor defects" and "cosmetic defects," respectively, do not have an impact on the functional performance of the software system. Thus, they have no effect on reliability quantification.

Table 6.6 (Table 3.48 in [Jones, 1996]) presents US averages for percentages of delivered defects by severity levels.

Table 6.6 Averages for Delivered Defects by Severity Level
(Adapted from Table 3.48 in [Jones, 1996])

Function points	Percentage of Delivered defects by Severity Level			
	Severity 1 (critical)	Severity 2 (significant)	Severity 3 (minor)	Severity 4 (cosmetic)
1	0	0	0	0
10	0	0	1	0
100	0.0256	0.1026	0.3590	0.5128
1000	0.0108	0.1403	0.3993	0.4496
10000	0.0150	0.1450	0.5000	0.3400
100000	0.0200	0.1200	0.5000	0.3600
Average	0.0197	0.1215	0.4996	0.3592

Using Table 6.6 and logarithmic interpolation, the percentages of delivered defects by severity level can be obtained for APP. For example, based on the assessment of the APP function point count (discussed in detail in Chapter 14), the percentage of delivered defects of severity 1 corresponding to $FP = 301$ ($100 < 301 < 1000$) is:

$$0.0256 + \frac{0.0108}{log_{10}(1000) - log_{10}(100)} \times [log_{10}(301) - log_{10}(100)] = 0.0185 \tag{6.2}$$

Table 6.7 presents the percentages of delivered defects by severity level for a system equivalent in size to FP..

Table 6.7 Delivered Defects by Severity Level for a System Equivalent in Functional Size to FP

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

The total percentage of Severity 1 (critical faults) and Severity 2 (significant faults) is:

$$0.0185 + 0.1206 = 0.1391 \tag{6.3}$$

Table 6.8 presents the partitioned defects (based on the severity level) for APP.

Table 6.8 Partitioned Defects (Based on Severity Level) for APP Using BLOC

Total Number of Defects	Defects (Critical)	Defects (Significant)	Defects (Minor)	Defects (Cosmetic)	Defects (Critical + Significant)
95	1.7575	11.457	38.9385	45.847	13.2

6.4 RePS Construction from BLOC

The probability of success-per-demand is obtained using Musa's exponential model [Musa, 1990] [Smidts, 2004]:

$$p_s(BLOC) = e^{\left(-K \times N_{BLOC} \times \tau / T_L\right)} \tag{6.4}$$

where

$p_s(BLOC)$ Reliability estimation for the APP system using the Bugs per Line of Code (BLOC) measure.

K Fault Exposure Ratio, in failure/defect.

N_{BLOC} Number of defects estimated using the BLOC measure.

τ Average execution-time-per-demand, in seconds/demand.

T_L Linear execution time of a system, in seconds.

Since *a priori* knowledge of the defects' location and their impact on failure probability is unknown, the average K value given in [Musa, 1979] [Musa, 1990] [Smidts, 2004] ($4.2 \times 10^{-7} failure/defect$) must be used.

The *linear execution time*, T_L, is usually estimated as the ratio of the execution time and the software size on a single microprocessor basis [Musa, 1987] [Musa, 1990] [Smidts, 2004]. However, in the case of the APP system, there are three parallel subsystems, each having a microprocessor executing its own software. Each of these three subsystems has an estimated linear execution time. Therefore, there are several ways to estimate the linear execution time for the entire APP system such as using the average value of these three subsystems.

For a safety-critical application, like the APP system, the UMD research team suggests making a conservative estimation of T_L by using the minimum of these three T_L values. Namely:

$$\begin{aligned} T_L &= \min\{T_L(\mu p1), T_L(\mu p2), T_L(CP)\} \\ &= \min\{0.018, 0.009, 0.021\} \\ &= 0.009 \text{ seconds} \end{aligned}$$

where

$T_L(\mu p1)$	Linear execution time of Microprocessor 1 (μp1) of the APP system. T_L *(μp1)* = 0.018 seconds (refer to Chapter 17).
$T_L(\mu p2)$	Linear execution time of Microprocessor 2 (μp2) of the APP system. T_L *(μp2)* = 0.009 seconds (refer to Chapter 17).
$T_L(CP)$	Linear execution time of Communication Microprocessor (CP) of the APP system. T_L *(CP)* = 0.021 seconds (refer to Chapter 17).

Similarly, the *average execution-time-per-demand*, τ, is also estimated on a single microprocessor basis. Each of the three subsystems in APP has an estimated average execution-time-per-demand. To make a conservative estimation, the average execution-time-per-demand for the entire APP system is the maximum of the three execution-time-per-demand values. Namely:

$$\begin{aligned}\tau &= \max\{\tau(\mu p1), \tau(\mu p2), \tau(CP)\} \\ &= \max\{0.082, 0.129, 0.016\} \\ &= 0.129 \text{ seconds/demand}\end{aligned}$$

where

$\tau(\mu p1)$	Average execution-time-per-demand of Microprocessor 1 (μp1) of the APP system. *τ(μp1)* = 0.082 seconds/demand (refer to Chapter 17).
$\tau(\mu p2)$	Average execution-time-per-demand of Microprocessor 2 (μp2) of the APP system. *τ(μp2)* = 0.129 seconds/demand (refer to Chapter 17).
$\tau(CP)$	Average execution-time-per-demand of Communication Microprocessor (CP) of the APP system. *τ(CP)* = 0.016 seconds/demand (refer to Chapter 17).

Thus the reliability for the APP system using the BLOC measure is given by:

$$\begin{aligned}p_s(BLOC) &= e^{(-4.2\times10^{-7}\times13.2\times0.129/0.009)} \\ &= 0.999920539\end{aligned}$$

6.5 Lessons Learned

It is well known that the lines of code measurement can be easily conducted because tools are available to support such measurements. BLOC measurement based on Equation 6.1 requires a clear definition of "module," which the author of BLOC did not provide. The existence of multiple definitions of the module concept [Schach, 1993] [IEEE, 1990] and the lack of consensus make its accurate measurement difficult. The research team explored two interpretations of "module" and conducted corresponding measurements as shown in Section 6.3. Based on the two sets of measurement results, a more meaningful interpretation was selected. The RePS based on BLOC is straightforward once the average execution-time-per-demand and the linear execution time are quantified.

6.6 References

[APP, Y1] "APP Module SF1 System Software code," Year Y1.

[APP, Y2] "APP SF1 Flux/Delta Flux/Flow Application code," Year Y2.

[APP, Y3] "APP Module μp2 System Software Source Code Listing," Year Y3.

[APP, Y4] "APP μp2 Flux/Delta Flux/Flow Application Software Source Code Listing," Year Y4.

[APP, Y5] "APP CP Source Code," Year Y5.

[Gaffney, 1984] J.E. Gaffney. "Estimating the Number of Faults in Code." *IEEE Transactions on Software Engineering*, vol. 10, pp. 459–64, 1984.

[Halstead, 1977] M.H. Halstead. *Elements of Software Science*. New York: Elsevier, 1977.

[IEEE, 1990] "IEEE Standard Glossary of Software Engineering Terminology," Std. 610.12-1990, 1990.

[Jones, 1995] C. Jones. "Backfiring Converting Lines of Code to Function Point," *Computer*, vol. 28, pp. 87–88, 1996.

[Musa, 1987] J.D. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Applications*. New York: McGraw-Hill, 1987.

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

[Rosenberg, 1997] J. Rosenberg. "Some Misconceptions about Lines of Code," in *Proc. 4th International Software Metrics Symposium*, 1997, pp. 137–142.

[Schach, 1993] S.R. Schach. *Software Engineering*. Homewood, IL: Aksen Associates Inc., 1993.

[Smidts, 2000] C. Smidts and M. Li, "Software Engineering Measures for Predicting Software Reliability in Safety Critical Digital Systems," NRC, Office of Nuclear Regulatory Research, Washington DC NUREG/GR-0019, 2000.

[Smidts, 2004] C. Smidts and M. Li, "Validation of a Methodology for Assessing Software Quality," NRC, Office of Nuclear Regulatory Research, Washington DC NUREG/CR-6848, 2004.

7. CAUSE-EFFECT GRAPHING

Cause-effect graphing (CEG) is a formal translation of a natural-language specification into a graphical representation of its input conditions and expected outputs. The graph depicts a combinatorial logic network. It illustrates the logical relationship between inputs and outputs along with the constraints among the inputs and outputs. Therefore, it could aid in identifying requirements that are incomplete and ambiguous in the SRS [Myers, 1976] [Myers, 1979] [Nuisimulu, 1995].

According to IEEE [IEEE, 1988], 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.

CEG can also be used to generate test cases in any type of computing application where the specification is clearly stated (that is, no ambiguities) and combinations of input conditions can be identified. It is used in developing and designing test cases that have a high probability of detecting faults that exist in programs. It is not concerned with the internal structure or behavior of the program [Elmendorf, 1973].

This measure can be applied as soon as the requirements are available. As listed in Table 3.3, the applicable life cycle phases for CEG are Requirements, Design, Coding, Testing, and Operation.

7.1 Definition

There are four primitives in this measure defined in [IEEE, 1988]:

1. List of causes: distinct input conditions
2. List of effects: distinct output conditions or system transformation (effects are caused by changes in the state of the system)
3. $A_{existing}$: number of ambiguities in a program remaining to be eliminated
4. A_{total}: total number of ambiguities identified

Then, the measure is computed as follows:

$$CE\% = 100 \times \left(1 - \frac{A_{existing}}{A_{total}}\right) \tag{7.1}$$

Cause effect graphing measures CE%, the percentage of the number of ambiguities remaining in a program over the total number of identified ambiguities through cause and effect graphing. The RePS which uses this measure is not based on the value of CE% but, rather, on the defects that were found in the SRS using an "inspection approach" based on cause and effect graphing. The impact of these defects is assessed using the PIE concept and, more specifically, an EFSM. The

defects themselves are characterized by their type and their location in the application. “Defect types” can be measured according to a nominal scale and “defect locations” can be measured according to an interval scale.

The detailed definitions of cause and effect are shown in the following subsections.

7.1.1 Definition of Cause

In the SRS, any functional event is identified as either an effect or a cause. A cause represents a distinct input condition or an equivalence class of input conditions. It is defined as an input event, typically triggered by a user.

A cause only has two mutually exclusive statuses: enabled (represented by “1”) or disabled (represented by “0”).

7.1.2 Definition of Effect

An effect might be a system output or a system action. There are two types of effects: user-observable effects and user-unobservable effects. User-observable effects, also called “primary effects,” are those effects that can be noticed by users. For example, the statuses of LEDs, either on or off, are user-observable effects. The user-unobservable effects will be treated as intermediate effects.

An effect only has three mutually exclusive statuses: present (represented by “1”), absent (represented by “0”), or non-existent (represented by “NULL”).

7.1.3 Definition of Logical Relationship and External Constraints

While constructing a cause-effect graph, both the cause-effect logical relationship and the external constraints can be identified by applying the so-called “pattern-matching method.”

There are four basic patterns of cause-effect logical relationships, which are shown in Table 7.1 [Myers, 1979]:

A constraint is a limitation (syntactic, environmental, or other) among causes or effects. There are five possible patterns of external constraints, which are shown in Table 7.2 [Myers, 1979].

Table 7.1 Cause-Effect Logical Relationships

Logical Relationship	Pattern
IDENTITY	IF cause *C1* THEN effect *E1*
NOT	IF NOT cause *C1* THEN effect *E1*
AND	IF cause *C1* AND *C2* THEN effect *E1*
OR	IF cause *C1* OR *C2* THEN effect *E1*

Table 7.2 Cause-Effect Constraints

External Constraints	Patterns
EXCLUSIVE	AT MOST ONE OF *a*, *b* CAN BE INVOKED
INCLUSIVE	AT LEAST ONE OF *a*, *b* MUST BE INVOKED
ONE-ONLY-ONE	ONE AND ONLY ONE OF *a*, *b* CAN BE INVOKED
REQUIRES	IF a IS INVOKED THEN b MUST BE INVOKED
MASKS	EFFECT *a* MASKS OBSERVANCE OF EFFECT *b*

7.2 Measurement Rules

The measurement rules for identifying causes, effects, logical relationships, and constraints are described in the following subsections, respectively.

7.2.1 Rule for Identifying Causes

To identify causes, one should read the specification carefully, underlining words or phrases that describe causes. Any distinct input condition or equivalence class of input conditions should be considered causes.

Only functional events in the specification are considered. Each cause is assigned to a unique number. None of the descriptive specifications are considered in identifying causes.

7.2.2 Rule for Identifying Effects

Effects can be identified by reading the specification carefully and underlining words or phrases that describe effects. Some intermediate effects are important for determining the status of the system. So both the primary effects and the intermediate effects are required to be considered.

Only functional events in the specification are considered. All the descriptive specifications are not considered in identifying effects. Each effect is assigned to a unique number.

7.2.3 Rule for Identifying Logical Relationship

The logical relationship between causes and effects can be identified by analyzing the semantic content of the specification linking the causes with the effects. Keywords such as "not," "or," "and" usually act as indicators of logical relationships. Other words denoting logical relationships, such as "both" and "neither" also should be addressed.

The logical relationships are primarily found in function specifications, but can also be found in some descriptive specifications. To ensure complete identification of all logical relationships between causes and effects, both function and descriptive specifications should be analyzed. The four basic logical relationships are shown in Table 7.1.

7.2.4 Rule for Identifying External Constraints

The external constraints among causes can be identified by checking for the occurrence of related causes specified in the SRS. The external constraints among effects can be identified by checking for the occurrence of related effects specified in the SRS. As with the logical relationships, the external constraints could be specified in both functional specifications and descriptive specifications. In order to identify all external constraints, both functional and descriptive specifications need to be analyzed. The five basic external constraints among causes and the external constraints among effects are shown in Table 7.2.

The following example shows how to apply the above measurement rules to a SRS:

Example #1: An application of these measurement rules

The following paragraph is excerpted from an APP requirement specification document for μp1 system software:

"*Upon power-up or a module reset, the first safety microprocessor shall perform the initialization algorithm. Below are the functional requirements performed in the sequence given unless stated otherwise. Refer to Figure 7.1 for high level flow chart.*"

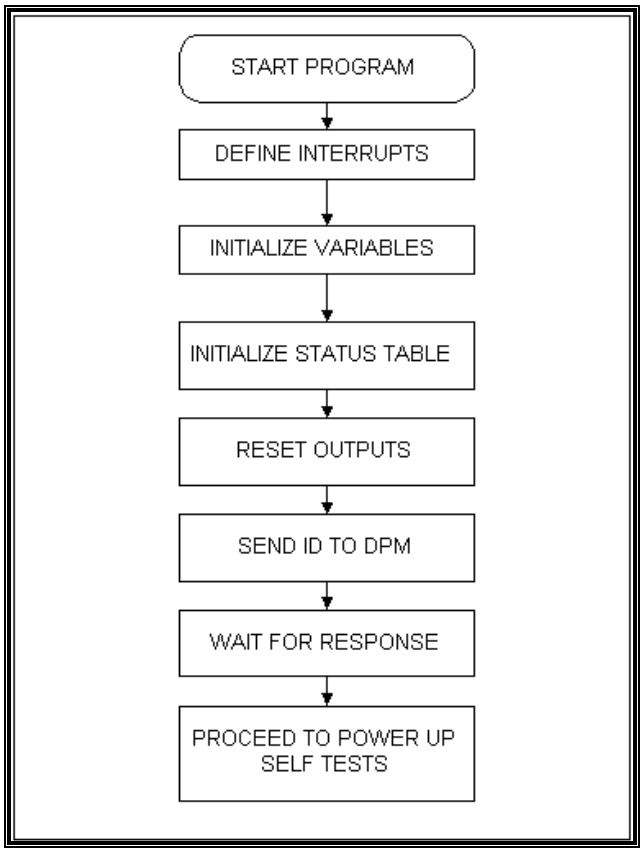


Figure 7.1 Initialization Flow Chart

1. "Upon power-up or a module reset, the first safety microprocessor shall perform the initialization algorithm" is a functional specification; below are the functional requirements performed in the sequence given unless stated otherwise. Refer to Fig. 7.1 [APP, Y1].
2. "Power-up" and "module reset" are two causes in the functional specification.
3. "The first safety μp shall perform the initialization algorithm" is the only identifiable effect from this specification. It is then necessary to determine if this effect is a prime effect or not. Because it is neither user-observable nor a system action, we consider it an intermediate effect (the prime effect is the detailed initialization algorithm). With this in mind, several prime effects can be identified from the figure.
4. The only logical relationship here is identifiable by the use of the keyword "or."
5. There are no constraints.

Based on the above rules, the CEG measurement results for this example are shown in Table 7.3.

Table 7.3 CEG Measurement Results Table for the Example

Causes	Relationships	Constraints	Effects
C1. Power up	C1 or C2	N/A	E1. Define interrupt E2. Initialize global variables E3. Initialize status table
C2. Module reset			E4. Reset outputs E5. Read ID from PROM E6. Send Module ID to DPM E7. Wait for response from CP

7.2.5 Rules for Constructing an Actual Cause-Effect Graph

An Actual Cause-Effect Graph (ACEG) is an implemented cause-effect graph constructed according to the SRS. The following steps show how to construct an ACEG based on an SRS:

1. Identify all requirements of the system and divide them into separate identifiable entities.
2. Carefully analyze the entities to identify all the causes and effects in the SRS and discern all the cause-effect logical relationships and constraints.
3. Represent each cause and each effect by a node identified by its unique number. For example, E1 for effect one or C1 for cause one.
4. Interconnect the cause and effect nodes by analyzing the semantic content of the specification and transforming it into a Boolean graph. Each cause and effect can be in one of two states: true or false. Using Boolean logic, set the possible states of the causes and determine under what conditions each effect will be present.
5. Annotate the graph with constraints describing combinations of causes and effects that are impossible because of semantic or environmental constraints.
6. Identify any defects in the SRS and map them to the ACEG.

7.2.6 Rules for Identifying Defects in ACEG

Defects can be any cause that does not result in a corresponding effect, any effect that does not originate with a cause, and effects that are inconsistent with the requirements specification or impossible to achieve. There are five main types of defects that can be found through constructing an ACEG:

1. Missing effect
2. Extra effect
3. Missing constraint
4. Extra constraint

5. Wrong Boolean function
    a. Missing cause in a Boolean function
    b. Extra cause in a Boolean function
    c. Wrong Boolean Operator

The detailed rules for identifying each type of defect are shown in the following

1. Missing effect:

While some missing effects may be obvious, in general, finding obscure missing effects requires mastery of the system. Thus, there is no straightforward process by which to identify missing effects.

2. Extra effect:

Extra effects are unnecessary effects. Therefore, to identify extra effects, an inspector must understand the physical meaning of the effect and determine whether or not it is necessary.

3. Missing constraint(s):

To identify missing constraints, the inspector should be capable of understanding the physical meaning of all the causes and effects in the ACEG.

The process for identifying missing constraints is:

a. Sequentially arrange all causes.
b. The REQUIRES constraint must be applied if two cause events occur sequentially. If it has not been applied, then it is a missing constraint.
c. For causes that occur simultaneously, examine if EXCLUSIVE, INCLUSIVE, or ONE-ONLY-ONE constraints were neglected.
d. Sequentially arrange all effects.
e. The MASKS constraint must be applied to effects that can occur simultaneously and if there is a risk for their co-existence. If it is missing, then it is a missing constraint.

4. Extra constraint(s):

To identify extra constraints, the inspector should be capable of understanding the physical meaning of all causes or effects in a constraint and determining whether the constraint is necessary or not.

The process for identifying extra constraints is:

a. Sequentially arrange all causes.
b. If two cause events do not occur sequentially, the REQUIRES constraint should not be applied to them. If applied, it is an extra constraint.

c. If two or more events do not occur simultaneously, EXCLUSIVE, INCLUSIVE or ONE-ONLY-ONE constraints should not be applied to them. If applied, it is an extra constraint.
d. Individually examine the MASKS constraints and determine if each is necessary or not. If not, it is an extra constraint.

5. Wrong Boolean function:

To identify a Wrong Boolean function, the inspector should be capable of understanding the physical meaning of all causes or effects. In addition, the inspector should have mastered the operation mechanism of the system to determine what logical relationships should be applied to the causes.

The process for identifying extra constraints is:

1. Consider one Boolean function at a time.
2. Individually check the causes in the Boolean function and determine whether or not a cause is necessary. An unnecessary cause is an extra cause in a Boolean function.
3. Consider the remaining causes in the ACEG. If any cause should have been involved in the Boolean function, it is a missing cause.
4. Consider other possible causes not included in the ACEG. If any cause should have been involved in the Boolean function, it is a missing cause.
5. Check all Boolean operators in the Boolean function to identify incorrect one(s).

7.2.7 Rules for Constructing a Benchmark Cause-Effect Graph

The Benchmark Cause-Effect Graph (BCEG) is constructed by removing all identified defects from an ACEG. Example #2 illustrates how to apply these rules for constructing an ACEG and its corresponding BCEG.

Example #2: An application of the ACEG and the BCEG and associated defects found by measurement rules

The following paragraph is excerpted from the APP requirement specification document for μp1 system software:

"*After completing all of the diagnostic tests, the Power-Up Self Tests algorithm shall reset the Power-Up Active flag and determine the integrity of each of the diagnostic test's results. If all tests passed, then the algorithm shall turn ON front panel LEDs, refresh the status relays and turn ON the μp status LED before proceeding to the Main Program.*"

Step 1: Apply measurement rules to the specification.

The measurement results table for this example is shown in Table 7.4:

Table 7.4 CEG Measurement Results for the Example

Causes	Relationships	Constraints	Effects
C1. RAM (DPM, Data bus line) test passed	All tests passed: C1 and C2 and C3 and C4 and C5 and C6 and C7 and C8	N/A	E1. Turn ON front panel LEDs
C2. Address bus line test passed			
C3. PROM checksum test passed			
C4. EEPROM checksum test passed			E2. Refresh the status relays
C5. Boards test passed			
C6. Algorithm test passed			E3. Turn ON the µp status LED
C7. Analog input circuits test passed			
C8. Discrete input circuits test passed			

Step 2: Draw the ACEG: the ACEG is shown in Figure 7.2.

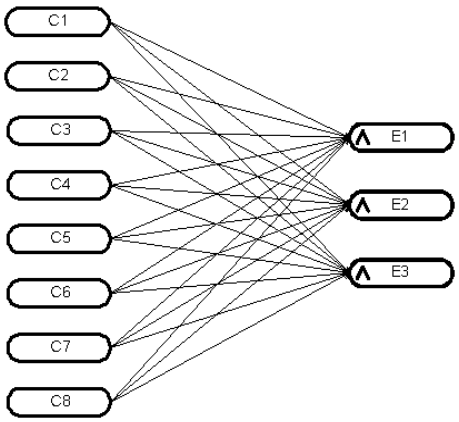


Figure 7.2 ACEG for Example #2[11]

Step 3: Check Defects:

Upon system inspection, an inspector would find that C4 is not the necessary cause for proceeding to the main program. Even if the EEPROM test fails in the power-on self test, the system can go into the main program. There is a special function in the main program to check the status of the EEPROM test. In summary, C4 is an extra cause.

Step 4: Draw the BCEG: By removing the defect from the ACEG, Figure 7.3 shows the BCEG for this example.

[11] A "–" mark indicates "IDENTIFY," a "^" mark indicates "AND," a "v" mark indicates "OR," and a "~" mark indicates "NOT."

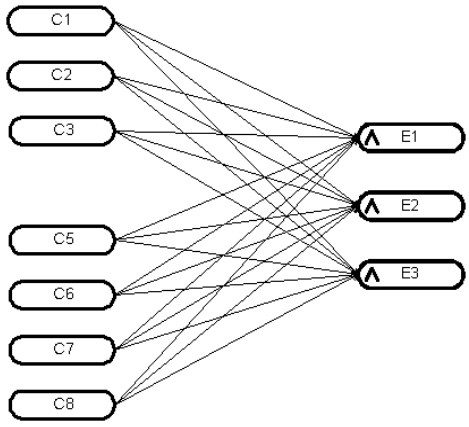


Figure 7.3 BCEG for Example #2[12]

7.3 Measurement Results

In this research, CEG measurement results are based only on the APP SRSs—the cause-effect structures in the SDD and the code have not been analyzed. However, the CEG method can be applied to those stages. If so, the reliability of the software in different stages can be obtained. The measurement rules for CEG in those stages have not been generated because that was out of the scope of this research.

A list of the defects found in the APP SRSs is shown in Table 7.5.

Table 7.5 List of Defects Found by CEG Based On the APP SRSs

Defect No.	Location	Defect Description	Cross-Reference
1	µp1	Extra cause (C11) in deciding whether to enter the main program.	Figure 7.4
2	µp1	Missing cause (C12) for setting the EEPROM test failure flag.	Figure 7.5
3	µp2	Extra cause (C10) in deciding whether to enter the main program.	Figure 7.6
4	µp2	Wrong Boolean function in setting the EEPROM test failure flag.	Figure 7.7
5	µp2	Wrong Boolean function in the RAM diagnostics test.	Figure 7.8
6	µp2	Missing effect (E3) for turning on TRIP LED.	Figure 7.9
7	CP	Missing cause (C16) in checking the diagnostics results.	Figure 7.10

The following figures show the ACEG and corresponding BCEG related to the above defects. On the left are the ACEGs and on the right are the BECGs.

[12] A solid line indicates "IDENTIFY," a ^ mark indicates "AND," a v mark indicates "OR," and a ~ mark indicates "NOT."

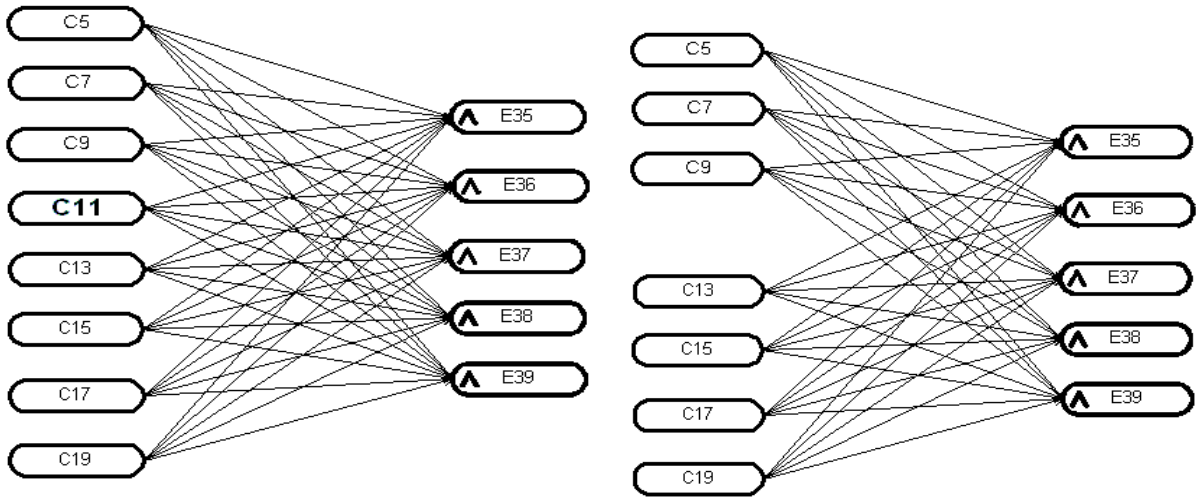


Figure 7.4 ACEG and BCEG for Defect #1[13]

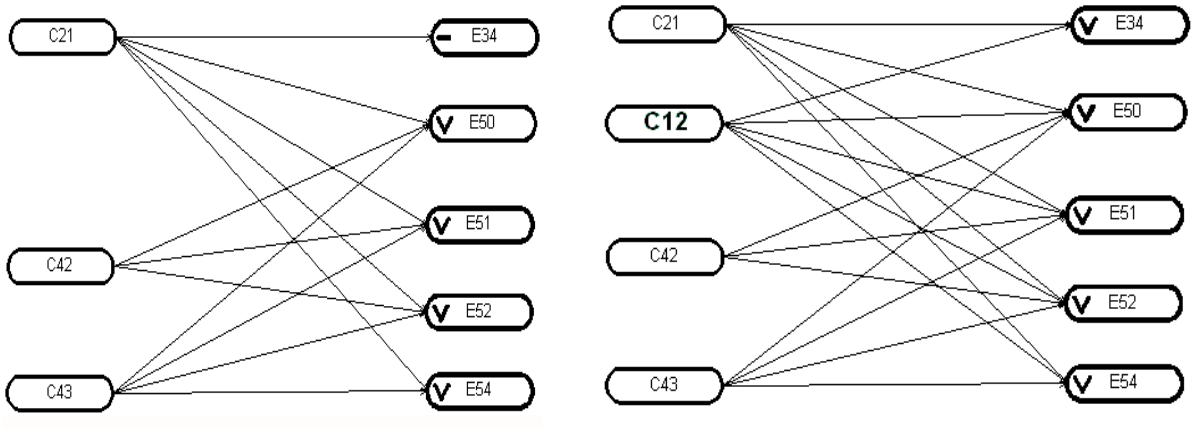


Figure 7.5 ACEG and BCEG for Defect #2

[13] A "–" mark indicates "IDENTIFY," a "^" mark indicates "AND," a "v" mark indicates "OR," and a "~" mark indicates "NOT."

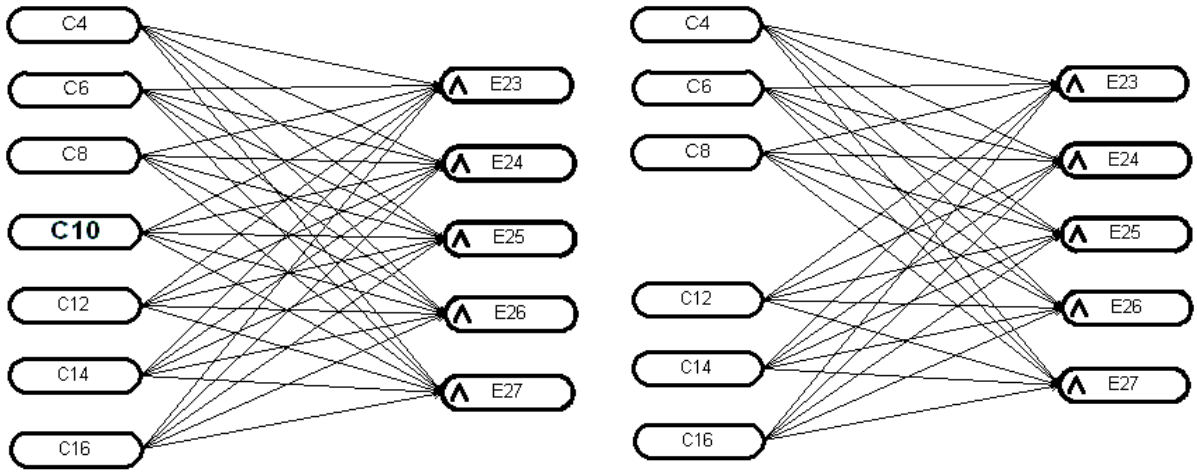


Figure 7.6 ACEG and BCEG for Defect #3[14]

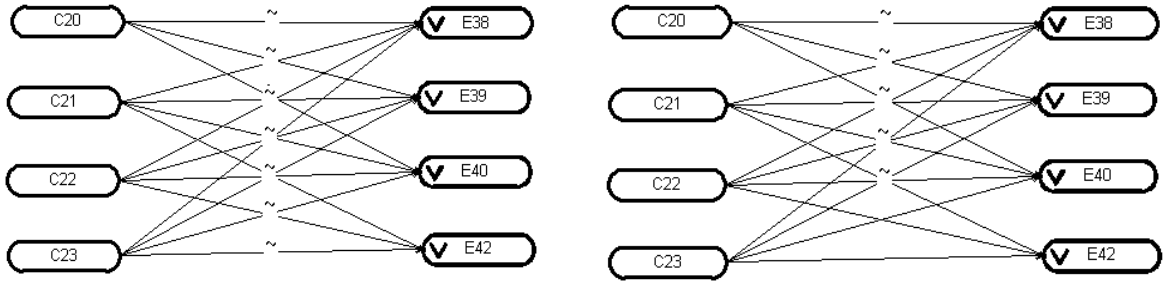


Figure 7.7 ACEG and BCEG for Defect #4

[14] A "–" mark indicates "IDENTIFY," a "^" mark indicates "AND," a "v" mark indicates "OR," and a "~" mark indicates "NOT."

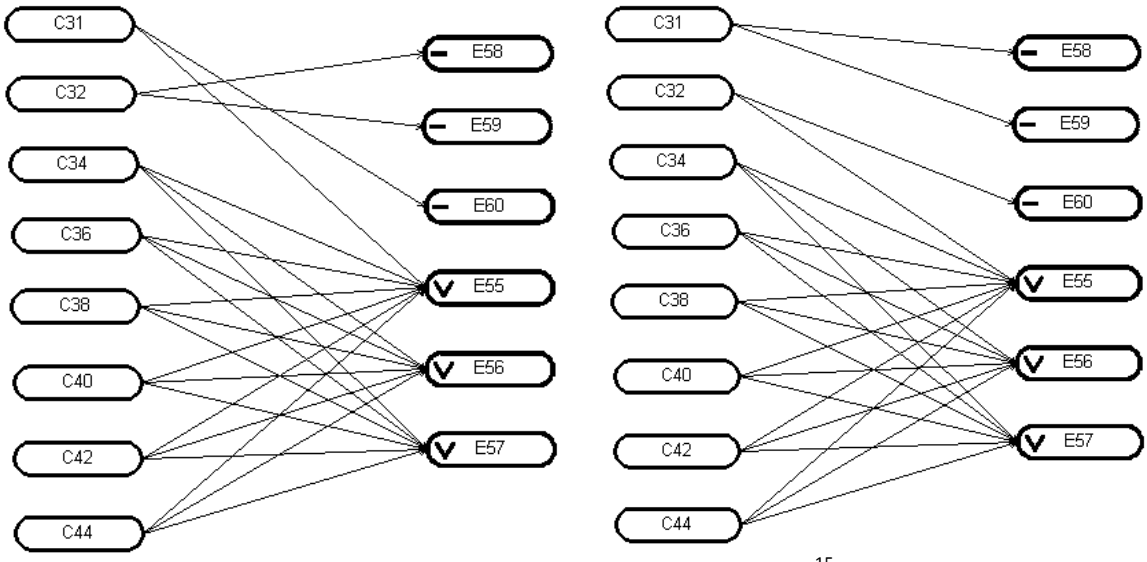


Figure 7.8 ACEG and BCEG for Defect #5[15]

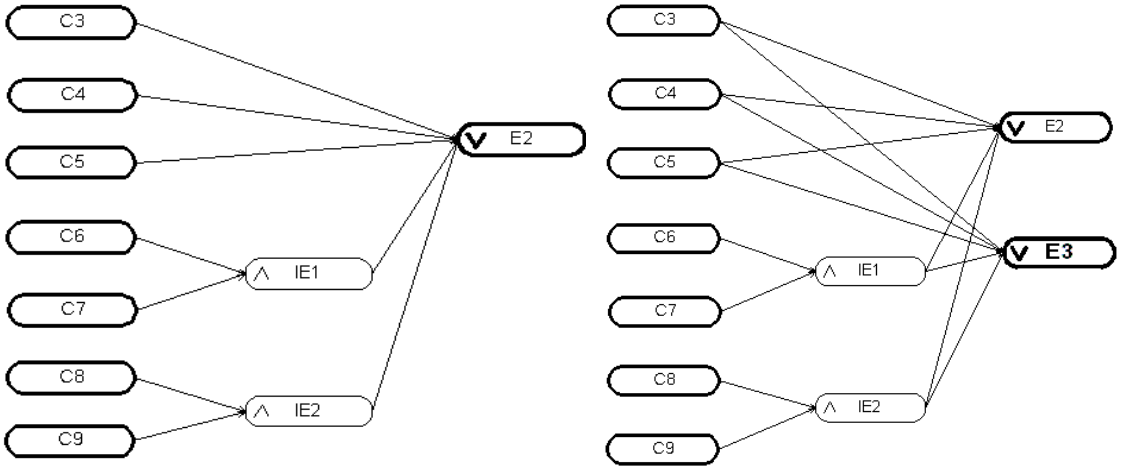


Figure 7.9 ACEG and BCEG for Defect #6

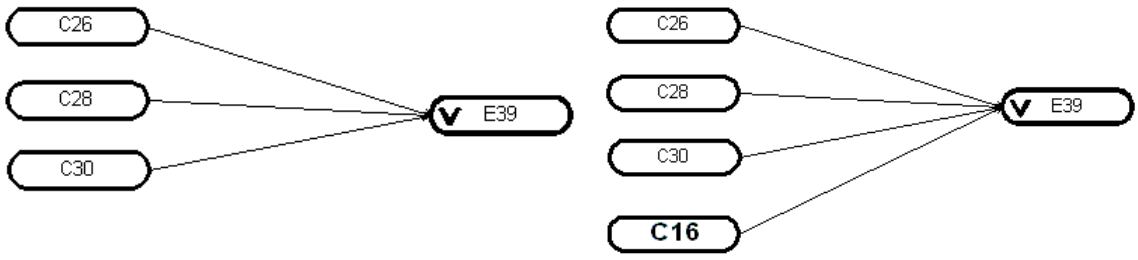


Figure 7.10 ACEG and BCEG for Defect #7

[15] A "–" mark indicates "IDENTIFY," a "^" mark indicates "AND," a "v" mark indicates "OR," and a "~" mark indicates "NOT."

As previously stated, all the above defects can be found in the SRS. Table 7.6 shows whether the defects found in the SRS were fixed (either in the SDD or in the code).

Table 7.6 Checking Results for Defects Found by CEG

Defect No.	Location	Defect Description	Fixed in SDD or in code?
1	µp1	Extra cause in deciding whether to enter the main program.	Y
2	µp1	Missing cause for setting the EEPROM test failure flag.	Y
3	µp2	Extra cause in deciding whether to enter the main program.	Y
4	µp2	Wrong Boolean function in setting the EEPROM test failure flag.	Y
5	µp2	Wrong Boolean function in the RAM diagnostics test.	Y
6	µp2	Missing effect for turning on TRIP LED.	Y
7	CP	Missing cause in checking the diagnostics results.	N

As specified in Table 7.6, six out of seven defects found were fixed. Only Defect No. 7 remains in the code. If the corresponding cause is triggered, the system will experience a catastrophic failure.

According to the CEG definition (Equation 7.1):

$$CE\% = 100 \times \left(1 - \frac{A_{existing}}{A_{total}}\right) = 100 \times \left(1 - \frac{1}{7}\right) = 85.71\%$$

The calculated value of $CE\%$ will not be used for reliability estimation. Section 7.4 shows the RePS construction from the CEG measure.

Because twelve measures are selected to evaluate the reliability of the system, an alternative to assigning an experienced analyst is to perform the measure during the later stages of the measurement. Usually, an analyst can gain knowledge of a system by performing other measurements such as the requirements traceability measure.

Cause-effect graphing is time-consuming work. For a large-scale application, it is only necessary to draw the defect-related portion(s). Doing so will save a great deal of time. The necessary documents are:

1. Complete list of causes and effects with their relationships and constraints in a table;
2. Defect-related ACEG and BCEG; and
3. Failure-relevant table for defects.

7.4 RePS Constructed from Cause-Effect Graphing

7.4.1 Reliability Prediction Based On CEG

Software reliability is estimated by first calculating the failure probability. Failure probability of an ACEG is assessed by comparing it with a corresponding BCEG and using a reduced-ordered binary-decision diagram (ROBDD) [Bryant, 1986] [Brace, 1990]. Figure 7.11 shows the generic fault tree for an ACEG.

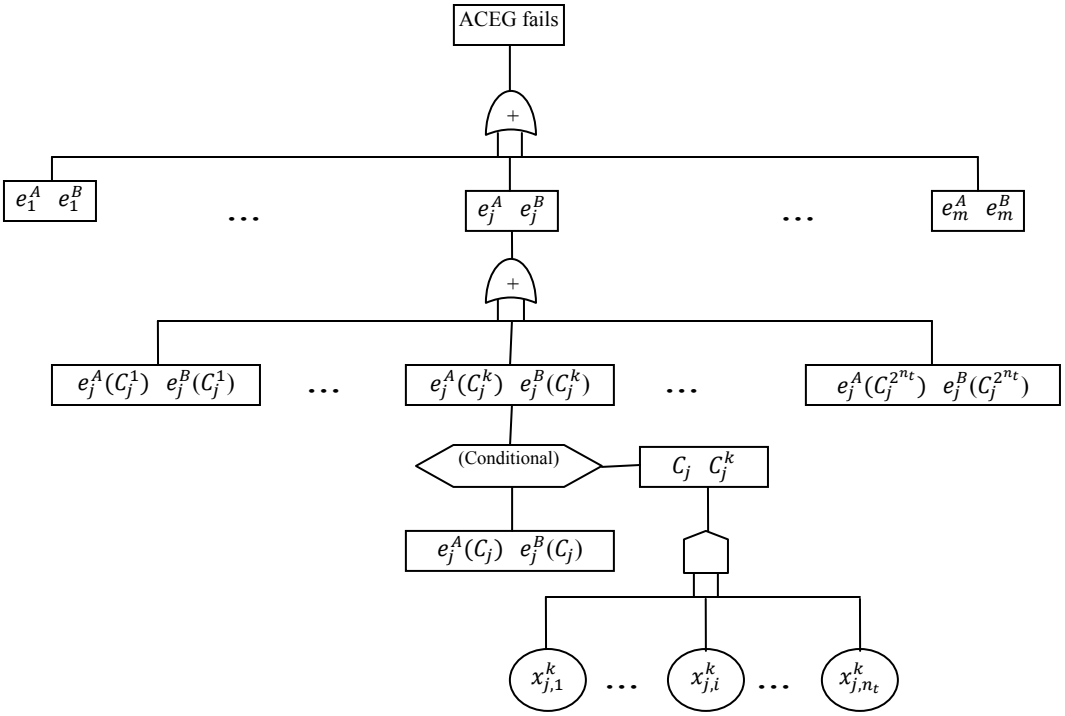


Figure 7.11 The Generic Fault Tree for an ACEG

An ACEG fails if one of the ACEG effects differs from its peer effect in the BCEG under a given cause-state combination. A complete nomenclature for CEG is given in the following:

$ACEG$ Actually implemented Cause-Effect Graph, constructed according to the SRS. $ACEG = \{C^A, E^A, F^A, CON^A\}$

C^A The cause set of the $ACEG$

E^A The observable effect set of the $ACEG$

F^A The Boolean function set of the $ACEG$

CON^A The constraint set of the $ACEG$

$BCEG$ Benchmark Cause-Effect Graph, constructed by removing all indentified defects from an $ACEG$. $BCEG = \{C^B, E^B, F^B, CON^B\}$

C^B The cause set of the $BCEG$

E^B The observable set of the $BCEG$

F^B The Boolean function set of the $BCEG$

CON^B The constraint set of the $BCEG$

e_j^A The j-th distinct observable effect in the $ACEG$; i.e., $e_j^A \in E^A, j = 1,2,\dots,m$

m The number of distinct effects in the union set $E^A \cup E^B$

e_j^B The peer observable effect in the $BCEG$ corresponding to e_j^A

f_j^A A Boolean function in F^A corresponding to e_j^A

f_j^B A Boolean function in F^B corresponding to e_j^B

C_j^A The set of causes appearing in f_j^A

C_j^B The set of causes appearing in f_j^B

C_j The union set of C_j^A and C_j^B; i.e., $C_j = C_j^A \cup C_j^B$

n_j The number of distinct causes in C_j

F An empty set

$\overrightarrow{C_j}$ A cause state vector that represents a state combination of all causes in C_j

$\overrightarrow{C_j}^k$ The k-th vector of $\overrightarrow{C_j}$

$\overrightarrow{C_j^k} = \left(x_{j,1}^k, x_{j,2}^k, \dots, x_{j,n}^k\right)$ where

$$x_{j,i}^k = \begin{cases} 1 \text{ if } c_{ji} \text{ occurs, } c_{ji} \in C_j & j = 1,2,\dots,m, i = 1,2,\dots,n_j \\ 0 \text{ otherwise} & k = 1,2,\dots,2^{n_j} \end{cases}$$

A three-step procedure is created to calculate the failure probability.

Step 1: Identify failure-relevant events for each effect pair.

If an effect relates to n causes, then compare the results from the ACEG with the results from the BCEG for 2^n times. It requires significant effort to draw the table and perform the comparison. Some of the causes are failure-irrelevant, which means changing their value will not affect the comparison results. So identifying the failure-relevant events is critical.

Step 2: Draw a decision table for every effect that is different between the ACEG and the BCEG.

A decision table is helpful for judging the equivalence of two effects with simple Boolean functions ($n_j \leq 10$). A sample decision table based on Boolean functions $f_1^A = c_1c_2 + c_3$, and $f_1^B =$

$c_1 + c_2c_3$, general constraints $CON_1^A = \{(c_1 \text{ requires } c_2)\}$, $CON_1^B = \{(c_2 \text{ requires } c_3)\}$ is shown in Table 7.7 below:

Table 7.7 Sample Decision Table for Judging Equivalence of Two Effects

k	$\overrightarrow{C_j^k}$			Conflict with CON_1^A	Conflict with CON_1^B	f_1^A	f_1^B	e_1^A	e_1^B	$e_1^A = e_1^B$?
	c_1	c_2	c_3							
1	0	0	0	N	N	0	0	0	0	Y
2	0	0	1	N	N	1	0	1	0	N
3	0	1	0	N	Y	0	0	0	NULL	N
4	0	1	1	N	N	1	1	1	1	Y
5	1	0	0	Y	N	1	1	NULL	1	N
6	1	0	1	Y	N	1	1	NULL	1	N
7	1	1	0	N	Y	1	1	1	NULL	N
8	1	1	1	N	N	1	1	1	1	Y

Step 3: Create a ROBDD for calculating the total system-failure probability.

In the field of reliability it is common knowledge that a BDD is a directed acyclic graph. The graph has two sink nodes labeled 0 and 1, representing the Boolean functions 0 and 1. Each non-sink node is labeled with a Boolean variable *v* and has two out-edges labeled 1 (or "then") and 0 (or "else"). Each non-sink node represents the Boolean function corresponding to its edge "1," if $v = 1$, or the Boolean function corresponding to its edge "0," if $v = 0$.

An Ordered BDD (OBDD) is a BDD in which each variable is encountered no more than once in any path and always in the same order along each path. A Reduced OBDD (ROBDD) is an OBDD in which no nodes have equivalent behavior.

The operational profile is required to do the calculation, and only the operational profile for defect-related causes is required. A revised recursive algorithm for calculating the probability of a ROBDD is shown in Figure 7.12.

```
bddProbCal(X)
/* X = ite (xi, H,L),
   H = "High" branch of node xi
   L = "Low" branch of node xi
   P_H = Probability of "High" branch reach terminal node "1"
   P_L = Probability of "Low" branch reach terminal node "1"   */
{
        /*Consider "True" branch*/
        If  H is terminal node "1"
                P_H = 1.0
        else if  H is terminal node "0"
                P_H = 0.0
        else
                /*Go deeper to find the probability of H by calling this function itself*/
                P_H = bddProbCal(H)

        /*Consider "False" branch*/
        If  L is terminal node "1"
                P_L = 1.0
        else if  "False" branch is terminal node "0"
                P_L = 0.0
        else
                /*Go deeper to find the probability of L by calling this function itself*/
                P_L = bddProbCal(L)

        Probability[X] = Probability[x_i] × P_H + (1- Probability[x_i]) × P_L
        Return (Probability[X])
}
```

Figure 7.12 Algorithm for Calculating the Probability of a ROBDD

7.4.2 Reliability Prediction Results

Based on Table 7.5, the probability of failure is 0.9963. Therefore, the reliability is 0.0037. Table 7.8 shows detailed results for each operational mode.

Table 7.8 Reliability Prediction Results for Four Distinct Operational Modes

Mode	Probability of Failure (per year)	Reliability (per year)
Power-on	0.000012	0.999988
Normal	0.99624	0.00376
Calibration	0.15376	0.84624
Tuning	0.15376	0.84624

The reliability of the APP system is low because all defects found in the SRS have been considered as the actual defects remaining in the APP system. Therefore, through the CEG measure based on the SRS, the reliability has been underestimated.

As shown in Table 7.6, six out of seven defects found in the SRS have been corrected (either in the SDD or in the code). Using this information, the system reliability can be updated. The probability of failure is calculated to be 6.732×10^{-13} per demand. Therefore, the reliability is 0.999999999999327 per demand. This reliability estimate is closer to the actual reliability than the previous estimate based only on SRS information.

It should be noted that the prediction of the probability of failure based on the CEG metric changes from 0.9963 to 6.732×10^{-13} per demand while the number of defects only changed from seven to one. This is due to the characteristics of the defects. As explained in chapter 5, the probability of a defect leading to a system failure depends on the defect execution, infection, and propagation probabilities. The defects that are more likely to lead to system failure were fixed either in the SDD or in the source code. The only defect remaining in the code actually has a fairly low probability to lead to failure.

7.5 Lessons Learned

It should be noted that if the CEG measurement is performed manually, the results depend on the ability of the individual performing the measurement. It is strongly recommended to assign an analyst who knows the software structure sufficiently well to perform the CEG measure. This is mainly because:

1. It is difficult to differentiate the prime effects from the intermediate effects if the analyst is unfamiliar with the system.
2. It is difficult to identify logical relationships between the causes and the constraints without adequate knowledge of the system. Consequently, defects found through CEG measurements may not be correctly interpreted and the final reliability estimation may not be very meaningful.

7.6 References

[APP, Y1] "APP Module First µp SRS," Year Y1.

[APP, Y2] "APP Flux/Delta Flux/Flow Application SRS for SF1," Year Y2.

[APP, Y3] "APP Module µp2 System Software SRS," Year Y3.

[APP, Y4] "APP µp2 Flux/Delta Flux/Flow Application Software SRS," Year Y4.

[APP, Y5] "APP Module Communication Processor SRS," Year Y5.

[APP, Y6] *APP Instruction Manual*.

[Brace, 1990] B.R. Rudell and R. Bryant. "Efficient Implementation of a BDD Package," in *Proc. 27th ACM/IEEE Design Automation Conference*, 1990.

[Bryant, 1986] R.E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation." *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, 1986.

[Elmendorf, 1973] W.R. Elmendorf. "Cause-Effect Graphs in Functional Testing." *TR-00.2487*, 1073; IBM Systems Development Division, 1973.

[IEEE, 1988] "IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software," IEEE Std. 982.2-1988, 1988.

[Myers, 1976] G.J. Myers. *Software Reliability: Principle and Practices*. New York: Wiley-Interscience, pp. 218–227, 1976.

[Myers, 1979] G.J. Myers. *The Art of Software Testing*. New York: Wiley-Interscience, pp. 56–76, 1979.

[Nuisimulu, 1995] K. Nursimulu and R.L. Probert. "Cause-Effect Graphing Analysis and Validation of Requirements," in *Proc. Conference of the Centre for Advanced Studies on Collaborative Research*, 1995, pp. 15–64.

8. CAPABILITY MATURITY MODEL

The software Capability Maturity Model (CMM) is a framework that describes key elements of an effective software process. It covers practices for planning, engineering, and managing software development and maintenance. When followed, these key practices improve the ability of organizations to meet goals for cost, schedule, functionality, and product quality [IEEE, 1988].

The goals of this measure are to describe the principles and practices underlying software-process maturity and to help software organizations improve the maturity of their software processes [IEEE, 1988].

The CMM was replaced in 2001 with the Capability Maturity Model Integrated (CMMI) [Royce, 2002]. While CMM was developed to account for management and software engineering activities, CMMI extends the CMM by including systems engineering and integrated product development activities. Although CMMI has superseded CMM, the research published in this report focuses on the measures ranked in NUREG/GR-0019. Since NUREG/GR-0019 pre-dated the introduction of CMMI, the report did not evaluate the latter metric. In addition, evidence linking CMMI to fault content remains sparse. Once available, such evidence can be used to revise the models presented in this chapter.

The CMM measure can be applied as soon as requirements are available for review. As listed in Table 3.3, the applicable life cycle phases for CMM are Requirements, Design, Coding, Testing, and Operation.

8.1 Definition

Continuous process improvement is based on small, evolutionary steps rather than on revolutionary innovations. The CMM provides a framework for organizing these evolutionary steps into five maturity levels that lay successive foundations for continuous improvement.[16]

8.1.1 Definition of the Five Maturity Levels

These five maturity levels define an ordinal scale for measuring the maturity of an organization's software process and for evaluating its software process capability [Paulk, 1995].[13]

[16] Carnegie Mellon University, Software Engineering Institute, THE CAPABILITY MATURITY MODEL: GUIDELINES FOR IMPROVING THE SOFTWARE PROCESS, pp. 15–19, © 1995 Addison-Wesley Publishing Company Inc. Reproduced by permission of Pearson Education, Inc.

The five levels can be described as the following:

1. **Initial:** The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort and "heroic" efforts by individuals.[17]

   At the Initial Level, the organization typically does not provide a stable environment for developing and maintaining software. During a crisis, projects typically abandon planned procedures and revert to coding and testing. Success depends entirely on having an exceptional manager and a seasoned and effective software team. Occasionally, capable and forceful software managers can resist the pressures to take shortcuts in the process; but when they leave the project, their stabilizing influence leaves with them. Even a strong engineering process cannot overcome the instability created by the absence of sound management practices.[14]

   The software process capability of Level 1 organizations is unpredictable because the software process is constantly being changed or modified as the work progresses (i.e., the process is ad hoc). Schedules, budgets, functionality, and product quality are generally unpredictable. Performance depends on the capabilities of individuals and varies with their innate skills, knowledge, and motivations. Few stable software processes are evident and performance can be predicted only by individual capability.

2. **Repeatable:** Basic project management processes are established to track cost, schedule, and functionality. The necessary discipline exists to repeat earlier successes on projects with similar applications.[14]

   At the Repeatable Level, policies for managing a software project and procedures to implement those policies are established. Planning and managing new projects are based on experience with similar projects. In Level 2 effective management processes for software projects are institutionalized, which allow organizations to repeat successful practices developed on earlier projects, even if the specific processes implemented by the projects may differ. An effective process should be practiced, documented, enforced, trained, measured, and capable of improvement.

   Projects in Level 2 organizations have installed basic software management controls. Realistic project commitments are based on the results observed in previous projects and on the requirements of the current project. [14] The software managers for a project track software costs, schedules, and functionality; problems in meeting commitments are identified when they arise. Software requirements and the products developed to satisfy them are baselined and their integrity is controlled. Software project standards are defined and the organization ensures they are faithfully followed. The software project works with its subcontractors, if any, to establish a strong customer-supplier relationship.[14]

---

[17] Carnegie Mellon University, Software Engineering Institute, THE CAPABILITY MATURITY MODEL: GUIDELINES FOR IMPROVING THE SOFTWARE PROCESS, pp. 15–19, © 1995 Addison-Wesley Publishing Company Inc. Reproduced by permission of Pearson Education, Inc.

The software process capability of Level 2 organizations can be summarized as disciplined, because the planning and tracking of the software project is stable and earlier successes can be repeated. The project's process is under the effective control of a project management system, following realistic plans based on the performance of previous projects.[18]

3. **Defined:** The software process for both management and engineering activities is documented, standardized, and integrated into a standard software process for the organization. All projects use an approved, tailored version of the organization's standard software process for developing and maintaining software. [15]

   At the Defined Level, the standard process for developing and maintaining software across the organization is documented, including both software engineering and management processes. These processes are integrated into a coherent whole. This standard process is referred to throughout the CMM as the organization's standard software process. Processes established at Level-3 are used (and changed, as appropriate) to help the software managers and technical staff perform more effectively. The organization exploits effective software engineering practices when standardizing its software processes. There is a group that is responsible for the organization's software-process activities, e.g., a software engineering-process group. An organization-wide training program is implemented to ensure that the staff and managers have the knowledge and skills required to fulfill their assigned roles.

   The software-process capability of Level-3 organizations can be summarized as standard and consistent, because both software engineering and management activities are stable and repeatable. Within established product lines, cost, schedule, and functionality are under control, and software quality is tracked. This process capability is based on a common, organization-wide understanding of the activities, roles, and responsibilities in a defined software process.[15]

4. **Managed:** Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled.[15]

   At the Managed Level, the organization sets quantitative quality goals for both software products and processes. Productivity and quality are measured for important software-process activities across all projects as part of an organizational measurement program. An organization-wide software-process database is used to collect and analyze the data available from the projects' defined software processes. Software processes are implemented with well-defined and consistent measurements at Level 4. These measurements establish the quantitative foundation for evaluating the projects' software processes and products.

[18] Carnegie Mellon University, Software Engineering Institute, THE CAPABILITY MATURITY MODEL: GUIDELINES FOR IMPROVING THE SOFTWARE PROCESS, pp. 15–19, © 1995 Addison-Wesley Publishing Company Inc. Reproduced by permission of Pearson Education, Inc.

The software-process capability of Level 4 organizations can be summarized as predictable, because the process is measured and operates within measurable limits. This level of process capability allows an organization to predict trends in process and product quality within the quantitative bounds of these limits. When these limits are exceeded, action is taken to correct the situation. Software products are of predictably high quality.

5. **Optimizing:** Continuous process improvement results from quantitative feedback and from piloting innovative ideas and technologies.[19]

   At the Optimizing Level, the entire organization is focused on continuous process improvement. The organization has the means to identify weaknesses and strengthen the process proactively, with the goal of preventing the occurrence of defects. Data on the effectiveness of the software process is used to perform cost-benefit analyses of new technologies and proposed changes to the organization's software process. Innovations that exploit the best software engineering practices are identified and transferred throughout the organization.

   Software project teams in Level 5 organizations analyze defects to determine their causes. Software processes are evaluated to prevent known types of defects from recurring, and lessons learned are disseminated to other projects.

   The software-process capability of Level 5 organizations can be characterized as continuously improving, because Level 5 organizations are continuously striving to improve the range of their process capability, thereby improving the process performance of their projects. Improvement occurs both by incremental advancements in the existing process and by innovations using new technologies and methods.

Organizing the CMM into the five levels shown in Figure 8.1 prioritizes improvement actions for increasing software process maturity. The labeled arrows in Figure 8.1 indicate the type of process capability being institutionalized by the organization at each step of the maturity framework [Paulk, 1993].[16]

[19] Carnegie Mellon University, Software Engineering Institute, THE CAPABILITY MATURITY MODEL: GUIDELINES FOR IMPROVING THE SOFTWARE PROCESS, pp. 15–19, © 1995 Addison-Wesley Publishing Company Inc. Reproduced by permission of Pearson Education, Inc.

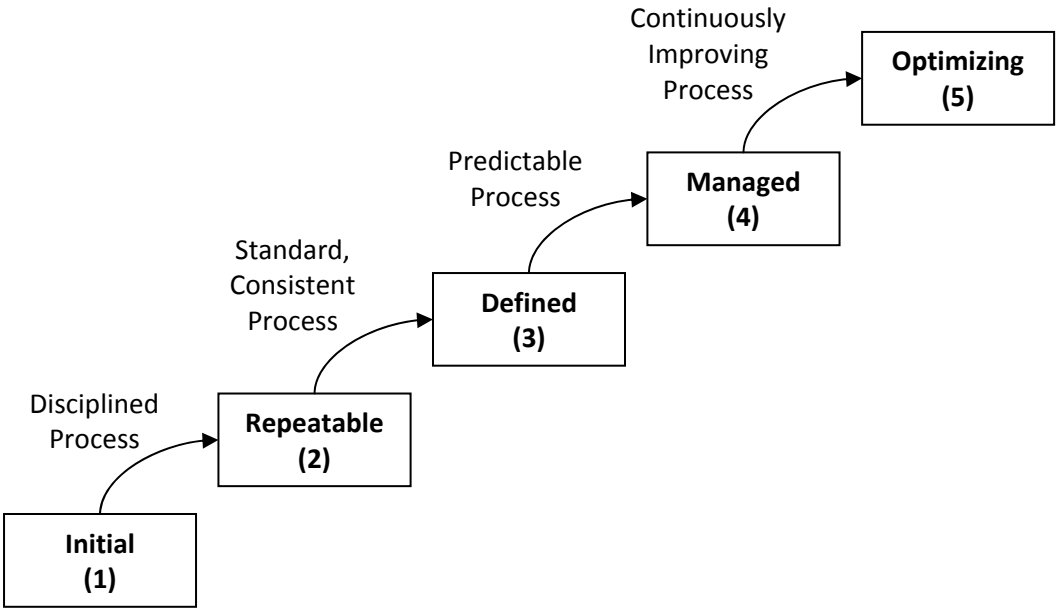


Figure 8.1 The Five Levels of Software Process Maturity[20]

8.1.2 Definition of the Key Process Areas (KPAs)

Each maturity level except for Level 1 (Initial) is divided into Key Process Areas (KPAs). Each KPA identifies a cluster of related activities that, when performed collectively, achieve a set of goals considered important for establishing process capability at that maturity level [Paulk, 1993]. The KPAs have been defined to reside at a single maturity level.

Figure 8.2 represents the KPAs by maturity levels.

[20] Carnegie Mellon University, Software Engineering Institute, THE CAPABILITY MATURITY MODEL: GUIDELINES FOR IMPROVING THE SOFTWARE PROCESS, pp. 15–19, © 1995 Addison-Wesley Publishing Company Inc. Reproduced by permission of Pearson Education, Inc.

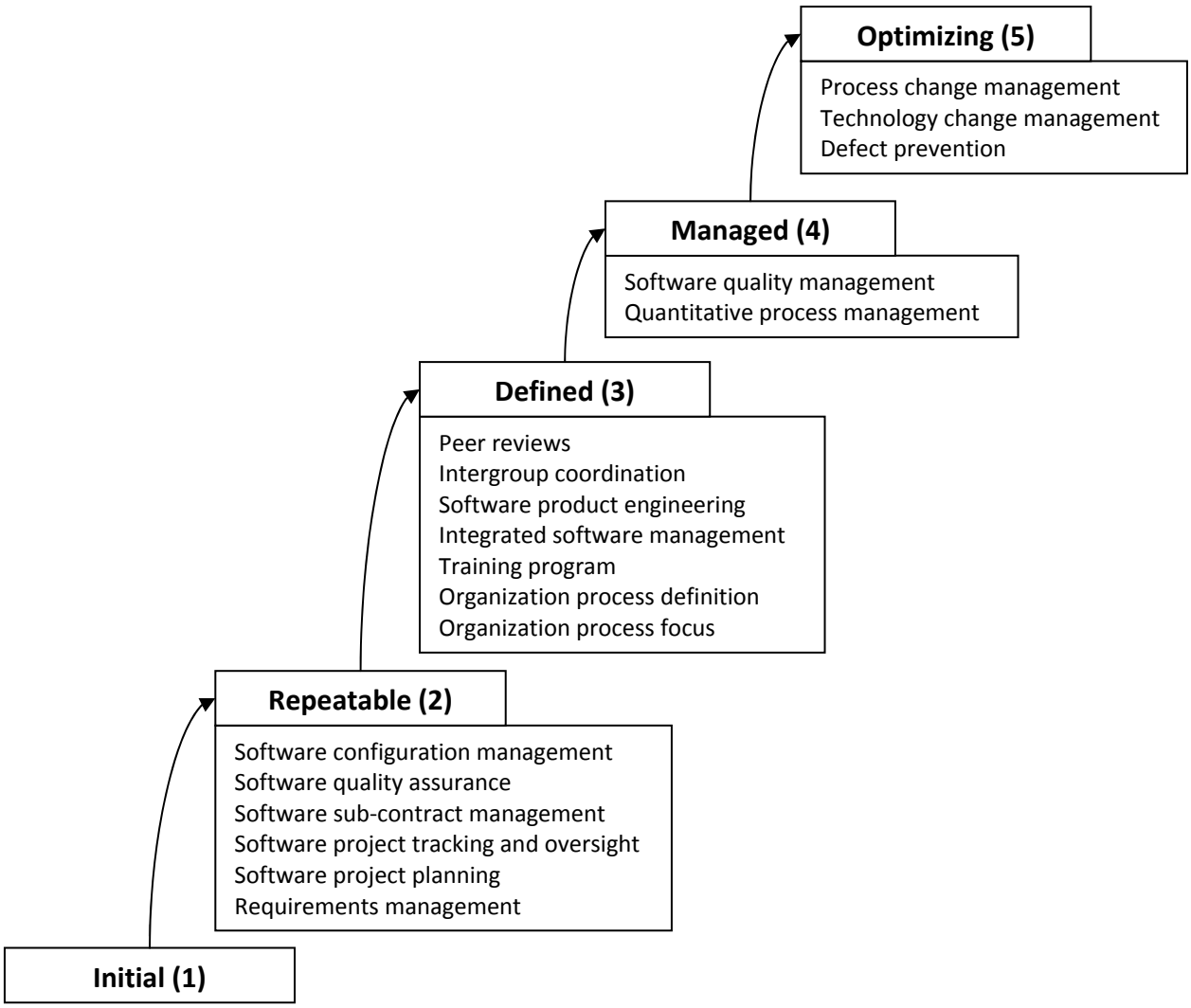


Figure 8.2 The Key Process Areas by Maturity Levels[21]

8.1.2.1 KPAs for Level 2

The KPAs at Level 2 focus on the software project's concerns related to establishing basic, project-management controls. Descriptions of each of the KPAs for Level 2 are given below:

1. **Requirements Management**: The purpose of Requirements Management is to establish a common understanding between the customer and the software project of the customer's requirements that will be addressed by the software project. This agreement with the customer is the basis for planning (as described in Software Project Planning) and managing (as described in Software Project Tracking and Oversight) the software

[21] Carnegie Mellon University, Software Engineering Institute, THE CAPABILITY MATURITY MODEL: GUIDELINES FOR IMPROVING THE SOFTWARE PROCESS, pp. 15–19, © 1995 Addison-Wesley Publishing Company Inc. Reproduced by permission of Pearson Education, Inc.

project. Control of the relationship with the customer depends on following an effective change control process (as described in Software Configuration Management).

2. **Software Project Planning**: The purpose of Software Project Planning is to establish reasonable plans for performing the software engineering and for managing the software project. These plans are the necessary foundation for managing the software project (as described in Software Project Tracking and Oversight). Without realistic plans, effective project management cannot be implemented.

3. **Software Project Tracking and Oversight**: The purpose of Software Project Tracking and Oversight is to establish adequate insight into actual progress, so that management can take effective actions if the software project's performance deviates significantly from the software plans.

4. **Software Subcontract Management:** The purpose of Software Subcontract Management is to select qualified software subcontractors and manage them effectively. It combines the concerns of Requirements Management, Software Project Planning, and Software Project Tracking and Oversight for basic management control, with the necessary coordination of Software Quality Assurance and Software Configuration Management, and applies these standards to the subcontractor as appropriate.

5. **Software Quality Assurance:** The purpose of Software Quality Assurance is to provide management with appropriate visibility into the process being used by the software project and of the products being built. Software Quality Assurance is an integral part of most software engineering and management processes.

6. **Software Configuration Management:** The purpose of Software Configuration Management is to establish and maintain the integrity of the products of the software project throughout the project's software life cycle. Software Configuration Management is an integral part of most software engineering and management processes.

8.1.2.2 KPAs for Level-3

The KPAs at Level-3 address both project and organizational issues, as the organization establishes an infrastructure that institutionalizes effective software engineering and management processes across all projects. Each of the KPAs for Level-3 is described below:

1. **Organization Process Focus**: The purpose of Organization Process Focus is to establish the organizational responsibility for software process activities that improve the organization's overall software-process capability. The primary result of the Organization Process Focus activities is a set of software process assets, which are described in Organization Process Definition. These assets are used by the software projects, as described in Integrated Software Management.

2. **Organization Process Definition**: The purpose of Organization Process Definition is to develop and maintain a usable set of software process assets that improve process performance across the projects and provide a basis for cumulative, long-term benefits to the organization. These assets provide a stable foundation that can be institutionalized via mechanisms such as training, which is described in Training Program.

3. **Training Program**: The purpose of the Training Program is to develop the skills and knowledge of individuals, so they can perform their roles effectively and efficiently. Training is an organizational responsibility, but each software projects should identify required skill sets and provide necessary training when the project's requirements are unique.

4. **Integrated Software Management**: The purpose of Integrated Software Management is to integrate the software engineering and management activities into a coherent, defined software process that is tailored from the organization's standard software process and related process assets, which are described in Organization Process Definition. This tailoring is based on the business environment and technical needs of the project, as described in Software Product Engineering. Integrated Software Management evolves from Software Project Planning and Software Project Tracking and Oversight at Level 2.

5. **Software Product Engineering**: The purpose of Software Product Engineering is to consistently perform a well-defined engineering process that integrates all the software engineering activities to produce correct, consistent software products effectively and efficiently. Software Product Engineering describes the technical activities of the project, e.g., requirements analysis, design, code, and testing.

6. **Intergroup Coordination:** The purpose of Intergroup Coordination is to establish a means for the software engineering group to participate actively with the other engineering groups, so the project is better able to satisfy the customer's needs effectively and efficiently. Intergroup Coordination is the interdisciplinary aspect of Integrated-Software Management not only should the software process be integrated, but the software engineering group's interactions with other groups must be coordinated and controlled.

7. **Peer Reviews:** The purpose of Peer Reviews is to remove defects from the software work products early and efficiently. An important corollary effect is to develop a better understanding of the software work products and of the defects that can be prevented. The peer review is an important and effective engineering method that is implemented in Software Product Engineering area by reviews and structured walkthroughs.

8.1.2.3 KPAs for Level 4

The KPAs at Level 4 focus on establishing a quantitative understanding of both the software process and the software work products being built. The two KPAs at this level, Quantitative

Process Management and Software Quality Management, are highly interdependent, as described below:

1. **Quantitative Process Management:** The purpose of Quantitative-Process Management is to quantitatively control the process performance of the software project. Software-process performance represents the actual results achieved from following a software process. The focus is on identifying special causes of variation within a measurably stable process and correcting, as appropriate, the circumstances that caused the transient variation to occur. Quantitative-Process Management adds a comprehensive measurement program to the practices of Organization-Process Definition, Integrated-Software Management, Intergroup Coordination, and Peer Reviews.

2. **Software Quality Management:** The purpose of Software Quality Management is to develop a quantitative understanding of the quality of the project's software products. Software Quality Management applies a comprehensive measurement program to the software work products described in Software Product Engineering.

8.1.2.4 KPAs for Level 5

The KPAs at Level 5 cover the issues that both the organization and the projects must address to implement continuous and measurable software-process improvement. Descriptions of each of the KPAs for Level 5 are given below:

1. **Defect Prevention:** The purpose of Defect Prevention is to identify the causes of defects and prevent them from recurring. The software project analyzes defects, identifies their causes, and changes its defined software process, as is described in Integrated-Software Management. Process changes of general value are communicated to other software projects, as is described in Process Change Management.

2. **Technology Change Management:** The purpose of Technology Change Management is to identify beneficial new technologies (i.e., tools, methods, and processes) and incorporate them into the organization in an orderly manner, as is described in Process Change Management. The focus of Technology-Change Management is on introducing innovation efficiently in an ever-changing world.

3. **Process Change Management:** The purpose of Process Change Management is to continually improve the software processes used in the organization with the intent of improving software quality, increasing productivity, and decreasing the cycle time for product development. Process Change Management takes the incremental improvements of Defect Prevention and the innovative improvements of Technology Change Management and makes them available to the entire organization.

8.2 Measurement Rules

8.2.1 Standard SEI-CMM Assessment

The Software Engineering Institute (SEI) assessment method used with the Capability Maturity Model for Software (SW-CMM) is named Capability Maturity Model-Based Appraisal for Internal Process Improvement (CBA IPI). This method is used by organizations to provide an accurate picture of the strengths and weaknesses of the organization's current software process, using the CMM as a reference model, and to identify KPAs for improvement.

The CBA IPI method is an assessment of an organization's software process capability by a trained group of professionals who work as a team to generate findings and ratings relative to the CMM KPAs within the assessment scope. The findings are generated from data collected from questionnaires, document review, presentations, and in-depth interviews with middle managers, project leaders, and software practitioners [Dunaway, 2001].

The CBA IPI method satisfies requirements established in the CMM Appraisal Framework (CAF), Version 1.0 [Masters, 1995]. Figure 8.3 illustrates the basic CAF activities.

Planning and preparation are the key to success of any appraisal. As illustrated in Figure 8.3, planning and preparation involve analyzing the appraisal's requirements, selecting and preparing the appraisal team, selecting and preparing the appraisal participants, and developing and documenting the appraisal plan.

Conducting an appraisal focuses on collecting and recording data in the form of notes, consolidating data into a manageable set of observations, determining their validity as findings, and their coverage of the appraisal scope and using those findings to produce ratings of the appraised entity's software process with respect to the CMM.

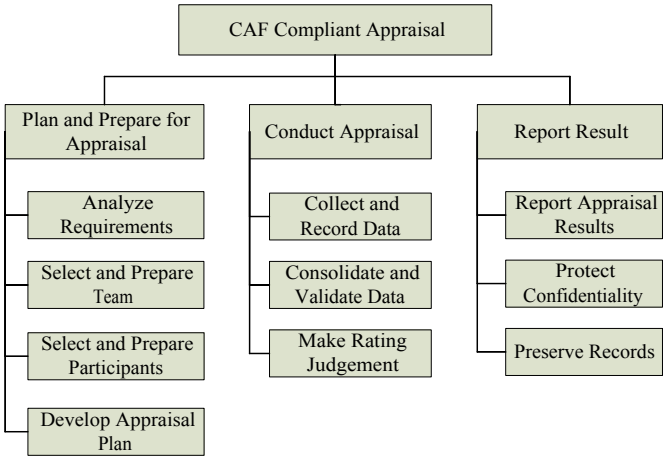


Figure 8.3 CMM Appraisal Framework Activities

The Reporting phase of an appraisal involves reporting appraisal results to sponsors, the appraisal method owner, the SEI, and, optionally, the appraised entity; and preserving appraisal records. In this research, this phase was not required.

The following are some general rules when conducting an appraisal:

1. The appraisal team must come to consensus on the ratings that it provides to an appraised entity. This consensus is one step in assuring that the entire team supports the appraisal report. Without consensus, the appraisal team cannot expect the appraised entity to have a high level of confidence in the contents of the report.
2. All ratings must be based on the CMM and only on the CMM. A CMM-based appraisal, by definition, is using the CMM as a framework for evaluating an appraised entity's software process. The appraisal team must, therefore, maintain fidelity to the model in its rating process. An appraisal method cannot add new KPAs to the model or delete existing KPAs.
3. Ratings must be based on the data the appraisal team collects during the appraisal process. By basing ratings on findings that have been validated by the team, and directly or indirectly by the appraised entity, the appraisal team can achieve a high level of confidence in their accuracy.

Lead Assessors are authorized by SEI to market and perform CBA IPI assessments either for third-party organizations or for their own organization's internal use. A list of SEI authorized Lead Assessors can be found at the SEI website [SEI, 2006]. The cost of a formal assessment conducted by an authorized lead assessor would be of the order of $50,000.

The key step of the assessment is to make rating judgments and determine the maturity level based on the collected data. Four rating values are provided for goals and KPAs: satisfied, unsatisfied, not applicable, or not rated. If a KPA is determined to be not applicable in the organization's environment, then all of the goals for that KPA are deemed not applicable. Conversely, if a KPA is determined to be applicable in the organizations environment, then all of the goals for that KPA are applicable.

In the following subsections, the detailed measurement rules are provided when conducting an appraisal.

8.2.1.1 Rules for Judging Satisfaction of Goals

1. Rate the goal "satisfied" if the associated findings indicate that this goal is implemented and institutionalized either as defined in the CMM with no significant weaknesses or that an adequate alternative exists.
2. Rate the goal "unsatisfied" if the associated findings indicate that there are significant weaknesses in the appraised entity's implementation and institutionalization of this goal as defined in the CMM and no adequate alternative is in place.
3. Rate the goal "not applicable" if the goal is not applicable in the organization's environment.

4. Rate the goal "not rated" if the associated findings do not meet the method's defined criteria for coverage or if the goal falls outside of the scope of the appraisal.

8.2.1.2 Rules for Judging Satisfaction of KPAs

1. Rate the KPA "satisfied" if all of the goals are rated "satisfied."
2. Rate the KPA "unsatisfied" if one or more goals are rated as "unsatisfied."
3. Rate the KPA "not applicable" if the KPA is not applicable in the organization's environment.
4. Rate the KPA "not rated" if any of the goals are rated "not rated" or if the KPA falls outside of the scope of the appraisal.

8.2.1.3 Rules for Determining Maturity Level

1. Maturity level ratings depend exclusively on KPA ratings. The appraisal team bases maturity level ratings solely on the KPA ratings. No additional team judgments are required.
2. A maturity level is satisfied if all KPAs within that level and each lower level are satisfied or not applicable. For example, rating of maturity level-3 requires that all KPAs within levels 2 and 3 be satisfied or not applicable.
3. The maturity level rating is that of the highest maturity level satisfied.

8.2.2 UMD-CMM Assessment

As far as the APP is concerned, a standard CMM level assessment had not been performed for the organization that developed the APP system. Furthermore, the APP system was 10 years old. As a consequence, any results of an assessment would have been post-mortem and as such, not qualify for a formal assessment.

To obtain an informal assessment, the SW-CMM Maturity Questionnaire [Zubrow, 1994] was provided to the remaining personnel involved in the development of the APP system.

UMD-CMM assessment followed the procedure defined in Section 8.2.1. The only discrepancy was in the composition of the team.

In order to conduct appraisals, a team of assessors who had gone through a complete training program and a lead assessor who had significant experience in the field of CMM appraisal was required. To become a SEI authorized assessor, normally the person should first attend a five-day course offered by SEI. After attending the course, participants would be qualified as candidate lead appraisers. To become authorized, candidate lead appraisers must be observed by a qualified observing lead appraiser and receive a satisfactory recommendation. Lead appraisers may provide appraisal services for their own organization or other organizations and deliver appraisal training to appraisal teams.

One UMD graduate student with experience in software engineering was sent to the five-day training course, was qualified as candidate lead appraiser, and performed the assessment.

8.3 Measurement Results

The Maturity Questionnaire was distributed to the APP development team members. Table 8.1 provides the summary of the answers to questions to this Questionnaire. The results in Table 8.1 show the ratio of the number of satisfied goals over the total applicable goals.

It should be noted that the summary was based on one respondent's answers since he was the manager of the APP development team. The respondent was explained the design of the questionnaire and told what KPAs meant and how the CMM levels are defined. He also had some prior knowledge about the CMM in general given his experience in the software field (22 years).

Table 8.1 Summary of the Answers to the Questions in the Maturity Questionnaire

CMM Level	No.	KPAs	Results
Repeatable (2)	1	Requirement Management	6/6
	2	Software Project Planning	7/7
	3	Software Project Tracking and Oversight	7/7
Repeatable (2)	4	Software Subcontract Management	Not Applicable
	5	Software Quality Assurance	7/7
	6	Software Configuration Management	8/8
Defined (3)	1	Organization Process Focus	2/7
	2	Organization Process Definition	4/6
	3	Training Program	7/7
	4	Integrated Software management	4/6
	5	Software Product Engineering	6/6
	6	Intergroup Coordination	6/7
	7	Peer Reviews	5/6

Table 8.2 shows the results obtained after the application of the measurement rules stated in Section 8.2.2.2 to the responses to the questions for each of the KPAs.

Table 8.2 Result of Application of KPA Satisfaction Level Measurement Rules

CMM Level	No.	KPAs	KPA Satisfaction Level
Repeatable (2)	1	Requirement Management	Satisfied
	2	Software Project Planning	Satisfied
	3	Software Project Tracking and Oversight	Satisfied
	4	Software Subcontract Management	Not Applicable
	5	Software Quality Assurance	Satisfied
	6	Software Configuration Management	Satisfied
Defined (3)	1	Organization Process Focus	Unsatisfied
	2	Organization Process Definition	Unsatisfied
	3	Training Program	Satisfied
	4	Integrated Software management	Unsatisfied
	5	Software Product Engineering	Satisfied
	6	Intergroup Coordination	Unsatisfied
	7	Peer Reviews	Unsatisfied

On analyzing the answers to the questions in the maturity questionnaire for the APP the following observations were made:

1. From the respondent's answers, it was clear the APP could not be assessed at CMM level-3. CMM level-3 focuses on having a generalized organizational level policy for all the activities in the software development process and that a project must tailor its own software process from these generalized organizational level policies. In this regard the respondent believed that the developer had some organizational level policies for both hardware and software systems developed by them. However, according to the rules, developer still did not reach CMM level-3.

2. The main focus of CMM level four is the collection of detailed measures of the software process and product. Both the software process and products are quantitatively understood and controlled. The developer did not have this kind of data collected across projects. This is why it could not be assessed above level-3.

According to the analysis, it is clear that all the KPAs in CMM level 2 are satisfied and five out of seven KPAs in CMM level-3 are not satisfied. Therefore, the APP is CMM level 2.

8.4 RePS Construction from CMM

In order to estimate reliability using CMM as the base measure, it is required to construct a model that links CMM to the number of defects in the software. Once there is a model to estimate the number of defects in the software using CMM as the base measure, then the exponential model can be applied to estimate the reliability of the software.

8.4.1 CMM Maturity Levels vs. Number of Defects

Historical industry data collected by Software Productivity Research Inc. [Jones, 1995] links the CMM level to the number of defects per function point. Table 8.3 presents this data.

Table 8.3 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

The CMM level of the APP is assessed to be CMM level 2. The functional size of the APP is 301 function points. Table 8.4 presents the estimation of defects for the APP.

Table 8.4 Defect Estimation for the APP Using CMM

CMM Level	Average Defects/Function Point	FP	Total Number of Defects
Level 2	0.44	301	132.44

The next step is the partitioning of the defects based on the criticality of the defects. Using Table 6.7, the partitioned number of defects (based on the severity level) for the APP using

CMM is presented in Table 8.5. The Table 6.7 values are listed in parentheses for each defect category.

Table 8.5 Partitioned Number of Defects (Based On Severity Level) for the APP Using CMM

Total Number of Defects	**Defects (Critical)** (0.0185)	**Defects (Significant)** (0.1206)	**Defects (Minor)** (0.3783)	**Defects (Cosmetic)** (0.4826)	**Defects (Critical + Significant)** (0.1391)
132.44	2.45	15.97	50.10	63.92	18.42

8.4.2 Reliability Estimation

The probability of success-per-demand is obtained using Musa's exponential model [Musa, 1990] [Smidts, 2004]:

$$p_s(CMM) = e^{\left(-K \times N_{CMM} \times \tau / T_L\right)} \tag{8.1}$$

and

$$N_{CMM} = N_{CMM,critical} + N_{CMM,significant} = 18.42 \tag{8.2}$$

where

$p_s(CMM)$	Reliability estimation for the APP system using the CMM measure.
K	Fault Exposure Ratio, in failure/defect.
N_{CMM}	Number of defects estimated using the CMM measure.
τ	Average execution-time-per-demand, in seconds/demand.
T_L	Linear execution time of a system, in seconds.
$N_{CMM,critical}$	Number of delivered *critical defects* (severity 1).
$N_{CMM,significant}$	Number of delivered *significant defects* (severity 2).

Since *a priori* knowledge of the defects' location and their impact on failure probability is not known, the average K value given in [Musa, 1987] [Musa, 1990] [Smidts, 2004], which is $4.2 \times 10^{-7} failure/defect$ must be used.

The *linear execution time*, T_L, is usually estimated as the ratio of the execution time and the software size on a single microprocessor basis [Musa, 1987] [Musa, 1990] [Smidts, 2004]. In the case of the APP system, however, there are three parallel subsystems, each of which has a microprocessor executing its own software. Each of these three subsystems has an estimated linear execution time. Therefore, there are several ways to estimate the linear execution time for the entire APP system, such as using the average value of these three subsystems.

For a safety-critical application, like the APP system, the UMD research team suggests to make a conservative estimation of T_L by using the minimum of these three subsystems' T_L. Namely,

$$\begin{aligned} T_L &= \min\{T_L(\mu p1), T_L(\mu p2), T_L(CP)\} \\ &= min\{0.018, 0.009, 0.021\} \\ &= 0.009 \text{ seconds} \end{aligned} \tag{8.3}$$

Where

$T_L(\mu p1)$	Linear execution time of Microprocessor 1 (μp1) of the APP system. T_L *(μp1)* = 0.018 seconds (refer to Chapter 17).
$T_L(\mu p2)$	Linear execution time of Microprocessor 2 (μp2) of the APP system. T_L *(μp2)* = 0.009 seconds (refer to Chapter 17).
$T_L(CP)$	Linear execution time of Communication Microprocessor (CP) of the APP system. T_L *(CP)* = 0.021 seconds (refer to Chapter 17).

Similarly, the *average execution-time-per-demand*, τ, is also estimated on a single microprocessor basis. Each of the three subsystems in APP has an estimated average execution-time-per-demand. To make a conservative estimation, the average execution-time-per-demand for the entire APP system is the maximum of the three subsystems' τ. Namely:

$$\begin{aligned} \tau &= \max\{\tau(\mu p1),\ \tau(\mu p2),\ \tau(CP)\} \\ &= \max\{0.082,\ 0.129,\ 0.016\} \\ &= 0.129 \text{ seconds/demand} \end{aligned} \tag{8.4}$$

Where

$\tau(\mu p1)$	Average execution-time-per-demand of Microprocessor 1 (μp1) of the APP system. *τ(μp1)* = 0.082 seconds/demand (refer to Chapter 17).
$\tau(\mu p2)$	Average execution-time-per-demand of Microprocessor 2 (μp2) of the APP system. *τ(μp2)* = 0.129 seconds/demand (refer to Chapter 17).
$\tau(CP)$	Average execution-time-per-demand of Communication Microprocessor (CP) of the APP system. *τ(CP)* = 0.016 seconds/demand (refer to Chapter 17).

Thus the reliability for the APP system using the CMM measure is given by:

$$\begin{aligned} p_s(CMM) &= e^{\left(-4.2\times10^{-7}\times18.42\times0.129/0.009\right)} \\ &= 0.999889118 \end{aligned} \tag{8.5}$$

A more accurate estimation of reliability using CMM for the APP system can be obtained by enhancing the estimation of K. A value of K for the safety-critical system, rather than the average value 4.2×10^{-7} failure/defect, should be used in Equation 8.1.

8.5 Lessons Learned

The standard CMM-level assessment was not performed for the company that developed the software module. Furthermore, the software module was more than ten years old and most of the members of the development team were no longer working with the company. The CMM assessment could only be conducted based on the "surviving" team member's answers to the Maturity Questionnaire. As a consequence, any results of an assessment are post-mortem and as such do not qualify for a formal assessment. The research team had to take an alternative informal approach as described in Section 8.2.2.

For recently developed software, the issues encountered during this research should not apply since more and more companies/organizations are encouraged to obtain a CMM (now CMMI) certification.

8.6 References

[Dunaway, 2001] D.K. Dunaway and S. Masters. "CMM-Based Appraisal for Internal Process Improvement (CBA IPI) Version 1.2 Method Description," Software Engineering Institute, CMU/SEI-2001-TR-033. Available: http://www.sei.cmu.edu/publications/documents/01.reports/01tr033.html [Nov. 2001].

[Jones, 1995] C. Jones. *Measuring Global Software Quality*. Burlington, MA: Software Productivity Research, 1995.

[Jones, 1997] C. Jones. *Applied Software Measurement: Assuring Productivity and Quality*. McGraw Hill, Inc., 1997.

[Masters, 1995] S. Masters and C. Bothwell. "CMM Appraisal Framework, Version 1.0," Software Engineering Institute, CMU/SEI-95-TR-001. Available: http://www.sei.cmu.edu/publications/documents/95.reports/95-tr-001/95-tr-001-abstract.html [Feb. 1995].

[Musa, 1987] J.D. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Applications*. New York: McGraw-Hill, 1987.

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

[Paulk, 1995] M.C. Paulk et al. *The Capability Maturity Model: Guidelines to improving the Software Process*. CMU: Addison-Wesley, 1995.

[Paulk, 1993] M.C. Paulk et al. *Key Practices of the Capability Maturity Model, Version 1.1*. CMU/SEI-93-TR-25, 1993.

[Royce, 2002] W. Royce. "CMM vs. CMMI: From Conventional to Modern Software Management," Available: http://www.cdainfo.com/down/1-Desarrollo/CMM2.pdf [Jul. 2010].

[SEI, 2006] Software Engineering Institute at Carnegie Mellon University. "SEI Appraisal Program Directories," Available: http://www.sei.cmu.edu/appraisal-program/directory/index.html [Jan. 2006].

[Smidts, 2004] C. Smidts and M. Li. "Validation of a Methodology for Assessing Software Quality," NRC, Office of Nuclear Regulatory Research, Washington DC NUREG/CR-6848, 2004.

[Zubrow, 1994] D. Zubrow et al. "Maturity Questionnaire." Software Engineering Institute, CMU/SEI-94-SR-7. Available: http://www.sei.cmu.edu/publications/documents/94.reports/94.sr.007.html [Jun. 1994].

9. COMPLETENESS

The completeness measure, COM, determines the completeness of the SRS.

The COM measure provides a systematic guideline to identify the incompleteness defects in the SRS. Also, the values determined for the primitives associated with the COM measure can be used to identify problem areas within the software specification.

The COM measure can be applied as soon as the requirements are available. As listed in Table 3.3, the applicable life cycle phases for the COM measure are Requirements, Design, Coding, Testing, and Operation.

9.1 Definition

The COM measure is the weighted sum of ten derived measures, D_1 through D_{10} [IEEE, 1988] [Murine, 1985]:

$$COM = \sum_{i=1}^{10} w_i D_i \tag{9.1}$$

where

COM	completeness measure,
w_i	the weight of the i-th derived measure,
D_i	the i-th derived measure,

Where for each i = 1, ..., 10, each weight w_i has a value between 0 and 1, the sum of the weights is equal to 1, and each D_i is a derived measure with a value between 1 and 0.

The weighting factor is dependent on the characteristics of the project. For example, a database project would be weighted heavily for the data-reference attribute. For each project, the weighting factors (w_i) should be determined by survey or expert opinion.

Since the value of the COM is subjectively determined, the RePS that uses the COM measure is based on the incompleteness defects identified in the SRS during the measurement but not on the value of COM (refer to Section 9.4).

Each derived measure is determined as follows:

$D_1 = (B_2 - B_1)/B_2$	fraction of functions satisfactorily defined
$D_2 = (B_4 - B_3)/B_4$	fraction of data references having an origin
$D_3 = (B_6 - B_5)/B_6$	fraction of defined functions used
$D_4 = (B_8 - B_7)/B_8$	fraction of referenced functions defined
$D_5 = (B_{10} - B_9)/B_{10}$	fraction of decision points whose conditions and condition options are all used
$D_6 = (B_{12} - B_{11})/B_{12}$	fraction of condition options having processing
$D_7 = (B_{14} - B_{13})/B_{14}$	fraction of calling routines whose parameters agree with the called routines defined parameters
$D_8 = (B_{12} - B_{15})/B_{12}$	fraction of condition options that are set
$D_9 = (B_{17} - B_{16})/B_{17}$	fraction of set condition options processed
$D_{10} = (B_4 - B_{18})/B_4$	fraction of data references having a destination

where B_1 to B_{18} are primitives defined as follows:

- B_1 number of functions not satisfactorily defined.
- B_2 number of functions.
- B_3 number of data references not having an origin.
- B_4 number of data references.
- B_5 number of defined functions not used.
- B_6 number of defined functions.
- B_7 number of referenced functions not defined.
- B_8 number of referenced functions.
- B_9 number of decision points missing condition(s).
- B_{10} number of decision points.
- B_{11} number of condition options having no processing.
- B_{12} number of condition options.
- B_{13} number of calling routines whose parameters are not agreeing with the called routines defined parameters.
- B_{14} number of calling routines.
- B_{15} number of condition options not set.
- B_{16} number of set condition options having no processing.
- B_{17} number of set condition options.
- B_{18} number of data references having no destination.

Assessment of some of the derived measures (D_i) may be more reliable at the design and coding level since they refer to design and coding characteristics described at a high level in the SRS. However, high-level estimates of D_i should be available during the requirements phase.

The following definitions were used while counting primitives:

Called Routine: a routine referred by another routine.
Called Routines Parameter: a prerequisite data used in the called routine in order to perform its required functions.
Calling Routine: a routine making reference to another routine.
Condition: a leaf-level expression which cannot be broken down into a simpler one.
Condition Option: one of the possible results determined by the condition.
Data Reference Origin: the source of the data manipulated by the data reference.
Data Reference: a *data reference* is a function which manipulates either internal or external data.
Data Reference Destination: the destination of the data manipulated by the data reference.
Decision Point: a process element that routes the system to one of several alternative outgoing paths, depending on its condition.
Defined Function: a function that is explicitly described in the SRS.
Function: a defined objective or characteristic action in the software requirement specification (SRS), usually involved in processing input(s) and/or generating output(s).
Processed Condition Option: a condition option is *processed* if a function is satisfactorily defined to process this condition option.
Referenced Function: a function that is implied or referred by another function.
Routine: a set of sequential functions. A routine is usually bulleted as a functional section in the SRS.
Satisfactorily Defined Function: a defined function that is correct, unambiguous, unique, and verifiable.
Set Condition: a condition is *set* if it is defined before it is used.
Set Condition Option: a condition option is *set* if all conditions are set.
Used Function: a function that is employed in the control flow or referred by other function(s) employed in the control flow.

9.2 Measurement Rules

The following measurement rules were tailored for the purpose of identifying defects (incomplete functional requirements) in the SRS and estimating software reliability.

9.2.1 B1: Number of Functions Not Satisfactorily Defined

Within the context of the COM measurement, a *satisfactorily defined function* is a function meeting the criteria specified in [IEEE, 1998].

Refer to Section 9.2.6 for the definition of a *defined function*.

More specifically, a function is a s*atisfactorily defined function* if it is defined and has all of the following attributes:

1. **unambiguous**: so that the customer, software analysts and other design stakeholders would have the same interpretation.
2. **complete**: there is sufficient information for the design of the software. Also, input functions should define responses to valid and invalid input values.
3. **verifiable**: so that a test case can be written for it.
4. **unique:** it is not redundant.
5. **consistent:** it does not contradict other requirements.
6. **correct**: the function should be approved by the customer or in agreement with a higher-level document, such as a project charter or high-level requirements.

The counting rule for B1 (the number of functions not satisfactorily defined) is to count all of the non-satisfactorily defined functions identified by the above rules.

The following are samples of satisfactorily defined and non-satisfactorily defined functions:

"Upon the μp addressing a board, a decoding chip on the board shall send a code back via the data bus lines." ([APP, Y5], Page 42) is a satisfactorily defined function;

"If all diagnostic tests are passed, then this algorithm shall light the MAINT LED for approximately one second" ([APP, Y5], Page 45) is a satisfactorily defined function;

"There shall be a delay between updates to give the communication μp time to access the Dual Port Ram" ([APP, Y1], Page 39) is a non-satisfactorily defined function since the duration of the delay time is not specified.

9.2.2 B2: Number of Functions

Within the context of the COM measurement, a function is a defined objective or characteristic action in the software requirement specification (SRS) that is usually involved in processing input(s) and/or generating output(s).

The defined objective or characteristic action is identified by analyzing the functional specifications at the word phrase level.

The following rules apply when identifying individual functions:

1. The Functional Requirements Section of the SRS is used to identify functions for this measure.
2. If there is no separate Functional Requirements Section, then use the requirements in the SRS that describe the inputs, processing, and outputs of the software. These are usually grouped by major functional description, sub-functions, and sub-processes.

3. Functions may also be displayed in data-related or object-oriented diagrams. In flow diagrams, functions are usually shown as ovals with arrows showing data flow or function inputs and outputs.
4. Each functional requirement is counted as a function. A functional requirement has the following characteristics:
    1. A function is the lowest-level characteristic of the software that usually has an input, processing, and an output.
    2. It is the most fundamental and testable characteristic and action that takes place in processing the inputs and generating the outputs. The inputs or outputs may be other functions, or inputs or outputs to the software system
    3. A functional requirement generally takes the form of a "noun-modal verb-action verb-object" sentence segment. The modal verb is usually a "shall," "should," "may," or "will" statement.
    4. A descriptive statement whose prototypical verb is a descriptive word, such as "contain," "indicate," "consider," and "include," is NOT a function.
    5. Compound sentence segments (joined with *and*, *or*, etc.) may describe separate functions.
    6. A chart or graphic may define one or more functions.
    7. A function may be implied. Such a function would not meet the requirement for a satisfactorily defined function.

Each functional specification is expressed as a fundamental and uncomplicated statement. Each function must be uniquely identified (usually numbered). Uniqueness is facilitated by the use of a consistent and logical scheme for assigning identification to each functional specification statement within the requirements document.

Non-functional requirements, as described in [IEEE, 1988], do not describe *what* the software will do, but *how* the software will perform the functions. Most of the non-functional requirements are not as important as the functional requirements. Typical non-functional requirements include:

- Performance Requirements (throughput, response time, transit delay, latency, etc.)
- Design Constraints
- Availability Requirements
- Security Requirements
- Maintainability Requirements
- External Interface Requirements
- Usability requirements (ease-of-use, learnability, memorability, efficiency, etc.)
- Configurability requirements
- Supportability requirements
- Correctness requirements
- Reliability requirements
- Fault tolerance requirements
- Operational scalability requirements (including support for additional users or sites, or higher transaction volumes)
- Localizability requirements (to make adaptations due to regional differences)

- Extensibility requirements (to add unspecified future functionality)
- Evolvability requirements (to support for new capabilities or ability to exploit new technologies)
- Composability requirements (to compose systems from plug-and-play components)
- Reusability—requirements
- System Constraints. (e.g., hardware and OS platforms to install the software, or legacy applications, or in the form of organizational factors or the process that the system will support.)
- User Objectives, Values and Concerns.

Normally, non-functional requirements are not considered while counting functions. However, in certain cases, non-functional requirements hide what really are functional requirements and may describe characteristics that are critical to safety and reliability, such as response time. These special cases should be identified by the analyst and included in the function count**.** Following are rules for counting functions implied in the non-functional requirements:

- A function in the non-functional requirements generally takes the form of a "noun-modal verb-action verb-object" sentence segment. The modal verb is usually a "shall," "should," "may," or "will" statement.
- A descriptive statement whose prototypical verb is a descriptive word, such as "contain," "indicate," "consider," and "include," is NOT a function.
- Compound sentence segments (joined with *and*, *or*, etc.) may describe separate functions.
- A chart or graphic may define one or more functions.

The counting rule for B2 (the number of functions) is to count all of the individual functions identified by the above rules.

The following are samples of functional and non-functional requirements:

"After power-up or reset, the CPU begins code execution from location 0000H" ([APP, Y5], Page 22) is a functional requirement;

"Upon a module power-up all table contents shall be reset to zero and then copied to specified locations in external RAM" ([APP, Y5], Page 25) is a functional requirement which defines two functions;

"This algorithm shall send a refresh signal to the watchdog timer" ([APP, Y5], Page 52) is a functional requirement;

"Time update variable shall contain eight bytes of data that represent the current data and time" ([APP, Y5], Page 54) is NOT a functional requirement since it is a descriptive statement;

"Memory mapping of the Dual Port Rams memory locations shall be specified in both safety µp and the communication Software Design Documents" ([APP, Y5], page 52) is NOT a functional requirement since it is a design requirement.

9.2.3 B3: Number of Data References Not Having an Origin

Within the context of the COM measurement, a *data reference origin* is the source of the data that is manipulated by the data reference. The origin of a data is either a system input or an outcome of other functions.

A data reference has an origin if and only if all data manipulated by this data reference have an identified source(s).

The counting rule for B3 (the number of data references not having an origin) is to count all of the identified individual data references that do not have an origin.

The following are samples of data references with and without data origin:

"The algorithm shall restore the data back to the two tested memory locations" ([APP, Y5], Page 33) is a data reference with an origin since the data is provided by another function. "Contents of the two data memory locations shall be stored in two CPU registers" ([APP, Y5], Page 33);

The data reference "Contents of the two data memory locations shall be stored in two CPU registers" ([APP, Y5], Page 33) has no data origin since no source provides the data "contents of the two data memory locations" (there is no statement to specify how to determine the memory locations).

9.2.4 B4: Number of Data References

Within the context of the COM measurement, a *data reference* is a function that manipulates either internal or external data.

The counting rule for B4 (the number of data references) is to count all of the individual data references identified by the above rules.

The following are samples of data references:

"The next step is to write the complement of the first byte to the first memory location and the complement of the second byte to the second location" ([APP, Y5], Page 33) is a data reference since it manipulates four data items: "the complement of the first byte," "the complement of the second byte," "the first memory location," and "the second location;"

"The algorithm shall restore the data back to the two tested memory locations" ([APP, Y5], Page 33) is a data reference;

"After power-up or reset, the CPU begins code execution from location 0000H" ([APP, Y5], Page 22) is NOT a data reference since it does not manipulate any data.

9.2.5 B5: Number of Defined Functions Not Used

Within the context of the COM measurement, a *used function* is a function that is either employed in the control flow or referenced by other used functions. Contrast this with a non-used function that is defined but neither employed in the control flow nor referenced by any other used function.

Refer to Section 9.2.2 for the definition of a function.

The counting rule for B5 (the number of defined functions not used) is to count all of the identified individual non-used functions.

The following are samples of used and non-used functions:

"This algorithm shall enter a loop which attempts to access the rights to the Semaphores for both Dual Port RAMs" ([APP, Y5], Page 45) is a used function since it is employed in the control flow;

The implied function "Allocate two separate bytes in external RAM" is a used function since it is referred by the used function "This algorithm shall read the hardwired code (one byte) and store the value in two separate bytes in external RAM" ([APP, Y5], Page 29);

"Next, the algorithm shall compare the lower five bits of the two safety µp to the hardware code stored in RAM and the identification code obtained from the Identity Chip visible on the module font panel" ([APP, Y5], Page 45) is a used function since it is referred by the used function "if the codes corresponds, then this algorithm shall write 55H to the 1 Function ID Status and 2 Function ID Status in the APP status table" ([APP, Y5], Page 45–46).

"Steps have to be taken to ensure that the program keeps track of which bank is being used." ([APP, Y5], Page 24) is a non-used function since it is neither employed in the control flow nor referred by any used function.

9.2.6 B6: Number of Defined Functions

Within the context of the COM measurement, a *defined function* is a function that is explicitly stated in the SRS. Contrast this with an implied function that is referenced but not defined.

Refer to Section 9.2.2 for the definition of a function.

The counting rule for B6 (the number of defined functions) is to count all defined functions identified by the above rules.

The following are samples of defined and implied functions:

In statement “This algorithm shall read the hardwired code (one byte) and store the value in two separate bytes in external RAM” ([APP, Y5], Page 29) there are two defined functions: “read the hardwired code (one byte)” and “store the value in two separate bytes in external RAM;”

The statement “This algorithm shall read the hardwired code (one byte) and store the value in two separate bytes in external RAM” ([APP, Y5], Page 29) implies an undefined function “Allocate two separate bytes in external RAM” since this function is not stated, but is required.

9.2.7 B7: Number of Referenced Functions Not Defined

Refer to Section 9.2.6 for the definition of a *defined function* and Section 9.2.8 for the definition of a *referenced function*.

The counting rule for B7 (the number of referenced functions not defined) is to count all of the individual referenced and non-defined functions.

9.2.8 B8: Number of Referenced Functions

Within the context of the COM measurement, a *referenced function* is a function that is referenced by any other functions within the same SRS.

Refer to Section 9.2.2 for the definition of a function.

The counting rule for B8 (the number of referenced functions) is to count all of the individual referenced functions identified by the above rules.

The following are samples of referenced functions:

In statement “If all diagnostic tests are passed, then this algorithm shall light the MAINT LED for approximately one second” ([APP, Y5], Page 45), “diagnostic tests” are referred functions since they are referred by the function “this algorithm shall light the MAINT LED for approximately one second;”

In statement “Upon completing the Initialization procedures above, the code execution shall proceed to the Power-Up Self Tests.” ([APP, Y5], Page 30) There are two functions which are referred: the “Initialization” function and the “Power-Up Self Tests” function.

9.2.9 B9: Number of Decision Points Missing Any Conditions

Refer to Section 9.2.10 for the definition of a *decision point* and to Section 9.2.12 for the definition of a *decision point condition*.

The counting rule for B9 (the number of decision points missing any condition) is to count all of the identified individual decision points in which a condition is missing.

9.2.10 B10: Number of Decision Points

Within the context of the COM measurement, a *decision point* is a process element that routes the system to one of several alternative outgoing paths, depending on its condition(s). In the requirement statements, the keywords, such as " = ", " < ," " > ", "compare," "verify" and "check," usually imply the existence of a decision point.

The counting rule for B10 (the number of decision points) is to count all of the identified individual decision points.

The following are samples of decision points:

"This algorithm shall compare the 5-bit codes sent from the safety μp to the code stored in the Identity Chip and the code hardwired to the module backplane connector" ([APP, Y5], Page 31) is a decision point;

"This algorithm shall read back the data in the data in the failure address line and then the base address data and compare the two values to check if the data are complements of each other" ([APP, Y5], Page 36) is a decision point.

9.2.11 B11: Number of Condition Options Having No Processing

Within the context of the COM measurement, a condition option is *processed* if a function is defined to take over the control flow given that the condition option is taken. Contrast this with an *unprocessed condition option* that no function is not defined to be in charge of the control flow.

Refer to Section 9.2.12 for the definition of a *condition option*.

The counting rule for B11 (the number of condition options having no processing) is to count all of the unprocessed condition options.

The following is an example of processed and unprocessed condition options:

The statements "This algorithm shall read the status flags generated by the On-Line Diagnostics. If a test status flag contains the value 55H, this shall..." ([APP, Y5], Page 49) imply four decision points, corresponding to the values taken by each of four test status flags: RAM Diagnostic Test Status Flag, Data Bus Lines Diagnostic Test Status Flag, Address Bus Lines Diagnostic Test Status Flag, and PROM Checksum Diagnostic Test Status Flag. The condition related to each decision point is "if the value of the test status flag is 55H." The options within each condition are "55H" and "other values." Option "55H" is processed since descendant functions are defined to handle this option (e.g., "read trip outputs"). However, option "other values" is unprocessed since there is no function defined to handle this option.

9.2.12 B12: Number of Condition Options

For the COM measure, a *condition* in a decision point is a leaf-level expression that cannot be broken down into a simpler expression. A *condition option* is one of the possible results determined by the condition.

The counting rule for B12 (the number of condition options) is to count the condition options of all identified individual conditions.

The following are samples of conditions and their condition options:

In the decision point "This algorithm shall compare the 5-bit codes sent from the safety μp to the code stored in the Identity Chip and the code hardwired to the module backplane connector" ([APP, Y5], Page 31), the condition is "if the two codes match or not;" the condition options are "the two codes match" and "the two codes mismatch;"

The statements "This algorithm shall read the status flags generated by the On-Line Diagnostics. If a test status flag contains the value 55H, this shall..." ([APP, Y5], Page 49) imply four decision points, corresponding to the values taken by each of four test status flags: RAM Diagnostic Test Status Flag, Data Bus Lines Diagnostic Test Status Flag, Address Bus Lines Diagnostic Test Status Flag, PROM Checksum Diagnostic Test Status Flag. The condition related to each decision point is "if the value of the test status flag is 55H." The options within each condition are "55H" and "other values."

9.2.13 B13: Number of Calling Routines Whose Parameters Do Not Agree with the Called Routines Defined Parameters

Refer to Section 9.2.14 for the definitions of a *calling routine*, a *called routine*.

The counting rule for B13 is to count the number of calling routines which can be separately identified and whose parameters do not agree with the parameters defined in the routines being called.

9.2.14 B14: Number of Calling Routines

Within the context of the COM measurement, a *routine* is a set of sequential functions. A routine is usually bulleted as a functional section in the SRS. A *calling routine* is a routine referring to other routine(s). A *called routine* is a routine referred by other routine(s).

The counting rule for B14 (the number of calling routines) is to count the calling routines which can be separately identified.

The following are samples of routines, calling routines and called routines:

"Check Diagnostic Test Status" ([APP, Y5], Page 49) is a routine since it consists of quite a few defined functions, such as "read the status flags generated by the On-Line Diagnostics," and

"stay in a loop which refreshes the watchdog timer and responds to the master station when polled;" However, it is neither a calling routine nor a called routine.

In routine "On-Line Diagnostics" ([APP, Y5], Page 53), a function is defined as "bring the system program CPU operation back to the Main Program." Obviously, the routine "Main Program" ([APP, Y5], Page 47) is called. Therefore, "On-Line Diagnostics" is a calling routine, and "Main Program" is the called routine.

9.2.15 B15: Number of Condition Options Not Set

Within the context of the COM measurement, a condition option is *set* if it is defined (explicitly stated) in the SRS. Contrast this with an *unset condition option* that is not defined.

Refer to Section 9.2.12 for the definition of a *condition option*.

The counting rule for B15 (the number of condition options not set) is to count the number of unset condition options of all identified conditions.

The following is an example of set and unset condition options:

The statements "This algorithm shall read the status flags generated by the On-Line Diagnostics. If a test status flag contains the value 55H, this shall..." ([APP, Y5], Page 49) imply four decision points, corresponding to the values taken by each of four test status flags: RAM Diagnostic Test Status Flag, Data Bus Lines Diagnostic Test Status Flag, Address Bus Lines Diagnostic Test Status Flag, and PROM Checksum Diagnostic Test Status Flag. The condition related to each decision point is "if the value of the test status flag is 55H." The options within each condition are "55H" and "other values." Option "55H" is set since it is explicitly stated, and option "other values" is unset since it is implied by using common sense.

9.2.16 B16: Number of Set Condition Options Having No Processing

Refer to Section 9.2.17 for the definition of a *set condition option* and Section 9.2.11 for .the definition of a *processed condition option*.

The counting rule for B16 (the number of set condition options having no processing) is to count the number of unprocessed condition options which are set.

9.2.17 B17: Number of Set Condition Options

Refer to Section 9.2.15 for the definition of a *set condition option*.

The counting rule for B17 (the number of set condition options) is to count the number of condition options related to all the conditions identified. B17 = B12 - B15.

9.2.18 B18: Number of Data References Having No Destination

Within the context of the COM measurement, a *data reference destination* is a place to which the outcome of the data reference is sent. The destination of a data is either a system output or an input of other functions.

A data reference has a destination if and only if all output data generated by this data reference have destination(s).

The counting rule for B18 (the number of data references having no destination) is to count the number of data references having no destination.

The following are samples of data references with and without destination:

The data reference "Contents of the two data memory locations shall be stored in two CPU registers" (CP System SRS document, Page 33) has a destination since its outcome is used by another function "The algorithm shall restore the data back to the two tested memory locations" (CP System SRS document, Page 33);

The data reference "Read data block size" ([APP, Y5], Page 57) has no destination since the "data block size" is not used by any other function.

9.2.19 Measurement Procedure

The purpose of the COM measurement is to identify defects (incomplete functional requirements) in the SRS and thereby estimate the software reliability.

An incompleteness defect in a software requirement specification (SRS) is one of the following:

1. An incomplete function:
   - An unsatisfactorily defined function; or
   - A defined function which is not used; or
   - A referenced function which is not defined; or
   - A data reference not having an origin; or
   - A data reference not having a destination.

2. An incomplete decision point:
   - A decision point missing a condition(s); or
   - A condition option not set; or
   - A condition option not processed.

3.
3. An incomplete calling routine:
   - A calling routine whose parameters disagree with the called routine's defined parameters.

Incompleteness defects in an SRS can be identified using the procedure shown in Figure 9.1.

Figure 9.2 presents the procedure to be followed to identify incomplete functions.

Figure 9.3 presents the procedure to be followed to identify incomplete decision points. The procedure to be followed to identify incomplete calling routines is shown in Figure 9.4

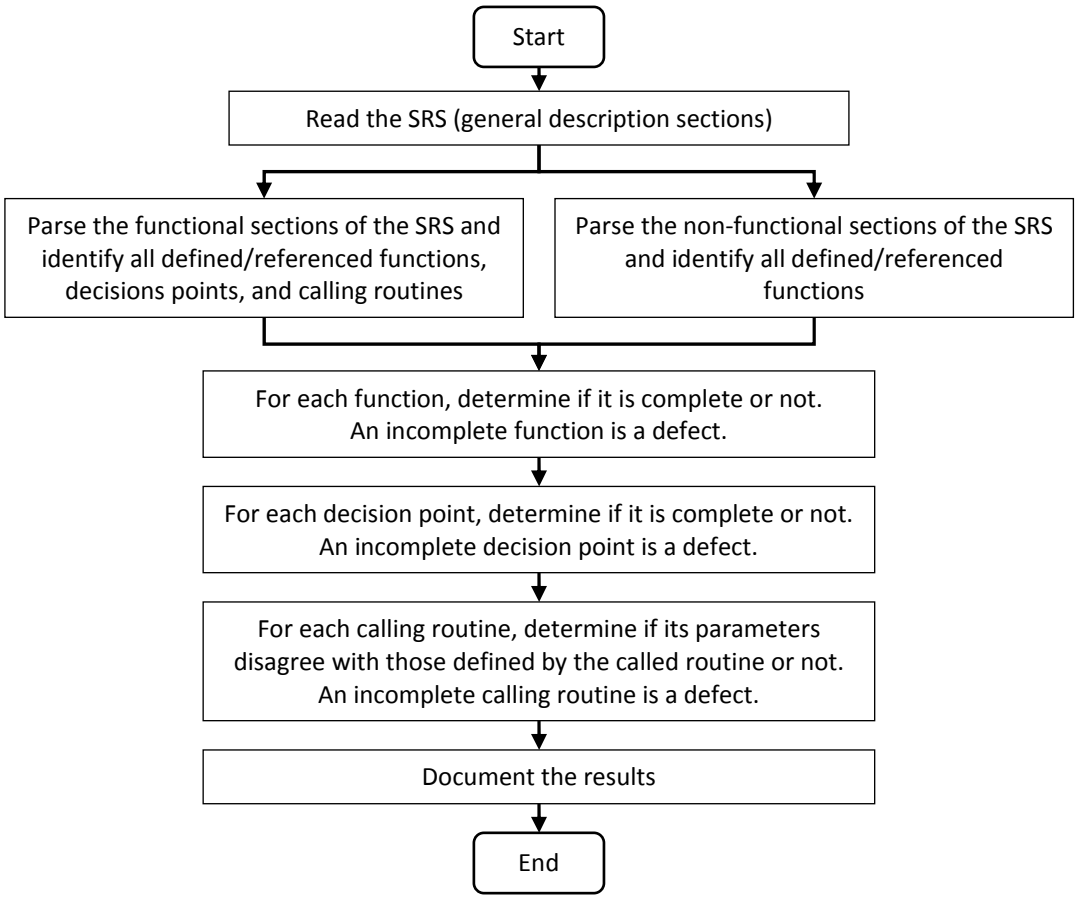


Figure 9.1 Procedure for Identifying Incompleteness Defects in the SRS

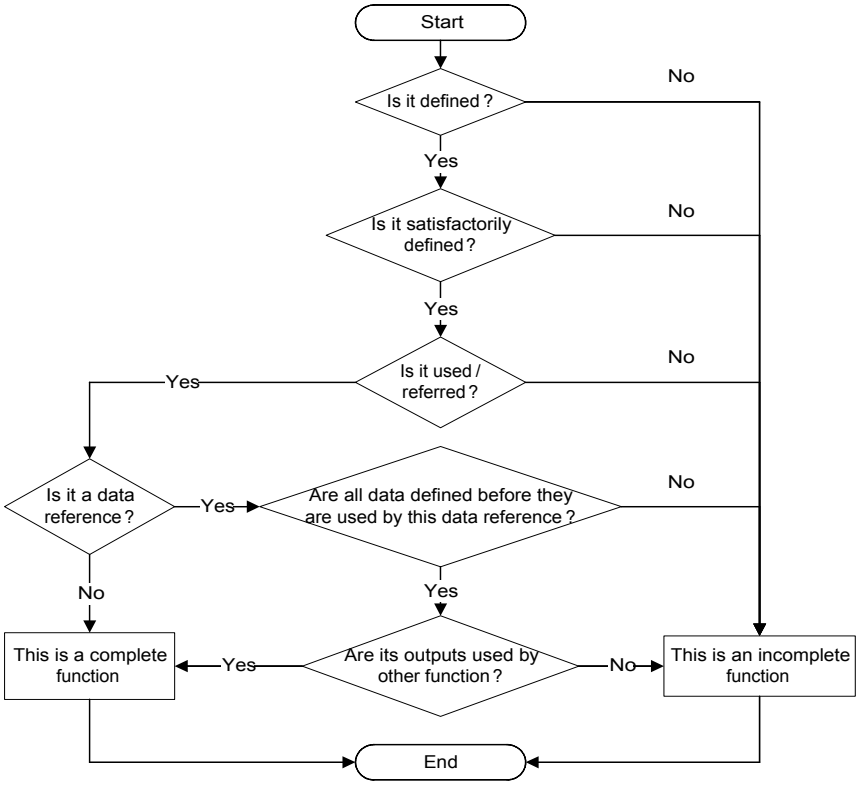


Figure 9.2 Procedure for Identifying Incomplete Functions in the SRS

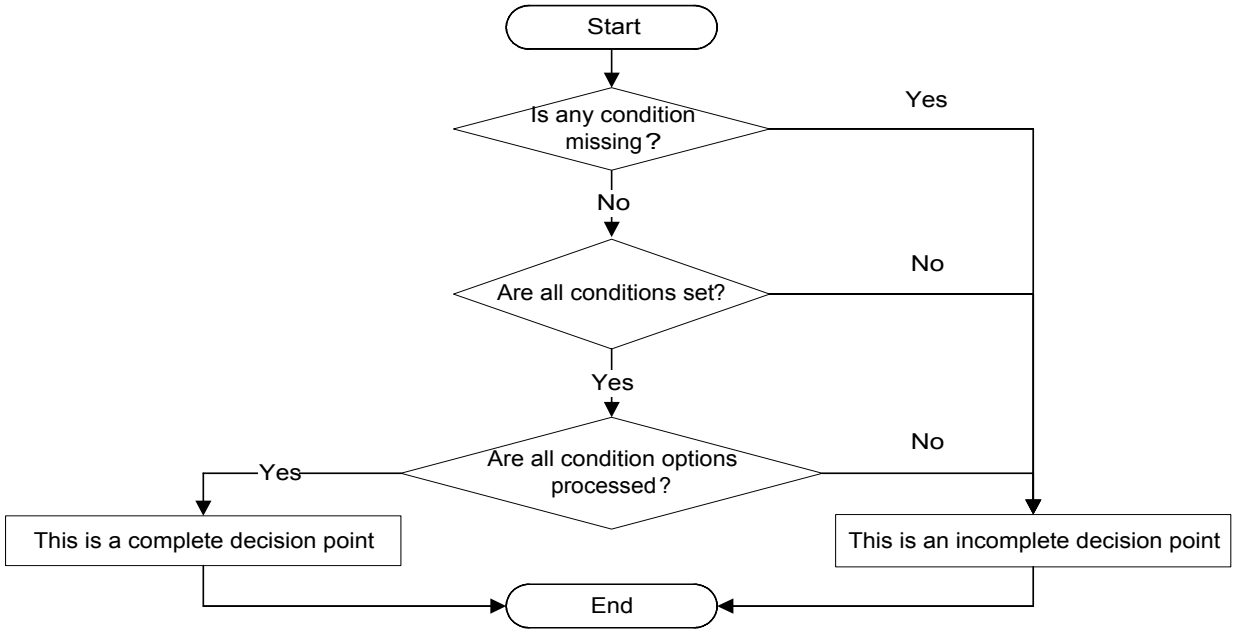


Figure 9.3 Procedure for Identifying Incomplete Decision Points in the SRS

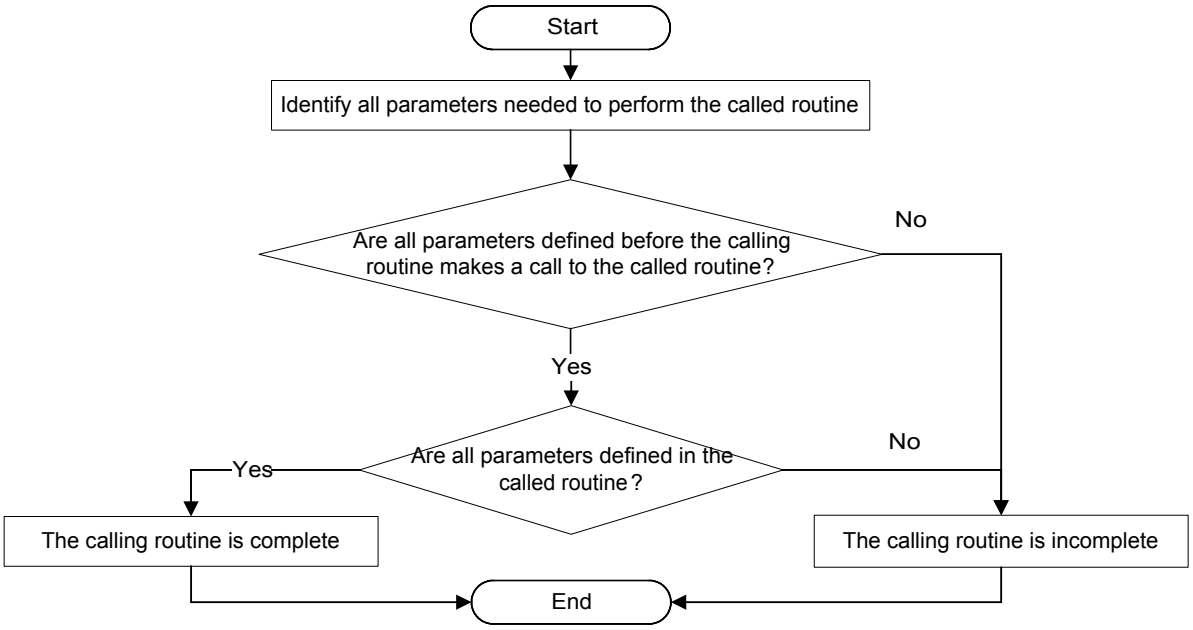


Figure 9.4 Procedure for Identifying Incomplete Calling Routines in the SRS

9.3 Measurement Results

The following documents were used to measure requirement completeness:

- APP Module μp1 System SRS [APP, Y1]
- APP Module μp1 Flux/Delta Flux/Flow Application SRS [APP, Y2]
- APP Module μp2 System SRS [APP, Y3]
- APP Module μp2 Flux/Delta Flux/Flow Application SRS APP Y4]
- APP Module Communication Processor SRS [APP, Y5]

The primitives are presented in Table 9.1.

Table 9.1 Primitives for APP Modules

Primitive	APP Module				
	CP System	**µp1 System**	**µp1 Application**	**µp2 System**	**µp2 Application**
B_1	14	19	3	5	4
B_2	190	301	61	218	29
B_3	2	8	0	4	0
B_4	138	225	60	184	25
B_5	9	8	0	0	0
B_6	182	292	60	218	25
B_7	7	4	1	0	4
B_8	125	93	40	74	20
B_9	2	1	0	0	0
B_{10}	28	28	11	52	6
B_{11}	2	1	1	0	0
B_{12}	63	57	34	110	20
B_{13}	0	1	0	0	0
B_{14}	18	26	1	7	0
B_{15}	2	1	2	0	0
B_{16}	0	1	0	0	0
B_{17}	63	56	32	110	20
B_{18}	3	5	0	0	0

Table 9.2 lists the weights, derived measures, and COM measures for the APP modules.

Table 9.2 Weights, Derived Measures, and COM Measures for the APP Modules

Weight, w_i[22]	Derived Measure	APP Module				
		CP System	**µp1 System**	**µp1 Application**	**µp2 System**	**µp2 Application**
$w_1 = 0.2$	D_1	0.926316	0.93688	0.95082	0.97706	0.862069
$w_2 = 0.1$	D_2	0.985507	0.96444	1	0.97826	1
$w_3 = 0.05$	D_3	0.950549	0.9726	1	1	1
$w_4 = 0.1$	D_4	0.944	0.95699	0.975	1	0.8
$w_5 = 0.1$	D_5	0.928571	0.96875	1	1	1
$w_6 = 0.05$	D_6	0.968254	0.98508	0.970588	1	1
$w_7 = 0.2$	D_7	1	0.96154	1	1	1
$w_8 = 0.05$	D_8	0.968254	0.95522	0.941176	1	1
$w_9 = 0.05$	D_9	1	0.98438	1	1	1
$w_{10} = 0.1$	D_{10}	0.978261	0.97778	1	1	1
COM		0.96325	0.9613	0.98325	0.99315	0.95241

The value of COM can be used as an indicator of the quality of an SRS. However, it should be made clear that the value of COM is partly subjective since the weights and the primitives are determined subjectively.

The identified incompleteness defects with severity level 1 and level 2 are summarized in Table 9.3. These defects are also categorized according to the operational modes to which they belong.

[22] These weights were obtained through expert opinion elicitation. The experts consisted of two software developers and a software reliability expert.

Table 9.3 Summary of Defects with Severity Level 1 and 2 Found in the SRSs of the APP System

Defect No.	SRS	Section Index in SRS	Operating Mode	Function Description	Incompleteness Type	Severity level	Defect Fixed in the code?	Where is the defect fixed in the source code?
1	CP System SRS	3.1.6	Power-on	Allocate two separate bytes in external RAM	Referenced function is not defined	2	Yes	[APP, Y10], Page 13, line 7–8
2		3.2.1	Power-on	Start from the top memory address to perform RAM test	Data reference does not have origin	2	No	(In the source code, RAM testing starts from the lowest address.) [APP, Y10], Page 18, line 13
3		3.2.3	Power-on	Start at the lowest address lines to perform Address Line test	Data reference does not have origin	2	Yes	[APP, Y10], Page 18, line 14
4		3.2.8	Power-on	Judge if both safety functions are checked	Referenced function is not defined	1	Yes	[APP, Y10], Page 20, line 30–54
5		3.3.1	Normal	Judge if all flags are read or not	Referenced function is not defined	1	Yes	[APP, Y10], Page 26, line 11–16

Table 9.3 Summary of Defects with Severity Level 1 and Level 2 Found in the SRSs of the APP System (continued)

Defect No.	SRS	Section Index in SRS	Operating Mode	Function Description	Incompleteness Type	Severity level	Defect Fixed in the code?	Where is the defect fixed in the source code?
6	CP System SRS	3.3.1	Normal	If not all flags are read, read next flag	Referenced function is not defined	1	No	[APP, Y10], Page 26, line 11–16
7		3.3.1	Calibration/ Tuning	If a Test status flag contains the value 55H, this shall indicate that the test has passed.	Condition option has no processing	1	No	In Binder #4, CP source code document, page 26, line 11–16, only Test status flag == BBH is checked; namely, any value 128 defined) is to count all of the individual referenced and no
8		3.3.3	Calibration/ Tuning	If switch line reset, judge which mode selected	Referenced function is not defined	1	Yes	[APP, Y10], Page 26, line 20–21
9		3.3.3	Calibration/ Tuning	Judge if the Cycle Monitor flag = 55H	Condition option has no processing	1	Yes	[APP, Y10], page 26, line 47

Table 9.3 Summary of Defects with Severity Level 1 and Level 2 Found in the SRSs of the APP System (continued)

Defect No.	SRS	Section Index in SRS	Operating Mode	Function Description	Incompleteness Type	Severity level	Defect Fixed in the code?	Where is the defect fixed in the source code?
10	CP System SRS	3.3.3	Calibration/ Tuning	Judge if key lock switch is moved from the TEST position	Referenced function is not defined	1	Yes	[APP, Y10], page 26, line 100–101
11	µp1 System SRS	3.2	Power-on	Call a diagnostic test	No parameter is defined for Calling routine	1	Yes	[APP, Y6], µp1 system source code document, page 23, line 13
12		3.2.1	Power-on	Start from the top memory address to perform RAM test	Data reference does not have an origin	2	No	(In the source code, RAM testing starts from the lowest address.) [APP, Y6], page 25, line 36 and page 92, line 10
13		3.2.4	Power-on	Start from the lowest memory address to perform Address Line test	Data reference does not have an origin	2	Yes	[APP, Y6], page 29, line 18–24

Table 9.3 Summary of Defects with Severity Level 1 and Level 2 Found in the SRSs of the APP System (continued)

Defect No.	SRS	Section Index in SRS	Operational Mode	Function Description	Incompleteness Type	Severity level	Defect Fixed in the code?	Where is the defect fixed in the source code?
14	µp1 System SRS	3.2.4	Power-on	Increase status counter	Data reference does not have an origin	1	Yes	[APP, Y6], page 17, line 31
15		3.2.7	Power-on	If not all boards tested, identify next board	Data reference does not have an origin	1	Yes	[APP, Y6], page 37, line 6–49
16		3.2.9	Power-on	Compare median with pre-stored value	Data reference does not have an origin	1	Yes	[APP, Y6], page 38, the last two lines
17		3.3.1	Normal	"A BBH test result should indicate a test failure which is considered to be fatal failure."	Condition option has no processing	1	No	[APP, Y6], page 22, line 20–28, only Test status flag != 55H is checked; namely, any value other than 55H is treated as BBH by default. However, this treatment is regarded as a fail-safe design.

Table 9.3 Summary of Defects with Severity Level 1 and Level 2 Found in the SRSs of the APP System (continued)

Defect No.	SRS	Section Index in SRS	Operating Mode	Function Description	Incompleteness Type	Severity level	Defect Fixed in the code?	Where is the defect fixed in the source code?
18	µp1 System SRS	3.3.2	Normal	Check if any Semaphore available	Referenced function is not defined	1	Yes	[APP, Y6], page 67, line 9
19		3.3.3	Calibration/ Tuning	Judge if port line is reset	Referenced function is not defined	1	Yes	[APP, Y6], page 10, line 44
20		3.3.3 A	Calibration/ Tuning	Obtain access rights to the DPR	Referenced function is not defined	1	Yes	[APP, Y6], page 10, line 42 and 67, line 13
21	µp1 Applica-tion SRS	3.2	Normal	Check if a Trip has occurred and removed	Referenced function is not defined	1	Yes	[APP, Y7], page 10, line 13–20
22	µp2 System SRS	3.1.2	Power-on	Judge if total sum = predetermined sum (PROM)	Data reference does not have an origin	1	Yes	[APP, Y8], page 31, line 9
23		3.1.2	Power-on	Judge if total sum = pre-determined sum (EEPROM)	Data reference does not have an origin	1	Yes	[APP, Y8], page 32, line 13

Table 9.3 Summary of Defects with Severity Level 1 and Level 2 Found in the SRSs of the APP System (continued)

Defect No.	SRS	Section Index in SRS	Operating Mode	Function Description	Incompleteness Type	Severity level	Defect Fixed in the code?	Where is the defect fixed in the source code?
24	µp2 System SRS	3.1.2	Power-on	Compare the generated values to known stored values (Algorithm)	Data reference does not have an origin	1	Yes	[APP, Y8], page 33, line 7 and Binder #3, µp2 application source code document, page 11, line 2–24
25		3.1.2	Power-on	Compare values to stored values (Analog input)	Data reference does not have an origin	1	Yes	[APP, Y8], page 34, line 13
26	µp2 Applica-tion SRS	3.1.3	Normal	Calculate Φ_U	Referenced function is not defined	1	Yes	[APP, Y9], page 8, line 5
27		3.1.3	Normal	Calculate Φ_L	Referenced function is not defined	1	Yes	[APP, Y9], page 8, line 6
28		3.1.3	Normal	Calculate ΔP_A	Referenced function is not defined	1	Yes	[APP, Y9], page 8, line 7
29		3.1.3	Normal	Calculate ΔP_B	Referenced function is not defined	1	Yes	[APP, Y9], page 8, line 8

9.4 RePS Construction Using Completeness Measurement

The APP system has four distinct operational modes: Power-on, Normal, Calibration, and Tuning [APP, 01]. The reliability of the APP system was estimated for each operational mode using a different Extended Finite State Machine (EFSM) model for each operational mode as defined in [Smidts, 2004].

The EFSM approach proceeds in three steps:

1. Construction of an EFSM model representing the user's requirements and embedding of the user's operational profile information.
2. Mapping of the identified defects into the EFSM model.
3. Execution of the EFSM model to evaluate the impact of the defects in terms of failure probability.

Figure 9.5 describes the approach used to estimate reliability. It should be noted that a defect belongs to only one operational mode.

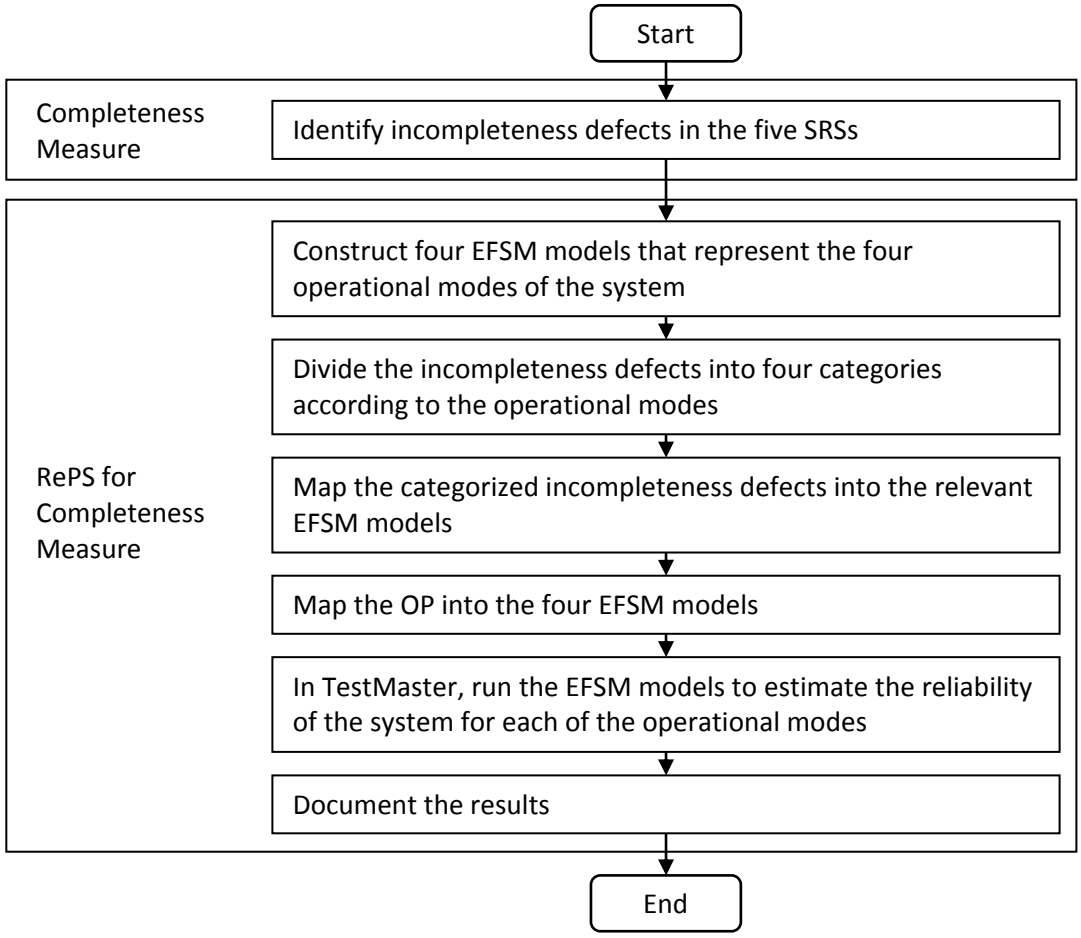


Figure 9.5 Approach used to estimate Reliability

Reliability estimation per operational mode is shown in Table 9.4 (Column 2 and Column 3).

Moreover, since some of the defects identified in the SRSs during the COM measurement might be fixed in later development phases, i.e., design phase and coding phase, one can use the approach described in Figure 9.5 to estimate software reliability based on the defects remaining in the source code, as shown in Table 9.4 (Column 4 and Column 5). All the values listed in the table were based on the EFSM analyses.

Table 9.4 Reliability Estimation for the Four Distinct Operational Modes

Mode	Based on all Severity Level 1 and Level 2 defects found in SRSs		Based on Severity Level 1 and Level 2 defects found in SRSs and remaining in the source codes	
	P_f	R	P_f	R
Power-on	1.000	0.000	0.000	1.000
Normal	2.582e-2	9.742e-1	0.000	1.000
Calibration	1.370e-2	9.863e-1	3.340e-11	1.000[23]
Tuning	1.370e-2	9.863e-1	3.340e-11	1.000[24]

Metrics used in the early phases of the development life cycle such as the COM measure and its derived measures can aid in detecting and correcting requirement defects.

The value of the COM measure is scaled between 0 and 1 by the appropriate weights. A score near 1 is considered to be better than a score near 0. Those values near zero should be highlighted and corresponding areas should be modified accordingly.

Also, the reliability based on Severity Level 1 and Level 2 defects found in SRSs and remaining in the source code is stated as 1 for the Power-on and Normal modes. This is because defects will not be triggered in Power-on and Normal mode and will only be triggered in Calibration and Tuning mode. The reliability based on Severity Level 1 and Level 2 defects found in SRSs and remaining in the source code is also stated as being 1 for the Calibration and Tuning. This is due to the need for a uniform number of significant figures in the measurements. The actual value is 0.9999999999666.

[23] This is the rounded up number. The actual number is 0.9999999999666.
[24] This is the rounded up number. The actual number is 0.9999999999666.

9.5 Lessons Learned

As a SRS-based measurement, the measurement process for COM is time-consuming. A considerable amount of time was spent in manually "parsing" the natural language of the SRS documents. Table 9.5 summarizes the effort expended to perform this measurement. The process of manually parsing the SRS is error-prone. The accuracy of the COM measure is highly dependent on the inspectors. A two-week period of training on the measurement and significant domain knowledge are required.

Some primitives are subjective, e.g., the number of satisfactorily defined functions. Repeatability of measurements is not guaranteed. The domain knowledge, physical status, and other subjective factors, to some extent, highly affect the inspector's judgment. Therefore, it is more appropriate to apply this measurement for identifying defects remaining in the SRS than for quantitatively assessing the quality of the SRS. Revisiting the defects found through the COM measurement and mapping them to the source code may significantly increase the chance of finding defects remaining in the source code.

Table 9.5 Effort Expended to Perform the Measurement of COM and Derived Measures

SRS	Pages of SRS	Pages of functional SRS	Time for			Total time	Total number of incompleteness defects identified in the functional SRS	Total number of identified incompleteness defects remaining in the source code
			Reading SRS' general description sections	Identifying and documenting functions, decision points, and calling routines	Identifying and documenting incomplete-ness defects			
CP System SRS	72	53	2.5 hrs	37.5 hrs	19.0 hrs	59.0 hrs	41 (10)	1 (1)
µp1 System SRS	106	70	3.5 hrs	56.0 hrs	21.5 hrs	81.0 hrs	48 (10)	0 (0)
µp1 Application SRS	23	12	1.0 hrs	12.5 hrs	7.5 hrs	21.0 hrs	7 (1)	0 (0)
µp2 System SRS	65	53	1.5 hrs	39.0 hrs	18.0 hrs	58.5 hrs	9 (4)	0 (0)
µp2 Application SRS	23	13	1.0 hrs	9.5 hrs	5.5 hrs	16.0 hrs	8 (4)	0 (0)
Sum	**289**	**201**	**9.5 hrs**	**154.5 hrs**	**71.5 hrs**	**235.5 hrs**	**113 (29)**	**1 (1)**

Note:
1. In Column 8 and Column 9, the numbers within the parentheses represent the number of defects of Severity Level 1 and Level 2, while the numbers outside the parentheses represent the number of defects of all severity levels.

9.6 References

[APP, 01] *APP Instruction Manual*.

[APP, Y1] "APP Module First µp SRS," Year Y1.

[APP, Y2] "APP Flux/Delta Flux/Flow Application SRS for SF1," Year Y2.

[APP, Y3] "APP Module µp2 System Software SRS," Year Y3.

[APP, Y4] "APP µp2 Flux/Delta Flux/Flow Application Software SRS," Year Y4.

[APP, Y5] "APP Module Communication Processor SRS," Year Y5.

[APP, Y6] "APP Module SF1 System Software code," Year Y6.

[APP, Y7] "APP SF1 Flux/Delta Flux/Flow Application code," Year Y7.

[APP, Y8] "APP Module µp2 System Software Source Code Listing," Year Y8.

[APP, Y9] "APP µp2 Flux/Delta Flux/Flow Application Software Source Code Listing," Year Y9.

[APP, Y10] "APP Communication Processor Source Code," Year Y10.

[IEEE, 1988] "IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software," IEEE Std. 982.2-1988, 1988.

[IEEE, 1998] "IEEE Recommended Practice for Software Requirements Specifications," IEEE Std. 830-1998, 1998.

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

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

10. COVERAGE FACTOR

A central problem in the validation of fault-tolerant systems such as those found in nuclear power plant safety systems is the evaluation of the efficiency of fault-tolerant mechanisms.

One parameter used to quantify this efficiency is the coverage factor (CF), which is defined as the probability of system recovery given that a fault exists. The sensitivity of dependability measures (such as reliability and availability) to small variations in the coverage factor is well known [Bouricius, 1969] [Arnold, 1973]. Consequently, it is important to determine coverage as accurately as possible [Powell 1993].

The CF reflects the ability of a system to automatically recover from the occurrence of a fault during normal operation. Fault-injection techniques can be used to determine the CF. Based on the fault-injection experiment results, the reliability of a fault-tolerant system can be estimated using the Markov chain modeling technique.

This chapter includes a definition of the CF, the introduction of Markov chain and fault-injection techniques, the application of Markov chain modeling and fault-injection techniques to the APP, and the process of calculating the reliability of the APP system.

This measure can only be applied when the source code is available. As listed in Table 3.3, the applicable life cycle phases for CF are Coding, Testing, and Operation.

10.1 Definition

CF is the probability that a system can recover from a fault given that a fault occurs [NUREG/GR - 0019]. A formal definition of the CF of a fault-tolerance mechanism is given as follows [Cukier, 1999]:

$$c = \Pr\left(H(g) = 1 | g \in G\right) \tag{10.1}$$

where

Pr the probability of $H(g) = 1$ when $g \in G$

H a variable characterizing the handling of a particular fault/activity pair,
$H(g) = \begin{cases} 1, if\ the\ mechanism\ correctly\ handles\ the\ fault/activity\ pairs\ g; \\ 0, otherwise \end{cases}$

G the global (i.e., complete) input space of a fault-tolerance mechanism, $G = F \times A$;

F fault space;

A activity space, or activation space, in which a single "activity" is a trajectory in the system's state space;

g a fault/activity pair, or a point in space *G*.

The CF is a function of the complete input space and is equal to the probability that a particular fault/activity pair is correctly handled given that a fault/activity pair is in the complete input space of a fault-tolerance mechanism. Actually, "$H = 1$" means that the system responds to the fault and recovers from the fault, "$g \in G$" indicates that a fault has happened, so the definition is the same as that in [NUREG/GR - 0019].

Mathematically, because H is a random variable that can take the values 1 or 0 for each element of the fault/activity space *G*, the CF can be the product of the probability of occurrence of *g* and of the value of *H* (0 or 1). Equation 10.1 can be expressed as [Cukier, 1999]:

$$c = \sum_{g \in G} H(g)p(g) \tag{10.2}$$

where

$p(g)$ The probability of occurrence of g*;*

$H(g)$ The value of H for a given point *g* ($g \in G$),
H(g) = 1 (if the system recovers) or 0 (if the system fails to recover).

Furthermore, the coverage can be viewed as the expected value of *H* from Equation 10.2 [Cukier, 1999], which means that Equation 10.2 can be transformed to:

$$c = E(H) \tag{10.3}$$

where

$E(H)$ expected value of *H*.

Without knowing the distribution *p(g)*, the best that can be done is to assume all fault/activity pairs in *G* are equally probable, i.e.:

$$p(g) = \frac{1}{(G)}$$

and to use the Coverage Proportion,

$$c = \frac{1}{|G|} \sum_{g \in G} H(g)$$

to describe the effectiveness of a given fault-tolerant mechanism**.**

10.2 Measurement Rules

Several techniques, such as testing and field data-collection, have been adopted to evaluate the dependability of a system. Fault/error injection has been recognized as the best approach to evaluate the behavior and performance of complex systems under faults and to obtain statistics on parameters such as coverage and latencies [Benso, 2003].

Especially for a highly dependable system, fault injection is a preferred method to accelerate the process of the quantitative evaluation of dependability since an unreasonable amount of time could be required to collect operating history results of statistical relevance. So the value of c is usually obtained by fault-injection experiments [Arlat, 1990] [Brombacher, 1999].

For the fault-injection approach, the most accurate way to determine c is to submit the system to all $g \in G$, and to observe all values of $H(g)$.

However, such exhaustive testing is rarely possible. In practice, the CF evaluation is carried out by submitting the system to a subset G^*, obtained by random sampling in the space G and then using statistics to estimate c.

The random sampling in space G is decomposed into two concurrent sampling processes: sampling a fault in space F and an activity in space A. Whereas the fault-space sampling process is explicit, the activity-sampling process is often achieved implicitly: the target system executes its operational workload and selected faults are injected asynchronously at random points in the workload execution. The activity-sampling process is distinct in this chapter.

An approximation of the CF is given by [Choi, 2000]:

$$c = \frac{the\ number\ of\ faults\ recovered\ by\ fault\ tolerant\ mechanisms}{the\ number\ of\ faults\ injected\ in\ the\ system} \tag{10.4}$$

Generally, four basic steps are required for CF Measurement:

1) Select a fault-injection technique;
2) Determine the sample input space;
3) Execute the fault-injection experiments;
4) Determine the CF applying Equation 10.4.

10.2.1 Selection of Fault-Injection Techniques

There are three kinds of fault-injection techniques:

1. Hardware-based (physical fault injection) which themselves can be classified into:

    a. Hardware fault injection with contact: the injector has direct physical contact with target system.
    b. Hardware fault injection without contact: the injector has no direct physical contact with the target system (radiation, air pressure, temperature, magnetism, humidity).

Hardware-based fault injection involves exercising a system under analysis with specially designed test hardware to allow the injection of faults into the target system and to examine the effects. Traditionally, these faults are injected at the integrated circuit (IC) pin level [Benso, 2003].

2. Software-based

Software-implemented fault injection (SWIFI): data is altered and/or timing of an application is influenced by software while running on real hardware.

Traditionally, software-based fault injection involves the modification of software executing on the system under analysis in order to provide the capability to modify the system state according to the programmer's view of the system. This is generally used on code that has communicative or cooperative functions so that there is enough interaction to make the fault injection useful [Benso, 2003].

3. Simulation-based

Simulation-based fault injection (SBFI): the whole system behavior is modeled and imitated using simulation.

Compared with the other two methods, simulation-based fault injection has the following advantages [Benso, 2003]:

- Simulation-based fault injection can support all system abstraction levels: axiomatic, empirical, and physical.
- There is no intrusion into the real system.
- Full control of both fault models and injection mechanisms is secured.
- Maximum observability and controllability are achieved.

For these reasons, simulation-based fault injection (SBFI) is selected to estimate the coverage factor of the APP system.

10.2.2 Determination of Sample Input Space

According to the definition in Equation 10.1, an input space is characterized by a fault space and an activity space. Therefore, the sampling of the input space for the fault-injection experiments consists of determining fault space and activity space, respectively.

10.2.2.1 Fault Space

One of the difficulties in fault injection is determining the fault-injection space (the set of faults that should be injected), since exhaustive testing of all possible faults that a system may encounter during its lifetime is impractical.

Generally, the fault space for a microprocessor-based embedded system has four dimensions:

Type: which kind of faults are injected
- a bit, bits, byte, word, or words
- permanent or transient

Location: where a fault is injected
- IU (Integer Unit)
- FPU (Float Point Unit)
- Data Unit (Data/Data Address)
- Register Array
- Instruction Unit (Code/Code Address)

Time: when a fault is injected.
- Pre-runtime
- Runtime (the number of executed instructions before the fault injection)

Duration: how long an injected fault lasts. (The duration is usually expressed in terms of the number of instructions executed after the fault was injected.)

Because the variables in the source code are stored in the RAM, fault injection was performed in the APP RAM. When hardware faults occur in the RAM, the values of variables will be changed, which injects faults into the system and may lead to system failure. Therefore, one can change the values of the variables to simulate faults in the RAM.

Many researchers have found that transient faults can be up to 100 times more frequent than permanent faults, and they are much more significant in terms of dependability simulation [Benso, 2003].

According to [Gil, 2002], the most used fault model is bit-flip for transient fault, which is produced in the memory circuit, so bit flip was selected as the fault type. In addition, pre-runtime fault injection was only suitable for a limited number of fault classes such as permanent faults [Hexel, 2003]. Therefore, runtime was taken as fault-injection time.

The fault space of the APP system is listed as follows:

Location: RAM
Type: Bit flip, Transient
Time: Runtime
Duration: Within a single execution cycle

10.2.2.2 Activity Space

The effect of an injected fault is dependent on system activity at the moment of its occurrence. So a sample space consists of the combination of the set of faults and system "activity."

The activity space for the APP system is divided into two categories: outside the "Barn shape" and inside the "Barn shape," as described in Chapter 4.

10.2.3 Applying the Simulation-Based Fault Injection Technique to the APP

There are two safety function processors in the APP System: one is an Intel 80c32 (μp1), and the other is a z80180 (μp2). Two simulated environments were set up to execute the fault injection experiments using KEIL μversion 2 and IAR, respectively.

1. KEIL μversion 2 (for μp1)

The processor of μp1 is the Intel 80c32, which belongs to the Intel 8051 family. KEIL develops C compilers, macro assemblers, real-time kernels, debuggers, simulators, integrated environments, and evaluation boards for the 8051, 251, ARM, and XC16x/C16x/ST10 microcontroller families. The KEIL μVision2 IDE provides control for the Compiler, Assembler, Real-Time OS, Project Manager, and Debugger in a single, intelligent environment.

The fault injection experiments for μp1 were carried out following the steps described below:

a. Install the KEIL μversion 2 software into the computer. The software was installed on the computer before the experiment began. KEIL μversion 2 was installed according to [KEIL, 2001] step-by-step instructions.
b. Create a project of KEIL μversion 2 for μp1. KEIL μversion 2 is designed for the 8051 family instead of only for Intel 80c32. Therefore an appropriate project had to be created for μp1 by setting up the appropriate configurations. This included selecting the type of

processor (Intel 80c32), the Memory model, and other configurations per the [KEIL, 2001] instructions.

c. Added μp1 source code to KEIL μversion 2 environment per [KEIL, 2001] instructions.
d. Executed fault-injection experiments for μp1. Injected the faults one after another by modifying the value of the variables in the watch window. Then, after running the system for at least one cycle, observed the system outputs. The outputs were the values of the indicator variables in the source code, which indicated whether the system sent a trip signal or intentionally halted. By comparing these results with the outputs obtained without the fault injected, the researchers determined in which state the system remained.
e. Collected the experimental results.

2. IAR Simulated Environment

IAR Systems provide a range of development tools for embedded systems: integrated development environments (IDE) with C/C++ compilers and debuggers, starter kits, hardware debug probes, and state machine design. The IAR C compiler for the Z80 offers the standard features of the C language, plus many extensions designed to take advantage of specific features of the Z80.

The fault-injection experiments for μp2 can be performed following these steps:

a. Install IAR on the computer following [IAR, 1997] instructions.
b. Create a project of IAR for μp2. IAR is designed for a range of different target processors. A project has to be created for μp2 to specify the processor under study. The steps are shown in [IAR, 1997].
c. Compile and link the project. It is necessary to compile and link the source files of μp2 with IAR before running μp2 in the environment. The steps are shown in [IAR, 1997].
d. Execute fault-injection experiments for μp2. Similar to step (4) of μp1.
e. Collect the results of all the experiments.

10.2.4 Determination of the CF

Table 10.1 presents six distinct states within which APP may reside. These six states describe the system in terms of the functional capabilities of its components at different instances of time; that is, the state in which the APP system is in at a particular time reflects whether the system is operational or whether it has failed.

If the experiments are separately executed based on each microprocessor, then the reliability of APP can be calculated based on the reliability value obtained for the two microprocessors (μp1 and μp2).

Table 10.1 Definition of States for Each Microprocessor

Name of State	Definition
Normal State	A fault-free state in which all outputs are correct with respect to the input.
Failure State 1	The Trip signal fails to be activated when it should be activated.
Failure State 2	The Trip signal is activated when it should not be activated.
Failure State 3	Other failures, which are indicated by other system outputs, such as LED, Semaphore, and Board ID sent from μp1 and μp2 to CP (Communication Microprocessor).
Recoverable State	A faulty state in which all outputs are correct with respect to the input.
Fail-safe State	The system is intentionally blocked by the FTM (Fault-tolerant mechanism), after trying to recover the error without success. The Trip signal is also activated.

It should be noted that it is impossible for the system to miss a trip signal when the analog input is inside the "Barn shape" because the system is not in a trip state. So from the definition of Failure State 1 in Table 10.1, it can occur only when the analog input is outside the "Barn shape." Similarly, Failure State 2 can occur only when the analog input is inside the "Barn shape." Failure State 3 can occur with analog input inside the "Barn shape" or outside the "Barn shape."

Table 10.2 shows the experimental results for the fault injection experiments. This table lists the number of occurrences of the states in which the APP remains for at least one cycle after a fault is injected. The number of occurrences of a state will be used to measure the CF (See Section 10.4).

The CF is the weighted sum of the probabilities of recovering from a fault with analog input inside the "Barn shape" and with analog input outside the "Barn shape:"

$$c = \frac{N_1+N_2}{N_{t1}} \times W_1 + \frac{N_3+N_4}{N_{t2}} \times W_2 \qquad (10.5)$$

where

N_1 the number of occurrences of the Normal State for an experiment such that the analog input is inside the "Barn shape" (e.g., as shown in Table 10.2 for μp1, $N_1 = 1195$);

N_2 the number of occurrences of the Fail-safe State for an experiment such that the analog input is inside the "Barn shape" (e.g., as shown in Table 10.2 for μp1, $N_2 = 355$);

N_3 the number of occurrences of the Normal State for an experiment such that the analog input is outside the "Barn shape" (e.g., as shown in Table 10.2 for µp1, N_3 = 1165);

N_4 the number of occurrences of the Fail-safe State for an experiment such that the analog input is outside the "Barn shape" (e.g., as shown in Table 10.2 for µp1, N_4 = 350);

N_{t1} the total number of experiments with analog input inside the "Barn shape" (e.g., as shown in Table 10.2 for µp1, N_{t1} = 2025);

N_{t2} the total number of experiments with analog input outside the "Barn shape" (e.g., as shown in Table 10.2 for µp1, N_{t2} = 2025);

W_1 the weight of experiments such that the analog input is inside the "Barn shape," $W_1 = 0.9999999943$, as determined in Chapter 4;

W_2 the weight of experiments such that the analog input is outside the "Barn shape," $W_2 = 5.7 \times 10^{-9}$, as determined in Chapter 4.

Table 10.2 Fault Injection Experimental Results

Safety System	Number of experiments		Normal State	Fail-safe State	Failure State 1	Failure State 2	Failure State 3	Recover-able State
µp1	Analog input inside the "Barn shape"	N_{t1}	N_1	N_2		N_9	N_{10}	N_5
		2025	1195	355	0	40	255	180
	Analog input outside the "Barn shape"	N_{t2}	N_3	N_4	N_7		N_8	N_6
		2025	1165	350	70	0	275	165
µp2	Analog input inside the "Barn shape"	N_{t1}	N_1	N_2		N_9	N_{10}	N_5
		3830	2210	510	0	95	630	385
	Analog input outside the "Barn shape"	N_{t2}	N_3	N_4	N_7		N_8	N_6
		3830	2175	480	155	0	610	410

10.3 Measurement Results

In order to obtain the experimental results, the following documents were used to measure the *coverage factor*:

- APP Module μp1 System SRS [APP, Y1]
- APP Module μp1 Flux/Delta Flux/Flow Application SRS [APP, Y2]
- APP Module μp2 System SRS [APP, Y3]
- APP Module μp2 Flux/Delta Flux/Flow Application SRS [APP, Y4]
- APP Module Communication Processor SRS [APP, Y5]
- APP Module μp1 System source code [APP, Y6]
- APP Module μp1 Flux/Delta Flux/Flow Application source code [APP, Y7]
- APP Module μp2 System source code [APP, Y8]
- APP Module μp2 Flux/Delta Flux/Flow Application source code [APP, Y9]
- APP Module Communication Processor System source code [APP, Y10]

The fault-injection experiments were performed to discover the effect of faults on the system given the existence of FTMs (fault-tolerant mechanisms) using the requirements and source code documents.

When a fault is injected, the APP system enters a Recoverable State. In most experiments, the system will come back to a Normal State from the Recoverable State or remain in the Recoverable State. A few injected faults will lead to Failure State 1, Failure State 2, or Failure State 3.

Experiments in which the Failure State was observed are presented in Table 10.3.

From Table 10.3, it can be seen that when the analog input condition is inside the "Barn shape," if a bit-flip fault occurs in the variable SA_TRIP_1_DEENRGZE (for μp1) and Trip_condition (for μp2) controlling the trip signal, the system will send a trip signal and enter a Failure State 2.

Referring to Table 10.3, when the analog input condition is outside the "Barn shape" the system should send a trip signal if no fault occurs. If a bit-flip fault occurs in the variable fAnalog_Input_6 (for μp1) and AIN[4] (for μp2) controlling one of the analog inputs, the system could miss a trip signal and enter a Failure State 1.

If a bit-flip fault occurs in the variable chLEDs_Outputs (for μp1), which indicates the status of the LED, and have_dpm (for μp2), which indicates whether the semaphore is available, the system will enter a Failure State 3.

Table 10.3 Example Experiments Leading to the System Failure

Safety system	Input condition	Variable in which a fault was injected	Time at which the fault is injected
µp1	Analog input inside the "Barn shape"	SA_TRIP_1_DEENRGZE	During RAM test of Diagnostic
		SA_TRIP_1_DEENRGZE	During PROM test of Diagnostic
		SA_TRIP_1_DEENRGZE	During Analog input test of Diagnostic
		SA_TRIP_1_DEENRGZE	During the execution of Main and after status checking
		SA_TRIP_1_DEENRGZE	During calculating analog input of Main
	Analog input outside the "Barn shape"	fAnalog_Input_6	During the execution of Main and after status checking
		SA_TRIP_1_DEENRGZE	During the execution of Main and after status checking
		chLEDs_Outputs	During the execution of Main and after status checking
µp2	Analog input inside the "Barn shape"	Trip_condition	Before RAM test of Diagnostic
		Trip_condition	During RAM test of Diagnostic
		Trip_condition	During the execution of Main and after status checking
	Analog input outside the "Barn shape"	AIN[4]	During the execution of Main and after status checking
		have_dpm	During update DPM of Main.

10.4 RePS Construction Using Coverage Factors of µp1 and µp2

The APP system has three microprocessors: µp1, µp2, and CP (Communication Processor). According to [APP, 01], the entire APP system has four distinct operational modes: Power-on, Normal, Calibration, and Tuning. Moreover, most fault-tolerant mechanisms (such as RAM Test and Address Bus Line Test) are only available during the Normal Operation Mode, in which CP is not involved. Therefore, the RePS for APP was constructed only for the Normal Operation Mode, and CP is not considered in this chapter.

Three steps are required to estimate the reliability of APP based on the coverage measurements:

- Construct CTMC (Continuous-time Markov Chain) Models for µp1 and µp2
- Estimate the reliability of µp1 and µp2 based on the CTMC Models, respectively
- Calculate the reliability of the APP based on the reliability estimates of µp1 and µp2

10.4.1 Construction of Continuous-Time Markov Chain Model for a Microprocessor

There are several different models found in the literature that help predict reliability using the coverage factor for a fault-tolerant system, such as ESPN (Extended Stochastic Petri Net), and DTMC (Discrete Time Markov Chain) [Smidts, 2000].

The CTMC (Continuous-time Markov Chain) model, defined by a discrete state space and continuous time parameter, is a stochastic model suitable for describing the behavior of complex fault-tolerant systems. It can represent hardware, software, and their combined interactions in a single model to provide various information. Furthermore, it can represent the rate at which the state changes occur, rather than simply probabilities as in the DTMC (Discrete Time Markov Chain) [Kaufman, 1999].

The statistical basis for this model is that of a Markov process whose fundamental premises, which are referred to as the memory-less property, are:

1. All past state information is irrelevant; that is, state memory is not required.
2. The length of time during which the current process has been in a given state is irrelevant; that is, state age memory is not required.

The CTMC models for µp1 and µp2 are similar to each other because both microprocessors implement the same fault-tolerant mechanisms, such as RAM Test, PROM Test, and EEPROM Test. The only difference between these two CTMC models lies in the values of the model parameters. The CTMC model for either µp1 or µp2 is shown in Figure 10.1.

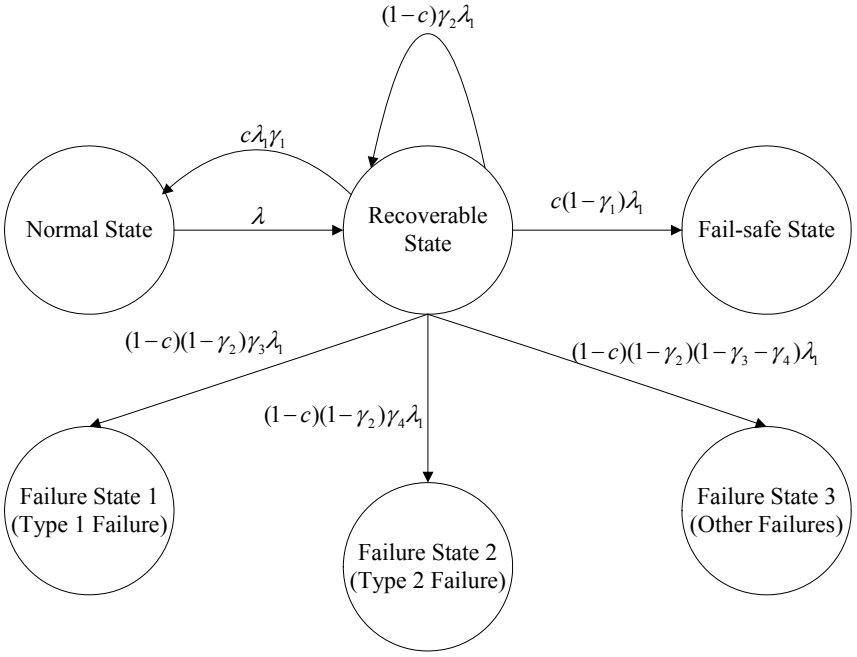


Figure 10.1 CTMC Model for µp1 or µp2

The state transition parameters that are required by such a Markov chain model are listed in Table 10.4.

Table 10.4 APP State Transition Parameters

Name of State Transition Parameter	Definition
λ	The rate at which an error occurs in the system (e.g., bit-flip in memory), independently of whether or not it is detected by the FTM (unit: per second)
λ_1	The rate at which the system deals with the fault injected and generates the result which indicates whether the fault can be recovered (unit: per second)
c	The coverage factor
γ_1	The probability that the system is brought back to the Normal State when an erroneous state is recovered
γ_2	The probability that the system remains in the Recoverable State when an erroneous state cannot be recovered
γ_3	The probability that the system enters the Failure State 1 when an erroneous state leads to the system failure
γ_4	The probability that the system enters the Failure State 2 when an erroneous state leads to the system failure

The steps to calculate the state transition parameters are:

1. Determining the Failure Rate of a microprocessor, λ, and the rate at which the system responds with the fault injected and generates the result that indicates whether or not the system can recover from the fault, λ_1.

The failure rate of a microprocessor, λ, is usually estimated by summing up the failure rates of all primary components:

$$\lambda = \sum_{all\ i} \beta_i \tag{10.6}$$

where

λ the failure rate of a microprocessor; and
β_i the failure rate of the i-th primary component.

The primary components for µp1 and µp2 are: CPU (Central Processing Unit), RAM (Random Access Memory), PROM (Programmable Read Only Memory), EEPROM (Electrical Erasable PROM), DPM (Dual Port RAM), and ABL (Address Bus Line). The failure rates of these five primary components are estimated by [Chu, 2005], as summarized in Table 10.5.

Table 10.5 Component Failure Rates

Failure Rate	Description	Value, in failure/hour
β_1	Failure rate of RAM	3.3E-07
β_2	PROM	2.6E-08
β_3	EEPROM	2.46E-09
β_4	DPM	1.7E-08
β_5	Address Bus Line	5.22E-07
β_6	CPU register	6.1E-8

The number of CPU registers in these two safety microprocessors are: 20 (µp1) [Dallas, 1995], and 22 (µp2) [ZiLOG, 2000].

Therefore, according to Equation 10.6, the failure rate of microprocessor µp1 is:

$$\lambda = (3.3 + 0.26 + 0.0246 + 0.17 + 5.22 + 0.61 \times 20) \times 10^{-7}/\text{hour}$$
$$= 2.117 \times 10^{-6}/\text{hour} = 5.883 \times 10^{-10}/\text{second}$$

The failure rate of microprocessor μp2 is:

$$\lambda = (3.3 + 0.26 + 0.0246 + 0.17 + 5.22 + 0.61 \times 22) \times 10^{-7}/\text{hour}$$
$$= 2.340 \times 10^{-6}/\text{hour} = 6.500 \times 10^{-10}/\text{second}$$

The rate $\hat{\lambda}_1$ at which the system deals with the fault injected and generates the result depends on the time required to tolerate the fault or experience a failure. In this chapter, an injected fault is generally recovered or causes the microprocessor failure in one program cycle time, otherwise it is regarded as latent in the Recoverable State. The rate λ_1 is the average rate for all the faults injected into the APP.

The time required to recover from the Recoverable State to the Normal State is one program cycle time, 0.129 s, therefore:

$$\lambda_1 = \frac{1}{0.129 \text{ second}} = 7.75/\text{second}$$

2. Determining the Transition Parameters γ_1, γ_2, γ_3, and γ_4.

The state transition parameters γ_1, γ_2, γ_3, and $\gamma_4$can be determined using the data in Table 10.2 and Equations 10.7 through 10.10:

$$\gamma_1 = \frac{N_1}{N_1+N_2} \times W_1 + \frac{N_3}{N_3+N_4} \times W_2 \quad (10.7)$$

$$\gamma_2 = \frac{N_5}{N_{t1}-(N_1+N_2)} \times W_1 + \frac{N_6}{N_{t2}-(N_3+N_4)} \times W_2 \quad (10.8)$$

$$\gamma_3 = \frac{N_7 \times W_2}{(N_7+N_8) \times W_2 + (N_9+N_{10}) \times W_1} \quad (10.9)$$

$$\gamma_4 = \frac{N_9 \times W_1}{(N_7+N_8) \times W_2 + (N_9+N_{10}) \times W_1} \quad (10.10)$$

Where

N_1, N_2, N_3, N_4, N_{t1}, N_{t2}, W_1, and $W_2$are the same as those in Equation 10.5;

N_5 the number of occurrences of the Recoverable State for an experiment such that the analog input is inside the "Barn shape" (shown in Table 10.2);

N_6 the number of occurrences of the Recoverable State for an experiment such that the analog input is outside the "Barn shape" (shown in Table 10.2);

N_7 the number of occurrences of the Failure State 1 for an experiment such that the analog input is outside the "Barn shape" (shown in Table 10.2);

N_8 the number of occurrences of the Failure State 3 for an experiment such that the analog input is outside the "Barn shape" (shown in Table 10.2);

N_9 the number of occurrences of the Failure State 2 for an experiment such that the analog input is inside the "Barn shape" (shown in Table 10.2);

N_{10} the number of occurrences of the Failure State 3 for an experiment such that the analog input is inside the "Barn shape" (shown in Table 10.2).

Table 10.6 summarizes the transition parameters for μp1 and μp2 based on Table 10.2 and Equation 10.5 through Equation 10.10.

Table 10.6 Transition Parameters (Probability)

Safety system	c	λ	λ_1	γ_1	γ_2	γ_3	γ_4
μp1	0.7654	5.883E-10/s	7.75/s	0.7710	0.3789	1.3525E-9	0.1356
μp2	0.7102	6.5E-10/s	7.75/s	0.8125	0.3468	1.2186E-9	0.1310

The parameter λ_1 of μp1 is the same as that of μp2 due to the fact that the failure rates of the hardware components, such as RAM, PROM, DPM, Address Bus Line, and EEPROM, are assumed to be the same for the two microprocessors.

10.4.2 Estimate the Reliabilities of μp1 and μp2

The CTMC (Continuous-time Markov Chain) can be used to estimate the probability of each state.

The steps of applying CTMC are:

- **Construct the differential equations governing a microprocessor's behavior**

According to [Carsten, 1973], the differential equation governing the relationship in the model is:

$$\frac{d\vec{P}(t)}{dt} = \bar{\bar{A}}\vec{P}(t) \tag{10.11}$$

where

$\vec{P}(t)$ a column vector whose elements are the system state probabilities at time t, $\vec{P}(t) = \left(p_1(t), p_2(t), \ldots, p_n(t)\right)^T$

$p_1(t)$ the probability that the system is in a state *i* at time *t*, i = 1, 2, 3, ..., n

n a finite and countable number of states for a state space

$\bar{\bar{A}}$ the $n \times n$ matrix of the transition rates

The following notations are used for the CTMC model shown in Figure 10.1:

$p_1(t)$ the probability that the system is in "Normal State" at time *t*

$p_2(t)$ the probability that the system is in "Recoverable State" at time *t*

$p_3(t)$ the probability that the system is in "Fail-safe State" at time *t*

$p_4(t)$ the probability that the system is in "Failure State 1" at time *t*

$p_5(t)$ the probability that the system is in "Failure State 2" at time *t*

$p_6(t)$ the probability that the system is in "Failure State 3" at time *t*

From Figure 10.1, one obtains Equation 10.12 and Equation 10.13:

$$\begin{cases} \frac{dp_1(t)}{dt} = -\lambda p_1(t) + c\gamma_1\lambda_1 p_2(t) \\ \frac{dp_2(t)}{dt} = \lambda p_1(t) - [1-(1-c)\gamma_2]\lambda_1 p_2(t) \\ \frac{dp_3(t)}{dt} = c(1-\gamma_1)\lambda_1 p_2(t) \\ \frac{dp_4(t)}{dt} = (1-c)(1-\gamma_2)\gamma_3\lambda_1 p_2(t) \\ \frac{dp_5(t)}{dt} = (1-c)(1-\gamma_2)\gamma_4\lambda_1 p_2(t) \\ \frac{dp_6(t)}{dt} = (1-c)(1-\gamma_2)(1-\gamma_3-\gamma_4)\lambda_1 p_2(t) \end{cases} \quad (10.12)$$

and

$$\bar{\bar{A}} = \begin{bmatrix} -\lambda & c\gamma_1\lambda_1 & 0 & 0 & 0 & 0 \\ \lambda & -[1-(1-c)\gamma_2]\lambda_1 & 0 & 0 & 0 & 0 \\ 0 & c(1-\gamma_1)\lambda_1 & 0 & 0 & 0 & 0 \\ 0 & (1-c)(1-\gamma_2)\gamma_3\lambda_1 & 0 & 0 & 0 & 0 \\ 0 & (1-c)(1-\gamma_2)\gamma_4\lambda_1 & 0 & 0 & 0 & 0 \\ 0 & (1-c)(1-\gamma_2)(1-\gamma_3-\gamma_4)\lambda_1 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (10.13)$$

- **Solve the differential equations to obtain the probability in each state**

As the number of system components and their failure modes increases, there is an exponential increase in system states, making the resulting reliability model more difficult to analyze. The large number of system states makes it difficult to solve the resulting model, to interpret state probabilities, and to conduct sensitivity analyses. However, this is not the case for the APP since the level of abstraction is such that the number of states is limited.

Knowing the initial conditions given by the state vector $\vec{P}(t=0)$ the set of simultaneous differential equations can be solved:

$$\vec{P}(t) = e^{\bar{\bar{A}}t}\vec{P}(t=0) \quad (10.14)$$

For a microprocessor, when it starts to work, the system is assumed to be in the Normal State, so the initial condition is:

$$p_1(0) = 1, p_2(0) = 0, p_3(0) = 0, p_4(0) = 0, p_5(0) = 0, p_6(0) = 0$$

namely,

$$\vec{P}(0) = (1,0,0,0,0,0)^T$$

Based on Table 10.6 and Equation 10.14, using the initial condition one obtains probabilities of the six states of μp1 and μp2 with $t = 0.129$ seconds, as listed in Table 10.7:

From Table 10.7, the probability of the Normal state is larger than that of other states because its failure rate is low and the FTMs in the microprocessor can recover most faults. In addition, the probability of Failure State 2 is much greater than that of Failure State 1 because most analog inputs are inside the "Barn shape" (Chapter 4).

Table 10.7 Probabilities of Six States of µp1 and µp2 with t = 0.129 Seconds

	$p_1(t)$	$p_2(t)$	$p_3(t)$	$p_4(t)$	$p_5(t)$	$p_6(t)$
µp1	9.99999999924109E-1	4.98E-11	5.02E-12	5.6415E-21	5.65E-13	3.61E-12
µp2	9.99999999916156E-1	5.53E-11	4.2258E-12	7.3204E-21	7.8695E-13	5.2203E-12

- **Calculate the reliability of a safety microprocessor**

In this experiment, the Normal State, the Recoverable State, and the Fail-safe State are regarded as reliable states because no failure occurs. The reliability of a safety microprocessor is the sum of the probabilities of these three states. Therefore:

$$R(t) = \sum_{i=1}^{3} p_i(t) \tag{10.15}$$

where

$R(t)$ the reliability of a microprocessor

$p_i(t)$ the probability that the microprocessor remains in the *i*-th reliable state, *i* = 1, 2, and 3, corresponding to the Normal State, the Recoverable State, and the Fail-safe State, respectively

From Table 10.7, based on Equation 10.15, the reliabilities of the two safety microprocessors at *t* =0.129 seconds are presented in Table 10.8.

Table 10.8 Reliabilities of µp1 and µp2 with t = 0.129 Seconds

Microprocessor	Reliability, $R(t)$
µp1	0.999999999978936
µp2	0.999999999975681

10.4.3 Reliability Calculation for the APP

For the whole APP system, there are also three types of independent failures: Type 1 Failure, Type 2 Failure, and Type 3 Failure (see Table 10.1). Therefore:

$$R_s(t) = 1 - \sum_{i=1}^{3} F_i(t) \tag{10.16}$$

where

$R_s(t)$ the reliability of the whole APP system

$F_i(t)$ the probability of the *i*-th type of failure, *i* = 1, 2, and 3

For Failure State 1, the APP system will miss a trip signal only when both microprocessors miss the trip signal, that is, APP will enter Failure State 1 only when both microprocessors enter Failure State 1. Therefore, μp1 and μp2 are logically in parallel. Then, the probability of Failure State 1 at *t* = 0.129 seconds for APP is:

$$F_1(t) = (5.64 \times 10^{-21}) \times (7.32 \times 10^{-21}) = 4.13 \times 10^{-41}$$

For Failure State 2, the APP system will send a trip signal once either microprocessor generates a trip signal, that is, APP will enter Failure State 2 when either safety system enters Failure State 2. So μp1 and μp2 are logically in series and the probability of Failure State 2 at *t* = 0.129 seconds for APP is:

$$F_2(t) = 1 - (1 - 5.65 \times 10^{-13}) \times (1 - 7.87 \times 10^{-13}) = 1.3526 \times 10^{-12}$$

The APP system will enter Failure State 3 when a microprocessor failure occurs, which is indicated by LED, Semaphore, or Board ID sent from μp1 and μp2 to CP. Therefore, μp1 and μp2 are logically in series and the probability of Failure State 3 at *t* = 0.129 seconds for the APP is:

$$F_3(t) = 1 - (1 - 3.61 \times 10^{-12}) \times (1 - 5.22 \times 10^{-12}) = 8.83 \times 10^{-12}$$

Based on Equation 10.16, the reliability of the whole APP system at t =0.129 seconds is:

$$R_s(t) = 1 - 4.13 \times 10^{-41} - 1.35 \times 10^{-12} - 8.83 \times 10^{-12} = 0.9999999999898$$

Fault-Tolerant Mechanisms (FTMs) are one of the major concerns of system design. A powerful FTM will increase the reliability and safety of the system, and decrease the probability of system failure. The CF is used to quantify the efficiency of the system FTM, which is a central problem in the validation of fault-tolerant systems [Powell, 1993]. By this measurement, the reliability of the system exceeds 0.999999999 per demand, which coincides with actual experience at the plant from which the operating data was obtained.

10.5 Lessons Learned

Fault-injection techniques have long been recognized as necessary to validate the dependability of a system. Artificial faults are injected into a system and the resulting behaviors are observed. Compared with other measurements, fault-injection techniques are useful in speeding up the occurrence and the propagation of faults into the system in order to observe the effects on the system performance. Fault injection techniques can be performed on either simulations and models or working prototypes and systems in the field. In this manner, the weaknesses of

interactions can be discovered. This approach is frequently used to test the resilience of a fault-tolerant system against known faults, and thereby measure the effectiveness of the fault-tolerant measures [Alfredo, 2003].

One difficulty of fault injection involves the simulation of temporary faults, which are the faults most likely to occur in a computer system. The nature of these temporary faults makes exhaustive testing exceedingly time-consuming. As a result, coverage evaluation is a problem of statistical estimation, where inferences about a population are based on sample observations.

When calculating the probability for each failure type (Type 1, Type 2, and Type 3), common cause failures were not considered. Common cause failure is a specific kind of dependent failure that arises in redundant components where simultaneous (or concurrent) multiple failures result in different channels from a single shared cause [Mauri, 2000] [Vesely, 2001] [Breakers, 2003]. Research on quantifying the impact of common-cause failures on fault-tolerant systems is beyond the scope of this report and is identified as a follow-on issue in Chapter 19.

10.6 References

[APP, 01] *APP Instruction Manual*.

[APP, Y1] "APP Module First SFP SRS," Year Y1.

[APP, Y2] "APP Flux/Delta Flux/Flow Application SRS for SF1," Year Y2.

[APP, Y3] "APP Module μp2 System Software SRS," Year Y3.

[APP, Y4] "APP μp2 Flux/Delta Flux/Flow Application Software SRS," Year Y4.

[APP, Y5] "APP Module Communication Processor SRS," Year Y5.

[APP, Y6] "APP Module SF1 System Software code," Year Y6.

[APP, Y7] "APP SF1 Flux/Delta Flux/Flow Application code," Year Y7.

[APP, Y8] "APP Module μp2 System Software Source Code Listing," Year Y8.

[APP, Y9] "APP μp2 Flux/Delta Flux/Flow Application Software Source Code Listing," Year Y9.

[APP, Y10] "APP Comm. Processor Source Code," Year Y10.

[Arlat, 1990] J. Arlat, M. Aguera and L. Amat. "Fault Injection for Dependability Validation: A Methodology and Some Applications." *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 166–182, 1990.

[Arnold, 1973] T.F. Arnold. "The Concept of Coverage and its Effect on the Reliability Model of a Repairable System." *IEEE Transactions on Computers*, vol. C-22, pp. 251–254, 1973.

[Benso, 2003] A. Benso. *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*, 2003.

[Bouricius, 1969] W.G. Bouricius, W.C. Carter and P.R. Schneider. "Reliability Modeling Techniques for Self-repairing Computer Systems," in *Proc. 24th Nut. Con., ACM*, 1969, pp. 295–309.

[Breakers, 2003] C. Breakers. "Common-Cause Failure Event Insights," US NRC, NUREG/CR-6819, vol. 4, 2003.

[Brombacher, 1999] A.C. Brombacher. "RIFIT: Analyzing Hardware and Software in Safeguarding Systems," *Reliability Engineering and System Safety*, pp. 149–156, 1999.

[Carsten, 1973] B. Carsten and T. Heimly. "A Reliability Model Using Markov Chains for Utility Evaluation of Computer Systems Onboard Chips," Winter Simulation Conference, 1973.

[Chu, 2005] T.L. Chu et al. *Collection of Failure Data and Development of Database for Probabilistic Modeling of Digital Systems*, 2005.

[Choi, 2000] J.G. Choi et al. "Reliability Estimation of Nuclear Digital I&C System using Software Functional Block Diagram and Control Flow," in *Proc. International Symposium on Software Reliability Engineering*, 2000.

[Cukier, 1999] M. Cukier and D. Powell. "Coverage Estimation Methods for Stratified Fault-Injection," *IEEE Transactions on Computers*, vol. 48, no. 7, pp. 707–723, 1999.

[Dallas, 1995] Dallas Semiconductor, DS80C320/DS80C323 High-Speed/Low-Power Micro, 1995.

[Dugan, 1989] J.B. Dugan. "Coverage Modeling for Dependability Analysis of Fault-Tolerant Systems," *IEEE Transactions on Computers*, vol. 38, No. 6, pp. 775–787, 1989.

[Gil, 2002] P. Gil and J. Arlat. "DBench - Fault Representativeness, Chapter 3, Deliverable from Dependability Benchmarking," European IST project (IST-2000-25425), 2002.

[Hexel, 2003] R. Hexel. "FITS - A Fault Injection Architecture for Time-Triggered Systems," in *Proc. 26th Australian Computer Science Conference*, 2003.

[IAR, 1997] IAR Systems. *IAR Embedded Workbench Interface Guide*, 1997.

[Kaufman, 1999] L.M. Kaufman and B.W. Johnson. "Embedded Digital System Reliability and Safety Analyses," NUREG/GR-0020, UVA Technical Report, 1999.

[KEIL, 2001] Keil Elektonik GmbH and Keil Software, Inc. *Getting Started with µversion 2 and the C51 Microcontroller Development Tools*, 2001.

[Mauri, 2000] G. Mauri. "Integrating Safety Analysis Techniques, Supporting Identification of Common Cause Failures," Ph.D. dissertation, Department of Computer Science, The University of York, 2000.

[NRC, 1990] "A Cause-Defense Approach to the Understanding and Analysis of Common-cause Failures," US NRC, NUREG/CR-5460, 1990.

[Powell, 1993] D. Powell, E. Martins and J. Arlat. "Estimators for Fault Tolerance Coverage Evaluation," *IEEE Transactions on Computers*, vol. 42, no. 8, pp. 775–787, 1993.

[Vesely, 2001] W.E. Vesely, F. Hsu and M. Stewart, "Common Cause Failure Analysis Guideline for the Space Shuttle Probabilistic Risk Assessment," SAIC/NASA, JSC PRA Documentation, 2001.

[ZiLOG, 2000] ZiLOG Worldwide Headquarters, *Z8018x Family MPU User Manual*, 2000.

11. CYCLOMATIC COMPLEXITY

This measure determines the structural complexity of a coded module.

The resulting measurement can then be used to inform the developer's decision to redesign the module to limit its complexity, thereby promoting understandability of the module and minimizing the number of logical testing paths [IEEE 982.2, 1988]. A module's cyclomatic complexity (CC) is also a strong indicator of its testability.

Based on this measure, a set of derived measures for the cyclomatic complexity of the entire software product was proposed in this chapter, which may be used to estimate the fault content in the delivered source code.

This measure can only be applied when detailed design information is available. As listed in Table 3.3, the applicable life cycle phases for CC are Design, Coding, Testing, and Operation.

11.1 Definition

The CC of a module is the number of linearly independent paths through a module. This is an indication of how much effort is required to test a module if the test plan is to supply diverse inputs so that all combinations of branches are executed.

The CC for the *i*-th module is defined by McCabe [McCabe, 1976] [McCabe, 1982] as:

$$CC_i = E_i - N_i + 2 \tag{11.1}$$

where

CC_i	is the cyclomatic complexity measure of the *i*-th 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).

A *module* corresponding to a single function or subroutine in a typical language has a single entry and exit point and is able to be used as a design component via a call/return function. In C language, a module is a function. This definition is different from that of the BLOC measure, in which a module is defined as a .c file together with all user defined .h files it includes (refer to Chapter 6).

A *node* is a sequential group of program statements.

An *edge* is the program flow between nodes.

McCabe's definition (Equation 11.1) applies to a representation of the model's control flow graph in which there is no edge between the exit node and the entry node [Jones, 1991] and as such is a non-strongly connected graph.

As an example, consider a module's control flow graph shown in Figure 11.1. Each node is numbered 0 through 6 and edges are displayed using solid lines connecting the nodes. The module's cyclomatic complexity is 4 (9 edges minus 7 nodes plus 2).

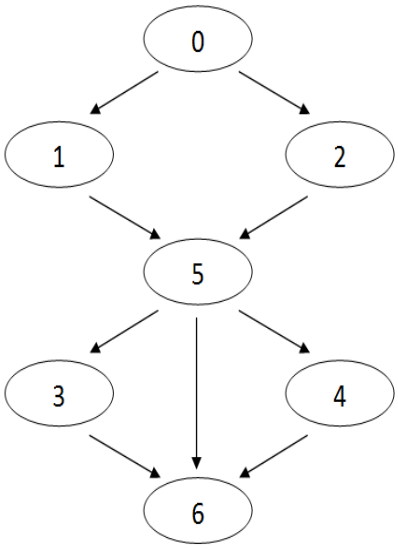


Figure 11.1 Control Flow Graph

When one uses a *strongly connected graph* to represent the module's control flow—where one fictitiously adds an edge from the exit node to the entry node—the Cyclomatic Complexity measure for the *i*-th module is [IEEE 982.2, 1988]:

$$CC_i = E_i - N_i + 1 \tag{11.2}$$

For the example above, the program-control-flow graph is not strongly connected. However, if we add a "virtual edge" to connect node 0 and node 6 (the dashed line in Figure 11.2), the flow graph becomes strongly connected. The number of nodes remains seven. The number of edges is now 10, thus the CC remains 4 (10 edges minus 7 nodes plus 1).

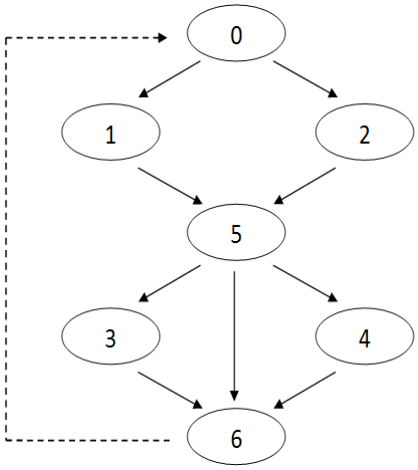


Figure 11.2 Control Flow Graph with a Virtual Edge

It should be noted that CC is a measure used for a single-coded module and not for an entire software product.

One way to characterize the cyclomatic complexity[25] of a software product is to use the following derived measures, which were proposed by the UMD research team based on Chapman's research [Chapman, 2002]:

$p_1\%$ Percentage of modules with CC < 4.
$p_2\%$ Percentage of modules with 4 ≤ CC < 10.
$p_3\%$ Percentage of modules with 10 ≤ CC < 16.
$p_4\%$ Percentage of modules with 16 ≤ CC < 20.
$p_5\%$ Percentage of modules with 20 ≤ CC < 30.
$p_6\%$ Percentage of modules with 30 ≤ CC < 80.
$p_7\%$ Percentage of modules with 80 ≤ CC < 100.
$p_8\%$ Percentage of modules with 100 ≤ CC < 200.
$p_9\%$ Percentage of modules with CC ≥ 200.

The percentage distribution of modules by CC level reflects the CC of a software product.

[25] Note that this is not the combined cyclomatic complexity of the software product. A combined cyclomatic complexity value is not necessary for RePS construction and reliability prediction.

11.2 Measurement Rules

The CC measure is based on the structure of a module's control-flow graph. Control-flow graphs describe the logic structure of software modules. Each flow graph consists of nodes and edges. Each possible execution path of a software module has a corresponding path from the entry node to the exit node of a module's control-flow graph.

For the remainder of this chapter, it is assumed that the constructed control-flow graphs are all *non-strongly-connected* (i.e., no edge exists between the entry and exit nodes).

Five steps are required to manually measure the CC of a module:

1. Beginning at the top of the source code, each non-comment line of code is numbered.
2. A circle is drawn to contain each number—each one is a "node."
3. All possible sequential nodes are joined with lines (i.e., "edges") to indicate the possible order in which the lines are executed.
4. The number of edges and the number of nodes in the control-flow graph are counted.
5. The CC of the *i*-th module is calculated using Equation 11.1.

It is time-consuming, tedious, and error-prone to manually construct the control-flow graphs and count the CC for each module. Fortunately, several easier methods to calculate CC exist in practice, ranging from counting decision predicates to using automated tools [Zuse, 1990] [Watson, 1996]. McCabe [McCabe, 1982] demonstrated that CC_i is also equal to the number of binary decision nodes in the control-graph plus one. Four basic rules can be used to calculate CC_i [McCabe, 1982] [Gill, 1997] [Hensen, 1978]:

1. Increment CC_i by one for every IF, CASE, or other alternate execution construct;
2. Increment CC_i by one for every Iterative DO, DO-WHILE, or other repetitive construct;
3. Add to CC_i the number of logical alternatives in a CASE minus two;
4. Add one to CC_i for each logical operator (AND, OR) in a conditional statement. Such statements include IF, CASE, DO, DO-WHILE, etc.

There are three variants of using the four rules mentioned above [Gill, 1997]:

a) Variant 1: all four rules are used, as in the original McCabe version.
b) Variant 2: only rules 1–3 apply, as proposed by [Myers, 1977].
c) Variant 3: only rules 1–2 apply, as suggested by Hansen [Hansen, 1978].

Variant 1 is widely recognized [Watson, 1996] [Gill, 1997] and is therefore adopted in this chapter.

In this report, RSM 6.8 [MST, 2005], a source code metrics and quality analysis tool for C, C++, Java, and C#, was used to measure the CC for all modules. This tool measures CC based on McCabe's four rules.

Once the CC for an individual module is obtained, the percentage distribution of modules by CC level should be determined using the following rules:

1. Divide the modules according to their level of cyclomatic complexity:

   Level 1: $0 \leq CC < 4$
   Level 2: $4 \leq CC < 10$
   Level 3: $10 \leq CC < 16$
   Level 4: $16 \leq CC < 20$
   Level 5: $20 \leq CC < 30$
   Level 6: $30 \leq CC < 80$
   Level 7: $80 \leq CC < 100$
   Level 8: $100 \leq CC < 200$
   Level 9: $CC \geq 200$

2. Count the number of modules for each cyclomatic complexity level;
3. Calculate the percentage distribution of modules by CC level according to Equation 11.3:

$$p_i = \frac{n_i}{\sum_{j=1}^{9} n_j} \tag{11.3}$$

where

p_i The percentage of modules with CC belonging to the *i*-th level. $i = 1, 2, ..., 9$.
n_j The number of modules with CC belonging to the *j*-th level. $j = 1, 2, ..., 9$.

One of the factors most often associated with successful and unsuccessful software projects [Jones, 1996] [Basili, 1984] [Stuzke, 2001] is the CC. In order to obtain a meaningful CC value for the entire software product, the concepts of Performance Influencing Factors (PIF) and Success Likelihood Index (SLI) are introduced. How good or how bad PIFs are in a given situation can be rated by experts and quantified by a SLI. SLI was used as an index that quantifies whether a particular environment will increase or decrease the human error probability (with respect to a "normal situation") [Stuzke, 2001]. The SLI ranges from 0 (error is likely) to 1 (error is not likely). This section discusses the rules for calculating the SLI of the CC.

It has been suggested that modules exceeding a threshold value of CC are difficult to test completely [Walsh, 1979] [McCabe, 1982] and incompletely tested software may be delivered with errors. According to McCabe [McCabe, 1982] modules with $CC > 10$ are at risk for deficient reliability. Walsh [Walsh, 1979] used $CC = 4$ as a threshold to estimate the defect density of the source code prior to unit testing.

Based on more recent research [Chapman, 2002], Equation 11.4 is proposed to quantify the impact of the CC factor on software quality:

$$SLI_1 = 1 - \sum_{i=1}^{9} f_i \times p_i\% \quad (11.4)$$

where

SLI_1	The SLI value of the CC factor
f_i	Failure likelihood f_i used for SLI_1 calculations, as shown in Table 11.1 (extracted from [Chapman, 2002])
p_i	Derived measures defined in Section 11.1, i = 1, 2, ... 9.

Table 11.1 Failure Likelihood f_i Used for SLI_1 Calculations

	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9
Value	0.08	0.15	0.25	0.35	0.45	0.55	0.65	0.75	1.0

Note that the above nine classes correspond to the complexity classes. The value of SLI_1 may be used as a quality indicator of a software product. SLI_1 is related to the likelihood that developers will err (i.e., introduce faults in the software product and/or fail to remove them) because of the CC of the modules.

11.3 Measurement Results

The following documents were used to measure module CC:

- APP Module µp1 System source code [APP, Y1]
- APP Module µp1 Flux/Delta Flux/Flow Application source code [APP, Y2]
- APP Module µp2 System source code [APP, Y3]
- APP Module µp2 Flux/Delta Flux/Flow Application source code [APP, Y4]
- APP Module Communication Processor System source code [APP, Y5]

The CC measures for all modules of the APP system are presented in Table 11.2.

Table 11.3 presents the counting of n_i for the APP system using the results in Table 11.2. The percentage distribution of modules for the APP system and the calculated SLI of the CC measure (SLI_1) are shown in Table 11.4 and 11.5, respectively.

Table 11.2 Measurement Results for CC_i

Software Name	File Name	Module Name	Cyclomatic Complexity
CP System Source Code	CMMONLI.c	Online Operation procedures	7
		Check cycle monitor procedure	11
		Check trip outputs procedure	17
		Test Mode procedure	2
	COMMPOW.c	Power Up Self Tests	24
		AM Tests procedure	6
		Address Line Tests procedure	4
		ROM Checksum procedure	2
		Board ID test procedure	3
		Halt procedure after diagnostic test failure	2
		Halt procedure after Module ID test failure	5
		Online diagnostic procedure	16
		Timer 0 Interrupt service routine	5

Table 11.2 Measurement Results for CC_i (continued)

Software Name	File Name	Module Name	Cyclomatic Complexity
CP System Source Code	COMMPROC.c	Main Program	2
		External interrupt 0 and 1 service routine	2
		Timer 1 Interrupt service routine	10
		Dual Port RAM Semaphore Handler function	3
		Disable Interrupt routine	2
		Enable Interrupt routine	2
		Initialization Procedure	3
		Process Serial Communication	23
		1 Transmit Buffer with a byte	4
		Get buffer size	3
		Get receive buffer byte	3
		Process time out counter	2
	COMMSER.c	Receive Dual Port RAM data	13
		Examine determine data direction transmission	8
		Receive Time of Day update	4
		Transmit dual port RAM data	11
		Transmit APP Status table	4
		Calculate CRC using CRC-CCITT methods	3
		Serial Communication Interrupt	9
µp1-Application Source Code	SF1APP.c	Application Program	36
		Application Program Diagnostic Test	24
		Square Root Function	2

Table 11.2 Measurement Results for CC_i (continued)

Software Name	File Name	Module Name	Cyclomatic Complexity
µp1- System Source Code µp1- System Source Code	SF1CALTN.c	Calibrate/Tune function	12
		Calibration function	6
		Tuning function	8
		Input calibration function	10
		Download tuning data from DPR function	4
		Handling input potentiometers function	28
	SF1FUNCT.c	Majority function	4
		Access semaphore function	9
		Averaging function	3
		Median function	5
		Read analog inputs function	3
		Copy status table to DPR function	2
		Generate discrete output signals function	11
		Generate front panel LEDs output signals	5
		Generate outputs function	3
		Generate status relays output signals	2
		Halt function	2
		Read module input signals function	53
		Reset outputs module	2
		Wait function	10
		Generate analog output signals function	1

Table 11.2 Measurement Results for CC_i (continued)

Software Name	File Name	Module Name	Cyclomatic Complexity
µp1- System Source Code	SF1FUNCT.c	Main function	2
		Initialization function	5
		Main program function	21
		External zero interrupt function	4
		External one interrupt function	4
		Serial interrupt function	4
		Timer 0 interrupt function	1
		Timer 1 interrupt function	2
		Timer 2 interrupt function	3
		Power-Up Self Tests function	2
	SF1TEST1.c	On-line diagnostics function	18
		External RAM test function	5
		DPR test function	2
		fun_perform memory R/W to external RAM/DPR	8
		Address lines test function	5
	SF1TEST2.c	PROM checksum test function	5
		EEPROM checksum test function	8
		fun_calculating checksum for PROM and EEPROM	6

Table 11.3 n_i Counts Per Subsystem

The number of modules whose CC belongs to *i*-th level, n_i	CP System	µp1 System	µp1 Application	µp2 System	µp2 Application
n_1	14	40	1	6	8
n_2	10	22	0	14	11
n_3	4	7	0	4	4
n_4	2	2	0	0	0
n_5	2	2	1	1	2
n_6	0	4	1	1	0
n_7	0	0	0	0	0
n_8	0	0	0	0	0
n_9	0	0	0	0	0

Table 11.4 Percentage Distribution of the APP System Modules

Derived Measure	Values of Derived Measure for				
	CP	µp1 System	µp1 Application	µp2 System	µp2 Application
p_1	43.75%	51.95%	33.33%	23.08%	32%
p_2	31.25%	28.57%	0	53.85%	44.0%
p_3	12.5%	9.09%	0	15.38%	16.0%
p_4	6.25%	2.6%	0	0	0
p_5	6.25%	2.6%	33.33%	3.85%	8%
p_6	0	5.19%	33.33%	3.85%	0
p_7	0	0	0	0	0
p_8	0	0	0	0	0
p_9	0	0	0	0	0

Table 11.5 SLI_1 for the Different Subsystems

	CP System	µp1 System	µp1 Application	µp2 System	µp2 Application
SLI_1	0.8369	0.8435	0.6400	0.8239	0.8324

11.4 RePS Construction Using the Cyclomatic Complexity Measure

Reliability prediction based on the CC measure consists of the following two steps:

- Estimate the fault contents in the delivered source code using the Success Likelihood Index Method (SLIM) (as described below).
- Calculate the reliability using Musa's Exponential Model (as described below).

11.4.1 Estimating the Fault Contents in the Delivered Source Code

Numerous influencing factors can be identified that potentially affect the magnitude of the intensity and probability-density functions. One method used in human reliability analysis to account for the quantitative aspects of influencing factors is the SLIM, developed in [Embrey, 1983], refined in [Dougherty, 1988] and critiqued in [Reason, 1990].

The SLIM is founded on three key assumptions:

1. The likelihood of an error occurring in a particular situation depends on the combined effects of a relatively small number of PIFs, which are represented by SLI.
2. Experts can numerically rate how good or bad these PIFs are in a given situation.
3. The probability of a human error is logarithmically proportional to the SLI.

$$SLI = a \ln HEP + b \tag{11.5}$$

where HEP = Human Error Probability and a and b are two constants to be determined using experimental data.

Based on the above SLIM method, Equation 11.6 is proposed for estimating the fault content in delivered source code with the assumption that the likelihood of an error occurring depends on the entire software product.

$$N = k \times A \times F^{1-2XSLI} \tag{11.6}$$

where

N	the number of faults remaining in the delivered source code
k	a universal constant, estimated by fitting experiment data
A	the amount of activity in developing the delivered source code
F	universal constant, estimated by fitting experiment data
SLI	the Success Likelihood Index of the entire software product

The UMD research team then examined twelve software products to find the values of constants *k* and *F*. The size of the source code (in terms of LOC) was chosen as a measure to quantify the amount of activity in developing the delivered source code. The data gave k = 0.036 and F = 20.

If the set of PIFs used in the SLIM model is restricted to the value of CC for the different factors, Equation 11.6 can be modified into Equation 11.7[26], which links the fault contents to the code size and CC. Further work is required to validate the values of *k* and *F* in Equation 11.6.

$$N = 0.036 \times SIZE \times (20)^{1-2\times SLI_1} \tag{11.7}$$

where

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

Table 11.6 summarizes the fault content calculation results for the APP system.

Table 11.6 Summary of Fault Content Calculation Results

	CP System	**µp1 System**	**µp1 Application**	**µp2 System**	**µp2 Application**
SIZE, in LOC	1,210	2,034	480	895	206
$SLI = SLI_1$	0.8369	0.8435	0.6400	0.8239	0.8324
Defects in source code	5.8	9.4	7.5	4.6	1.0

The estimated number of faults in the entire APP system is:

$$N_{CC} = 5.8 + 9.4 + 7.5 + 4.6 + 1.0 = 28.3\ defects \tag{11.8}$$

11.4.2 Calculating the Reliability Using the Fault-Contents Estimation

The probability of success-per-demand is obtained using Musa's exponential model [Musa, 1990] [Smidts, 2004]:

$$p_s(CC) = e^{\left(-K\times N_{CC}\times \tau / T_L\right)} \tag{11.9}$$

where

$P_s(CC)$ Reliability estimation for the APP system accounting for the effect of CC.
K Fault Exposure Ratio, in failures/defect.
N_{CC} Number of defects estimated using the CC measure.
τ Average execution-time-per-demand, in seconds/demand.
T_L Linear execution time of a system, in seconds.

[26] Parameters *K* and *F* are determined using severity level 1 and 2 defects only. Thus, the number of defects obtained from Equation 11.7 is for severity level 1 and 2 defects only.

Since *a priori* knowledge of the defect locations and impact of the defects on failure probability is unknown, the average *K* value given in [Musa, 1990], which is 4.2×10^{-7} failure/defect, must be used.

For the APP system, N_{CC} = 28.3, as calculated in Section 11.4.1.

The *linear execution time*, T_L, is usually estimated as the ratio of the execution time and the software size on a single microprocessor basis [Musa, 1990] [Smidts, 2004]. In the case of the APP system, however, there are three parallel subsystems (μp1, μp2, and CP), each of which has a microprocessor executing its own software. Each of these three subsystems has an estimated linear execution time. Therefore, there are several ways to estimate the linear execution time for the entire APP system, such as using the average value of these three subsystems.

For a safety-critical application, such as the APP system, the UMD research team suggests a conservative estimation of T_L by using the minimum T_L of the three values. Namely:

$$\begin{aligned} T_L &= \min\{T_L(\mu p1), T_L(\mu p2), T_L(CP)\} \\ &= \min\{0.018, 0.009, 0.021\} \\ &= 0.009\ seconds \end{aligned} \tag{11.10}$$

where

$T_L(\mu p1)$: Linear execution time of Microprocessor 1 (μp1) of the APP system. T_L *(μp1)* = 0.018 second, as determined in Chapter 17;

$T_L(\mu p2)$: Linear execution time of Microprocessor 2 (μp2) of the APP system. T_L *(μp2)* = 0.009 second, as determined in Chapter 17;

$T_L(CP)$: Linear execution time of Communication Microprocessor (CP) of the APP system. T_L *(CP)* = 0.021 second, as determined in Chapter 17.

Similarly, the *average execution-time-per-demand*, τ, is also estimated on a single microprocessor basis. Each of the three subsystems in the APP has an estimated average execution-time-per-demand. To make a conservative estimation, the average execution-time-per-demand for the entire APP system is the maximum of the three values. Namely:

$$\begin{aligned} \tau &= \max\{\tau(\mu p1), \tau(\mu p2), \tau(CP)\} \\ &= \max\{0.082, 0.129, 0.016\} \\ &= 0.129 seconds/demand \end{aligned} \tag{11.11}$$

Where

$\tau(\mu p1)$ Average execution-time-per-demand of μp1 of the APP system. *τ(μp1)* = 0.082 seconds/demand, as determined in Chapter 17;

$\tau(\mu p2)$ Average execution-time-per-demand of μp2 of the APP system. *τ(μp2)* = 0.129 seconds/demand, as determined in Chapter 17;

$\tau(CP)$ Average execution-time-per-demand of CP\ of the APP system. *τ(CP)* = 0.016 seconds/demand, as determined in Chapter 17.

Thus the reliability for the APP system using the CC measure is given by:

$$p_s(CC) = \exp\left(-4.2 \times 10^{-7} \times 28.3\ defects \times \frac{0.129\ s/demand}{0.009\ s}\right) = 0.9998296\ \ defect/demand \quad (11.12)$$

11.4.3 An Approach to Improve the Prediction Obtained from the CC Measure

The UMD approach described in sections 11.4.1 and 11.4.2 relates CC and the number of defects directly using the SLI concept and the SLIM model. However, it is obvious that the number of defects in the software is affected by many other factors besides CC. Thus, estimation based only on CC is inaccurate. To improve the prediction of the number of defects and the reliability prediction, other factors (PIFs) that could affect predicted defect number should be incorporated in the SLIM model as additional *support measures*. These factors include:

- Development Schedule Factor (SCED)
- Experience Factor
  - Application Experience (APEX)
  - Platform Experience (PLEX)
  - Language and Tool Experience (LTEX)
- Capability Factor
  - Analyst Capability (ACAP)
  - Programmer Capability (PCAP)
  - Tester Capability (TCAP)
  - Personnel Continuity (PCON)
- Development Tools Factor (TOOL)
- Development Site Factor (SITE)
- Team Cohesion Factor (TEAM)
- Management Style Factor (STYLE)
- Process Maturity Factor (PMAT)
- Requirement Evolution Factor (REVL)

A justification for using such factors to predict the number of defects remaining in the software is found in the software engineering literature:

1. SCED, APEX, PLEX, LTEX, ACAP, PCAP, PCON, TOOL, SITE, TEAM, and PMAT are factors defined in COQUALMO. COQUALMO is a quality model extension of the existing COCOMO II [Boehm, 2000]. It is based on the software Defect Introduction and Defect Removal model described by Boehm [Boehm, 1981]. All the factors identified in COQUALMO are related to defects content in the software.

2. TCAP, STYLE, and REVL are identified as important influencing factors to software-failure density by a team of experienced software developers [Stutzke, 2001].
3. STYLE and REVL are also factors identified as two out of the 32 factors influencing software reliability by Pham [Pham, 2000].

Definitions, measurement rules, and SLI ratings for each of the above factors are presented in the following sections. If data for a PIF is unavailable, the value 0.5 (corresponding to a nominal/average situation) for the corresponding SLI should be used.

11.4.3.1 Development Schedule Factor (SCED)

This factor measures the schedule constraint imposed on the project team developing the software. The rating scales for SCED are defined in terms of the percentage of schedule stretch-out or acceleration with respect to a nominal schedule.

The development schedule factor can be estimated using [Boehm, 1982] [Boehm, 2000]:

$$SCED = \frac{TDEV_{actual}}{TDEV_{nominal}} \times 100\% \tag{11.13}$$

$$TDEV_{nominal} = 3.67 \times \left[2.94 \times \left(SIZE_{developed}\right)^{1.15}\right]^{0.328} \tag{11.14}$$

$$SIZE_{developed} = SIZE_{delivered} + SIZE_{discarded} \tag{11.15}$$

where

$TDEV_{actual}$	Actual time to develop the software, in calendar months.
$TDEV_{nominal}$	Nominal time to develop the software, in calendar months.
$SIZE_{developed}$	The size of developed source code, in KLOC.
$SIZE_{delivered}$	The size of finally delivered source code, in KLOC.
$SIZE_{discarded}$	The size of source code discarded during development, in KLOC.

Either $SIZE_{delivered}$ or $SIZE_{discarded}$ is given by:

$$SIZE_{delivered\ or\ discarded} = SIZE_{new} + ESIZE_{adapted} + ESIZE_{reused} + ESIZE_{COTS} \tag{11.16}$$

where

$SIZE_{new}$	The size of new code developed, in KLOC (Kilo Line of Code).
$ESIZE_{adapted}$	The equivalent size of adapted code, in *KLOC*. Adapted code is preexisting code that is treated as a white-box and is modified for use with the product.

$ESIZE_{reused}$	The equivalent size of reused code, in *KLOC*. Reused code is preexisting code that is treated as a black-box and plugged into the product without modification.
$ESIZE_{COTS}$	The equivalent size of off-the-shelf software, in *KLOC*. There may be some new interface code associated with it, which also needs to be counted as new code.

The equivalent size of adapted, reused, or COTS code is calculated according to the following sizing equations:

$$ESIZE_{adapted,reused,or\ COTS} = SIZE_{adapted,reused,or\ COTS} \times \left(1 - \frac{AT}{100}\right) \times AAM \quad (11.17)$$

$$\begin{cases} \frac{[AA+AAF(1+(0.02\times SU\times UNFM))]}{100} & for\ AAF \leq 50 \\ \frac{[AA+AAF(SU\times UNFM)]}{100} & for\ AAF > 50 \end{cases} \quad (11.18)$$

$$AAF = (0.4 \times DM) + (0.3 \times CM) + (0.3 \times IM) \quad (11.19)$$

where

AA	Assessment and Assimilation Increment
AAF	Adaptation Adjustment Factor
AAM	Adaptation Adjustment Modifier
AT	Percentage of Code Re-engineered by Automation
CM	Percentage of Code Modified
DM	Percentage of Design Modified
IM	Percentage of Integration Effort Required for Integrating Adapted or Reused Software.
SU	Percentage of Software Understanding
$UNFM$	Programmer Unfamiliarity with Software

If the software is developed without using any adapted, reused, or COTS source code (like the APP system), the $ESIZE_{adapted,reused,or\ COTS} = 0$. Otherwise, it is necessary to measure *AT*, *CM*, *DM*, and *IM*, and estimate the value of *AA*, *SU*, and *UNFM* to quantify the $ESIZE_{adapted,reused,or\ COTS}$ and the *SLI* of the development schedule factor.

Assessment and Assimilation (AA) assesses the degree of effort ("increment") necessary to determine whether a reused software module is appropriate for the application, and to integrate its description into the overall product description. Table 11.7 provides the Rating Scales and values for the assessment assimilation increment. "AA" and "AA increment" are used interchangeably in this report.

Table 11.7 Rating Scales for Assessment and Assimilation Increment (AA)

AA Increment	*Level of AA Effort*
0	None
2	Basic module search and documentation
4	Some module Test and Evaluation (T&E), documentation
6	Considerable module T&E, documentation
8	Extensive module T&E, documentation

The Software Understanding increment (SU) is obtained from Table 11.8. If the software is rated very high on structure, applications clarity, and self-descriptiveness, the software understanding and interface-checking penalty is 10%. If the software is rated very low on these factors, the penalty is 50%. SU is determined by taking the subjective average of the three categories.

Table 11.8 Rating Scales for Software Understanding Increment (SU)

	Very Low	*Low*	*Nominal*	*High*	*Very High*
Structure	Very low cohesion, high coupling, spaghetti code.	Moderately low cohesion, high coupling.	Reasonably well-structured; some weak areas.	High cohesion, low coupling.	Strong modularity, information hiding in data/control structures.
Application clarity	No match between program and application worldviews.	Some correlation between program and application.	Moderate correlation between program and application.	Good correlation between program and application.	Clear match between program and application worldviews.
Self-descriptiveness	Obscure code; documentation missing, obscure or obsolete.	Some code commentary and headers; some useful documentation.	Moderate level of code commentary, headers, documentation.	Good code commentary and headers; useful documentation; some weak areas.	Self-descriptive code; documentation up-to-date, well-organized, with design rationale.
SU Increment	50	40	30	20	10

UNFM is the indicator for the programmer's relative unfamiliarity with the software. If the programmer works with the software every day, the 0.0 multiplier for UNFM will add no software understanding effort increment. If the programmer has never seen the software before,

the 1.0 multiplier will add the full software understanding effort increment. The rating for UNFM is shown in Table 11.9.

Table 11.9 Rating Scales for Programmer Unfamiliarity (UNFM)

UNFM Increment	*Level of UNFM*
0.0	Completely familiar
0.2	Mostly familiar
0.4	Somewhat familiar
0.6	Considerably unfamiliar
0.8	Mostly unfamiliar
1.0	Completely unfamiliar

Table 11.10 summarizes the guidelines and constraints to estimate the parameters used in the sizing equations (Equation 11.15 to Equation 11.17).

Table 11.10 Guidelines and Constraints to Estimate Reuse Parameters

	Reuse Parameters					
	DM	***CM***	***IM***	***AA***	***SU***	***UNFM***
New code	N/A	N/A	N/A	N/A	N/A	N/A
Adapted code	0–100%	0–100%	0–100+% (can be > 100%)	0–8%	0–50%	0–1
Reused code	0%	0%	0–100%	0–8%	N/A	N/A
COTS	0%	0%	0–100%	0–8%	N/A	N/A

AAM uses the factors described above, Software Understanding (SU), Programmer Unfamiliarity (UNFM), and Assessment and Assimilation (AA) with the Adaptation Adjustment Factor (AAF), which is given by Equation 11.19.

In order to obtain the SLI of the Schedule Pressure factors (denoted by SLI_2), the UMD research team investigated the Yerkes-Dodson law [Yerkes, 1908]. This "law" states that the quality of performance on any task is an inverted U-shaped function of arousal, as shown on Figure 11.3. With increasing arousal, performance first improves, up to an optimal level, and then deteriorates when arousal is too high.

The range over which performance improves with increasing arousal varies with task

complexity, as shown on Figure 11.4 [Huey, 1993]. A simple task needs a higher amount of arousal than a more complex task to reach a maximal quality of performance.

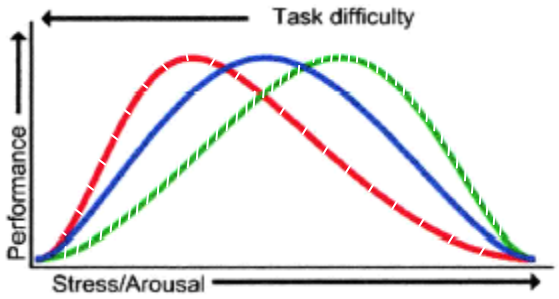


Figure 11.3 The Yerkes-Dodson Law with Three Levels of Task Difficulty

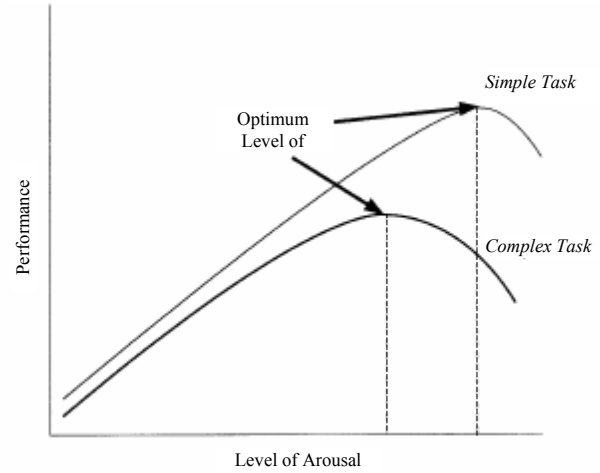


Figure 11.4 U-Function Relating Performance to Arousal

For a "nominal" task with medium level of difficulty, it is reasonable to postulate a symmetric bell-shaped function that relates SLI to SCED. Assume:

$$SLI_2 = k\ exp\left[-\left(\frac{1/SCED - \mu}{\sigma}\right)^2/2\right]$$

with conditions:

$$SLI_{2|SCED=200\%} = Maximum \Rightarrow \mu = 0.5$$
$$SLI_{2|SCED=200\%} = 1.0 \Rightarrow k = 1$$

and

$$SLI_{2|SCED=100\%} = 0.5 \Rightarrow \sigma = \frac{1}{\sqrt{8 \ln 2}}$$

Therefore, Equation 11.20:

$$SLI_2 = \exp\left[-\ln 16 \times \frac{1}{(SCED-0.5)^2}\right] \quad (11.20)$$

This equation gives results consistent with those given by [Gertman, 2005]. A follow on effort is required to validate this equation.

11.4.3.2 Experience Factor

11.4.3.2.1 Application Experience (APEX)

The rating scales for APEX are defined in terms of the project team's level of experience with this type of application [Boehm, 1982] [Boehm, 2000]. See Table 11.11 for APEX ratings.

Table 11.11 Rating Scales for APEX

APEX Descriptors	2 months	6 months	1 year	3 years	6 years
Rating Levels	Very Low	Low	Nominal	High	Very high
Rating Value	1	2	3	4	5

11.4.3.2.2 Platform Experience (PLEX)

The rating scales for PLEX are defined in terms of the project team's equivalent level of experience with the development platforms, including Graphical User Interface (GUI), database, Operating System, hardware platform, networking platform, etc. [Boehm, 1982] [Boehm, 2000]. See Table 11.12 for PLEX ratings.

Table 11.12 Rating Scales for PLEX

PLEX Descriptors	2 months	6 months	1 year	3 years	6 years
Rating Levels	Very Low	Low	Nominal	High	Very high
Rating Value	1	2	3	4	5

11.4.3.2.3 Language and Tool Experience (LTEX)

LTEX is a measure of the level of programming language and software tool experience of the

project team developing the software system or subsystem [Boehm, 1982] [Boehm, 2000]. See Table 11.13 for LTEX ratings.

Table 11.13 Rating Scales for LTEX

LTEX Descriptors	2 months	6 months	1 year	3 years	≥ 6 years
Rating Levels	Very Low	Low	Nominal	High	Very high
Rating Value	1	2	3	4	5

Refer to Table 11.14 to estimate the SLI value for the Experience factor (denoted by SLI_3)

Table 11.14 Experience SLI Estimation

Sum of Rating Values of APEX, PLEX and LTEX	3, 4	5, 6	7, 8	9, 10	11, 12	13, 14	15
Rating Levels	Extra Low	Very Low	Low	Nominal	High	Very high	Extra High
SLI Value	0.0	0.17	0.34	0.50	0.67	0.84	1.0

11.4.3.3 Measurement for Capability Factor

11.4.3.3.1 Analyst Capability (ACAP)

Analysts are personnel who work on requirements, high-level design, and detailed design. The rating scales for ACAP are expressed in terms of percentiles with respect to the overall population of analysts [Boehm, 1982] [Boehm, 2000]. The major attributes that should be considered in this rating are:

1. Analysis and design ability
2. Efficiency and thoroughness
3. Ability to communicate and cooperate

Note:

- These attributes should be approximately equally weighted in the evaluation.
- The evaluation should not consider the level of experience of the analysts; experience effects are covered by other factors.
- The evaluation should be based on the capability of the analysts as a team rather than as individuals.

See Table 11.15 for ACAP ratings.

Table 11.15 Rating Scales for ACAP

ACAP Descriptors	15th percentile	35th percentile	55th percentile	75th percentile	90th percentile
Rating Levels	Very Low	Low	Nominal	High	Very high
Rating Value	1	2	3	4	5

11.4.3.3.2 Programmer Capability (PCAP)

The rating scales for PCAP are expressed in terms of percentiles with respect to the overall population of programmers. Unit testing is regarded as one of the tasks performed by the programmers. The major factors that should be considered in the rating are [Boehm, 1982] [Boehm, 2000]:

1. Programmer ability
2. Efficiency and thoroughness
3. Ability to communicate and cooperate

Note:

- These attributes should be approximately equally weighted in the evaluation.
- The evaluation should not consider the level of experience of the programmers; experience effects are covered by other factors.
- The evaluation should be based on the capability of the programmers as a team rather than as individuals.

See Table 11.16 for PCAP ratings.

Table 11.16 Rating Scales for PCAP

PCAP Descriptors	15th percentile	35th percentile	55th percentile	75th percentile	90th percentile
Rating Levels	Very Low	Low	Nominal	High	Very high
Rating Value	1	2	3	4	5

11.4.3.3.3 Tester Capability (TCAP)

The rating scales for TCAP are expressed in terms of percentiles with respect to the overall population of testers. Unit testing is regarded as one of the tasks performed by the programmers, not by the testers. The major factors that should be considered in the rating are [Boehm, 1982] [Boehm, 2000]:

1. Tester ability
2. Efficiency and thoroughness
3. Ability to communicate and cooperate

Note:

- These attributes should be approximately equally weighted in the evaluation.
- The evaluation should not consider the level of experience of the testers; experience effects are covered by other factors.
- The evaluation should be based on the capability of the testers as a team rather than as individuals.

See Table 11.17 for TCAP ratings.

Table 11.17 Rating Scales for TCAP

TCAP Descriptors	15th percentile	35th percentile	55th percentile	75th percentile	90th percentile
Rating Levels	Very Low	Low	Nominal	High	Very high
Rating Value	1	2	3	4	5

11.4.3.3.4 Personnel Continuity (PCON)

The rating scales for PCON measures the project's annual personnel turnover [Boehm, 1982] [Boehm, 2000]. See Table 11.18 for PCON ratings.

Table 11.18 Rating Scales for PCON

PCON Descriptors	48% per year	24% per year	12% per year	6% per year	3% per year
Rating Levels	**Very Low**	**Low**	**Nominal**	**High**	**Very high**
Rating Value	1	2	3	4	5

Refer to Table 11.19 or Table 11.20 to estimate the SLI value of CAPABILITY Factor (denoted by SLI_4) for either capability excluded from the rating or capability included in the rating, respectively.

Table 11.19 Estimating SLI Value of Capability (Tester Capability Excluded)

Sum of SLI Values of ACAP, PCAP, and PCON	3, 4	5, 6	7, 8	9, 10	11, 12	13, 14	15
Rating Levels	**Extra Low**	**Very Low**	**Low**	**Nominal**	**High**	**Very high**	**Extra High**
SLI Value	0	0.17	0.24	0.50	0.67	0.84	1

Table 11.20 Estimating SLI Value of Capability (Tester Capability Included)

Sum of SLI values of ACAP, PCAP, PCON and TCAP	4, 5	6, 7	8–10	11–13	14–16	17–19	20
Rating Levels	**Extra Low**	**Very Low**	**Low**	**Nominal**	**High**	**Very high**	**Extra High**
SLI Value	0	0.17	0.24	0.50	0.67	0.84	1

11.4.3.4 Measurement For Development Tools Factor

The major factors that should be considered in this rating are [Boehm, 1982] [Boehm, 2000]:

1. Capability of the tools employed within the life cycle of a project.
2. Maturity of the tools
3. Integration of the tools

Refer to Table 11.21 for TOOL ratings and SLI estimation (denoted by SLI_5).

Table 11.21 Rating Scales for TOOL Factor

Tool Descriptors	Minimal tools for document editing, coding, compiling, and debugging	Simple life-cycle tools, little integration	Basic life-cycle tools, moderately integrated	Strong, mature life-cycle tools, moderately integrated	Strong, mature, proactive life-cycle tools, well integrated with processes, methods, and reuse
Rating Levels	Very Low	Low	Nominal	High	Very high
SLI Value	0	0.25	0.5	0.75	1

11.4.3.5 Measurement for Development Site Factor (SITE)

Determining the rating of the SITE factor involves the assessment and combination of two factors: site collocation and communication support. When making the subjective average of these two components of SITE, 70% and 30% weights are recommended for site collocation and communication support, respectively, as shown in Table 11.22 and Table 11.23 [Boehm, 1982] [Boehm, 2000].

Table 11.22 Rating Scales for Site Collocation

Site Collocation Descriptors	Inter-national	Multi-city and Multi-company	Multi-city or Multi-company	Same city or metro area	Same building or complex	Fully cooperative
Rating Levels	Very Low	Low	Nominal	High	Very High	Extra High
Rating Value	0	1	2	3	4	5

Table 11.23 Rating Scales for Communication Support

Site Communi-cation Descriptors	Some phone, mail	Individ. phone, FAX	Narrow-band e-mail	Wideband e-comm.	Wideband e-comm. occas. Video conference	Interactive multi-media
Rating Levels	Very Low	Low	Nominal	High	Very High	Extra High
Rating Value	0	1	2	3	4	5

Refer to Table 11.24 for SITE ratings and SLI estimation (denoted by SLI_6), and Table 11.25 for determining the weighted sum by the rating values of collocation and communication.

Table 11.24 SITE Ratings and SLI Estimation

Weighted Sum of SLI values of Site Collocation and Site Comm.	0.0–0.9	1.0–1.5	1.6–2.9	3.0–3.8	4.0–4.4	4.7–5.0
Rating Levels	**Very Low**	**Low**	**Nominal**	**High**	**Very high**	**Extra High**
SLI Value	0.0	0.25	0.50	0.67	0.84	1.0

Table 11.25 Determining the Weighted Sum by the Rating of Collocation and Communication

Communication Rating → Collocation Rating ↓	0	1	2	3	4	5
0	0	0.3	0.6	0.9	1.2	1.5
1	0.7	1	1.3	1.6	1.9	2.2
2	1.4	1.7	2	2.3	2.6	2.9
3	2.1	2.4	2.7	3	3.3	3.6
4	2.8	3.1	3.4	3.7	4	4.3
5	3.5	3.8	4.1	4.4	4.7	5

11.4.3.6 Measurement for Team Cohesion Factor (TEAM)

TEAM accounts for the sources of project turbulence and extra effort caused by difficulties in synchronizing the project's stakeholders: users, customers, developers, maintainers, and others. See Table 11.26 for TEAM ratings and SLI estimation (denoted by SLI_7) and Table 11.27 for the components comprising TEAM ratings. [Boehm, 1982] [Boehm, 2000]

Table 11.26 Rating Scales for TEAM

TEAM Descriptors	Very difficult interactions	Some difficult interactions	Basically cooperative interactions	Largely cooperative	Highly cooperative	Seamless interactions
Rating Levels	**Very Low**	**Low**	**Nominal**	**High**	**Very High**	**Extra High**
SLI Value	0	0.25	0.5	0.67	0.84	1

Table 11.27 TEAM Rating Components

Characteristic	Very Low	Low	Nominal	High	Very High	Extra High
Consistency of stakeholder objectives and cultures	Little	Some	Basic	Considerable	Strong	Full
Ability, willingness of stakeholders to accommodate other stakeholders' objectives	Little	Some	Basic	Considerable	Strong	Full
Experience of stakeholders in operating as a team	None	Little	Some	Basic	Considerable	Extensive
Stakeholder team building to achieve shared vision and commitments	None	Little	Some	Basic	Considerable	Extensive

11.4.3.7 Measurement for Management Style Factor (STYLE)

This factor captures the impact of management style on the quality of a project. Refer to Table 11.28 for STYLE ratings and SLI estimation (denoted by SLI_8).

Table 11.28 Rating Scales for STYLE

Style Descriptors	Highly Intrusive	Moderately Intrusive	Neither Intrusive nor Supportive	Moderately Supportive	Highly Supportive
Rating Levels	Very Low	Low	Nominal	High	Extra High
SLI Value	0	0.25	0.5	0.75	1

11.4.3.8 Measurement for Process Maturity Factor (PMAT)

PMAT captures the capability level of an organization based on the software Engineering Institute's Capability Maturity Model (CMM) (Refer to Chapter 8 for CMM measurement). Refer to Table 11.29 for PMAT SLI Estimation (denoted by SLI_7) [Boehm, 1982] [Boehm, 2000] .

Table 11.29 Rating Scales and SLI Estimation for PMAT

PMAT Descriptors	CMM level 1 (lower half)	CMM level 1 (upper half)	CMM level 2	CMM level 3	CMM level 4	CMM level 5
Rating Levels	Very Low	Low	Nominal	High	Very High	Extra High
SLI Value	0	0.25	0.5	0.67	0.84	1

11.4.3.9 Measurement for Requirements Evolution Factor (REVL)

Different from the definition given by COCOMO II [Boehm, 2000], REVL here is defined in terms of the percentage of code change due to the evolution of requirements since the initial SRS baseline. Refer to Chapter 15 for details.

See Table 11.30 for REVL ratings and SLI estimation (denoted by SLI_{10}).

Table 11.30 Rating Scales and SLI Estimation for REVL

REVL Descriptors	5% code change	20% code change	35% code change	50% code change	65% code change	80% code change
Rating Levels	Very Low	Low	Nominal	High	Very High	Extra High
SLI Value	1	0.75	0.5	0.34	0.16	0

11.4.3.10 Measurement results for the support measures

Table 11.31 summarizes the measurement results for all Performance Influencing Factors.

Table 11.31 PIF Measurement Results for the APP System

Influence Factors	Primitives	Values of Primitives for				
		CP	µp1 System	µp1 Application	µp2 System	µp2 Application
EXPERIENCE	APEX	5	5	5	5	5
	PLEX	3	3	3	3	3
	LTEX	3	3	3	3	3
CAPABILITY	ACAP	4	4	4	4	4
	PCAP	3	3	3	3	3
	PCON	5	5	5	5	5
	TCAP	3	3	3	3	3
SCED	$TDEV_{actual}$, in calendar months	25	25	13	25	19
	$SIZE_{delivered}$, in KLOC	1.21	2.034	0.48	0.895	0.206
	$SIZE_{discarded}$, in KLOC	0.150	0.270	0.045	0.180	0.190
Use of Methods/ Notation/TOOL	TOOL	3	3	3	3	3

Table 11.31 PIF Measurement Results for the APP System (continued)

Influence Factors	Primitives	Values of Primitives for				
		CP	µp1 System	µp1 Application	µp2 System	µp2 Application
SITE	COLLO-CATION	4	4	4	4	4
	COMMU-NICATION	1	1	1	1	1
Team Relationships	TEAM	3	3	3	4	4
Management Style	STYLE	4	4	4	4	4
PMAT	CMM	2	2	2	2	2
Requirement Volatility	REVL	10.6%	3.8%	3.0%	9.1%	3.9%

The data for APEX, PLEX, LTEX, ACAP, PCAP, PCON, TCAP, $TDEV_{actual}$, TOOL, COLLOCATION, COMMUNICATION, TEAM, STYLE, and CMM were extracted from responses to a questionnaire distributed to the APP system manufacturer. Refer to Chapter 15 for details of obtaining the data for REVL.

The data for $SIZE_{discarded}$ was obtained by the following procedure.

1. Identify the discarded code segment/module documented in [APP, Y1], [APP, Y2], [APP, Y3], [APP, Y4], and [APP, Y5].
2. Count the size of the discarded code by using the code size measurement rules defined in Chapter 6.

Table 11.32 summarizes the SLIs for the APP system calculated by applying the measurement rules of the PIFs to the data in Table 11.31.

Table 11.32 Summary of SLI Calculations

	CP System	µp1 System	µp1 Application	µp2 System	µp2 Application
Cyclomatic Complexity	0.8369	0.8435	0.6400	0.8239	0.8324
SECD	0.7857	0.8347	0.8395	0.8057	0.7768
EXPERIENCE	0.67	0.50	0.50	0.7692	0.7314
CAPABILITY	0.84	0.67	0.67	0.84	0.84

Table 11.32 Summary of SLI Calculations (continued)

	CP System	µp1 System	µp1 Application	µp2 System	µp2 Application
TOOL	0.50	0.50	0.50	0.50	0.50
SITE	0.50	0.50	0.50	0.50	0.50
TEAM	0.67	0.67	0.67	0.84	0.84
STYLE	0.75	0.75	0.75	0.75	0.75
PMAT	0.5	0.5	0.5	0.5	0.5
REVL	0.9067	1.00	1.00	0.9317	1.00

The SLI of the entire software product is given by the weighted sum of all PIF SLIs:

$$SLI = \sum_{i=1}^{10} W_i SLI_i \tag{11.21}$$

where

W_i weight of the *i*-th influence factor. Table 11.33 provides the values of weights used for *SLI* calculation [Stutzke, 2001].

SLI_i the *SLI* value of the i-th influence factor.

Table 11.33 Values of Weights Used for SLI Calculation

	W_1	W_2	W_3	W_4	W_5	W_6	W_7	W_8	W_9	W_{10}
Value	0.204	0.037	0.148	0.093	0.056	0.167	0.019	0.037	0.074	0.167

11.4.3.11 RePS with supportive measures

Equation 11.22 will be used to estimate the fault content.

$$N = 0.036 \times SIZE \times (20)^{1-2\times SLI} \tag{11.22}$$

Table 11.34 summarizes the SLI values and the fault content of the delivered source codes with and without using the support measures respectively.

The estimated number of faults in the APP using the support measures in addition to CC is:

$$N_{CC^+} = 11.8 + 22.7 + 6.9 + 7.8 + 1.7 = 50.9 \quad (11.23)$$

Thus, the APP reliability prediction (using support measures in addition to CC) is given by:

$$p_s(CC^+) = \exp\left(-4.2 \times 10^{-7} \times 50.9\ defects \times \frac{0.129\ s/demand}{0.009\ s}\right) = 0.9996936\ defect/demand \quad (11.24)$$

The above results show that the estimated number of defects using the support measures (i.e., 50.9 from Equation 11.23) is larger than the estimated number of defects obtained using only CC (i.e., 28.3 defects from Equation 11.8). Consequently, the reliability will be less using the support measures (i.e. 0.9996936 defect/demand from Equation 11.24) than using only *CC* (i.e., 0.9998296 defect/demand from Equation 11.12). As shown in Table 11.34, the SLI values for many of the influencing factors are lower than the SLI values for CC. This means the APPs performance on these factors was low and consequently the number of defects estimated using all the factors should be higher.

It should be noted that the use of supportive measures in this chapter is for illustration only. The purpose of this exercise is to show how supportive measures could be used to improve a reliability prediction based on CC. The results analysis in Chapter 19 uses the reduced CC RePS (i.e., without supportive measures).

Table 11.34 Summary of Fault Content Calculation

		CP System	**µp1 System**	**µp1 Application**	**µp2 System**	**µp2 Application**
SIZE, in LOC		1210	2034	480	895	206
Without using the support measures	$SLI = SLI_1$	0.8369	0.8435	0.64	0.8239	0.8324
	The number of defects in the source code.	5.8	9.4	7.5	4.6	1
Using the support measures	$SLI = \sum_{i=1}^{10} W_i SLI_i$	0.7175	0.6952	0.6539	0.7377	0.7441
	The number of defects in the source code.	11.8	22.7	6.9	7.8	1.7

11.5 Lessons Learned

The measurement of CC can be supported by automation tools. The RePS based on CC is straightforward once the average execution-time-per-demand and the linear execution-time are quantified. Thus, CC is a convenient measure for software-reliability prediction. However, there are two issues with this measure. First, as is the case for BLOC, the measurement of CC also requires the concept of software "module" while there is no clear definition of "module" provided in the current literature. Second, the CC RePS uses empirical industry-data to link the CC value with the number of defects. Thus, reliability prediction from CC is not as good as the predicted reliability obtained from other measures that deal with real defects of the application.

11.6 References

[APP, Y1] "APP Module SF1 System Software code," Year Y1.

[APP, Y2] "APP SF1 Flux/Delta Flux/Flow Application code," Year Y2.

[APP, Y3] "APP Module μp2 System Software Source Code Listing," Year Y3.

[APP, Y4] "APP μp2 Flux/Delta Flux/Flow Application Software Source Code Listing," Year Y4.

[APP, Y5] "APP Comm. Processor Source Code," Year Y5.

[Basili, 1984] V.R. Basili and B.T. Perricone. "Software Errors and Complexity: An Empirical Investigation," *Communications of the ACM*, 1984.

[Boehm, 1982] B. Boehm. *Software Engineering Economics*. Prentice Hall, Inc., 1982.

[Boehm, 2000] B. Boehm et al. *Software Cost Estimation With COCOMO II*. Prentice-Hall, Inc., 2000.

[Chapman, 2002] R.M. Chapman and D. Solomon. "Software Metrics as Error Predictors," NASA, 2002. Available: http://sarpresults.ivv.nasa.gov/ViewResearch/289/23.jsp

[Dougherty, 1988] E.M. Dougherty and J.R. Fragola. *Human Reliability Analysis: A System Engineering Approach with Nuclear Power Plant Applications*. John Wiley & Sons, 1988.

[Embrey, 1983] D.E. Embrey. "The Use of Performance Shaping Factors and Quantified Expert Judgment in the Evaluation of Human Reliability: An Initial Appraisal," US NRC, NUREG/CR-2986, 1983.

[Fenton, 1999] N.E. Fenton and M. Neil. "A Critique of Software Defect Prediction Models," *IEEE Transactions on Software Engineering*, vol. 25, pp. 675–689, 1999.

[Gertman, 2005] D.I . Gertman et al. "The SPAR-H Human Reliability Analysis Method," US NRC, NUREG/CR-6883, 2005.

[Gill, 1997] G.K. Gill and C.F. Kemerer. "Cyclomatic Complexity Density and Software Maintenance Productivity," *IEEE Transactions on Software Engineering*, vol. 17, pp. 1284–1288, 1991.

[Hansen, 1978] W. J. Hansen. "Measurement of Program Complexity by the Pair (Cyclomatic Number, Operator Count)," ACM SIGPLAN Notices, vol. 13, no. 3, pp. 29–33, 1978.

[IEEE 982.2, 1988] "IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software," IEEE Std. 982.2-1988, 1988.

[Jones, 1991] C. Jones. *Applied Software Measurement: Assuring Productivity and Quality*. New York, NY: McGraw-Hill, 1991.

[Jones, 1996] C. Jones. *Software Systems Failure and Success*. International Thomson Computer Press, Inc., 1996.

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

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

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

[McCabe, 1989] T.J. McCabe and C.W. Butler. "Design Complexity Measurement and Testing," *Communications of the ACM*, vol. 32, pp. 1415–1425, 1989.

[McCabe, 1994] T.J. McCabe and A.H. Watson. "Software Complexity." Crosstalk, *Journal of Defense Software Engineering*, vol. 7, pp. 5–9, 1994.

[MST, 2005] M Squared Technology, RSM (Resource Standard Metrics) Version 6.80, 2005. Available: http://msquaredtechnologies.com/m2rsm/index.htm

[Myers, 1977] G.J. Myers. "An Extension to the Cyclomatic Measure of Program Complexity," *SIGPLAN Notices*, vol. 12, no. 10, pp. 61–64, 1977.

[Pham, 2000] X. Zhang and H. Pham. "An analysis of Factors Affecting Software Reliability," *The Journal of Systems and Software*, vol. 50, pp. 43–56, 2000.

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

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

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

[Takahashi, 1997] R. Takahashi. "Software Quality Classification Model Based on McCabe's Complexity Measure," *Journal of Systems and Software*, vol. 38, pp. 61–69, 1997.

[Walsh, 1979] T. Walsh. "A Software Reliability Study Using a Complexity Measure," in *Proc. AFIPS Conference*, 1979.

[Watson, 1996] A.H. Watson and T.J. McCabe. "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric," NIST Special Publication 500-235, 1996. Available: http://www.mccabe.com/pdf/nist235r.pdf

[Zuse, 1990] H. Zuse. "Software Complexity: Measures and Methods," Hawthorne, NJ: Walter de Gruyter Co., 1990.

12. DEFECT DENSITY

The Defect Density measure indicates whether the inspection process is effective. If the defect density is outside the norm after several inspections, it is an indication that the inspection process requires further scrutiny.

This measure can only be applied after the requirements, design, and source code inspections are completed. As listed in Table 3.3, the applicable life cycle phases for the DD measure are Testing and Operation.

12.1 Definition

Defect density is defined in this study as the ratio of defects remaining to the number of lines of code in the software. This definition is consistent with the "Code Defect Density", which is defined in [IEEE 982.2, 1988] and [Smidts, 2000]. The defects are discovered by independent inspection. The inspection process is discussed below.

To calculate defect density, severity levels for defect designation[27] are established first. In this particular case, all defects discussed below belong to the level 1 category.[28]

Defect Density, *DD*, is given as:

$$DD = \left(\frac{1}{KLOC}\right)\left[\sum_{i=1}^{3}\sum_{j=1}^{N} D_{i,j} - \sum_{l=1}^{3}\sum_{k>l}^{3} DF_{l,k} - \sum_{m=2}^{N} DU_m(m-1)\right] \quad (12.1)$$

where

i An index reflecting the development stage. A value of 1 represents the requirements stage, a value of 2 represents the design stage and a value of 3 represents the coding stage.

j The index identifying the specific inspector. This index ranges from 1 to *N*.

$D_{i,j}$ The number of unique defects detected by the *j*-th inspector during the *i*-th development stage in the current version of the software.

$DF_{l,k}$ The number of defects found in the l-th stage and fixed in the *k*-th stage, $1 \leq l < k \leq 3$.

[27] Refer to Chapter 6 for a definition of severity levels.
[28] No severity level 2 defects were found.

DU_m The number of defects found by exactly m inspectors and remaining in the code stage. The value of m ranges from 2 to N.

N Total number of inspectors.

$KLOC$ The number of source lines of code (LOC) in thousands. The LOC counting rule is defined in Chapter 6.

The numerator in Equation 12.1 is the number of defects discovered by the inspection but remaining unresolved in the APP. The first term of the numerator is the total number of defects found by all inspectors and from all life cycle phases (requirements, design, code, and testing). Among these defects, some are fixed in the succeeding life cycles (for instance, a defect is found in the requirements phase but later fixed in the testing phase); some are found by multiple inspectors simultaneously (for instance, Inspector I found defect A and Inspector II found defect A, too). The second term in the numerator represents the former case (defects fixed in a later stage), and the third term represents the latter situation, i.e., a duplicate count for one defect.

12.2 Measurement

The IEEE standard [IEEE 982.2, 1988] specified that Defect Density can be measured using software inspection. It did not specify, however, which software inspection procedure should be conducted. In this study, the authors utilized the Fagan [Fagan, 1976] approach to conduct the software inspection. Fagan's method was further developed by Robert Ebenau and described in [Strauss, 1993].

The inspection conducted in this study is not in the development process. As such, the inspection stages described in [Strauss, 1993] were tailored in this study. Only the planning, preparation, and meeting stage from [Strauss, 1993] were considered. The inspectors (or checkers); the documents under inspection; the documents required (also called source document, for example, the user requirements, the system requirements or other background knowledge); and the rules or checklists were identified in the planning stage. The individual checking activities were performed in the preparation stage. The findings were then summarized in the meeting stage. No process improvement activities are required in the inspection process.

The checklists used for the requirements, design, and code inspection are presented in [Strauss, 1993]. The requirements, design, and code inspection are formalized in the following sub-sections.

12.2.1 Requirements Inspection

Products Under Inspection

1. APP module first safety function processor SRS [APP, Y3]
2. APP Flux/Delta Flow Application SRS SF1 [APP, Y6]
3. TAR module µp2 system software SRS [APP, Y9]
4. APP µp2 Flux/Delta Flux/Flow application software SRS [APP, Y12]
5. APP module communication processor SRS [APP, Y15]

Source Documents

1. APP instruction manual [APP, Y1]
2. APP module - design specification [APP, Y2]

Participants:

1. Two Inspectors
2. One Moderator

The inspectors inspected the products independently and recorded all ambiguous, incorrect, or incomplete statements and locations. The moderator reviewed the logs and corrected mistakes made during the inspection process[29]. The values of $D_{1,j}$ were obtained during this stage.

12.2.2 Design Inspection

Products Under Inspection

1. APP module first safety function processor SDD [APP, Y4]
2. APP Flux/Delta Flux/Flow Application SDD for µp1 [APP, Y7]
3. APP µp2 SDD for system software [APP, Y10]
4. APP µp2 Flux/Delta Flux/Flow application software SDD [APP, Y13]
5. APP communication processor SDD [APP, Y16]

Source Documents:

1. APP instruction manual [APP, Y1]
2. APP module - design specification [APP, Y2]
3. APP module first safety function processor SRS [APP, Y3]
4. APP Flux/Delta Flow Application SRS for SF1 [APP, Y6]
5. APP module µp2 system software SRS [APP, Y9]

[29]By "mistake" refers to cases where a defect found by inspection was determined not to be a defect per se.

6. APP μp2 Flux/Delta Flux/Flow application software SRS [APP, Y12]
7. APP module communication processor SRS [APP, Y15]
8. The list of defects generated in the requirements inspection cycle.

Participants:

1. Two Inspectors
2. One Moderator

The inspectors inspected the products independently and recorded defects (for example, any ambiguity, incorrectness, inconsistency, or incompleteness).

The moderator reviewed all defects discovered in the design stage, and corrected the mistakes made during the inspection.

The inspectors identified the defects found by the requirements inspection and fixed in the design stage ($DF_{1,2}$) as well as the defects that originated during the design process ($D_{2,j}$).

12.2.3 Source Code Inspection

Products Under Inspection

1. APP module SF1 system software code [APP, Y5]
2. APP SF1 Flux/Delta Flux/Flow application code [APP, Y8]
3. APP μp2 system software source code listing [APP, Y11]
4. APP μp2 Flux/Delta Flux/Flow application software source code listing [APP, Y14]
5. APP communication processor source code [APP, Y17]

Source Documents:

1. APP instruction manual [APP, Y1]
2. APP module-design specification [APP, Y2]
3. APP module first safety function processor SRS [APP, Y3]
4. APP Flux/Delta Flow Application SRS for SF1 [APP, Y6]
5. APP module μp2 system software SRS [APP, Y9]
6. APP μp2 Flux/Delta Flux/Flow application software SRS [APP, Y12]
7. APP module communication processor SRS [APP, Y15]
8. APP module first safety function processor SDD [APP, Y4]
9. APP Flux/Delta Flux/Flow Application SDD for SF1 [APP, Y7]
10. APP μp2 SDD for system software [APP, Y10]
11. APP μp2 Flux/Delta Flux/Flow application software SDD [APP, Y13]

12. APP communication processor SDD [APP, Y16]
13. The list of defects generated in the requirements inspection cycle.
14. The list of defects generated in the design inspection cycle.

Participants:

1. Two Inspectors
2. One Moderator

The inspectors inspected the source code independently and recorded defects with an emphasis on the following types of defects: data reference, data declaration, computation, comparison, control flow, interface, input/output, and missing code.

The moderator reviewed all defects discovered in the code stage, and corrected mistakes made during the inspection.

The inspectors identified the number of defects found by the requirements inspection that were fixed in the code ($DF_{1,3}$), the number of defects found by the design inspection that were in the code ($DF_{2,3}$), and the number of defects that originated in the code $D_{3,j}$.

12.2.4 Lines of Code Count

The number of source lines of code was counted by one of the inspectors using the counting rules defined in Chapter 6.

12.3 Results

The values of the different primitives required to evaluate defect density are shown in Table 12.1 through Table 12.4. Only Level 1 and 2 defects were considered.

Table 12.1 Values of the Primitives $D_{i,j}$

$D_{i,j}$		**Development Stage** (j)		
		Requirements	Design	Code
Inspector (i)	1	0	0	0
	2	2	4	0

Table 12.2 Values of the Primitives $DF_{l,k}$

$DF_{l,k}$		Development Stage During which Defects Were Fixed		
		Requirements	Design	Code
Development Stage During which Defects were Introduced	Requirements	0	0	0
	Design	N/A	0	2
	Code	N/A	N/A	0

Table 12.3 Values of the Primitives DU_m

m	DU_m
2	0

Based on these results, the value of the numerator is obtained in Equation 12.1 (where $n = 2$):

$$DD = \left(\frac{1}{KLOC}\right)\left[\sum_{i=1}^{3}\sum_{j=1}^{2} D_{i,j} - \sum_{l=1}^{3}\sum_{k>l}^{3} DF_{l,k} - \sum_{m=2}^{2} DU_m(m-1)\right] = \frac{4}{KLOC} \qquad (12.2)$$

Table 12.4 lists the number of lines of code.

Table 12.4 Primitive LOC

LOC	4825

Therefore

$$DD = \frac{4 \text{ defects}}{4825 \text{ LOC}} = \frac{0.829 \text{ defects}}{\text{KLOC}}$$

Table 12.5 gives a detailed description of the unresolved defects found during inspection.

Table 12.5 Unresolved Defects Leading to Level 1 Failures Found during Inspection

Defect Number	Location	Defect Description	Severity Level	SRS	SDD	Source Code
1	Page 62 of µp1 System SRS, address line check	The check algorithm cannot detect coupling failure but only stuck at high or low failures.	Level 1	Originated	Remains	Remains
2	Page 41 of µp2 System SDD, main_program trip calculation logic	If trip condition is calculated, the logic will force another calculation. The final decision then completely depends on the result of this round of calculation. This logic is problematic in case of this scenario: a real trip first, then a false non-trip. Although it is less likely but possible.	Level 1	Not specified	Originated	Remains
3	Page 45 of µp2 System SDD, address_line_test function	Z180 has 16 bits of address line but only the least 13 bits are examined. The most significant three are not considered. In case those three are in a bad situation, the test is not able to reveal it.	Level 1	Not clearly specified	Originated	Remains
4	Page 38 of CP SRS, address line check	The check algorithm cannot detect coupling failure but only stuck at high or low failures.	Level 1	Originated	Remains	Remains

12.4 RePS Construction and Reliability Estimation

Chapter 5 explained in greater detail how to utilize Extended Finite State Machine (EFSM) models to propagate defects against an operational profile. Such EFSM models and the operational profile constitute the RePS for Defect Density.

12.4.1 Result

The defect-density-based failure-probability prediction was obtained through execution of the EFSM model. Detailed EFSM construction procedures are provided in Appendix A. The estimation of APP probability of failure-per-demand based on the defect density RePS is 2.31×10^{-10}. Hence $p_s = 1 - 2.31 \times 10^{-10} = 0.9999999997688$.

12.5 Lessons Learned

The measurement of DD is a labor-intensive process. The use of a well-defined checklist can facilitate the process. However, a large number of items in the checklist must be verified for a single segment of requirement or design specification or source-code module. Some of the items are high level and cannot be verified systematically nor answered objectively. For instance, the checklist does not provide a clear definition of "complete," "correct," and "unambiguous" for an item such as: "Are the requirements complete, correct, and unambiguous?" Thus, the larger the application, the more difficult a complete measurement of DD becomes.

12.6 References

[APP, Y1] *APP Instruction Manual*.

[APP, Y2] "APP Module-Design Specification," Year Y2.

[APP, Y3] "APP Module First SFP SRS," Year Y3.

[APP, Y4] "APP Module First SFP SDD," Year Y4.

[APP, Y5] "APP Module SF1 System Software code," Year Y5.

[APP, Y6] "APP Flux/Delta Flux/Flow Application SRS for SF1," Year Y6.

[APP, Y7] "APP Flux/Delta Flux/Flow Application SDD for SF1," Year Y7.

[APP, Y8] "APP SF1 Flux/Delta Flux/Flow Application Code," Year Y8.

[APP, Y9] "APP Module μp2 System Software SRS," Year Y9.

[APP, Y10] "APP Module μp2 SDD for System Software," Year Y10.

[APP, Y11] "APP Module μp2 System Software Source Code Listing," Year Y11.

[APP, Y12] "APP μp2 Flux/Delta Flux/Flow Application Software SRS," Year Y12.

[APP, Y13] "APP μp2 Flux/Delta Flux/Flow Application Software DD," Year Y13.

[APP, Y14] "APP μp2 Flux/Delta Flux/Flow Application Software Source Code Listing," Year Y14.

[APP, Y15] "APP Module Communication Processor SRS," Year Y15.

[APP, Y16] "APP Module Communication Processor SDD," Year Y16.

[APP, Y17] "APP Communication Processor Source Code," Year Y17.

[Fagan, 1976] M.E. Fagan. "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal*, vol. 15, pp. 182–211, 1976.

[IEEE 982.2, 1988] "IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software," IEEE Std. 982.2-1988, 1988.

[Smidts, 2004] C. Smidts and M. Li, "Validation of a Methodology for Assessing Software Quality," NRC, Office of Nuclear Regulatory Research, Washington DC NUREG/CR-6848, 2004.

[Strauss, 1993] S.H. Strauss and R.G. Ebenau. "Software Inspection Process," New York: McGraw-Hill, Inc., 1993.

[Voas, 1992] J.M. Voas. "PIE: A Dynamic Failure-Based Technique," *IEEE Transactions on Software Engineering*, vol. 18, pp. 717–27, 1992.

13. FAULT-DAYS NUMBER

The fault-days number (FDN) measure represents the number of days that faults remain in the software system from introduction to removal.

It should be noted that this measure is more suitable for assessing a development process than for assessing a product.

The effectiveness of the software design and development process depends upon the timely removal of faults across the entire life cycle. This measure is an indicator of the quality of the software system design and of the development process. A high value may be indicative of delayed removal of faults and/or presence of many faults, due to an ineffective development process [Smidts, 2000].

This measure encourages timely inspections and testing and can also assist in the management of improving the design and development process [Smidts, 2000].

Although limited published research is available, this measure can be used in a software reliability program to monitor the quality of process and product development. Careful collection of primitive data is essential to the successful use of this measure [Smidts, 2000].

This measure can be applied as soon as the requirements are available. As listed in Table 3.3, the applicable life cycle phases for the FDN measure are Requirements, Design, Coding, Testing, and Operation.

13.1 Definition

The fault-day metric 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 [Smidts, 2000] [Herrmann, 2000], such that:

$$FD_i = f_{out_i} - f_{in_i} \tag{13.1}$$

and

$$FD = \sum_{i=1}^{I} FD_i \tag{13.2}$$

where

- FD Fault-days for the total system
- FD_i Fault-days for the *i*-th fault
- f_{in_i} Date at which the *i*-th fault was introduced into the system
- f_{out_i} Date at which the *i*-th fault was removed from the system
- I Total number of faults

It is difficult to determine the exact fault content introduced into a system during the life cycle phases. One way is to use the industry-average data to estimate the fault content based on the size of a system (in terms of function point), as described later.

The "waterfall model," sometimes called the "classic life cycle," is a model of the software development process in which the constituent activities, typically a concept phase, requirements phase, design phase, coding phase, integration and test phase, and installation and checkout phase, are performed in that order, possibly with overlap but with little or no iteration [IEEE 610.12, 1990].

For a software product whose development process follows a sequential development life cycle model (such as the waterfall model), the FDN measure is counted on a phase-by-phase basis.

Despite the criticism of its efficacy in all situations [Hanna, 1995], the waterfall model is suitable for use when [Pressman, 2004]:

1. The requirements of a problem are reasonably well understood
2. Work flows from communication through deployment in a reasonably linear fashion
3. Well-defined adaptations or enhancements to an existing system must be made

The definitions of the phases in the waterfall model are as follows (according to their typical sequence of occurrence in the model):

Requirements Phase: the period of time in the software life cycle during which the requirements for a software product are defined and documented [IEEE 610.12, 1990]. Requirements Review is part of this phase, in which a process or meeting during which the requirements for a system, hardware item, or software item are presented to project personnel, managers, users, customers, or other interested parties for comment or approval. Types of requirements reviews include system requirements review, and software requirements review [IEEE 610.12, 1990].

Design Phase: the period of time in the software life cycle during which the designs for architecture, software components, interfaces, and data are created, documented, and verified to satisfy requirements. Types of design phases include detailed design and preliminary design [IEEE 610.12, 1990]. The Design Review is a process or meeting during which a system, hardware, or software design is presented to project personnel, managers, users, customers, or other interested parties for comment or approval. Types of design reviews include critical design review, preliminary design review, system design review [IEEE 610.12, 1990].

Coding Phase: sometimes called the "implementation phase," the period of time in the software life cycle during which a software product is created from design documentation and debugged [IEEE 610.12, 1990]. Code Inspection is a process or meeting during which software code is

presented to project personnel, managers, users, customers, or other interested parties for comment or approval [IEEE 610.12, 1990].

Test Phase: the period of time in the software life cycle during which the components of a software product are evaluated and integrated, and the software product is evaluated to determine whether or not requirements have been satisfied [IEEE 610.12, 1990].

The following abbreviations for typical development phases appear in this chapter.

RQ	Requirements Phase
RR	Requirements Review
DE	Design Phase
DR	Design Review
CO	Coding (or Implementation) Phase
CI	Code Inspection
TE	Testing Phase

13.2 Measurement Rules

This section presents eight rules for counting the FDN of a system. In Section 13.3 we use the APP system to illustrate the application of these rules.

Rule 13.1: The FDN is counted on a workday basis.

Rule 13.2: The FDN for a system is the sum of the FDN of all faults, including faults removed during the development life cycle, and faults remaining in the delivered source code.

Rule 13.3: The FDN of a fault is calculated according to Equation 13.1.

Rule 13.4: If the exact date at which the fault was introduced is unknown, it is assumed to have occurred during the middle of the corresponding phase [Smidts, 2000]; i.e.:

$$f_{in} = \{t_{begin}\}_{\phi} + \frac{\{t_{end}\}_{\phi} - \{t_{begin}\}_{\phi}}{2} \tag{13.3}$$

where

f_{in}	Date at which the fault was introduced into the system
$\{t_{end}\}_{\phi}$	Ending date of the phase ϕ in which the fault was introduced
$\{t_{begin}\}_{\phi}$	Beginning date of the phase ϕ in which the fault was introduced

Rule 13.5: If the exact date at which the fault was removed is unknown, it is assumed to have occurred during the middle of the corresponding phase [Smidts, 2000]; i.e.:

$$f_{out} = \{t_{begin}\}_{\phi} + \frac{\{t_{end}\}_{\phi} - \{t_{begin}\}_{\phi}}{2} \tag{13.4}$$

where

f_{out}	Date at which the fault was removed from the system
$\{t_{end}\}_{\phi}$	Ending date of the phase ϕ in which the fault was removed
$\{t_{begin}\}_{\phi}$	Beginning date of the phase ϕ in which the fault was removed

Three steps are required in order to apply Rule 13.4 or Rule 13.5:

1. Identify the *beginning date* and the *ending date* of each life cycle phase.

The *beginning date* of a phase is the date at which initial activities belonging to that phase are conducted. The *ending date* of a phase is the date of release of the first version of all deliveries belonging to that phase. These dates are usually recorded on development documents and/or quality assurance documents, such as the SRS and the Verification and Validation (V&V) Summary Report.

2. Construct the sequential development life cycle according to the occurrence sequence of all phase beginning dates.

3. Divide the documented faults into several categories according to their originating phase.

For example, most faults are usually introduced during *RQ*, *DE*, and *CO* phases. Therefore, the faults are divided into *Requirements Faults*, *Design Faults*, and *Coding Faults*.

Requirements faults originate in the *requirements phase* and can be detected in the *requirements review*, *design*, *design review*, *code*, *code review*, or *testing phase* of the software life cycle.

Design faults originate in the *design phase* and can be detected in the *design review*, *coding*, *code review*, or *testing phases* of the software life cycle.

Code faults originate in the *coding phase* and can be detected in the *code inspection* or *testing phases* of the software life-cycle.

The FDN of the requirements faults, design faults, and coding faults is thus counted phase-by-phase.

Rule 13.6: The removal date of a fault remaining in the delivered source code is estimated to be the ending date of the last phase of the software-development life-cycle.
Rule 13.7: The fault content of requirements faults, design faults, and code faults, respectively, are estimated using the industry average data.

According to [Stutzke, 2001], the expected fault content function is determined by solving the following differential equation:

$$\frac{d\{\mu_U(t)\}_{j,\phi}}{dt} = \{v(t)\mu_H(t)\}_{j,\phi} + \{z_a(t)\}_{j,\phi}\{\mu_R(t)\}_{\phi}\{\mu_U(t)\}_{j,\phi} \quad (13.5)$$

where

$\{\mu_U(t)\}_{j,\phi}$	expected category "j" fault count at time t
j	a category of faults introduced during phase j, j = *RQ*, *DE*, or *CO*, corresponding to Requirements Faults, Design Faults, and Coding Faults, respectively
ϕ	a life cycle phase, ϕ = *RQ*, *RR*, *DE*, *DR*, *CO*, *CI*, or *TE*
t	life cycle time
$\{v(t)\mu_H(t)\}$	estimate of "j" fault introduction rate in phase ϕ
$\{z_a(t)\}_{j,\phi}$	intensity function of per-fault detection in phase ϕ, for category "j" faults
$\{\mu_R(t)\}_{\phi}$	expected change in fault count due to each repair in phase ϕ, for category "j" faults

Equation 13.5 is usually only applied to Requirements Faults, Design Faults, and Coding Faults (j = *RQ*, *DE*, and *CO*) because most faults are introduced into a software system during the RQ, DE, and CO phases.

The component $\{v(t)\mu_H(t)\}_{j,\phi}$ addresses the introduction of faults. The component $\{z_a(t)\}_{j,\phi}\{\mu_R(t)\}_{\phi}\{\mu_U(t)\}_{j,\phi}$ addresses the detection and removal of faults.

Three steps are required to set-up Equation 13.5.

1. Estimate of $\{v(t)\mu_H(t)\}_{j,\phi}$

Assuming that the fault-introduction rate within a phase is constant, the estimate of the fault-introduction rate is given by [Stutzke, 2001]:

$$\{v(t)\mu_H(t)\}_{j,\phi} = \begin{cases} \left\{\overline{v(t)\mu_H(t)}\right\}_{j,\phi} \times F_{j,\phi}^{1-2\times SLI} & j,\phi = j \\ 0 & \phi \neq j \end{cases} \quad (13.6)$$

and

$$\overline{\{v(t)\mu_H(t)\}}_{j,\phi} = \frac{DP \cdot fd_{j,\phi}}{\bar{t}_{fp,\phi}} \tag{13.7}$$

where

$\overline{\{v(t)\mu_H(t)\}}_{j,\phi}$	unadjusted estimate of the fault-introduction rate of the *j*-th fault categories
j	a category of faults introduced during phase ϕ, j = *RQ*, *DE*, or *CO*
$F_{j,\phi}$	a constant
SLI	Success Likelihood Index for the FDN measure which varies between 0 (error is likely) and 1 (error is not likely)
DP	fault potential per function point
$fd_{j,\phi}$	fraction of faults of type j that originated in phase ϕ
$\bar{t}_{fp,\phi}$	mean effort necessary to develop a function point in phase ϕ

In Equation 13.6, $\{v(t)\mu_H(t)\}_{j,\phi} = 0$ while $j \neq \phi$. The reason is that each category of faults is only introduced in a phase. For example, the Requirements Faults ($j = RQ$) are introduced in the requirement phase ($\phi = RQ$). Therefore, the introduction rate of the Requirements Faults is zero during other phases ($\phi \neq RQ$). We will thus write $\{v(t)\mu_H(t)\}_{j,\phi}$ as well as other parameters and variables in Section 13.6 and 13.7 as dependents on ϕ only.

Stutzke [Stutzke, 2001] proposed a method for estimating the SLI and F_ϕ. For F_ϕ, the following transformations should be made. The upper and the lower bounds on $\{v(t)\mu_H(t)\}_\phi$ (corresponding to the extreme values of SLI: 0 and 1) are:

$$\{v(t)\mu_H(t)\}_{min,\phi} = \frac{1}{F_\phi} \times \overline{\{v(t)\mu_H(t)\}}_\phi \text{ (corresponding to } SLI = 1)$$

and

$$\{v(t)\mu_H(t)\}_{max,\phi} = F_\phi \times \overline{\{v(t)\mu_H(t)\}}_\phi \text{ (corresponding to } SLI = 0)$$

Therefore,

$$F_\phi = \sqrt{\frac{\{v(t)\mu_H(t)\}_{max,\phi}}{\{v(t)\mu_H(t)\}_{min,\phi}}} \tag{13.8}$$

According to Equation 13.7, to obtain the upper and lower bounds of the $\{v(t)\mu_H(t)\}_\phi$ in the development phase, the upper and lower bounds of $DP.fd_\phi$ and $\bar{t}_{fp,\phi}$ should be obtained first.

Based on Capers Jones' data [Jones, 2002], the average defect potential per function point per phase for a software is shown in the "Average Defect Potential" column in Table 13.1. The upper bound (worst software case) and the lower bound (best software case) of the defect potential per

function point per phase for a software program are shown in the "Upper Bound of the Defect Potential" column and the "Lower Bound of the Defect Potential" column, respectively.

Table 13.1 $DP.fd_{\phi}$ Per Function Point Per Phase

Defect Origins	Average Defect Potential	Upper Bound of the Defect Potential	Lower Bound of the Defect Potential
Requirements	1.00	1.50	0.40
Design	1.25	2.20	0.60
Coding	1.75	2.50	1.00
Documents	0.60	1.00	0.40
Bad fixes	0.40	0.80	0.10
Total	5.00	8.00	2.50

The value of $\bar{t}_{fp,\phi}$ is determined according to the "Mean" column of Table 13.2 (adapted from Table 3.17 in [Jones, 1996]).

Table 13.2 $\bar{t}_{fp,\phi}$, Mean Effort Per Function Point for Each Life Cycle Phase ϕ, in Staff Hours

Phase, ϕ	Max	Mode	Min	Mean*
RQ	2.64	0.75	0.38	1.00
RR	1.76	0.59	0.33	0.74
DE	9.24	2.07	1.03	3.09
DR	1.76	0.60	0.33	0.75
CO	8.8	2.64	0.66	3.34
CI	1.76	0.88	0.44	0.95
Independent Validation & Verification	1.76	1.06	0.66	1.11
Unit Testing	1.89	0.88	0.33	0.96
Function Testing	5.28	0.88	0.44	1.54
Integration Testing	1.76	0.75	0.33	0.85
System Testing	1.32	0.66	0.26	0.70
Independent Testing	1.32	0.66	0.44	0.73
Field Testing	1.76	0.59	0.26	0.73
Acceptance Testing	1.76	0.38	0.22	0.58

*Note: Mean was calculated using Equation 12 in [Stutzke, 2001]:

$$Mean = \frac{1}{6}(Min + 4 \times Mode + Max)$$

Therefore, Table 13.3 provides the boundary information for $DP.fd_\phi$ and $\bar{t}_{fp,\phi}$.

Table 13.3 Boundary Information for $DP.fd_\phi$ and $\bar{t}_{fp,\phi}$

	Requirements			Design Phase			Coding Phase		
	Max	Mean	Min	Max	Mean	Min	Max	Mean	Min
$DP.fd_\phi$	1.5	1.00	0.4	2.2	1.25	0.6	2.5	1.75	1.0
$\bar{t}_{fp,\phi}$	2.64	1	0.38	9.24	3.09	1.03	8.8	3.34	0.66

Normally, there are enough reasons to believe that the defect potential will become smaller if more effort is spent on the development process. Thus, the maximum defect potential is corresponding to the minimum effort and the minimum defect potential is corresponding to the maximum effort. Therefore, the upper bound of the $\{v(t)\mu_H(t)\}_\phi$ can be obtained by using the maximum defect potential divided by the minimum development effort. Similarly, the lower bound of the $\{v(t)\mu_H(t)\}_\phi$ is the minimum defect potential over the maximum development effort. The results of the boundary of $\{v(t)\mu_H(t)\}_\phi$ are shown in Table 13.4.

Table 13.4 Boundary Information for $\{v(t)\mu_H(t)\}_\phi$

	Requirements			Design Phase			Coding Phase		
	Max	**Mean**	**Min**	**Max**	**Mean**	**Min**	**Max**	**Mean**	**Min**
$\{v(t)\mu_H(t)\}_\phi$	3.95	0.5	0.15	2.14	0.49	0.065	3.79	0.49	0.11

Thus, the value of ϕ for each development phase can be obtained from Equation 13.8 and is shown in Table 13.5**.**

Table 13.5 Values of F_ϕ for Different Fault Categories

	RQ	**DE**	**CO**
F_ϕ	5.13	5.74	5.87

If there is no data available in the documents for determining the value of SLI, it is recommended to use 0.5 for SLI, which corresponds to the average. Thus, Equation 13.6 becomes:

$$\{v(t)\mu_H(t)\}_{j,\phi} = \begin{cases} \{\overline{v(t)\mu_H(t)}\}_{\phi} & \phi = j \\ 0 & \phi \neq j \end{cases} \quad (13.9)$$

APP's SLI is given in Chapter 11 (Cyclomatic Complexity).

2. Estimate of $\{\mu_R(t)\}_{\phi}$

The expected change in fault count due to one repair for the life cycle phase ϕ is [Stutzke, 2001]:

$$\{\mu_R(t)\}_{\phi} = -\frac{\{N_{fixed}\}_{\phi}}{\{N_{rr}\}_{\phi}} \quad (13.10)$$

where

$\{\mu_R(t)\}_{\phi}$	Expected change in fault count due to one repair in the life-cycle phase ϕ
t	Life-cycle time
ϕ	A life-cycle phase, ϕ = *RQ*, *RR*, *DE*, *DR*, *CO*, *CI*, or *TE*
$\{N_{fixed}\}_{\phi}$	Number of requested repairs that are fixed in the life-cycle phase ϕ
$\{N_{rr}\}_{\phi}$	Number of repairs requested in the life-cycle phase ϕ

An industry average value of -0.7 should be used when the data for estimating $\{\mu_R(t)\}_{\phi}$ is not available (especially for RQ, DE, and CO, in which the debugging activities are rarely documented) [Stutzke, 2001].

3. Estimate of $\{z_a(t)\}_{j,\phi}$

The intensity function of per-fault detection in phase ϕ, $\{z_a(t)\}_{j,\phi}$ is estimated as follows:

According to Stutzke [Stutzke, 2001], $z_a(t)$ can be determined by Equation 13.11: [30]

$$z_a(t) = \frac{r \cdot x \cdot t_{fp}}{t - t_0} \quad (13.11)$$

where:

$z_a(t)$	Intensity function of per-fault detection
r	Fault-detection rate
x	Fault-detection efficiency
t_{fp}	Effort necessary to develop a function point
t	Time
t_0	Time at which the considered phase originates

[30] We omit the indices j, ϕ for the current discussion.

The fault detection efficiency x has the same characteristics as function $\{v(t)\mu_H(t)\}_{j,\phi}$. Thus, similarly, according to:

$$1 - x = \overline{1 - x} \cdot F^{(1-2SLI)}$$

the upper and lower bounds on $(1 - x)$ (corresponding to the extreme values of SLI: 0 and 1) are:

$$(1 - x)_{min} = \left(\frac{1}{F}\right)(\overline{1 - x}) \quad \text{(corresponding to } SLI = 1\text{)},$$

and

$$(1 - x)_{max} = (F)(\overline{1 - x}) \quad \text{(corresponding to } SLI = 0\text{)}.$$

Based on the data by Capers Jones [Jones, 1986], Table 13.6 presents the fault-detection efficiency during the development phases.

Table 13.6 Upper and Lower Bounds of the Fault Detection Efficiency during Development Phases

Removal Step	Lowest Efficiency	Modal Efficiency	Highest Efficiency
Desk checking of design	15%	35%	70%
Desk checking of code	20%	40%	60%

Therefore, the mean fault-detection-efficiency can be calculated using Equation 12 in [Stutzke, 2001] and F can be obtained easily. These results are provided in Table 13.7.

Table 13.7 Mean Fault Detection Efficiency and F for Fault Detection Efficiency x

Removal Step	Mean Efficiency	*F*
Desk checking of design	37.5%	1.68
Desk checking of code	40%	1.41

Therefore, $x = 1 - 0.625 \times 1.68^{(1-2SLI)}$ for RQ and DE documents; $x = 1 - 0.6 \times 1.41^{(1-2SLI)}$ for CO documents.

Estimations of the inspection speed are shown in Table 13.8.

As shown in Table 13.8, the peer-review speed is around four times the formal documents inspection rate and three times the code-inspection rate. The average effort and reviewing speed for the peer review can be estimated based on Table 13.2 and is shown in Table 13.9

Table 13.8 Estimations of the Reviewing Speed

Phase	Peer Review Speed	Inspection Rate
Requirement	20 pages/hour[31]	5 pages/hour
External Design		4 pages/hour
Internal Design		200 lines/hour
Code		150 non-comment source lines/hour
Test Plan		4 pages/hour

Table 13.9 Average Peer Review Effort and Reviewing Speed

Phase	Peer Review Effort (staff hour/function point)	Reviewing Speed (function point/staff hour)
RQ	0.74/4 = 0.185	5.41
DE	0.75/4 = 0.188	5.32
CO	0.95/3 = 0.32	3.13

Having the above information on x and r, the intensity function of per-fault detection in phase ϕ $\{z_a(t)\}_{j,\phi}$, is estimated and is shown in Table 13.10.

Rule 13.8: Only critical and significant faults should be considered when calculating the FDN for a system.

The fault content of a system estimated according to Rule 13.7 does not distinguish faults by their severity levels.

Furthermore, the measurements use empirical data and subjective assessments. The empirical data used in this research is based on a significant amount of industry data. The associated assessments are based on the best knowledge and information available to the research team after communications with the developers. Also, much of the modeling is based on direct measurements of the APP system and, as such, is purely objective in nature.

[31] It is assumed that each page contains 30 lines of requirements/design description in natural language or 30 lines of code.

Table 13.10 Intensity Function of Per-fault Detection of Requirements, Design, and Coding Faults

Phase ϕ	Intensity Function of Per-fault Detection, $\{z_a(t)\}_{j,\phi}$		
	Requirements Faults (j = RQ)	**Design Faults (j = DE)**	**Coding Faults (j = CO)**
RQ	$5.41 \times [1 - 0.625 \times 1.68^{(1-2SLI)}]/t$	0	0
RR	$[1 - 0.625 \times 1.68^{(1-2SLI)}]/(N_{fp} \times 0.74)$	0	0
DE	$[1 - 0.625 \times 1.68^{(1-2SLI)}]/(N_{fp} \times 3.09)$	$15.88 \times [1 - 0.625 \times 1.68^{(1-2SLI)}] /(t - \{t_{begin}\}_{DE})$	0
DR	$[1 - 0.625 \times 1.68^{(1-2SLI)}]/(N_{fp} \times 0.75)$	$[1 - 0.625 \times 1.68^{(1-2SLI)}]/(N_{fp} \times 0.74)$	0
CO	$[1 - 0.625 \times 1.68^{(1-2SLI)}]/(N_{fp} \times 3.34)$	$[1 - 0.625 \times 1.68^{(1-2SLI)}]/(N_{fp} \times 3.34)$	$10.41 \times [1 - 0.6 \times 1.41^{(1-2SLI)}] /(t - \{t_{begin}\}_{CO})$
CI	$[1 - 0.625 \times 1.68^{(1-2SLI)}]/(N_{fp} \times 0.95)$	$[1 - 0.625 \times 1.68^{(1-2SLI)}]/(N_{fp} \times 0.95)$	$[1 - 0.6 \times 1.41^{(1-2SLI)}]/(N_{fp} \times 0.95)$
IV&V	$[1 - 0.625 \times 1.68^{(1-2SLI)}]/(N_{fp} \times 1.11)$	$[1 - 0.625 \times 1.68^{(1-2SLI)}]/(N_{fp} \times 1.11)$	$[1 - 0.6 \times 1.41^{(1-2SLI)}]/(N_{fp} \times 1.11)$
UT	$[1 - 0.625 \times 1.68^{(1-2SLI)}]/(N_{fp} \times 0.96)$	$[1 - 0.625 \times 1.68^{(1-2SLI)}]/(N_{fp} \times 0.96)$	$[1 - 0.6 \times 1.41^{(1-2SLI)}]/(N_{fp} \times 0.96)$
FT	$[1 - 0.625 \times 1.68^{(1-2SLI)}]/(N_{fp} \times 1.54)$	$[1 - 0.625 \times 1.68^{(1-2SLI)}]/(N_{fp} \times 1.54)$	$[1 - 0.6 \times 1.41^{(1-2SLI)}]/(N_{fp} \times 1.54)$
IgT	$[1 - 0.625 \times 1.68^{(1-2SLI)}]/(N_{fp} \times 0.85)$	$[1 - 0.625 \times 1.68^{(1-2SLI)}]/(N_{fp} \times 0.85)$	$[1 - 0.6 \times 1.41^{(1-2SLI)}]/(N_{fp} \times 0.85)$
ST	$[1 - 0.625 \times 1.68^{(1-2SLI)}]/(N_{fp} \times 0.7)$	$[1 - 0.625 \times 1.68^{(1-2SLI)}]/(N_{fp} \times 0.7)$	$[1 - 0.6 \times 1.41^{(1-2SLI)}]/(N_{fp} \times 0.7)$
IpT	$[1 - 0.625 \times 1.68^{(1-2SLI)}]/(N_{fp} \times 0.73)$	$[1 - 0.625 \times 1.68^{(1-2SLI)}]/(N_{fp} \times 0.73)$	$[1 - 0.6 \times 1.41^{(1-2SLI)}]/(N_{fp} \times 0.73)$
FiT	$[1 - 0.625 \times 1.68^{(1-2SLI)}]/(N_{fp} \times 0.73)$	$[1 - 0.625 \times 1.68^{(1-2SLI)}]/(N_{fp} \times 0.73)$	$[1 - 0.6 \times 1.41^{(1-2SLI)}]/(N_{fp} \times 0.73)$
AT	$[1 - 0.625 \times 1.68^{(1-2SLI)}]/(N_{fp} \times 0.58)$	$[1 - 0.625 \times 1.68^{(1-2SLI)}]/(N_{fp} \times 0.58)$	$[1 - 0.6 \times 1.41^{(1-2SLI)}]/(N_{fp} \times 0.58)$

*Notes:

IV&V	Independent Validation & Verification
UT	Unit Testing
FT	Function Testing
IgT	Integration Testing
ST	System Testing
IpT	Independent Testing
FiT	Field Testing
AT	The number of function points for a system
N_{fp}	The number of function points for a system
t	The life-cycle time, in staff-hours
$\{t_{begin}\}_{DE}$	The beginning date of design phase
$\{t_{begin}\}_{CO}$	The beginning date of coding phase

13.3 Measurement Results

The following documents were used to measure FDN of the APP system:

- APP Module Software V&V PLAN (SVVP) [APP, Y1]
- Final Verification and Validation Report for APP Module Software [APP, Y2]
- APP Module µp1 System SRS [APP, Y3]
- APP Module µp1 Flux/Delta Flux/Flow Application SRS [APP, Y4]
- APP Module µp2 System SRS [APP, Y5]
- APP Module µp2 Flux/Delta Flux/Flow Application SRS [APP, Y6]
- APP Module Communication Processor SRS [APP, Y7]
- APP Module µp1 SDD [APP, Y8]
- APP Flux/Delta Flux/Flow Application SDD for µp1 [APP, Y9]
- APP µp2 System Software SDD [APP, Y10]
- APP µp2 Flux/Delta Flux/Flow Application Software SDD [APP, Y11]
- APP Communication Processor SDD [APP, Y12]
- APP Module µp1 Flux/Delta Flux/Flow Application source code [APP, Y14]
- APP Module µp2 System source code [APP, Y15]
- APP Module µp2 Flux/Delta Flux/Flow Application source code [APP, Y16]
- APP Module Communication Processor System source code [APP, Y17]
- Test Summary Report for µp1 [APP, Y18]
- Test Summary Report for µp2 [APP, Y19]
- Test Summary Report for Communication Processor [APP, Y20]

13.3.1 Phases in the Development Life Cycle

According to the documents cited above, the APP system was developed according to the waterfall model. The phases in the development life cycle are ordered as follows: RQ, RR, DE, DR, CO, CI, and TE.

13.3.2 Duration of Each Life-Cycle Phase

The APP system has five components: the μp1 System, the μp1 Application, the μp2 System, the μp2 Application, and the CP System. The μp1 System and the μp1 Application were developed by one team, while the μp2 System, the μp2 Application, and the CP System were developed by another team. The debugging phases (RR, DR, CI [Code Review Phase], and TE) were conducted by a third independent team.

The beginning dates and ending dates of RR, DR, and CI for the five components were obtained from [APP, Y1] and [APP, Y2].

The beginning dates and ending dates of TE for μp1 System, μp2 System, and CP System were obtained from [APP, Y18], [APP, Y19], and [APP, Y20]. There is no independent testing for μp1 Application and μp2 Application.

The ending dates of RQ, DE, and CO were obtained from [APP, Y2]. However, the beginning dates of RQ, DE, and CO were not documented. In Table 13.11, the beginning dates of RQ, DE and CO were estimated by the manufacturer of the APP system. These estimates can strongly influence the accuracy of the measurement results. Given the beginning date and the ending date, the length of a phase is estimated on a 20-workdays-per-month basis according to the manufacturer of the APP system. These data also are summarized in Table 13.11.

Based on the collected information from the developer (5 staff-hours/workday), the total effort (in staff-hours) of each life-cycle phase of the APP system development effort can be obtained and is shown in Table 13.12.

Table 13.11 Measurement of Length of Each Life Cycle-Phase for the APP System

		Phase, ϕ						
		RQ	**RR**	**DE**	**DR**	**CO**	**CI**	**TE**
µp1 System	Begin date	05/12/93	08/19/93	09/07/93	01/14/94	03/05/94	06/15/94	06/14/94
	End date	06/28/93	09/07/93	01/03/94	03/04/94	04/05/94	06/24/94	09/12/94
	$T_{\phi 1}$ in workdays	30	13	69	34	21	7	61
µp1 Appli-cation	Begin date	11/08/93	01/06/94	11/24/93	03/07/94	04/06/93	07/21/94	06/14/94
	End date	11/23/93	02/14/94	12/09/93	03/31/94	06/23/94	07/21/94	09/12/94
	$T_{\phi 2}$ in workdays	10	26	11	17	53	1	61
µp2 System	Begin date	09/29/93	11/12/93	10/14/93	02/21/94	04/07/94	07/07/94	08/10/94
	End date	10/13/93	01/28/94	12/08/93	04/06/94	05/16/94	08/15/94	11/02/94
	$T_{\phi 3}$ in workdays	13	25	38	30	27	27	57
µp2 Appli-cation	Begin date	10/13/93	01/19/94	10/23/93	03/11/94	05/17/94	08/09/94	08/10/94
	End date	10/22/93	02/10/94	12/10/93	03/24/94	07/07/94	08/19/94	11/02/94
	$T_{\phi 4}$ in workdays	7	15	33	9	35	7	57
CP	Begin date	08/09/93	10/27/93	09/10/93	02/16/94	12/14/94	07/08/94	10/04/94
	End date	09/09/93	12/01/93	12/13/93	03/17/94	04/11/94	08/08/94	11/09/94
	$T_{\phi 5}$ in workdays	20	23	64	20	73	21	25
T_ϕ in workdays $(T_\phi = T_{\phi 1} + T_{\phi 2} + T_{\phi 3} + T_{\phi 4} + T_{\phi 5})$		80	102	215	110	209	63	261

Table 13.12 Duration Estimation for All Life Cycle Phases of the APP

	Phase,						
	RQ	**RR**	**DE**	**DR**	**CO**	**CI**	**TE**
Total effort (in staff-hours)	400	510	1075	550	1045	315	1305

13.3.3 Software Development Life Cycle

Based on the data in Table 13.12, the entire software development life cycle timeline for the APP system can be reconstructed, as shown in Figure 13.1 (unit: staff-hours).

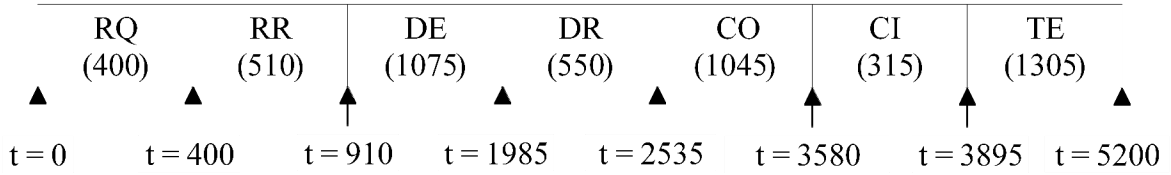


Figure 13.1 Software Development Life Cycle for APP

Table 13.13 summarizes the beginning date of each life cycle phase for the APP system. This data is used in Section 13.3.5 to estimate the intensity function of the per-fault detection for the development phases (RQ, DE, and CO).

Table 13.13 Beginning Time of Each Life-Cycle Phase for the APP

	Phase						
	RQ	**RR**	**DE**	**DR**	**CO**	**CI**	**TE**
Beginning time of phase , , in staff-hours	0	400	910	1985	2535	3580	3895

13.3.4 Introduction Rates of Requirements Faults, Design Faults, and Coding Faults

The introduction rates of requirements faults, design faults, and code faults, $\{v(t)\mu_H(t)\}_{j,\phi}$, are estimated according to Equations 13.6 and 13.7.

The function-point count for the APP system is 301 function points as determined in Section 14.3.3. Moreover, the APP system falls into the category of "system software" (see Section 14.4.1). Therefore, the fault-potential-per-function-point for the APP system can be obtained from the "Systems" column in Table 13.14 (extracted from Table 3.44 in [Jones, 1996]), using a logarithmic interpolation for 301 function points (100 < 301 < 1000).

$$\begin{aligned} DP(APP) &= 5 + \frac{6-5}{\log_{10}(1000)-\log_{10}(100)} \times [\log_{10}(301) - \log_{10}(100)] \\ &= 5.48 \text{ fault potential/function point} \end{aligned} \tag{13.12}$$

where $DP(APP)$ is the fault potential per function point for the APP system.

Table 13.14 Fault Potential Per Function Point, DP

Function Points	End User	MIS	Outsourced	Commercial	Systems	Military	Average
1	1	1	1	1	1	1	1.00
10	2.5	2	2	2.5	3	3.25	2.54
100	3.5	4	3.5	4	5	5.5	4.25
1,000	N/A	5	4.5	5	6	6.75	4.54
10,000	N/A	6	5.5	6	7	7.5	5.33
100,000	N/A	7.25	6.5	7.5	8	8.5	6.29

The value of fd_ϕ is obtained from the "Systems" column of Table 13.15 (extracted from Table 3.15 in [Jones, 1996]).

Table 13.15 fd_ϕ, Fraction of Faults Originated in Phase ϕ

Phase, ϕ	fd_ϕ						
	End User	**MIS***	**Out-sourced**	**Commer-cial**	**Systems**	**Military**	**Average**
RQ	0.00	0.15	0.20	0.10	0.10	0.20	0.1250
DE	0.15	0.30	0.25	0.30	0.25	0.20	0.2417
CO	0.55	0.35	0.35	0.30	0.40	0.35	0.3833
User Document	0.10	0.10	0.10	0.20	0.15	0.15	0.1333
Bad Fix	0.20	0.10	0.10	0.10	0.10	0.10	0.1167

*Note: "MIS" is "Management Information System"

The mean effort per function point, $\bar{t}_{fp,\phi}$, is obtained from the "Mean" column of Table 13.2.

Table 13.16 summarizes the data required to calculate $\{v(t)\mu_H(t)\}_{j,\phi}$ for the APP system.

Table 13.16 Data Required to Calculate $\{v(t)\mu_H(t)\}_{j,\phi}$ for APP

	Phase, ϕ						
	RQ	**RR**	**DE**	**DR**	**CO**	**CI**	**TE**
DP	5.48						
fd_ϕ	0.10	N/A	0.25	N/A	0.40	N/A	N/A
$\bar{t}_{fp,\phi}$ in staff hrs	1	0.74	3.09	0.75	3.34	0.95	3.12

*Note: Only Function Testing (FT), Integration Testing (IgT), and Independent Testing (IpT) were conducted during the testing phase, according to [APP, Y18], [APP, Y19], and [APP, Y20]. Therefore, $\bar{t}_{fp,\phi}$ for the testing phase is the sum of values of FT, IgT, and IpT.

Using Equations 13.6 and 13.7, Table 13.5 and Table 13.16 with SLI equals 0.71 (See Chapter 11), the introduction rates of requirements faults, design faults, and code faults can be calculated, as summarized in Table 13.17.

Table 13.17 Introduction Rates of Requirements, Design, and Coding Faults for APP

		Phase, ϕ						
		RQ	**RR**	**DE**	**DR**	**CO**	**CI**	**TE**
Fault Introduction Rate $\{v(t)\mu_H(t)\}_{j,\phi}$ faults/staff-hour	Requirements Faults ($j = RQ$)	0.28	0	0	0	0	0	0
	Design Faults ($j = DE$)	0	0	0.21	0	0	0	0
	Coding Faults ($j = CO$)	0	0	0	0	0.31	0	0

13.3.5 The Expected Change in Fault Count Due to One Repair

The expected change in fault count due to one repair in each phase, $\{\mu_R(t)\}_\phi$, is estimated according to Equation 13.10. However, the numbers of repair requests and the numbers of fixed-repair requests for the APP system are not available. Therefore, the industry average was used for all life-cycle phases; namely, $\{\mu_R(t)\}_\phi = -0.7$ for ϕ = RQ, RR, DE, DR, CO, CI, and TE.

13.3.6 Estimate of the Intensity Function of Per-Fault Detection

The intensity function of per-fault detection of requirements faults, design faults, and coding faults during RQ, RR, DE, DR, CO, or CI phase, $\{z_a(t)\}_{j,\phi}$ (ϕ = RQ, RR, DE, DR, CO, CI), is calculated according to Table 13.10. The number of function points for the APP system is 301, as determined in Chapter 14.

Only Function Testing (FT), Integration Testing (IgT), and Independent Testing (IpT) were conducted during the testing phase according to [APP, Y18], [APP, Y19], and [APP, Y20]. Therefore, $\{z_a(t)\}_{j,\phi}$ for the testing phase (TE) is the sum of values of FT, IgT, and IpT (see Table 13.18, calculated according to Table 13.10).

Table 13.18 Intensity Function of Per-Fault Detection Faults for APP

Phase ϕ	Intensity Function of Per-fault Detection $\{z_a(t)\}_{j,\phi}$		
	Requirements Faults $(j = RQ)$	**Design Faults** $(j = DE)$	**Coding Faults** $(j = CO)$
RQ	$2.691/t$	0	0
RR	0.00223	0	0
DE	0.00053	$8.177/(t-910)$	0
DR	0.00223	0.00223	0
CO	0.00049	0.00049	$5.022/(t-2535)$
CI	0.00172	0.00172	0.00166
TE $(TE = FT + IgT + IpT)$	0.00528	0.00528	0.00510

13.3.7 Expected Content of Requirements Faults, Design Faults, and Coding Faults

The expected content of requirements faults, design faults, and coding faults, $\{\mu_U(t)\}_{j,\phi}$, is obtained using the results in Section 13.3.2 through 13.3.6 to solve Equation 13.5.

For example, during the requirement-analysis phase ($j = RQ$, $\phi = RQ$), $\{v(t)\mu_H(t)\}_{j,\phi} = 0.28$, (determined in Table 13.17), $\{\mu_R(t)\}_{\phi} = -0.7$ (determined in Section 13.3.5), and $\{z_a(t)\}_{j,\phi} = 2.691/t$ (determined in Table 13.18). Therefore, Equation 13.5 becomes:

$$\begin{aligned}\frac{d\{\mu_U(t)\}_{RQ,RQ}}{dt} &= 0.28 + \frac{2.691}{t} \times (-0.7) \times \{\mu_U(t)\}_{RQ,RQ} \\ &= 0.28 - \frac{1.88}{t} \times \{\mu_U(t)\}_{RQ,RQ}\end{aligned} \tag{13.13}$$

Since $\{\mu_U(t)\}_{RQ,RQ}|_{t=0} = 0$ (there is no fault introduced into a system when $t = 0$), Equation 13.15 yields:

$$\{\mu_U(t)\}_{RQ,RQ} = 0.097t \qquad 0 \le t \le 400 \tag{13.14}$$

During RR, $\{v(t)\mu_H(t)\}_{j,\phi} = 0$, $\{\mu_R(t)\}_{\phi} = -0.7$, and $\{z_a(t)\}_{j,\phi} = 0.00223$.
Therefore, Equation 13.5 becomes:

$$\frac{d\{\mu_U(t)\}_{RQ,RR}}{dt} = 0 + 0.00223 \times (-0.7) \times \{\mu_U(t)\}_{RQ,RR} = -0.00156 \times \{\mu_U(t)\}_{RQ,RR} \quad (13.15)$$

Since continuity dictates that $\{\mu_U(t)\}_{RQ,RR}|_{t=400} = \{\mu_U(t)\}_{RQ,RQ}|_{t=400} = 0.097 \times 400 = 38.8$ Equation 13.13 yields:

$$\{\mu_U(t)\}_{RQ,RR} = 38.8 \times \exp[-0.00156(t-400)] \qquad 400 \le t \le 910 \quad (13.16)$$

In the same way, the expected content of requirements faults can be obtained by solving Equation 13.5 phase-by-phase, as shown in Equation 3.17:

$$\{\mu_U(t)\}_{RQ} = \begin{cases} 0.097t & 0 \le t \le 400 \\ 38.80 \times \exp[-0.00156(t-400)] & 400 \le t \le 910 \\ 17.50 \times \exp[-0.00037(t-910)] & 910 \le t \le 1985 \\ 11.70 \times \exp[-0.00156(t-1985)] & 1985 \le t \le 2535 \\ 4.95 \times \exp[-0.00034(t-2535)] & 2535 \le t \le 3580 \\ 3.44 \times \exp[-0.0012(t-3580)] & 3580 \le t \le 3895 \\ 2.36 \times \exp[-0.0037(t-3895)] & 3895 \le t \le 5200 \end{cases} \quad (13.17)$$

Similarly, using the results in Section 13.3.2 through 13.3.6 to solve Equation 13.5 yields the expected content of design faults:

$$\{\mu_U(t)\}_{DE} = \begin{cases} 0 & 0 \le t \le 400 \\ 0 & 400 \le t \le 910 \\ 0.0312t & 910 \le t \le 1985 \\ 33.58 \times \exp[-0.00156(t-1985)] & 1985 \le t \le 2535 \\ 14.21 \times \exp[-0.00034(t-2535)] & 2535 \le t \le 3580 \\ 9.90 \times \exp[-0.0012(t-3580)] & 3580 \le t \le 3895 \\ 6.77 \times \exp[-0.0037(t-3895)] & 3895 \le t \le 5200 \end{cases} \quad (13.18)$$

Using the results in Section 13.3.2 through 13.3.6 to solve Equation 13.5 yields the expected content of coding faults:

$$\{\mu_U(t)\}_{CO} = \begin{cases} 0 & 0 \le t \le 400 \\ 0 & 400 \le t \le 910 \\ 0 & 910 \le t \le 1985 \\ 0 & 1985 \le t \le 2535 \\ 0.069t & 2535 \le t \le 3580 \\ 71.74 \times \exp[-0.0016(t-3580)] & 3580 \le t \le 3895 \\ 49.71 \times \exp[-0.00357(t-3895)] & 3895 \le t \le 5200 \end{cases} \quad (13.19)$$

The total expected fault content of the APP system is

$$\{\mu_U(t)\} = \{\mu_U(t)\}_{RQ} + \{\mu_U(t)\}_{DE} + \{\mu_U(t)\}_{CO} \tag{13.20}$$

13.3.8 Count of Fault-Days Number

The FDN for the APP system is determined according to the eight measurement rules (Rule 13.1 to Rule 13.8) described in Section 13.2. The time unit in this section is converted from staff-hours to workdays (on a 5-staff-hours/day basis), which is given by the manufacturer of the APP system.

Three steps are required to count the FDN of a system:

1. Calculate the FDN for faults removed during the development life cycle
2. Calculate the FDN for faults remaining in the delivered source code
3. Calculate the FDN of the system, which is the sum of the results of the previous two steps

13.3.8.1 Calculation of FDN for Faults Removed During the Development Life Cycle

Table 13.19 summarizes the required data for counting the FDN, in which $\{t_{begin}\}_\phi$ and $\{t_{end}\}_\phi$ were determined according to Figure 13.1.

As mentioned in Section 13.2, faults are classified based on the phase during which they are introduced into a system. For example, the requirements faults are introduced into a system only during RQ. Therefore, the introduction date of type j faults, $\{f_{in}\}_j$, according to Rule 13.4, is:

$$\{f_{in}\}_j = \begin{cases} \{t_{begin}\}_\phi + \frac{\{t_{end}\}_\phi - \{t_{begin}\}_\phi}{2} & if\ \phi = j \\ Not\ applicable & if \phi \neq j \end{cases} \tag{13.21}$$

where

$\{f_{in}\}_j$	date at which type j faults are introduced into a system
j	a category of faults introduced, j = RQ, DE, or CO
ϕ	a life cycle phase, ϕ = RQ, RR, DE, DR, CO, CI, or TE
$\{t_{end}\}_\phi$	ending date of the phase ϕ
$\{t_{begin}\}_\phi$	beginning date of the phase ϕ

Similarly, a fault of type j cannot be removed from a system until it has been introduced into a system. Therefore, the date at which type j faults are removed from a system during phase ϕ, $\{f_{out}\}_{j,\phi}$ according to Rule 13.5, is:

$$\{f_{out}\}_{j,\phi} = \begin{cases} Not\ applicable & if\ \phi\ occurs\ ahead\ of\ j \\ \{t_{begin}\}_{\phi} + \frac{\{t_{end}\}_{\phi} - \{t_{begin}\}_{\phi}}{2} & otherwise \end{cases} \quad (13.22)$$

where

$\{f_{out}\}_{j,\phi}$	date at which type j faults are removed from a system
j	a category of faults introduced, j = RQ, DE, or CO
ϕ	a life cycle phase, ϕ = RQ, RR, DE, DR, CO, CI, or TE
$\{t_{end}\}_{\phi}$	ending date of the phase ϕ
$\{t_{begin}\}_{\phi}$	beginning date of the phase ϕ

$\{\mu_U(t_{begin})\}_{j,\phi}$ and $\{\mu_U(t_{end})\}_{j,\phi}$ were calculated and shown in Table 13.19 according to Equation 13.17 ($j = RQ$), Equation 13.18 ($j = DE$), and Equation 13.19 ($j = CO$), respectively.

Using Table 13.19, the FDN for each fault category can be calculated phase-by-phase, as presented in Table 13.20.

According to Rule 13.8, only critical faults and significant faults should be considered while calculating the FDN. Moreover, the fraction of critical faults and significant faults for the APP system is 0.1391, as calculated in Equation 6.3. Therefore, the number of type j faults (critical and significant) removed from the APP system during phase ϕ is:

$$\Delta_{j,\phi} = \begin{cases} Not\ applicable & if\ \phi\ occurs\ ahead\ of\ j \\ \left[\{\mu_U(t_{begin})\}_{j,\phi} - \{\mu_U(t_{end})\}_{j,\phi}\right] \times 0.1391 & otherwise \end{cases} \quad (13.23)$$

where

$\Delta_{j,\phi}$	number of type j faults (critical and significant) removed during phase ϕ
j	a category of faults introduced, j = RQ, DE, or CO
ϕ	a life cycle phase, ϕ = RQ, RR, DE, DR, CO, CI, or TE
$\{\mu_U(t_{begin})\}_{j,\phi}$	Expected number of type j faults at the beginning of phase ϕ
$\{\mu_U(t_{end})\}_{j,\phi}$	Expected number of type j faults at the end of phase ϕ

The FDN per fault of type j removed during phase ϕ is:

$$\{FDN_{j,\phi}\}_{per\ fault} = \begin{cases} Not\ applicable & if\ \phi\ occurs\ ahead\ of\ j \\ \{f_{out}\}_{j,\phi} - \{f_{in}\}_j & otherwise \end{cases} \quad (13.24)$$

where

$\{FDN_{j,\phi}\}_{per\ fault}$	fault-days number per fault of type j removed during phase ϕ
$\{f_{in}\}_j$	date at which type j faults are introduced into a system
$\{f_{out}\}_{j,\phi}$	date at which type j faults are removed from a system
j	a category of faults introduced, j = RQ, DE, or CO
ϕ	a life-cycle phase, ϕ = RQ, RR, DE, DR, CO, CI, or TE

Table 13.19 Data Required to Calculate FDN for Faults Removed during the Development Life Cycle

		Phase, ϕ						
		RQ	**RR**	**DE**	**DR**	**CO**	**CI**	**TE**
$\{t_{begin}\}_j$		0	80	182	397	507	716	779
$\{t_{end}\}_\phi$		80	182	397	507	716	779	1040
Date at which type j faults are introduced into APP, $\{f_{in}\}_j = \{t_{begin}\}_\phi + \frac{\{t_{end}\}_\phi - \{t_{begin}\}_\phi}{2}$	*Requirements Faults (j = RQ)*	40	N/A	N/A	N/A	N/A	N/A	N/A
	Design Faults (j = DE)	N/A	N/A	289.5	N/A	N/A	N/A	N/A
	Coding Faults (j = CO)	N/A	N/A	N/A	N/A	611.5	N/A	N/A

Table 13.19 Data Required to Calculate Fault-days Number for Faults Removed during the Development Life Cycle (continued)

		Phase, ϕ						
		RQ	**RR**	**DE**	**DR**	**CO**	**CI**	**TE**
$\{t_{begin}\}_j$		0	80	182	397	507	716	779
$\{t_{end}\}_\phi$		80	182	397	507	716	779	1040
Date at which type j faults are removed from APP $\{f_{out}\}_{j,\phi} = \{t_{begin}\}_\phi + \frac{\{t_{end}\}_\phi - \{t_{begin}\}_\phi}{2}$	*Requirements Faults (j = RQ)*	40	131	289.5	452	611.5	747.5	909.5
	Design Faults (j = DE)	N/A	N/A	289.5	452	611.5	747.5	909.5
	Coding Faults (j = CO)	N/A	N/A	N/A	N/A	611.5	747.5	909.5
Number of faults at the beginning of phase ϕ, $\{\mu_U(t_{begin})\}_{j,\phi}$	*Requirements Faults (j = RQ)*	0	38.8	17.5	11.7	5.0	3.4	2.4
	Design Faults (j = DE)	0	0	0	33.6	14.2	9.9	6.8
	Coding Faults (j = CO)	0	0	0	0	0	71.7	49.7
Number of faults at the end of phase ϕ, $\{\mu_U(t_{end})\}_{j,\phi}$	*Requirements Faults (j = RQ)*	38.8	17.5	11.7	5.0	3.4	2.4	0.02
	Design Faults (j = DE)	0	0	33.6	14.2	9.9	6.8	0.05
	Coding Faults (j = CO)	0	0	0	0	71.7	49.7	0.47

The fault-days number for a fault of type j removed during phase ϕ is:

$$FDN_{j,\phi} = \begin{cases} Not\ applicable & if\ \phi\ occurs\ ahead\ of\ j \\ \{FDN_{j,\phi}\}_{per\ fault} \times \Delta_{j,\phi} & otherwise \end{cases} \tag{13.25}$$

where

$FDN_{j,\phi}$ fault-days number of type j faults (critical and significant) removed during phase ϕ;

$\{FDN_{j,\phi}\}_{per\ fault}$ fault-days number per fault of type j removed during phase ϕ;

$\Delta_{j,\phi}$ number of type j faults (critical and significant) removed during phase ϕ;

j a category of faults introduced during phase j, j = RQ, DE, or CO;

ϕ a life cycle phase, ϕ = RQ, RR, DE, DR, CO, CI, or TE.

Table 13.20 Calculation of FDN for Faults Removed during the Development Life Cycle

		Phase ϕ						
		RQ	**RR**	**DE**	**DR**	**CO**	**CI**	**TE**
Fault-days number per fault of type j removed during phase ϕ, $\{FDN_{j,\phi}\}_{perfault}$ in workdays	*Requirements Faults (j = RQ)*	0	91	249.5	412	571.5	707.5	869.5
	Design Faults (j = DE)	N/A	N/A	0	162.5	322	458	620
	Coding Faults (j = CO)	N/A	N/A	N/A	N/A	0	136	298
Number of type j faults (critical and significant) removed during phase ϕ, $\Delta_{j,\phi} = \begin{bmatrix} \{\mu_U(t_{begin})\}_{j,\phi} \\ -\{\mu_U(t_{end})\}_{j,\phi} \end{bmatrix} \times 0.1391$	*Requirements Faults (j = RQ)*	0	3.0	0.8	0.9	0.2	0.2	0.3
	Design Faults (j = DE)	N/A	N/A	0	2.7	0.6	0.4	0.9
	Coding Faults (j = CO)	N/A	N/A	N/A	N/A	0	3.1	6.8
Fault-days number of type j faults removed during phase ϕ, $FDN_{j,\phi}$	*Requirements Faults (j = RQ)*	0	273.0	199.6	370.8	114.3	141.5	260.9
	Design Faults (j = DE)	N/A	N/A	0	438.8	193.2	183.2	558.0
	Coding Faults (j = CO)	N/A	N/A	N/A	N/A	0	421.6	2026.4

13.3.8.2 Calculation of FDN for Faults Remaining in the Delivered Source Code

The fault-days number for faults remaining in the delivered source code is calculated using Rule 13.3, 13.4, 13.6, and 13.8, as summarized in Table 13.21.

The date at which type *j* faults are introduced into the APP, $\{f_{in}\}_j$, is determined in Table 13.19. According to Rule 13.6, the removal date for a fault remaining in the delivered source code is the ending date of TE (the last phase of the development life cycle), namely,

$$\{f_{out}\}_{remain} = \{t_{end}\}_{TE} \tag{13.26}$$

where

$\{f_{out}\}_{remain}$ removal date of faults remaining in the delivered source code;
$\{t_{end}\}_{TE}$ ending date of testing phase, which is the last phase in the software development life cycle of the APP system.

The FDN per fault of type *j* remaining in the delivered source code, according to Rule 13.3, is:

$$\{FDN_{j,remain}\}_{per\ fault} = \{f_{out}\}_{remain} - \{f_{in}\}_j \tag{13.27}$$

where

$\{FDN_{j,remain}\}_{per\ fault}$ fault-days number per fault of type *j* remaining in the delivered source code;
$\{f_{out}\}_{remain}$ removal date of faults remaining in the delivered source code;
$\{f_{in}\}_j$ date at which type *j* faults are introduced into a system.

The number of type *j* faults (critical and significant) remaining in the delivered source code was estimated using Equation 13.17 ($j = RQ$), Equation 13.18 ($j = DE$), and Equation 13.19 ($j = CO$), respectively:

$$N_{j,remain} = \{\mu_U(t)\}_j|_{t=\{t_{end}\}_{TE}} \times 0.1391 \tag{13.28}$$

where

$N_{j,remain}$ number of type j faults (critical and significant) remaining in the delivered source code;
$\{\mu_U(t)\}_j$ expected content of type *j* faults at life cycle time t;
$\{t_{end}\}_{TE}$ ending date of the testing phase.

Using Equations 13.27 and 13.28, the fault-days number for type j faults remaining in the delivered source code can be calculated:

$$FDN_{j,remain} = \{FDN_{j,remain}\}_{per\ fault} \times N_{j,remain} \tag{13.29}$$

Table 13.21 Calculation of Fault-days Number for Faults Remaining in the Delivered Source Code

	Requirements Faults ($j = RQ$)	**Design Faults ($j = DE$)**	**Coding Faults ($j = CO$)**
Date at which type j faults are introduced into APP $\{f_{in}\}_j$ *in workdays*	40	289.5	611.5
Removal date of faults remaining in the delivered source code $\{f_{out}\}_{remain} = \{t_{end}\}_{TE}$ *in workdays*	1040		
Fault-days number per fault remaining in the delivered source code, in workdays $\{FDN_{j,remain}\}_{per\ fault} = \{f_{out}\}_{remain} - \{f_{in}\}_j$	1000	750.5	428.5
Number of type j faults (critical and significant) remaining in the delivered source code $N_{j,remain} = \{\mu_U(t)\}_j\|_{t=\{t_{end}\}_{TE}} \times 0.1391$	0.00264	0.00757	0.06537
Fault-days number of type j faults remaining in the delivered source code $FDN_{j,remain} = \{FDN_{j,remain}\}_{per\ fault} \times N_{j,remain}$	2.6	5.7	28.0

13.3.8.3 Calculation of FDN for the APP

Using the results in Table 13.20 and 13.21, the fault-days number for the APP system is:

$$FDN = \sum_{j=RQ,DE,CO}\left(\sum_{all\ \phi} FDN_{j,\phi}\right) + \sum_{j=RQ,DE,CO} FDN_{j,remain} = 5217.6\ \text{fault} \cdot \text{workday} \quad (13.30)$$

13.4 RePS Construction Using the Fault-Days Number Measure

Based on the cumulative characteristic of the Fault-days Number measure and by using the concepts introduced by Stutzke [Stutzke, 2001], one can show that FDN is related to $\mu_U(t)$ by the following equation:

$$\langle FDN(t+\Delta t)\rangle = \langle FDN(t)\rangle + v(t)\mu_H(t)\Delta t \cdot \Delta t + [1 - z_a(t)\mu_R(t)\Delta t]\mu_U(t) \cdot \Delta t \quad (13.31)$$

where

$\langle FDN(t+\Delta t)\rangle$	the Fault-days Number at time $t+\Delta t$
$\langle FDN(t)\rangle$	the Fault-days Number at time t
$v(t)\mu_H(t)$	estimate of fault introduction rate
$z_a(t)$	intensity function of per-fault detection
$\mu_R(t)$	expected change in fault count due to each repair
$\mu_U(t)$	expected fault count at time t

Equation 13.33 can be simplified to Equation 13.32:

$$\frac{d\langle FDN\rangle}{dt} = \mu_U(t) \quad (13.32)$$

This equation shows the direct relationship between the measured real FDN and the corresponding fault count number. Once the real FDN is measured, the number of faults can be obtained by this equation. However, the real FDN cannot be obtained experimentally because not all the faults can be discovered during the inspection. One can only obtain the apparent FDN, FDN_A. "Apparent" refers to only removed faults logged during the development process. One can relate FDN_A to FDN by Equation 13.33 knowing v, z_a, μ_R, μ_U.

$$\frac{d\langle FDN_A\rangle}{dt} = \gamma(t; v, z_a, \mu_R, \mu_H) \cdot \frac{d\langle FDN\rangle}{dt} \quad (13.33)$$

where

FDN_A	the apparent Fault-days Number
$\gamma(t; v, z_a, \mu_R, \mu_H)$	a function of v, z_a, μ_R, μ_U which relates FDN_A to FDN
FDN	the exact Fault-days Number

Therefore, one can still obtain the fault count based on the measured apparent FDN as shown by Equation 13.34.

$$\mu_U(t) = \frac{d\langle FDN_A\rangle}{dt} \cdot \frac{1}{\gamma(t; v, z_a, \mu_R, \mu_H)} \quad (13.34)$$

Thus ideally, six steps are required to estimate software reliability using the Fault-days Number measure:

1. Measure the apparent FDN
2. Map the faults discovered into the EFSM
3. Execute the EFSM and obtain the failure probability
4. Calculate the per-fault Fault Exposure Ratio (vK)
5. Calculate the number of faults (N) remaining in the source code using FDN measurement results by Equation 13.34
6. Calculate the failure probability using Musa's exponential model

In the case of the APP system, the above procedures are difficult to apply because:

- The apparent FDN may be unobtainable because no record of removed faults exists. One can only obtain the average introduction and removal date of a category of faults during a specific development phase. Therefore, the FDN obtained in Section 13.3.8 is not the apparent FDN of the APP system, it is an estimated FDN.
- There may be no record of the description of each fault found during the development process. Thus, it may be impossible to map the faults discovered into the EFSM and execute the EFSM to obtain the failure probability and the exact per-fault Fault Exposure Ratio for the APP system. One substitute method is to use the testing data and its corresponding Fault Exposure Ratio. (This will be shown in Chapter 17).

The research team was aware of these difficulties and adopted the following steps to estimate the reliability of the APP system using the Fault-days Number measure:

1. Measure the estimated FDN shown in Section 13.3.8
2. Estimate the number of faults (N) remaining in the source code using the Fault-days Number measure
3. Estimate the number of delivered critical and significant faults
4. Calculate the failure probability using Musa's Exponential Model and the new Fault Exposure Ratio

13.4.1 Estimate of Number of Faults Remaining in the Source Code Using FDN

According to Figure 13.1 the APP system was released by the end of TE, when $t = 5200$ staff-hours. Therefore, the delivered fault content is:

$$
\begin{aligned}
N_{FDN,total} &= \{\mu_U(t)\} \\
&= \{\mu_U(t)\}_{RQ}|_{t=5200} + \{\mu_U(t)\}_{DE}|_{t=5200} + \{\mu_U(t)\}_{CO}|_{t=5200} \\
&= 2.36 \times \exp[-0.0037 \times (5200 - 3895)] \\
&\quad +6.97 \times \exp[-0.0037 \times (5200 - 3895)] \\
&\quad +49.71 \times \exp[-0.0037 \times (5200 - 3895)] \\
&= 0.5
\end{aligned}
\tag{13.35}
$$

Where

$N_{FDN,total}$	total number of delivered faults in the APP estimated using the Fault-days Number (FDN) measure
$\mu_U(t)$	total expected fault content of the APP as a function of life cycle time

13.4.2 Estimate of the Number of Delivered Critical and Significant Faults

Given the total number of delivered defects, $N_{FDN,total}$, and the percentages of delivered defects by severity level as determined in Table 6.7, the number of delivered defects by severity level can be calculated. For example, the number of delivered defects of severity 1 for the APP system is 0.543 × 0.0185 = 0.01.

Table 13.22 presents the number of delivered defects by severity level for the APP system.

Table 13.22 Number of Delivered Defects by Severity Level for the APP System

	Severity 1 (critical)	**Severity 2 (significant)**	**Severity 3 (minor)**	**Severity 4 (cosmetic)**
Number of delivered defects	0.01	0.065	0.205	0.262

13.4.3 Reliability Calculation from Delivered Critical and Significant Defects

According to Musa's exponential model [Musa, 1990] [Smidts, 2004], the reliability of a software product is given by:

$$
p_s(FDN) = \exp\left(-K \times N_{FDN} \times {}^{\tau}/_{T_L}\right) \tag{13.36}
$$

and

$$N_{FDN} = N_{FDN,critical} + N_{FDN,significant} \quad (13.37)$$

where

$p_s(FDN)$	reliability estimation for the APP system using the Fault-days Number (FDN) measure;
K	Fault Exposure Ratio, in failures/fault;
N_{FDN}	Number of defects in APP estimated using the FDN measure;
$N_{FDN,critical}$	Number of delivered *critical defects* (severity 1) estimated using the FDN measure;
$N_{FDN,significant}$	Number of delivered *significant defects* (severity 2) estimated using the FDN measure;
τ	Average execution-time-per-demand, in seconds/demand;
T_L	Linear execution time, in seconds.

The value of the new fault exposure ratio is 4.5×10^{-12} failure/defect. This is determined later through Equation 17.14 and shown in section 19.2.2.3.

As shown in Table 13.22, the APP system $N_{FDN,critical} = 0.01$, and $N_{FDN,significant} = 0.065$. Therefore, according to Equation 13.37, $N_{FDN} = 0.01 + 0.065 = 0.075$ which we round to 1.

The *linear execution time*, T_L, is usually estimated as the ratio of the execution time and the software size on a single microprocessor basis [Musa, 1990] [Smidts, 2004]. In the case of the APP system, however, there are three parallel subsystems (μp1, μp2, and CP), each of which has a microprocessor executing its own software. Each of these three subsystems has an estimated linear execution time. Therefore, there are several ways to estimate the linear execution time for the entire APP system, such as using the average value of these three subsystems.

For a safety-critical application like the APP system, the UMD research team suggests to make a conservative estimation of T_L by using the minimum of the three subsystems. Namely,

$$\begin{aligned} T_L &= \min\{T_L(\mu p1), T_L(\mu p2), T_L(CP)\} \\ &= \min\{0.018, 0.009, 0.021\} \\ &= 0.009 \text{ second} \end{aligned} \quad (13.38)$$

where

$T_L(\mu p1)$	Linear execution time of Microprocessor 1 (μp1) of the APP system. T_L *(μp1)* = 0.018 second (refer to Chapter 17)
$T_L(\mu p2)$	Linear execution time of Microprocessor 2 (μp2) of the APP system. T_L *(μp2)* = 0.009 second (refer to Chapter 17)

$T_L(CP)$ Linear execution time of Communication Microprocessor (CP) of the APP system. $T_L\ (CP)$ = 0.021 second (refer to Chapter 17)

Similarly, the *average execution-time-per-demand*, τ, is also estimated on a single microprocessor basis. Each of the three subsystems in APP has an estimated average execution-time-per-demand. To make a conservative estimation, the average execution-time-per-demand for the entire APP system is the maximum of the three subsystems. Namely,

$$\begin{aligned}\tau &= \max\{\tau(\mu p1), \tau(\mu p2), \tau(CP)\} \\ &= \max\{0.082, 0.129, 0.016\} \\ &= 0.129 \text{ seconds/demand}\end{aligned} \tag{13.39}$$

where

$T_L(\mu p1)$ Average execution-time-per-demand of Microprocessor 1 (μp1) of the APP system. $\tau(\mu p1)$ = 0.082 seconds/demand (refer to Chapter 17)

$T_L(\mu p2)$ Average execution-time-per-demand of Microprocessor 2 (μp2) of the APP system. $\tau(\mu p2)$ = 0.129 seconds/demand (refer to Chapter 17)

$T_L(CP)$ Average execution-time-per-demand of Communication Microprocessor (CP) of the APP system. $\tau(CP)$ = 0.016 seconds/demand (refer to Chapter 17)

Thus the reliability for the APP system using the Fault Days Number measure is given by:

$$\begin{aligned}p_s(FDN) &= e^{\left(-4.5\times10^{-12}\times1\times{}^{0.129}/_{0.009}\right)} \\ &= 0.99999999999355\end{aligned} \tag{13.40}$$

A more accurate estimation of reliability using the *Fault-days Number* measure for the APP system can be obtained by the following:

1. Obtaining the accurate dates at which faults are introduced into a system and removed from a system;
2. Obtaining actual dates at which phases of the development life cycle start;
3. Considering the existence of multiple versions of documentation for each phase;
4. Considering the overlap between two development life cycle phases;
5. Considering the iteration of the development life cycle phases;
6. Obtaining better documentation on debugging activities during RQ, DE, and CO phases;
7. Estimating the fault introduction rate in each development life cycle phase using the data for safety-critical applications, rather than the data for industry average;
8. Collecting data to estimate the Success Likelihood Index for the Fault-days Number measure, SLI_{FDN}, for the safety-critical application.
9. Using the concept of vK as discussed in Chapter 19.

13.5 Lessons Learned

The measurement of FDN requires data on the software-development process. This data was unavailable to the research team because it was either undocumented or unclearly documented in the software-development documents (SRS, SDD code, and V&V). For example, the exact effort for each development phase could not be obtained for each team member because it was not recorded during the original development. Even if these data had been recorded, the exact effort for each phase would have been difficult to measure since the development did not follow a waterfall development model because the developers returned to work on the SRS after the code was written.

13.6 References

[APP, Y1] "APP Module Software V&V PLAN (SVVP)," Year Y1.

[APP, Y2] "Final V&V Report for APP Module Software," Year Y2.

[APP, Y3] "APP Module First Safety Function Processor SRS," Year Y3.

[APP, Y4] "APP Flux/Delta Flux/Flow Application SRS for SF1," Year Y4.

[APP, Y5] "APP Module µp2 System Software SRS," Year Y5.

[APP, Y6] "APP µp2 Flux/Delta Flux/Flow Application Software SRS," Year Y6.

[APP, Y7] "APP Module Communication Processor SRS," Year Y7.

[APP, Y8] "APP Module First Safety Function Processor SDD," Year Y8.

[APP, Y9] "APP Flux/Delta Flux/Flow Application SDD for SF1," Year Y9.

[APP, Y10] "APP µp2 System Software SDD," Year Y10.

[APP, Y11] "APP µp2 Flux/Delta Flux/Flow Application Software SDD," Year Y11.

[APP, Y12] "APP Communication Processor SDD," Year Y12.

[APP, Y13] "APP Module SF1 System Software code," Year Y13.

[APP, Y14] "APP SF1 Flux/Delta Flux/Flow Application code," Year Y14.

[APP, Y15] "APP Module µp2 System Software Source Code Listing," Year Y15.

[APP, Y16] "APP µp2 Flux/Delta Flux/Flow Application Software Source Code Listing," Year Y16.

[APP, Y17] "APP Communication Processor Source Code," Year Y17.

[APP, Y18] "Test Summary Report for µp1," Year Y18.

[APP, Y19] "Test Summary Report for µp2," Year Y19.

[APP, Y20] "Test Summary Report for Communication Processor," Year Y20.

[Hanna, 1995] M. Hanna. "Farewell to Waterfalls," *Software Magazine*, pp. 38–46, 1995.

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

[IEEE 610.12, 1990] "IEEE Standard Glossary of Software Engineering Terminology," IEEE Std. 610.12-1990, 1990.

[Jones, 1986] C. Jones. *Programming Productivity*. McGraw-Hill, Inc., 1986.

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

[Jones, 2002] C. Jones. *Software Quality in 2002: A Survey of the State of Art*. Burlington, MA, 2002.

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

[Pressman, 2004] R. Pressman. *Software Engineering: A Practitioner's Approach*. New York: McGraw Hill, 2004.

[Shepard, 1979] S.B. Shepard and T. Love. "Modern coding practices and programmer performance," *Computer*, vol. 12, no. 12. pp. 41–49, 1979.

[Smidts, 2000] C. Smidts and M. Li, "Software Engineering Measures for Predicting Software Reliability in Safety Critical Digital Systems," NRC, Office of Nuclear Regulatory Research, Washington DC NUREG/GR-0019, 2000.

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

[Stutzke, 2001] M. Stutzke and C. Smidts. "A Stochastic Model of Fault Introduction and Removal during Software Development," *IEEE Transactions on Reliability Engineering*, vol. 50, 2001.

14. FUNCTION POINT

Function Point (FP) is a measure designed to determine the functional size of software.[32]

FP measures the entire size of an application including enhancements regardless of the technology used for development and/or maintenance. FPs have gained acceptance as a primary measure of software size [IEEE 982.2, 1988].

This measure can be applied as soon as the requirements are available. As listed in Table 3.3, the applicable life-cycle phases for FP are Requirements, Design, Coding, Testing, and Operation.

14.1 Definition

The *Function Point Counting Practices Manual* is the definitive description of the Function Point Counting Standard, despite the fact that there are many resources addressing FP counting (such as [Heller, 1996] and [Garmus, 2001]). Several versions of the manual are available, the latest, Release 4.3.1, was published in 2004 [IFPUG, 2004].

However, unless otherwise specified, information in this chapter is intended to be consistent with Release 4.1 [IFPUG, 2000]. This is because this report is a follow-up of previous research [Smidts, 2004] in which Release 4.1 was used in FP counting.

According to [IFPUG, 2000], "Function Point" is a unit of measure of functionality of a software project or application from a logical (not physical) point of view. A "function point" is defined as one end-user business function, such as a query for an input [IFPUG, 2000].

The primary terms used in FP counting are alphabetically listed as follows [IFPUG, 2000]:

Data Element Type (DET): A unique, user-recognizable, non-repeated field.

External Inputs (EIs): An elementary process in which data crosses the boundary from outside to inside. This data may come from a data-input screen or another application. The data may be used to maintain one or more internal logical files. The data is either control or business information. If the data is control information, it does not have to update an internal logical file.

[32] Used with permission from the International Function Point Users' Group (IFPUG) (http://www.ifpug.org/).

External Interface Files (EIFs): A user-identifiable group of logically related data that is used for reference purposes only. The data resides entirely outside the application and is maintained by another application. The external interface file is an internal logical file for another application.

External Inquiries (EQs): An elementary process with both input and output components that result in data retrieval from one or more ILFs and EIFs. The input process does not update any ILFs, and the output side does not contain derived data.

External Outputs (EOs): An elementary process in which derived data passes across the boundary from inside to outside. Additionally, an EO may update an ILF. The data creates reports or output files sent to other applications. These reports and files are created from one or more ILFs and EIFs.

Internal Logical Files (ILFs): A user-identifiable group of logically related data that resides entirely within the application's boundary and is maintained through external inputs.

Record Element Type (RET): A user-recognizable subgroup of data elements within an ILF or EIF.

14.2 Measurement Rules

The FP count for the APP system was outsourced to Charlie Tichenor, an IFPUG Certified Function Point Specialist.[33] The advantages of outsourcing the FP counting are [SCT, 1997]:

Expertise - The major FP consultants have experience with many organizations and diverse technologies. They can ensure FP analysis is properly utilized in the metrics program and the software development process as a whole.

Current Knowledge - Staying up to date with FP counting is a problem for most in-house practitioners. If they count a system only once every few months, their knowledge of more convoluted rules fades. Often they lack the time and budget to update their knowledge at IFPUG conferences or other training events.

Credibility - In many situations, credibility of the in-house counters is an issue. Outside consultants often have greater credibility due to their expertise and the currency of their

[33]In this study, FP counting was outsourced to a specialist whereas the CMM appraisal was conducted by the UMD research team. Indeed, a formal FP count is not as expensive as a formal CMM appraisal, i.e., a formal FP count remains affordable even within a limited budget.

information. Sometimes the mere fact that they are outsiders increases their credibility. An outside consultant should be an IFPUG Certified Function Point Specialist.

Consistency - Consistency is a key to successful use of any measure. One requirement for consistency is the use of a small group of counters who are in constant communication with one another regarding counting practices. Furthermore, this group must have ties to the FP counting community as a whole, that is, membership and participation in the IFPUG.

Independence - Bias can be a problem in FP counting. Project personnel may overstate counts because they may feel they will be judged on the size of their delivered system. Project customers may understate the size to push for quicker and cheaper delivery. There is a need for an unaffiliated third-party who is judged only on the accuracy of the count and any associated estimates. This is the role of an independent consultant.

Frees Resources - In many development groups, the counting is done by developers who have other project responsibilities. Often they are under pressure to continue with their other responsibilities. They often do not feel their job security or advancement is related to counting FPs. Outsourcing the counts can make both developers and their managers happier.

Versions of the IFPUG's FP counting manual preceding 1994 did not provide clear counting rules for real-time systems. As a consequence, the applicability of FPs to real-time systems was judged as questionable by many practitioners and researchers [Abran]. Counting rules specifically dedicated to the evaluation of real-time systems were added to versions of the manual published after 1994. These updated rules were used for the APP system FP count.

The total process to size FPs can be summarized by the following seven steps [Garmus, 2001]:

1. Determine the type of FP count.
2. Identify the counting scope and application boundary.
3. Identify all data functions (ILFs and EIFs) and their complexity.
4. Identify all transactional functions (EIs, EO, and EQs) and their complexity.
5. Determine the unadjusted FP count.
6. Determine the Value Adjustment Factor, which is based on the 14 general system characteristics.
7. Calculate the adjusted FP count.

Sections 14.2.1 to 14.2.5 provide a brief description on how to conduct FP counting (adopted from [Garmus, 2001]). For a complete description refer to [IFPUG, 2000].

14.2.1 Determining the Type of FP Count

The three types of FP counts are [Garmus, 2001]:

1. **Development Project**: Measures the functionality provided to end users with the first installation of the application.
2. **Enhancement Project**: Measures modifications to existing applications and includes the combined functionality provided to users by adding new functions, deleting old functions, and changing existing functions.
3. **Application**: Measures an installed application.

There are some minor differences between the three types [IFPUG, 2000].

14.2.2 Identifying the Counting Scope and Application Boundary

The counting scope defines the functionality that will be included in a particular FP count. [IFPUG, 2000]

The application boundary indicates the border between the software being measured and the user [IFPUG, 2000].

14.2.3 Identifying Data Functions and Their Complexity

Data functions represent the functionality provided to the user to meet internal and external data requirements. Data functions are either Internal Logical Files (ILFs) or External Interface Files (EIFs) [IFPUG, 2000].

In the analysis, these two components are ranked as *low*, *average*, or *high* complexity. The ranking is based on the number of Record Element Types (RETs) and the number of Data Element Types (DETs) [IFPUG, 2000].

A weight is assigned to these components by complexity level according to a rating matrix [IFPUG, 2000], which is summarized in Table 14.1.

Table 14.1 Rating Matrix for Five Components in Function Point Counting (Adapted from [IFPUG, 2000])

Type of component	Weight of components with complexity of		
	Low	Average	High
Internal Logical Files (ILFs)	× 7	× 10	× 15
External Interface Files (EIFs)	× 5	× 7	× 10
External Inputs (EIs)	× 3	× 4	× 6
External Outputs (EOs)	× 4	× 5	× 7
External Inquiries (EQs)	× 3	× 4	× 6

14.2.4 Identifying Transactional Functions and Their Complexity

Transactional functions represent the functionality provided to the user to process data. Transactional functions are either External Inputs (EIs), External Outputs (EOs), or External Inquiries (EQs) [IFPUG, 2000].

In the analysis, these three components are ranked as *low*, *average*, or *high* complexity. The ranking is based on the number of files updated or referenced (FTRs) and the number of Data Element Types (DETs) [IFPUG, 2000].

A weight is assigned to these components by complexity level according to the rating matrix summarized in Table 14.1[IFPUG, 2000].

14.2.5 Determining the Unadjusted Function Point Count

The Unadjusted Function Point Count (UFPC) reflects the specific functionality provided to the user by the project or application [IFPUG, 2000]. The UFPC is given by [IFPUG, 2000]:

$$\begin{aligned} Unadjusted\ function\ point \\ &= (No.of\ Internal\ Logic\ Files \\ &\times Weight\ of\ Internal\ Logic\ Files) \\ &+ (No.of\ External\ Logic\ Files \\ &\times Weight\ of\ External\ Logic\ Files) \\ &+ (No.of\ External\ Input \\ &\times Weight\ of\ External\ Input) \\ &+ (No.of\ External\ Output \\ &\times Weight\ of\ External\ Output) \\ &+ (No.of\ External\ Inquiries \\ &\times Weight\ of\ External\ Inquiries) \end{aligned} \tag{14.1}$$

14.2.6 Determining the Value Adjustment Factor

The Value Adjustment Factor (VAF) is based on 14 general system characteristics (GSCs) that comprise the general functionality of the application being counted.

Each characteristic has associated descriptions that help determine the degrees of influence of the characteristics. The degrees of influence range from 0 to 5, from no influence to strong influence, respectively [IFPUG, 2000].

The IFPUG Counting Practices Manual [IFPUG, 2000] provides detailed evaluation criteria for each of the GSCs. The list below provides an overview of each GSC.

1. **Data Communications**. The data and control information used in the application are sent or received over communication facilities.
2. **Distributed Data Processing**. Distributed data or processing functions are a characteristic of the application within the application boundary.
3. **Performance Application**. Performance objectives, stated or approved by the user, in either response or throughput, influence (or will influence) the design, development, installation, and support of the application.
4. **Heavily Used Configuration**. A heavily used operational configuration, requiring special design considerations, is a characteristic of the application.
5. **Transaction Rate**. The transaction rate is high and influences the design, development, installation, and support.
6. **Online Data Entry**. Online data entry and control information functions are provided in the application.

7. **End-User Efficiency**. The online functions provided emphasize a design for end-user efficiency.
8. **Online Update**. The application provides online update for the ILFs.
9. **Complex Processing**. Complex processing is a characteristic of the application.
10. **Reusability.** The application and the code in the application have been specifically designed, developed, and supported to be usable in other applications.
11. **Installation Ease**. Conversion and installation ease are characteristics of the application. A conversion and installation plan and/or conversion tools were provided and tested during the system test phase.
12. **Operational Ease**. Operational ease is a characteristic of the application. Effective start-up, backup, and recovery procedures were provided and tested during the system test phase.
13. **Multiple Sites**. The application has been specifically designed, developed, and supported for installation at multiple sites for multiple organizations.
14. **Facilitate Change**. The application has been specifically designed, developed, and supported to facilitate change.

Equation 14.2 converts the total degrees of influence assigned above to the Value Adjustment Factor [IFPUG, 2000] into the Value Adjustment Factor:

$$Value\ Adjustment\ Factor = (Total\ Degrees\ of\ Influence) \times 0.01 + 0.65 \quad (14.2)$$

14.2.7 Calculating the Adjusted Function Point Count

The Adjusted Function Point Count (AFPC) is calculated using Equation 14.3 for a development project, enhancement project, or application (system baseline) function point count [IFPUG, 2000]:

$$Adjusted\ Function\ Points = (Unadjusted\ Function\ Points) \times (Value\ Adjustment\ Factor) \quad (14.3)$$

The number of adjusted FPs, or simply "Function Points" (FPs), represents the size of the application and can be used to compute several measures discussed in other sections of this document.

14.3 Measurement Results

The following documents were used to count FPs for the APP system:

1. APP Module µp1 System SRS [APP, Y1]
2. APP Module µp1 Flux/Delta Flux/Flow Application SRS [APP, Y2]
3. APP Module µp2 System SRS [APP, Y3]
4. APP Module µp2 Flux/Delta Flux/Flow Application SRS [APP, Y4]
5. APP Module Communication Processor SRS [APP, Y5]

14.3.1 The Unadjusted Function Point

Table 14.2 and Table 14.3 list the measurement results of ILFs, EIFs, EIs, EOs, and EQs for the APP system from the IFPUG Certified Function Point Specialist, complying with the IFPUG Function Point Counting Practices Manual Release 4.1.1 [IFPUG, 2000].

The data shown in Table 14.2 and Table 14.3 can be used to count the unadjusted FPs of the five components, including ILFs, EIFs, EIs, EOs, and EQs (refer to Section 14.2.3 and Section 14.2.4), and thereby determine the unadjusted FPs of the entire system (refer to Section 14.2.5).

Table 14.2 Measurement Results of Data Functions for the APP System

ILF or EIF Descriptions	ILF				EIF			
	DET	RET	#	LVL*	DET	RET	#	LVL*
µp1	22	1	1	L				
µp2	22	1	1	L				
Set Points (Flux/Flow Imbalance Algorithm)	16	1	1	L				
Commands	< 50	1	1	L				
µp Cycle Timer							1	L
Communications Processor Cycle Timer							1	L
Input Range Table			1	L				
Flux/Flow/Imbalance Algorithm	< 50		1	L				
Trip Data Storage			1	L				

*Note: LVL stands for level of complexity.

Table 14.3 Measurement Results of Transaction Functions for the APP System[34]

Section	Descriptions	EIs		EOs		EQs	
		#	LVL*	#	LVL*	#	LVL*
Discrete Inputs	DIN1	1	L				
	DIN2	1	L				
	DIN3	1	L				
	DIN4	1	L				
	DIN5	1	L				
	DIN6	1	L				
	DIN7	1	L				
	DIN8	1	L				
	DIN9	1	L				
	DIN10	1	L				
	DIN11	1	L				
	DIN12	1	L				

*Note: LVL stands for level of complexity.

[34]There should be mostly empty cells in this form as only one kind of function is entered per row.

Table 14.3 Measurement Results of Transaction Functions for the APP System[35] (continued)

Section	Descriptions	EIs		EOs		EQs	
		#	LVL*	#	LVL*	#	LVL*
µp Diagnostics	Screen Display					1	A
	Main Program Running			1	L		
	Processor POST			1	L		
	Main Program Timeout			1	L		
	Dual Port RAM Test			1	L		
	RAM Test			1	L		
	Address Line test			1	L		
	PROM Checksum test			1	L		
	EEPROM Checksum test			1	L		
	Application Program test			1	L		
	Proc. Bd in Correct Slot			1	L		
	Installed Boards			1	L		
	Multiplexer/ADC test			1	L		
	Analog output Test			1	L		
	Discrete Input Test			1	L		
	TUNE mode			1	A		
	CAL mode			1	A		
Analog Inputs	AIN 1	1	L				
	AIN 2	1	L				
	AIN 3	1	L				
	AIN 4	1	L				
	AIN 5	1	L				
	AIN 6	1	L				
	AIN 7	1	L				
	Trip Reset Button	1	L				
	Key-Lock switch	1	L				

*Note: LVL stands for level of complexity.

[35]There should be mostly empty cells in this form as only one kind of function is entered per row.

Table 14.3 Measurement Results of Transaction Functions for the APP System[36] (continued)

Section	Descriptions	EIs		EOs		EQs	
		#	LVL*	#	LVL*	#	LVL*
Discrete Outputs	Trip 1			1	A		
	Trip 2-Trip 4 (Not Used)						
	Status 1			1	L		
	Status 2 (Not Used)						
	Aux1			1	L		
	Aux2			1	L		
	Aux3-6 (Not Used)						
Analog Outputs	AOUT1			1	L		
	AOUT2			1	L		
	AOUT3			1	L		
	AOUT4			1	L		
LED's	Processors are operating LED			1	L		
	Trip LED			1	L		
	MAINT LED			1	L		
Comm. Processor Diagnostics	RAM Test			1	L		
	Address Line test			1	L		
	PROM Checksum test			1	L		
	Processor Bd In Correct Slot			1	L		
	Test Bd in Correct Slot			1	L		
	Module date			1	L		
	Module time			1	L		
	TEST mode			1	L		
	Online RAM Test			1	L		
	Online Address Line test			1	L		
	Online PROM Checksum test			1	L		

*Note: LVL stands for level of complexity.

[36]There should be mostly empty cells in this form as only one kind of function is entered per row.

Table 14.3 Measurement Results of Transaction Functions for the APP System[37] (continued)

Section	Descriptions	EIs		EOs		EQs	
		#	LVL*	#	LVL*	#	LVL*
APP Processing	Initialization	1	A				
	Power-on self test (counted)						
	Main Program (counted)						
	Update Dual port RAM	1	L				
	Calibrate and tune (counted)						
	Read Discrete inputs and analog outputs(counted)						
	Application (counted)						
	Generate discrete and analog outputs(counted)						
	Output refresh (On/Off)			1	L		
Application	Flux/Flow/Imbalance algorithm (counted)						
Comm. Processor	Slot ID	1	L				
	ID Chip	1	L				
	Initialization	1	A				
	Power-on self test	1	L				

*Note: LVL stands for level of complexity.

Table 14.4 summarizes the numbers of ILFs, EIFs, EIs, EOs, and EQs for three complexity levels (*Low*, *Average*, and *High*) based on the data in Table 14.2 and Table 14.3.

[37]There should be mostly empty cells in this form as only one kind of function is typed in per row.

Table 14.4 The Counts of Components with Different Complexity Level

Type of component	Number of components with complexity of		
	Low	Average	High
Internal Logical Files (ILFs)	7	0	0
External Interface Files (EIFs)	2	0	0
External Inputs (EIs)	25	2	0
External Outputs (EOs)	32	6	0
External Inquiries (EQs)	1	1	0

Table 14.5 summarizes the unadjusted FPs of ILFs, EIFs, EIs, EOs, and EQs based on the data (the numbers of the five components) in Table 14.4 and the data (the weights of the five components for three different complexity levels) in Table 14.1.

The total unadjusted FPs for the APP system is 307.

Table 14.5 The Counts of the Unadjusted Function Points

Type of component	Unadjusted function points of components with complexity of			Sum of unadjusted FPs
	Low	Average	High	
Internal Logical Files	7 × 7 = 49	0 × 10 = 0	0 × 15 = 0	49
External Interface Files	2 × 5 = 10	0 × 7 = 0	0 × 10 = 0	10
External Inputs	25 × 3 = 75	2 × 4 = 8	0 × 6 = 0	83
External Outputs	32 × 4 = 128	6 × 5 = 30	0 × 7 = 0	158
External Inquiries	1 × 3 = 3	1 × 4 = 4	0 × 6 = 0	7
Total Unadjusted FP Count for the APP system				307

14.3.2 The Value Adjustment Factor

Table 14.6 presents the measurement results of the General System Characteristics for the APP system. The results were obtained from the IFPUG Certified Function Point specialist, who complied with the IFPUG Function Point Counting Practices Manual Release 4.1.1 [IFPUG, 2000].

Table 14.6 Measurement Results of General System Characteristics for the APP System

General System Characteristics	Degree of Influence
Data Communications	4
Distributed Processing	4
Performance	4
Heavily Used Configuration	1
Transaction Rates	0
Online Data Entry	5
End-User Efficiency	2
Online Update	4
Complex Processing	1
Reusability	1
Installation Ease	0
Operational Ease	5
Multiple CPU Sites	0
Facilitate Change	2
Total Degree of Influence	**33**

According to Equation 14.2, the Value Adjustment Factor (VAF) is:

$$VAF = 33 \times 0.01 + 0.65 = 0.98 \quad (14.4)$$

14.3.3 The Adjusted Function Point

According to Equation 14.3, the value of the adjusted FPs for the APP system is:

$$Adjusted\ FPs = 307 \times 0.98 = 300.8 \quad (14.5)$$

which is rounded up to 301.

14.4 RePS Construction from Function Point

Two steps are required to estimate software reliability using the FP measure:

1. Estimate the number of delivered defects based on the FP measurement (refer to Section 14.4.1)
2. Calculate the reliability using Musa's Exponential Model (refer to Section 14.4.2)

14.4.1 Estimating the Number of Delivered Defects

There is no proposed model in the literature linking FP to the estimated total number of delivered defects. However, there is data for the state-of-the-practice of the U.S. averages for delivered defects summarized in [Jones, 1996]. This data links the FP to the number of defects per FPs for different categories of applications. The definitions of different types of software systems are given as follows [Jones, 1996]:

End-user software: applications written by individuals who are neither professional programmers nor software engineers.

Management information system (MIS): applications produced by enterprises in support of their business and administrative operations, e.g., payroll systems, accounting systems, front- and back-office banking systems, insurance claims handling systems, airline reservation systems, and so on.

Outsourced and contract software: outsourced software is software produced under a blanket contract by which a software-development organization agrees to produce all, or specific categories, of software for the client organization. Contract software is a specific software project that is built under contract for a client organization.

Commercial software: applications that are produced for large-scale marketing to hundreds or even millions of clients. Examples of commercial software are Microsoft Word, Microsoft Excel, etc.

System software: software that controls physical devices. They include the operating systems that control computer hardware, network switching systems, automobile fuel-injection systems, and other control systems.

Military software: software produced for a uniformed military service.

Furthermore, only defects of Severity 1 and Severity 2—called *critical defects* and *significant defects*—should be considered when estimating software reliability.

14.4.1.1 Estimating the Total Number of Delivered Defects

Table 14.7 (Table 3.46 in [Jones, 1996]) provides the average numbers for delivered defects per FP for different types of software systems.

Table 14.7 Averages for Delivered Defects Per Function Point
(Extracted From Table 3.46 in [Jones, 1996])

FPs	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	N/A	0.85	0.59	0.4	0.36	0.47	0.56
10000	N/A	1.5	0.83	0.6	0.49	0.68	0.84
100000	N/A	2.54	1.3	0.9	0.8	0.94	1.33
Average	**0.23**	**0.90**	**0.49**	**0.36**	**0.30**	**0.39**	**0.53**

The APP system software falls into the category of "system software" according to the previous definitions.

The FP count for the APP system is 301 (100 < 301 < 1000), as calculated in Section 14.3.3.

Therefore, according to Table 14.7, the delivered defect density (the number of total delivered defects per FP) for the APP system is calculated using logarithmic interpolation:

$$DDD(APP) = 0.1 + \frac{0.36 - 0.1}{\log_{10}(1000) - \log_{10}(100)} \times [\log_{10}(301) - \log_{10}(100)] = 0.2244 \quad (14.6)$$

where

$DDD(APP)$ = the delivered defect density for the APP system in defects/FP.

The number of total delivered defects for the APP system is given by:

$$N_{FP,total} = DDD(APP) \times FP(APP) = 0.2244 \times 301 = 67.54 \quad (14.7)$$

where

$N_{FP,total}$	the number of total delivered defects for the APP system.
$DDD(APP)$	the delivered defect density for the APP system. $DDD(APP) = 0.2244$ defects/FP.
$FP(APP)$	the FP count for the APP system. $FP(APP) = 301$ (refer to Section 14.3.3).

14.4.1.2 Estimating the Number of Delivered Critical and Significant Defects

Table 14.8 (Table 3.48 in [Jones, 1996]) presents U.S. averages for percentages of delivered defects by severity levels.

Using Table 14.8 and logarithmic interpolation, the percentages of delivered defects by severity level can be obtained. For example, the percentage of delivered defects of severity 1 corresponding to FP = 301 (100 < 301 < 1000) is:

$$DDD(APP) = 0.0256 - \frac{0.0256 - 0.0108}{\log_{10}(1000) - \log_{10}(100)} \times [\log_{10}(301) - \log_{10}(100)] = 0.0185 \quad (14.8)$$

Table 14.8 Averages for Delivered Defects by Severity Level
(Adapted From Table 3.48 in [Jones, 1996])

FPs	Percentage of Delivered Defects by Severity Level			
	Severity 1	**Severity 2**	**Severity 3**	**Severity 4**
1	0	0	0	0
10	0	0	1	0
100	0.0256	0.1026	0.359	0.5128
1000	0.0108	0.1403	0.3993	0.4496
10000	0.015	0.145	0.5	0.34
100000	0.02	0.12	0.5	0.36
Average	0.0179	0.1270	0.5517	0.4156

Given the total number of delivered defects, $N_{FP,total}$ (refer to Section 14.4.1.1), and the percentages of delivered defects by severity level (refer to Table 6.7), the number of delivered defects by severity level can be calculated. For example, the number of delivered defects of severity 1 for the APP system is: 67.54 × 0.0185 = 1.249.

Table 14.10 presents the numbers of delivered defects by severity level for the APP system.

Table 14.9 Number of Delivered Defects by Severity Level for the APP System

	Severity 1 (critical)	**Severity 2 (significant)**	**Severity 3 (minor)**	**Severity 4 (cosmetic)**
Number of delivered defects	1.249	8.1	25.6	32.6

For the APP system, the number of delivered defects of severity 1 is 1.249 and the number of delivered defects of severity 2 is 8.1.

14.4.2 Reliability Calculation from Delivered Critical and Significant Defects

The probability of success-per-demand is obtained using Musa's exponential model [Musa, 1990] [Smidts, 2004]:

$$p_s(FP) = exp\left(\frac{-K \times N_{FP} \times \tau}{T_L}\right) \quad (14.9)$$

and

$$N_{FP} = N_{FP,critical} + N_{FP,significant} \quad (14.10)$$

where

$p_s(FP)$	Reliability estimation for the APP system using the FP measure.
K	Fault Exposure Ratio, in failure/defect.
N_{FP}	Number of defects estimated using the FP measure.
τ	Average execution-time-per-demand, in seconds/demand.
T_L	Linear execution time of a system, in seconds.
$N_{FP,critical}$	Number of delivered *critical defects* (severity 1).
$N_{FP,significant}$	Number of delivered *significant defects* (severity 2).

Since *a priori* knowledge of the defect locations and their impact on failure probability is unknown, the average K value given in [Musa, 1987] [Musa, 1990] must be used: 4.2×10^{-7} failure/defect.

For the APP system, $N_{FP,critical} = 1.3$, and $N_{FP,significant} = 8.1$, as calculated in Section 14.4.1.2. Therefore, according to Equation 14.10, $N_{FP} = 1.3 + 1.8 = 9.4$.

The *linear execution time*, T_L, is usually estimated as the ratio of the execution time and the software size on a single microprocessor basis [Musa, 1987] [Musa, 1990] [Smidts, 2004]. In the case of the APP system, however, there are three parallel subsystems (μp1, μp2, and CP), each of which has a microprocessor executing its own software. Each of these three subsystems has an estimated linear-execution time. Therefore, there are several ways to estimate the linear-execution time for the entire APP system, such as using the average value of these three subsystems.

For a safety-critical application like the APP system, the UMD research team suggests a conservative estimation of T_L by using the minimum of these three subsystems' values. Namely,

$$T_L = \min\{T_L(\mu p1), T_L(\mu p2), T_L(CP)\} = \min\{0.018, 0.009, 0.021\} = 0.009 \text{ second} \quad (14.11)$$

where

$T_L(\mu p1)$ Linear execution time of Microprocessor 1 (μp1) of the APP system. T_L *(μp1)* = 0.018 second (refer to Chapter 17).

$T_L(\mu p2)$ Linear execution time of Microprocessor 2 (μp2) of the APP system. T_L *(μp2)* = 0.009 second (refer to Chapter 17).

$T_L(CP)$ Linear execution time of Communication Microprocessor (CP) of the APP system. T_L *(CP)* = 0.021 second (refer to Chapter 17).

Similarly, the *average execution-time-per-demand*, τ, is also estimated on a single-microprocessor basis. Each of the three subsystems in APP has an estimated average execution-time-per-demand. To make a conservative estimation, the average execution-time-per-demand for the entire APP system is the maximum of the three subsystems' values. Namely,

$$\tau = \max\{\tau(\mu p1), \tau(\mu p2), \tau(CP)\} = \max\{0.082, 0.129, 0.016\} = 0.129 \text{ seconds/demand} \quad (14.12)$$

where

$T_L(\mu p1)$ Average execution-time-per-demand of Microprocessor 1 (μp1) of the APP system. *τ(μp1)* = 0.082 seconds/demand (refer to Chapter 17).

$T_L(\mu p2)$ Average execution-time-per-demand of Microprocessor 2 (μp2) of the APP system. *τ(μp2)* = 0.129 seconds/demand (refer to Chapter 17).

$T_L(CP)$ Average execution-time-per-demand of Communication Microprocessor (CP) of the APP system. *τ(CP)* = 0.016 seconds/demand (refer to Chapter 17).

Thus, the reliability for the APP system using the FP measure is given by:

$$p_s(FP) = exp\left(\frac{-K \times N_{FP} \times \tau}{T_L}\right) = 0.999943414 \quad (14.13)$$

14.5 Lessons Learned

The measurement of FP can be systematically conducted based on the rules published by IFPUG. As for BLOC, CMM, and CC, empirical industry data was used to build correlations between the

value of FP and the number of defects residing in the software. Thus, reliability-prediction results based on FP are not as good as the ones obtained from other measures which deal with the real defects of the application.

14.6 References

[APP, Y1] "APP Module First Safety Function Processor SRS," Year Y1.

[APP, Y2] "APP Flux/Delta Flux/Flow Application SRS for SF1," Year Y2.

[APP, Y3] "APP Module μp2 System Software SRS," Year Y3.

[APP, Y4] "APP μp2 Flux/Delta Flux/Flow Application Software SRS," Year Y4.

[APP, Y5] "APP Module Communication Processor SRS," Year Y5.

[Garmus, 2001] D. Garmus and D. Herron. "Function Point Analysis: Measurement Practices for Successful Software Project," Addison-Wesley, 2001.

[Heller, 1996] R. Heller. "An Introduction to Function Point Analysis," in *Newsletter from Process Strategies*, 1996.

[IEEE 982.2, 1988] "IEEE Guide for the use of Standard Dictionary of Measures to Produce Reliable Software," IEEE Std. 982.2-1988, 1988.

[IFPUG, 2000] IFPUG, "Function Point Counting Practices Manual (Release 4.1.1)," International Function Point Users Group, 2000.

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

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

[Musa, 1987] J.D. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Applications*. New York: McGraw-Hill, 1987.

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

[Pressman, 1992] R. Pressman. *Software Engineering: A Practitioner's Approach*. New York: McGraw-Hill, 1992.

[SCT, 1997] Software Composition Technologies, "Frequently Asked Questions (and Answers) Regarding Function Point Analysis," Software Composition Technologies, Inc., Available: http://ourworld.compuserve.com/homepages/softcomp/fpfaq.htm [Jun. 25, 1997].

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

15. REQUIREMENTS SPECIFICATION CHANGE REQUEST

Requirements evolution is considered one of the most critical issues in developing computer-based systems. The sources of changes may come from dynamic environments such as a changing work environment, changes in government regulations, organizational complexity, and conflict among stakeholders in deciding on a core set of requirements [Barry, 2002].

The *requirements specification change request measure*, denoted by *RSCR*, indicates the stability and/or growth of the functional requirements. Moreover, it provides an additional view of the effectiveness of the functional specification process used and has the potential of adding credibility to the product [Smidts, 2000].

It has been observed that a significant cause of project failure and poor quality in software systems is frequent changes to requirements. RSCR is an indication of the quality of the resulting software system. Evidence suggests that the system quality decreases as the size of requirements specification change requests increases [Smidts, 2000].

However, RSCR can not reflect the contents of requirements specification change requests. Based on the results from applying the requirements specification change requests measurement to the APP system, the UMD research team does not recommend using RSCR to estimate the reliability of a software product.

Instead, the UMD research team suggests using a derived measure, the Requirements Evolution Factor (REVL), which links requirements specification change requests to the changed source code. REVL can be used to estimate the reliability of a software product, as described in Section 15.4. REVL has not been validated thoroughly to date.

RSCR and REVL are related in the sense that both measures reflect the effect of changes to requirements that occur during the software development life cycle after requirements have been frozen. However, REVL may yield a better estimation of impact than RSCR because, in REVL, the size of code impacted is incorporated into the measure.

RSCR can be applied as soon as the requirements are available. As listed in Table 3.3, the applicable life cycle phases for this measure are Requirements, Design, Code, Testing and Operation.

REVL, on the other hand, is not available until the delivery of the source code.

15.1 Definition

The *requirements specification change request measure (RSCR)* is defined as the number of change requests that are made to the requirements specification. The requested changes are counted from the first release of the requirements specification document to the time when the product begins its operational life. Thus, RSCR is defined as [Smidts, 2000]:

$$RSCR = \sum(requested\ changes\ to\ the\ requirements\ specification) \tag{15.1}$$

where the summation is taken over all requirements change requests initiated during the software development life cycle (after the first release of the requirements specification document). It should be noted that the definitions of RSCR published in the software-engineering literature fail to clearly state what type of requirements (functional or non-functional requirements) should be included in the RSCR count.

Most of the non-functional requirements are not as important as the functional requirements. They do not describe what the software will do, but how the software will perform its functions. Normally, non-functional requirements are not included in the evaluation of reliability based on requirements change requests. However, in certain cases, non-functional requirements hide what really are functional requirements or may describe characteristics that are critical such as response time. These special cases should be identified by the analyst and included in the measurement. In this research, some of the non-functional requirements for the APP system such as the timing requirements are also crucial. Thus, such implied functional requirements in the non-functional requirements section also are considered.

RSCR only quantifies the "number" of requirements specification change requests, and can be used as an indicator of the stability and/or growth of the functional requirements. However, RSCR cannot reflect the contents of requirements specification change requests. Therefore, it is inappropriate to use RSCR to estimate the reliability of a software system.

To link requirements specification change requests to the reliability of a software system, the UMD research team recommends a derived measure called REVL, which is defined as:

$$REVL = \frac{SIZE_{changed\ due\ to\ RSCR}}{SIZE_{delivered}} \times 100\% \tag{15.2}$$

where

$REVL$	measure of requirements Evolution and Volatility Factor
$SIZE_{changed\ due\ to\ RSCR}$	size of changed source code corresponding to requirements specification change requests, in Kilo Line of Code (KLOC)
$SIZE_{delivered}$	size of the delivered source code, in *KLOC*

The concept of Requirements Evolution and Volatility Factor was originally proposed in [Boehm, 1982] and further developed in [Boehm, 2000] for the purpose of estimating the development effort of a software project at the early stages of the development life cycle. UMD quantified REVL based on [Boehm, 2000] and [Stutzke, 2001], as shown in Equation 15.2.

The size of changed source code corresponding to requirements specification change requests is given by

$$SIZE_{changed\ due\ to\ RSCR} = SIZE_{added\ due\ to\ RSCR} + SIZE_{deleted\ due\ to\ RSCR} + SIZE_{modified\ due\ to\ RSCR} \quad (15.3)$$

where

$SIZE_{added\ due\ to\ RSCR}$	size of added source code corresponding to requirements specification change requests, in *KLOC*
$SIZE_{deleted\ due\ to\ RSCR}$	size of deleted source code corresponding to requirements specification change requests, in *KLOC*
$SIZE_{modified\ due\ to\ RSCR}$	size of modified source code corresponding to requirements specification change requests, in *KLOC*

15.2 Measurement Rules

Five steps are required to measure the impact of Requirements Evolution and Volatility Factor on the reliability of a software system:

1. Identify requirements specification change requests during the software development life cycle
2. Identify the changed source code corresponding to requirements specification change requests
3. Measure the size of the changed source code corresponding to requirements specification change requests
4. Calculate REVL

A comparison between the first and last version of the source code will not result in a correct measurement of REVL because some of the code changes do not correspond to requirements specification change requests but instead to code fixes related to coding or design errors.

15.2.1 Identifying Requirements Specification Change Requests

A *requirements specification change request* has the following essential attributes:

- It is an authorized change of the SRS
- It is a change of the functional requirements of the software
- It is a documented change of requirements, usually in the final version of the SRS
- It is proposed between the release of the first version of the SRS and the time the software product is delivered to the customer

For example, "Changed MVOLT to mvolt" ([APP, Y1], Page 2) is not considered as a requirements specification change request because it is not a change of the functional requirements of the software.

"Changed Analog Inputs = 14 to Analog Inputs = 28" ([APP, Y1], Page 11) is regarded as a requirements specification change request.

The counting rule for RSCR is to count the number of identified software functional requirements change requests.

RSCR is counted for the purpose of comparison between RSCR and REVL, as described in Section 15.3. It is not used when constructing the RePS based on REVL, as described in Section 15.4.

15.2.2 Identifying the Changed Source Code Corresponding to RSCR

The changed source code corresponding to requirements specification change requests is identified by mapping all requirements specification change requests identified in the previous step to the delivered source code. Mapping a requirements specification change request to source code means linking the changed functional requirement(s) to the affected line(s) of the source code.

The mapping relationships between the source code and a requirements specification change request may be one-to-one, one-to-many, or many-to-one.

15.2.3 Measuring the Size of the Changed Source Code Corresponding to RSCR

The changes of source code due to requirements specification change requests are divided into three categories: added, deleted, and modified.

It should be noted that not all changed source code but only the changes corresponding to requirements specification change requests should be considered while counting the following

three quantities: $SIZE_{\text{added due to RSCR}}$, $SIZE_{\text{deleted due to RSCR}}$, and $SIZE_{\text{modified due to RSCR}}$ (see Section 15.1).

The rules to measure the size of the changed source code are the same as those used to measure the size of the source code for the BLOC measure (See Section 6.2).

The size of the changed source code corresponding to requirements specification change requests is calculated according to Equation 15.3.

15.2.4 Calculating REVL

REVL is calculated by applying Equation 15.2 to the results obtained in Section 15.2.3.

15.3 Measurement Results

The following documents were used to measure RSCR and REVL:

- APP Module μp1 System SRS [APP, Y1]
- APP Module μp1 Flux/Delta Flux/Flow Application SRS [APP, Y2]
- APP Module μp2 System SRS [APP, Y3]
- APP Module μp2 Flux/Delta Flux/Flow Application SRS [APP, Y4]
- APP Module Communication Processor SRS [APP, Y5]
- APP Module μp1 System source code [APP, Y6]
- APP Module μp1 Flux/Delta Flux/Flow Application source code [APP, Y7]
- APP Module μp2 System source code [APP, Y8]
- APP Module μp2 Flux/Delta Flux/Flow Application source code [APP, Y9]
- APP Module Communication Processor System source code [APP, Y10]

The APP system has five components: the μp1 System, the μp1 Application, the μp2 System, the μp2 Application, and the CP System. The measurement results for the APP system are presented in Table 15.1 (see Section 15.2 for the measurement rules).

From Table 15.1, one may notice that the size of the changed source code corresponding to requirements specification change requests is not proportional to RSCR. For example, RSCR for the μp2 System is 7 and the size of the correspondingly changed source code is 72 LOC, whereas RSCR for the μp1 System is 26 and the size of the correspondingly changed source code is 27 LOC.

REVL and RSCR are not linearly related because:

- Requirements specification change requests may have different levels of granularity. Consequently, some requirements specification change requests lead to changing more

lines of source code than others. This is also why RSCR is not good at capturing the impact of requirements specification change requests on the software product.

- A requirements specification change request may affect multiple functions in the source code ("one-to-many"). This occurs if the code contains multiple implementations of the same function.
- Multiple requirements specification change requests may correspond to the same line(s) of changed source code ("many-to-one").

Despite the benefits exhibited by REVL, the following limitations of REVL also should be noted and understood:

- REVL does not capture requirements specification change requests proposed in the requirements analysis phase because these changes are invisible from the point of view of the source code.
- REVL does not capture requirements specification change requests proposed in the design phase because these changes, too, are invisible from the point of view of the source code.

Table 15.1 Measurement Results for RSCR and REVL for the APP System

	CP System	**µp1 System**	**µp1 Application**	**µp2 System**	**µp2 Application**
RSCR	4	26	14	7	5
$SIZE_{\text{delivered}}$, *in KLOC*	1.21	2.034	0.48	0.895	0.206
$SIZE_{\text{added due to RSCR}}$, *in KLOC*	0	0.003	0.003	0.006	0
$SIZE_{\text{deleted due to RSCR}}$, *in KLOC*	0.129	0.007	0	0.003	0
$SIZE_{\text{modified due to RSCR}}$, *in KLOC*	0	0.027	0.011	0.072	0.008
$SIZE_{\text{changed due to RSCR}} = SIZE_{\text{added due to RSCR}} + SIZE_{\text{deleted due to RSCR}} + SIZE_{\text{modified due to RSCR}}$ *(in KLOC)*	0.129	0.037	0.014	0.081	0.008
$REVL = \frac{SIZE_{changed\ due\ to\ RSCR}}{SIZE_{delivered}} \times 100\%$	10.7%	1.8%	2.9%	9.1%	3.9%

Further development of REVL is required for quantifying the impact of requirements specification change requests at the early stages of the development life cycle.

To resolve this issue, the UMD research team suggests linking requirements specification change requests to the affected function points and quantifying the impact of this change on defect density through empirical analysis or expert opinion elicitation.

15.4 RePS Construction Based On REVL

Currently there are three approaches found in the literature that attempt to estimate the fault content of a software system based on requirements volatility. These only focus on linking requirements volatility to the changed source code, partly because it is too difficult to quantify the impact of requirements specification change requests at the design phase, as discussed in Section 15.3.

The first approach is to link requirements volatility to the defect density of the source code, assuming that the software has been modified in response to changed functional requirements and that the modification process is imperfect [Malayia, 1998].

The second approach is to use *Code Churn* to estimate the impact of code changes corresponding to requirements specification change requests [Munson, 2003].

The third approach is to use the Success Likelihood Index Methodology (SLIM) to integrate the human analysis of the Performance Influencing Factors [Stutzke, 2001], as described in Section 11.4.1.

Due to the difficulty in obtaining data required to estimate the model parameters of Malayia's and Munson's approaches [Malayia, 1998] [Munson, 2003], the third approach was adopted.

Four steps are required to estimate the reliability of a software product using SLIM [Stutzke, 2001]:

1. Measure REVL, as described previously, and other Performance Influencing Factors, as described in Section 11.2.1 to 11.2.9.
2. Estimate SLI for requirements Evolution and Volatility Factor.
3. Estimate the fault content in the delivered source code using SLIM, as described below.
4. Calculate reliability using Musa's Exponential Model, as described below.

15.4.1 Estimating the Value of SLI for Requirements Evolution and Volatility Factor

Requirements Evolution and Volatility Factor was regarded as one of the Performance Influencing Factors (PIFs) leading to the success or failure of a project [Jones, 1995].

The effect of PIFs on software development can be quantified by a Success Likelihood Index (SLI), which ranges from 0 (error is likely) to 1 (error is not likely) [Stutzke, 2001].

SLI for the Requirements Evolution and Volatility Factor, denoted by SLI_{10}, is estimated using the value of REVL, as shown in Table 15.2. If necessary, piecewise linear interpolation is used.

The SLI_{10} scale for REVL (in Table 15.2) is based on COCOMO II [Boehm, 2000]. The assumption made for the SLI_{10} ratings is that the relationship between REVL and SLI is an S-shaped curve, as shown in Figure 15.1.

Further investigation is required to validate the relationship between SLI_{10} and REVL.

The values of SLI_{10} for the five components of the APP system are summarized in Table 15.3. For example, REVL for the μp1 Application is 1.8%, as determined in Table 15.1, which is less than 5%. According to Table 15.2, the value of the SLI_{10} is 1 when $REVL \leq 5\%$. Therefore, the value of SLI_{10} for the μp1 Application is 1.

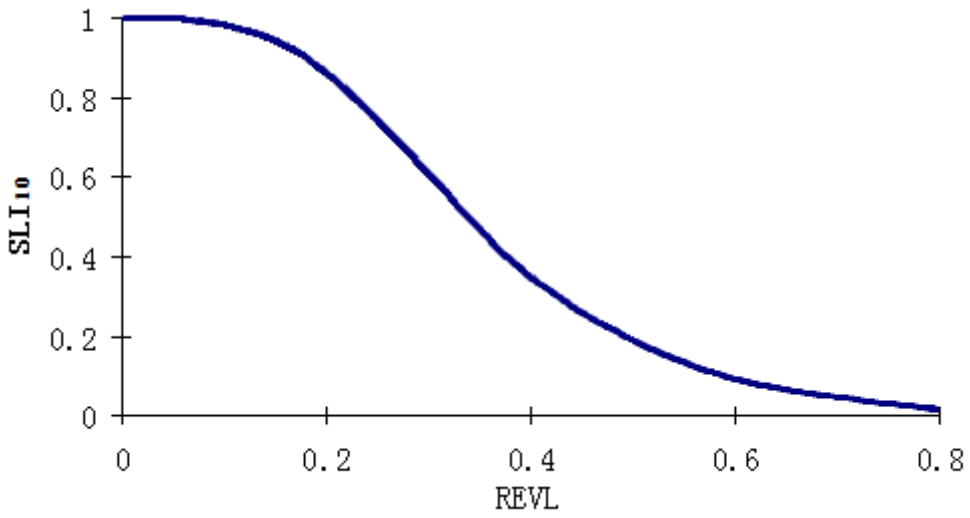


Figure 15.1 Relationship between SLI_{10} and REVL

Table 15.2 Rating Scale and SLI Estimation for REVL

REVL Descriptors	**5%**	**20%**	**35%**	**50%**	**65%**	**80%**
Rating Levels	Very Low	Low	Nominal	High	Very High	Extra High
Value of SLI_{10}	1	0.75	0.5	0.34	0.16	0

15.4.2 Estimating the Fault Content in the Delivered Source Code

The fault content of the source code is given by (see Section 11.4.1 for details):

$$N = 0.036 \times SIZE \times (20)^{1-2\times SLI} \tag{15.4}$$

where

N	number of faults remaining in the delivered source code
$SIZE$	size of the delivered source code in terms of *LOC*
SLI	Success Likelihood Index of a software product

According to Equation 15.4, the fault content varies with SLI: the fault content is maximum when SLI = 0 and minimum when SLI = 1, as shown in Equation 15.5 and 15.6:

$$N_{max} = 0.72 \times SIZE, when\ SLI = 0 \tag{15.5}$$

$$N_{min} = 0.0018 \times SIZE, when\ SLI = 1 \tag{15.6}$$

To validate the expert-opinion-based ranking [Smidts, 2004], where the target measure must be isolated from other measures, the SLI of a software product is represented by that of REVL; i.e.:

$$SLI = SLI_{10} \tag{15.7}$$

However, the UMD research team recommends using other measures in addition to REVL while using SLIM to estimate the source code fault content because this method usually yields more accurate results. The SLI of a software product is given by the weighted sum of all PIF SLIs:

$$SLI = \sum_{i=1}^{10} W_i SLI_i \tag{15.8}$$

Table 15.3 summarizes both SLI values and the fault content of the delivered source code with and without using the supportive measures, respectively. In Table 15.3, the values of SLIs for the five components of the APP system are found in Table 11.30 (Row 5).

Table 15.3 Summary of Fault-Content Calculation

		CP System	µp1 System	µp1 Application	µp2 System	µp2 Application
LOC		1210	2034	480	895	206
Without using supportive measures	$SLI = SLI_{10}$	0.9067	1	1	0.9317	1
	Number of defects in source code	3.8	3.7	0.9	2.4	0.4
Using supportive measures	$SLI = \sum_{i=1}^{10} W_i SLI_i$	0.7175	0.6952	0.6539	0.7377	0.7441
	Number of defects in source code	11.8	22.7	6.9	7.8	1.7

The estimated number of faults in the entire APP system based on the requirements specification change request measurement is:

$$N_{REVL} = 3.8 + 3.7 + 0.9 + 2.4 + 0.4 = 11.2\ (without\ using\ the\ supportive\ measures) \quad (15.9)$$

or

$$N_{REVL} = 11.8 + 22.7 + 6.9 + 7.8 + 1.7 = 50.9\ (using\ the\ supportive\ measures) \quad (15.10)$$

15.4.3 Calculating Reliability Using the Defect Content Estimation

The probability of success-per-demand is obtained using Musa's exponential model [Musa, 1990] [Smidts, 2004]

$$p_s(REVL) = \exp - \left(\frac{K \times N_{REVL} \times \tau}{T_L}\right) \quad (15.11)$$

where

$p_s(REVL)$ Reliability estimation for the APP system based on REVL

K Fault Exposure Ratio, in failure/defect

N_{REVL}	Number of defects estimated based on REVL
τ	Average execution-time-per-demand, in seconds/demand
T_L	Linear execution time of a system, in seconds

Since *a priori* knowledge of defect locations and their impact on failure probability is not known, the average K value given in [Musa, 1990] must be used: 4.2×10^{-7} failure/defect.

For the APP system, $N_{REVL} = 11.2$ (without using the supportive measures), and $N_{REVL} = 50.9$ (using the supportive measures), as calculated in Section 15.4.2.

The *linear execution time*, T_L, is usually estimated as the ratio of the execution time and the software size on a single microprocessor basis [Musa, 1990] [Smidts, 2004]. In the case of the APP system, however, there are three parallel subsystems (μp1, μp2, and CP), each of which has a microprocessor executing its own software. Each of these three subsystems has an estimated linear-execution time. Therefore, there are several ways to estimate the linear-execution time for the entire APP system, such as using the average value of these three subsystems.

For a safety-critical application, such as the APP system, the UMD research team suggests making a conservative estimation of T_L by using the minimum of these three subsystems' values. Namely,

$$\begin{aligned} T_L &= \min\{T_L(\mu p1), T_L(\mu p2), T_L(CP)\} \\ &= \min\{0.018, 0.009, 0.021\} \\ &= 0.009\ second \end{aligned} \tag{15.12}$$

where

$T_L(\mu p1)$	Linear execution time of Microprocessor 1 (μp1) of the APP system. T_L *(μp1)* = 0.018 second, as determined in Chapter 17;
$T_L(\mu p2)$	Linear execution time of Microprocessor 2 (μp2) of the APP system. T_L *(μp2)* = 0.009 second, as determined in Chapter 17;
$T_L(CP)$	Linear execution time of Communication Microprocessor (CP) of the APP system. T_L *(CP)* = 0.021 second, as determined in Chapter 17.

Similarly, the *average execution-time-per-demand*, τ, is estimated on a single microprocessor basis. Each of the three subsystems in APP has an estimated average execution-time-per-demand. To make a conservative estimation, the average execution-time-per-demand for the entire APP system is the maximum of the three subsystems' values. Namely,

$$\begin{aligned}\tau &= \max\{\tau(\mu p1), \tau(\mu p2), \tau(CP)\} \\ &= \max\{0.082, 0.129, 0.016\} \\ &= 0.129 \text{ second/demand}\end{aligned} \quad (15.13)$$

where

$T_L(\mu p1)$ Average execution-time-per-demand of Microprocessor 1 (μp1) of the APP system. *τ(μp1)* = 0.082 second/demand, as determined in Chapter 17;

$T_L(\mu p2)$ Average execution-time-per-demand of Microprocessor 2 (μp2) of the APP system. *τ(μp2)* = 0.129 second/demand, as determined in Chapter 17;

$T_L(CP)$ Average execution-time-per-demand of Communication Microprocessor (CP) of the APP system. *τ(CP)* = 0.016 second/demand, as determined in Chapter 17.

Thus the reliability of the APP system based on REVL is given by:

$$p_s(REVL) = \exp\left[\frac{(-4.2 \times 10^{-7})(11.2)(0.129)}{0.009}\right] = 0.999933 \quad (15.14)$$

without using supportive measures, or

$$p_s(REVL) = \exp\left[\frac{(-4.2 \times 10^{-7})(50.9)(0.129)}{0.009}\right] = 0.999694 \quad (15.15)$$

with using supportive measures.

15.5 Lessons Learned

Empirical industry data was used to build the relation between REVL/RSCR and the number of defects residing in the software. Thus, reliability-prediction results based on REVL/RSCR are not as good as those obtained from other measures which deal with actual defects in the application.

A more accurate estimation of reliability based on REVL for the APP system can be obtained by:

1. Obtaining better documentation on requirements change requests;
2. Collecting data to estimate the SLI of the REVL factor for safety-critical applications;
3. Combining REVL with RSCR for quantifying the impact of requirements specification change requests;
4. Measuring REVL at the sub-system level.
5. Enhancing the estimation of K. A value of K for the safety-critical system, rather than the average value failure/defect, should be used in Equation 15.14 and 15.15;

15.6 References

[APP, 01] *APP Instruction Manual*.

[APP, Y1] "APP Module First Safety Function Processor SRS," Year Y1.

[APP, Y2] "APP Flux/Delta Flux/Flow Application SRS for SF1," Year Y2.

[APP, Y3] "APP Module μp2 System Software SRS," Year Y3.

[APP, Y4] "APP μp2 Flux/Delta Flux/Flow Application Software SRS," Year Y4.

[APP, Y5] "APP Module Communication Processor SRS," Year Y5.

[APP, Y6] "APP Module SF1 System Software code," Year Y6.

[APP, Y7] "APP SF1 Flux/Delta Flux/Flow Application code," Year Y7.

[APP, Y8] "APP Module μp2 System Software Source Code Listing," Year Y8.

[APP, Y9] "APP μp2 Flux/Delta Flux/Flow Application Software Source Code Listing," Year Y9.

[APP, Y10] "APP Comm. Processor Source Code," Year Y10.

[Boehm, 1982] B. Boehm. *Software Engineering Economics*,. Prentice-Hall, Inc., 1982.

[Boehm, 2000] B. Boehm et al. *Software Cost Estimation With COCOMO II*. Prentice-Hall, Inc., 2000.

[Barry, 2002] E.J. Barry, T. Mukhopadhyay and S. Slaughter. "Software Project Duration and Effort: An Empirical Study, 2002," *Information Technology and Management*, vol. 3, pp. 113–136, 2002.

[Jones, 1995] C. Jones. *Patterns of Software Systems Failure and Success*. Thompson Computer Press, 1995.

[Malayia, 1998] Y. Malayia and J. Denton. "Requirements Volatility and Defect Density," in *Proc. 10th International Symposium on Software Reliability Engineering*, 1998.

[Munson, 2003] J.C. Muson. *Software Engineering Measurement*. AUERBACH Publications, CRC Press LLC, 2003.

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

[Smidts, 2000] C. Smidts and M. Li, "Software Engineering Measures for Predicting Software Reliability in Safety Critical Digital Systems," NRC, Office of Nuclear Regulatory Research, Washington DC NUREG/GR-0019, 2000.

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

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

16. REQUIREMENTS TRACEABILITY

Traceability is defined as the degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another [IEEE, 1990].

According to IEEE [IEEE, 1988], the requirements traceability (RT) measure aids in identifying requirements that are either missing from, or in addition to, the original requirements.

This measure can be applied as soon as the design is available. As listed in Table 3.3, the applicable life cycle phases for RT are Design, Coding, Testing, and Operation.

16.1 Definition

RT is defined as:

$$RT = \frac{R_1}{R_2} \times 100\% \tag{16.1}$$

where

RT the value of the measure requirements traceability,
R_1 the number of requirements met by the architecture, and
R_2 the number of original requirements.

Ideally, tracing should be done from the user's requirements specification to the SRS and then to the SDD, if a SDD is produced. Furthermore, if the source code is available, tracing can be done from the SDD to the Code or from the user's requirements specification to the Code directly. In this research, because the original user's requirements specification was not available, tracing could only be performed from the SRS to the other products. Normally, from a software-reliability point-of-view, it is better to trace from the SRS to the Code directly. This is because SDD is only an intermediate product and it is the code that affects the reliability of the software system. However, derived requirements may exist in the SDD. These requirements also should be identified and counted as the original requirements. Thus, the definition of requirement traceability is modified as follows:

$$RT = \frac{R_1}{R_2} \times 100\% \tag{16.2}$$

where

RT the value of the measure requirements traceability,
R_1 the number of requirements implemented in the source code, and

R_2 the number of original requirements specified in the SRS and derived requirements specified in the SDD.

It should be noted that, here, the calculated *RT* acts only as an indicator of the RT measure. The RePS using this measure is not based on the value of *RT* but based on the actual defects found between code and SRS (refer to Section 16.4).

16.2 Measurement Rules

The definition of RT specifically recommends *backward traceability* to all previous documents and *forward traceability* to all spawned documents [Gotel, 1994] [Wilson, 1997] [Ramesh, 1995]. A three-step measurement approach, however, was customized for the purpose of assessing the reliability of the software. The three steps in this approach are:

Step 1. Identify the set of Original Requirements in the SRS and in the SDD. (Refer to Section 16.2.1)
Step 2. Forward Tracing (Refer to Section 16.2.2)
Step 3. Backward Tracing (Refer to Section 16.2.3)

According to the definition, this three-step approach was applied to the APP by tracing only forward and backward between the original requirements identified in the SRS and the derived requirements identified in the SDD, and the requirements implemented in the delivered source codes.

16.2.1 Original Requirements Identification

Generally, there are two kinds of requirements in an SRS:

1. Functional Requirements
2. Non-functional Requirements

These terms are defined in [IEEE, 1998]:

Functional Requirement - A system/software requirement that specifies a function that a system/software system or system/software component must be capable of performing. These are software requirements that define behavior of the system, that is, the fundamental process or transformation that software and hardware components of the system perform on inputs to produce outputs.

Non-functional Requirement - In software system engineering, a software requirement that describes not what the software will do, but how the software will do it. For example, software-

performance requirements, software external interface requirements, software-design constraints, and software-quality attributes are non-functional requirements.

Functional requirements (FRs) capture the intended behavior of the system in terms of services, tasks or functions the system is required to perform. On the other hand, *Non-functional Requirements* (NRs) are requirements that impose restrictions on the product being developed (product requirements), on the development process (process requirements), or they specify external constraints that the product/process must meet (external requirements). These constraints usually narrow the choices for constructing a solution to the problem.

As stated earlier in this report, most of the non-functional requirements are not as important as the functional requirements. They do not describe what the software will do, but how the software will perform its functions. Normally, non-functional requirements are not included in the evaluation of reliability based on RT. However, this statement must be considered with caution. In certain cases, non-functional requirements hide functional requirements, or may describe characteristics that are critical, such as response time. These special cases should be identified by the analyst and included in the measurement of RT. In this research, some of the non-functional requirements for the APP system such as the timing requirements are crucial.

In the following subsections, the rules for distinguishing FRs from NRs are given. The counting rules for identifying each type of requirement in a SRS also are provided.

16.2.1.1 Distinguishing FRs from NRs

The following rules apply when distinguishing FRs from NRs:

1. "Functional" refers to the set of functions a system is to offer. "Non-functional" refers to the manner in which such functions are performed.
2. Functional requirements are the most fundamental and testable characteristics and actions that take place in processing function inputs and generating function outputs.
3. Functional requirements might be characterized in data-related or object-oriented diagrams. In flow diagrams, functional requirements usually are shown as ovals with arrows showing data flow or function inputs and outputs.
4. Functional requirements describe what it is that a customer needs to be able to do with the software. They may be documented in the form of rigorously specified Process Models or Use Cases, or they may simply be lists of required features and functions. Whatever the form used, functional requirements should always identify the minimum functionality necessary for the software to be successful.
5. Functional requirements typically are phrased with subject/predicate constructions, or noun/verb constructions. For example, "The system prints invoices" is a functional requirement.
6. Non-functional requirements may be found in adverbs or modifying clauses, such as "The system prints invoices *quickly*" or "The system prints invoices *with confidentiality*."

7. NFRs are focused on how the software must perform something instead of focused on what the software must do.
8. NFRs express constraints or conditions that need to be satisfied by functional requirements and/or design solutions.
9. Different from functional requirements that can fail or succeed, NFRs rarely can be completely met—they are satisfied within acceptable limits.

The following requirements should NOT be considered functional requirements:

a. Performance Requirements (throughput, response time, transit delay, latency, etc.)[38]
b. Design Constraints
c. Availability Requirements
d. Security Requirements
e. Maintainability Requirements
f. External Interface Requirements
g. Usability Requirements (ease-of-use, learnability, memorability, efficiency, etc.)
h. Configurability Requirements
i. Supportability Requirements
j. Correctness Requirements
k. Reliability Requirements
l. Fault tolerance Requirements
m. Operational Scalability Requirements (including support for additional users or sites, or higher transaction volumes)
n. Localizability Requirements (to make adaptations due to regional differences)
o. Extensibility Requirements (to add unspecified future functionality)
p. Evolvability Requirements (to support new capabilities or the ability to exploit new technologies)
q. Composability Requirements (to compose systems from plug-and-play components)
r. Reusability Requirements
s. System Constraints (e.g., hardware and OS platforms to install the software, or legacy applications, or in the form of organizational factors or the process that the system will support.)
t. User Objectives, Values, and Concerns.

The most common method of distinguishing functional requirements from non-functional requirements is to ask the appropriate decision maker(s) a series of qualifying questions for each category: “What,” “Who,” “Where,” “When,” and “How.” In addition, the “How” category can be broken down into four subcategories, specifically, “How Many,” “How Often,” “How Fast,”

[38] In the case of APP, some performance requirements need to be traced. See Section 16.2.1.3 for details.

and “How Easy”, as shown in Table 16.1 [Xu, 2005] [Hayes, 2004] [Sousa, 2004] [Matthia, 1998].

Table 16.1 Distinguishing Functional Requirements from Non-Functional Requirements

Problem Categories	Requirement type
What?	Functional Requirements
Who?	Security Requirements
Where?	Topographical Requirements
When?	Timing Requirements
How Often?	Frequency Requirements
How Fast?	Performance Requirements
How Many?	Scalability Requirements
How Easy?	Usability Requirements

16.2.1.2 Functional Requirements (Functions) Identification

The following counting rules apply when identifying functions in a SRS:

1. The Functional Requirements Section of the SRS is used to identify functional requirements for this measure.
2. If there is no separate Functional Requirements Section, then use the requirements in the SRS that describe the inputs, processing, and outputs of the software. These usually are grouped by major functional description, sub-functions, and sub-processes. A sub-function or sub-process is defined as a logical grouping of activities that generate a definable product or service.
3. The Software Design Document (SDD) is used to identify derived functional requirements. Normally, most of the functions defined here correspond to the functional requirements described in the SRS. If there exist functions that were not defined in the SRS, these functions should be considered derived requirements.
4. Each functional requirements specification is re-expressed as a fundamental and uncomplicated statement.
5. Each statement of functional requirements must be uniquely identified to achieve traceability. Uniqueness is facilitated by the use of a consistent and logical scheme for assigning identification to each specification statement within the requirements document.
6. Each uniquely identified (usually numbered) functional requirement is counted as an *Original Requirement*.

Figure 16.1 presents the procedures for identifying functions in a SRS.

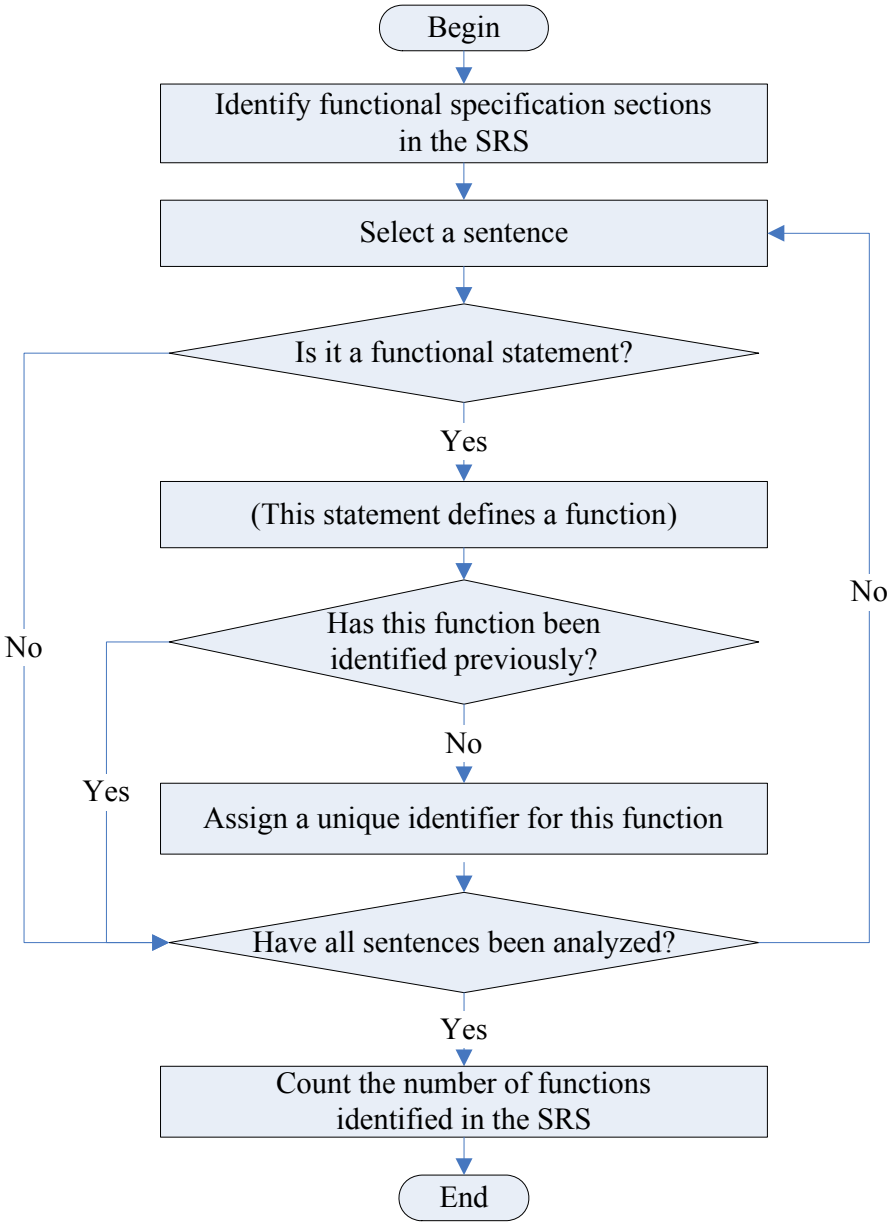


Figure 16.1 Procedure to Identify Functions in a SRS

16.2.1.3 Non-functional Requirements Identification

Since the APP system is a real-time system, it should continuously react with its environment and must satisfy timing constraints to properly respond to all the external events. Therefore, in this research, some of the non-functional requirements for the APP system also should be traced. The non-functional requirements that need to be traced are listed below:

1. Timing requirements
2. Frequency requirements
3. Performance requirements

The following counting rules apply when identifying non-functional requirements in a SRS:

1. Most of the timing and frequency requirements are specified in the Performance Requirements Section in the SRS. Some of these requirements also may be found in the External Interface Requirements Section in the SRS.
2. All of the performance requirements can be identified in the Performance Requirements Section in the SRS.
3. Each non-functional requirements specification is re-expressed as a fundamental and uncomplicated statement.
4. Each non-functional requirement statement must be uniquely identified to achieve traceability. Uniqueness is facilitated by the use of a consistent and logical scheme for assigning identification to each specification statement within the requirements document.
5. Each uniquely identified (usually numbered) non-functional requirement is counted as an *Original Requirement*.

Figure 16.2 describes the general procedures for identifying counted non-functional requirements in a SRS.

16.2.2 Forward Tracing

Forward tracing in the RT measurement is used to determine the counterparts of the original requirements of the SRS/SDD in the source code. In this step, the original requirements identified in Step 1 are mapped into the delivered source code, one after another, primarily for the purpose of identifying unimplemented SRS/SDD original requirements and uncovered source code. Figure 16.3 presents the procedure of forward tracing (from the SRS/SDD to the source code).

An *unimplemented SRS/SDD original requirement* is a requirement that is identified in the SRS but has no counterpart found in the delivered source code. Contrast this with an *implemented SRS/SDD original requirement* that is identified in the SRS/SDD and has counterpart(s) found in

the delivered source code. Each unimplemented SRS/SDD original requirement is a defect in the delivered source code.

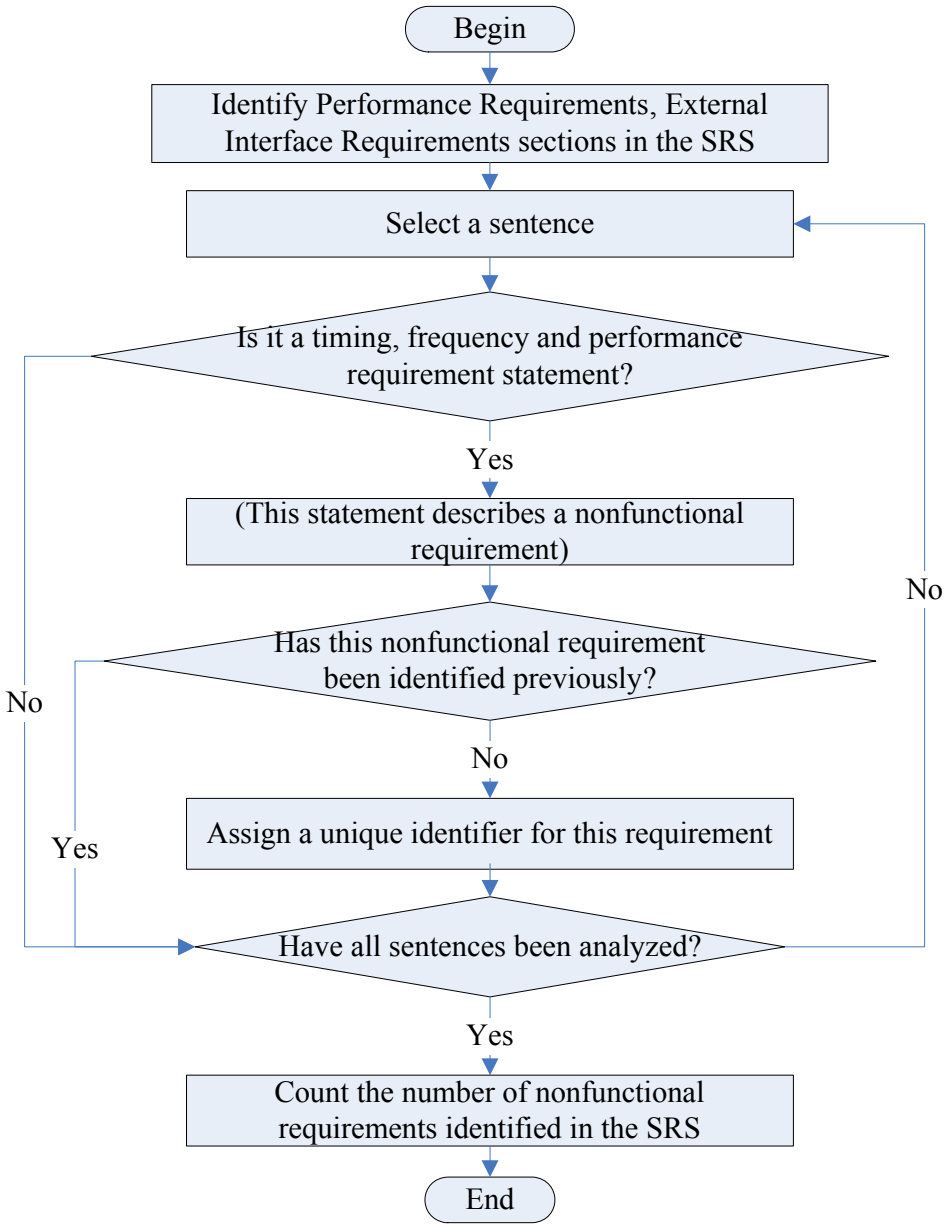


Figure 16.2 Procedure to Identify Non-functional Requirements in a SRS

The *uncovered source code* is the source code that does not correspond to any original requirements identified in the SRS/SDD**.** This can be contrasted with *covered source code,* which has a counterpart identified in the SRS/SDD.

It should be noted that understanding the lines of code corresponding to the original requirements is not easy, especially for a large system. However, existing commercial tools such as the Rational software developed by IBM are very helpful in this process.

16.2.3 Backward Tracing

The primary concerns of backward tracing are to identify the extra requirements and to count the number of requirements implemented in the delivered source code (R1).

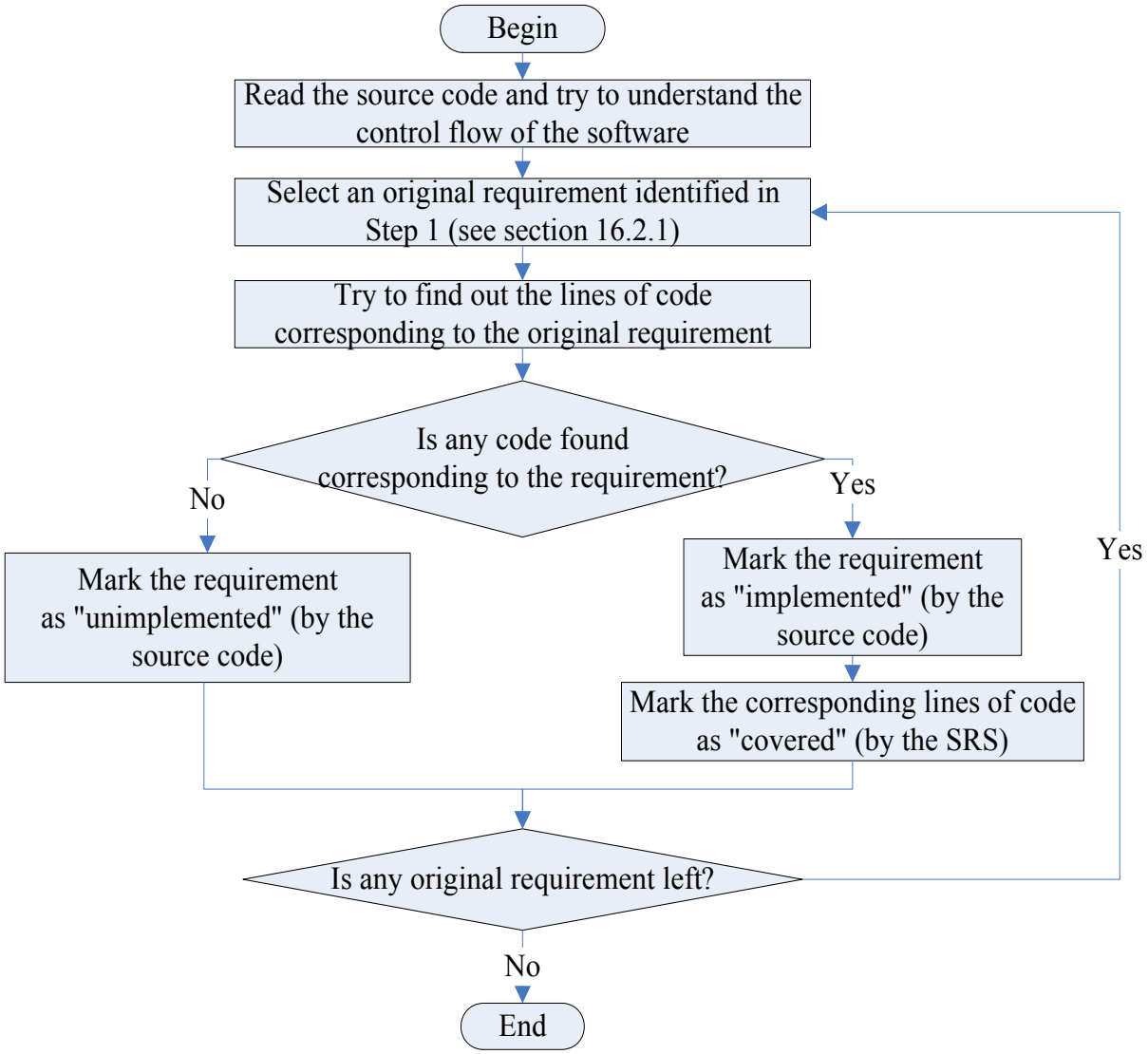


Figure 16.3 Procedure for Forward Tracing

An *extra requirement* is a requirement that is not identified in the SRS/SDD but is implemented in the delivered source code. Each extra requirement is a defect in the delivered source code because it may introduce risk into the system.

In this step, the uncovered lines of source code identified in Step 2 are analyzed and then the corresponding extra requirements are represented using the same level of granularity as used to identify requirements in the SRS/SDD. Figure 16.4 describes the procedure for backward tracing (from the source code to the SRS/SDD).

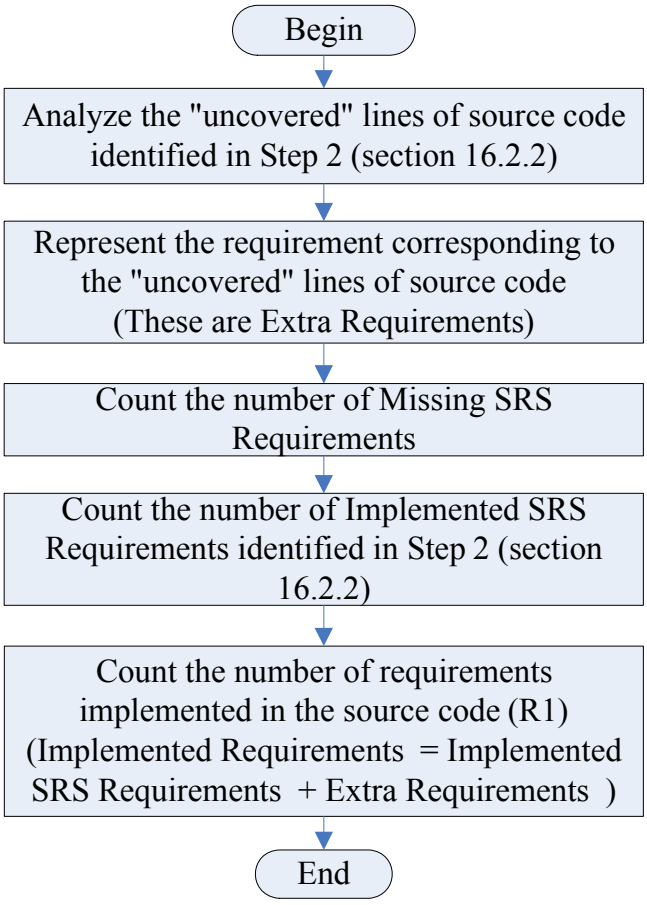


Figure 16.4 Procedure for Backward Tracing

16.3 Measurement Results

The following documents were used to measure the requirements traceability between the APP SRSs and the codes:

- APP Module μp1 System SRS [APP, Y1]
- APP Module μp1 Flux/Delta Flux/Flow Application SRS [APP, Y2]
- APP Module μp2 System SRS [APP, Y3]
- APP Module μp2 Flux/Delta Flux/Flow Application SRS [APP, Y4]
- APP Module Communication Processor SRS [APP, Y5]
- APP module first safety function processor SDD
- APP Flux/Delta Flux/Flow Application SDD for SF1
- APP μp2 SDD for system software
- APP μp2 Flux/Delta Flux/Flow application software SDD
- APP communication processor SDD
- APP Module μp1 System Software Code [APP, Y6]
- APP μp1 Flux/Delta Flux/Flow Application Software Source Code [APP, Y7]
- APP Module μp2 System Software Source Code [APP, Y8]
- APP μp2 Flux/Delta Flux/Flow Application Software Source Code [APP, Y9]
- APP Communication. Processor Source Code [APP, Y10]

Quantities R_1 and R_2 are counted at the primitive level. The tables below (Table 16.2 through Table 16.6) present the measurements.

Table 16.2 Summary of the Requirements Traceability Measurement for μp1 System Software

No.	Section No.	Section Name	R_1	R_2	RT
1	SRS 3.1	Initialization	48	48	100%
2	SRS 3.2	Power-up self test	175	176	99.432%
3	SRS 3.3	Main Program	135	135	100%
4	SRS 3.3.3.A	Calibration	40	42	95.238%
5	SRS 3.3.3.B	Tune	16	16	100%
6	SRS 3.4	On-line diagnostics	144	144	100%
7	SDD	Decomposition Description	2	2	100%

Table 16.3 Summary of the Requirements Traceability Measurement for µp1 Application Software

No.	Section No.	Section Name	R_1	R_2	RT
1	SRS 3.0	Specific Requirements	67	67	100%
2	SRS 3.1	Other Requirements	3	3	100%

Table 16.4 Summary of the Requirements Traceability Measurement for µp2 System Software

No.	Section No.	Section Name	R_1	R_2	RT
1	SRS 3.1.1	Initialization	10	9	111.111%
2	SRS 3.1.2	Power-up self test	32	33	96.970%
3	SRS 3.1.3	Main Program	56	56	100%
4	SRS 3.1.4	Calibration	25	25	100%
5	SRS 3.1.5	Tune	12	12	100%
6	SRS 3.1.6	On-line diagnostics	46	46	100%
7	SRS 3.2	External Interface Requirements	3	3	100%
8	SRS 3.3	Performance Requirements	4	4	100%
9	SDD	Decomposition Description	4	4	100%

Table 16.5 Summary of the Requirements Traceability Measurement for µp2 Application Software

No.	Section No.	Section Name	R_1	R_2	RT
1	3.1	Functional Requirements	25	25	100%
2	3.2	External Interface Requirements	3	3	100%

The challenge in forward tracing and backward tracing arises from understanding the activities of the source code. Mastering the control flow of the source code and thus grasping the big picture is usually the first step to understanding the source code. Comments in the source code,

along with other documents such as the Design document, Testing Plan, and V&V reports will be helpful for performing the tracing.

During the measurement it was observed that the requirements for μp1 were written to a higher level of detail as compared to the requirements for μp2.

The ratio of R_1 and R_2 is somewhat subjective because the granularity level of the original requirements used for counting R_1 and R_2 is subjective. As stated in the definition section, the RePS using this measure is not based on the value of *RT* but is based instead on the actual defects found between SRS and code. A defect was identified when either a requirement was not implemented in the code or if extra code was implemented for a requirement that did not exist.

Table 16.6 Summary of the Requirements Traceability Measurement for CP

No.	Section No.	Section Name	R_1	R_2	RT
1	3.1	Initialization	18	18	100%
2	3.2	Power-up self test	96	97	98.969%
3	3.3	Main Program	45	45	100%
4	3.4	On-line diagnostics	69	69	100%
5	3.5	Time of the day	4	4	100%
6	3.6	Serial Communications	64	64	100%

Table 16.7 Description of the Defects Found in APP by the Requirements Traceability Measure

No.	Location	Requirement Description	Defect Type	Severity Level
1	μp1 Section 3.2	Increment the EEPROM test counter if the Tuning in Progress flag setup.	Requirement not implemented in the code	3
2	μp1 Section 3.2	This algorithm shall detect coupling faults between two address lines.	Requirement not implemented in the code	1

Table 16.7 Description of the Defects Found in APP by the Requirements Traceability Measure (continued)

No.	Location	Requirement Description	Defect Type	Severity Level
3	µp1 Section 3.3.3.A	Copy the contents of the table to the Dual Port RAM.	Requirement not implemented in the code	1
4	µp1 Section 3.3.3.A	Give up the Semaphore	Requirement not implemented in the code	1
5	µp2 Section 3.1.1	N/A	Code not mentioned in SRS	3
6	µp2 Section 3.1.2.3	This algorithm shall detect coupling faults between two address lines.	Requirement not implemented in the code	1
7	CP Section 3.2.3	This algorithm shall detect coupling faults between two address lines.	Requirement not implemented in the code	1

16.4 RePS Construction from Requirements Traceability

The APP system has four distinct operational modes: Power-on, Normal, Calibration, and Tuning [APP, 01]. The reliability of the APP system was estimated on a one-by-one operational mode basis using the Extended Finite State Machine (EFSM) model approach [Smidts, 2004].

This approach proceeds in three steps:

1. Construct an EFSM model representing the user's requirements and embedding the user's operational profile information.
2. Map the identified defects to the EFSM model.
3. Execute the EFSM model to evaluate the impact of the defects in terms of the failure probability.

Figure 16.5 presents the entire approach to estimate reliability. It should be noted that it is possible for a defect to be involved in more than one operational mode.

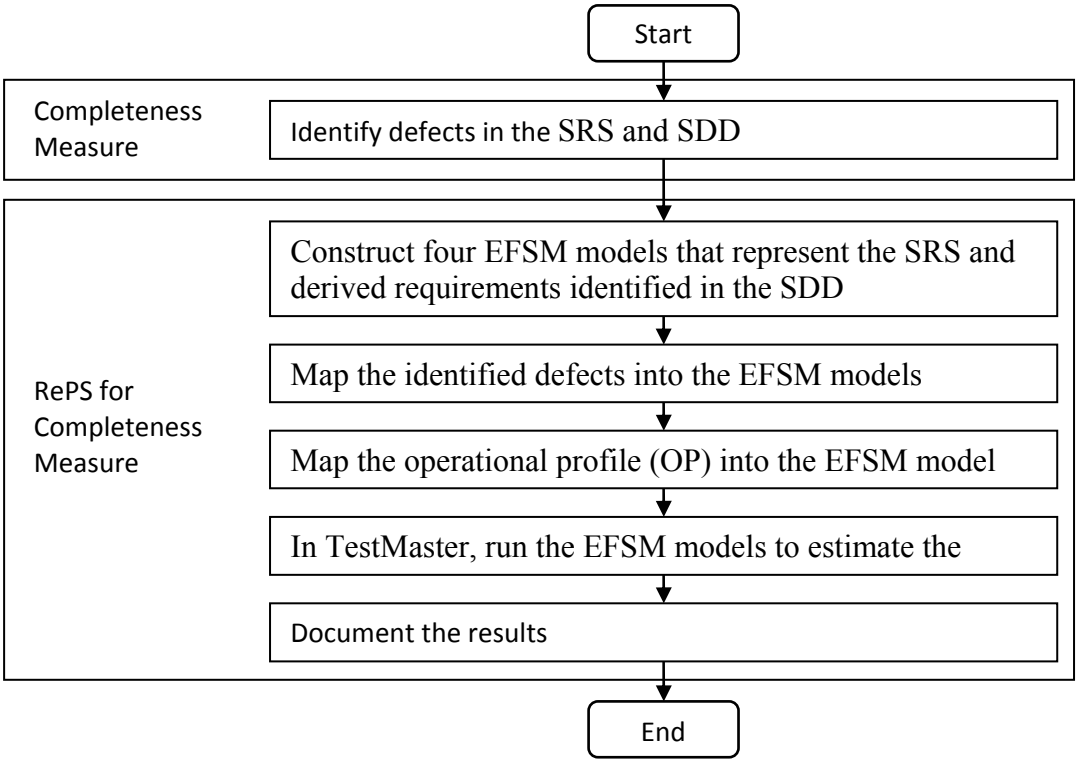


Figure 16.5 Approach of Reliability Estimation Based on the EFSM Model

The estimation of APP probability of failure-per-demand based on the RT RePS is 3.28×10^{-10}. Hence:

$$p_s = 1 - 3.28 \times 10^{-10} = 0.9999999996720$$

The reliability estimation for each of the four operational modes using the defects found through the requirement traceability measurement is shown in Table 16.8.

Table 16.8 Reliability Estimation for Four Distinct Operational Modes

Mode	Probability of Failure
Power-on	2.06×10^{-10}
Normal	3.28×10^{-10}
Calibration	6.72×10^{-13}
Tuning	0

16.5 Lessons Learned

The measurement of RT is a labor-intensive process but it can be assisted by building a formal approach as illustrated in Figure 16.1 to Figure 16.4. Unlike the DD measurement, which requires the verification of a large number of items, the measurement of RT only requires verifying the presence or absence of an item in the requirements documents and the code. Thus, the RT measurement process is not as error-prone some other measures.

16.6 References

[APP, 01] *APP Instruction Manual*.

[APP, Y1] "APP Module First Safety Function Processor SRS," Year Y1.

[APP, Y2] "APP Flux/Delta Flux/Flow Application SRS for SF1," Year Y2.

[APP, Y3] "APP Module μp2 System Software SRS," Year Y3.

[APP, Y4] "APP μp2 Flux/Delta Flux/Flow Application Software SRS," Year Y4.

[APP, Y5] "APP Module Communication Processor SRS," Year Y5.

[APP, Y6] "APP Module SF1 System Software Code," Year Y6.

[APP, Y7] "APP SF1 Flux/Delta Flux/Flow Application Code," Year Y7.

[APP, Y8] "APP Module μp2 System Software Source Code Listing," Year Y8.

[APP, Y9] "APP μp2 Flux/Delta Flux/Flow Application Software Source Code Listing," Year Y9.

[APP, Y10] "APP Comm. Processor Source Code," Year Y10.

[Gotel, 1994] O. Gotel and A. Finkelstein. "An Analysis of the Requirements Traceability Problem," in *Proc. of the 1st International Conference on Requirements Engineering*, pp. 94–101, 1994.

[Hayes, 2004] J.H. Hayes et al. "Helping Analysts Trace Requirements: An Objective Look," *in Proc. of IEEE Requirements Engineering Conference*, 2004, pp. 249–261.

[IEEE, 1988] "IEEE Guide for the Use of Standard Dictionary of Measures to Produce Reliable Software," IEEE Std. 982.2-1988, 1988.

[IEEE, 1990] "IEEE Standard Computer Dictionary. A Compilation of IEEE Standard Computer Glossaries," IEEE Std. 610, 1991.

[IEEE, 1998] "IEEE Recommended Practice for Software Requirements Specifications," IEEE Std. 830-1998, 1998.

[Matthia, 1998] J. Matthias. "Requirements Tracing." *Communications of the ACM*, vol. 41, 1998.

[Ramesh, 1995] B. Ramesh, L.C. Stubbs and M. Edwards. "Lessons Learned from Implementing Requirements Traceability," *Crosstalk, Journal of Defense Software Engineering*, vol. 8, pp. 11–15, 1995.

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

[Sousa, 2004] G. Sousa and J.F.B. Castro. "Supporting Separation of Concerns in Requirements Artifacts," in *Proc. 1st Brazilian Workshop on Aspect-Oriented Software Development*, 2004.

[Wilson, 1997] W.M. Wilson, L.H. Rosenberg and L.E. Hyatt, "Automated Analysis of Requirement Specifications," in *Proc. International Conference on Software Engineering*, 1997.

[Xu, 2005] L. Xu, H. Ziv and D. Richardson. "Towards Modeling Non-Functional Requirements in Software Architecture," in *Proc. SESSION: Workshop on Architecting Dependable Systems*, 2005, pp. 1–6.

17. TEST COVERAGE

The concept of test coverage (TC) is applicable to both hardware and software. In the case of hardware, coverage is measured in terms of the number of possible faults covered. In contrast, the number of software faults is unknown. TC in the case of software systems is measured in terms of structural or data-flow units that have been exercised.

According to [IEEE, 1988], TC is a measure of the completeness of the testing process from both a developer and a user perspective. The measure relates directly to the development, integration, and operational test stages of product development: unit, system, and acceptance tests. The measure can be applied by developers in unit tests to obtain a measure of the thoroughness of structural tests.

This measure can be applied once testing is completed. As listed in Table 3.3, the applicable life cycle phases for TC are Testing and Operation.

17.1 Definition

As described in [IEEE, 1988], the primitives for TC are divided in two classes: program and requirement. For the program class, there are two types of primitives: functional and data. The program functional primitives are either modules, segments, statements, branches (nodes), or paths. Program data primitives are equivalence classes of data. Requirement primitives are either test cases or functional capabilities.

TC is the percentage of requirement primitives implemented multiplied by the percentage of primitives executed during a set of tests. A simple interpretation of TC can be expressed by Equation 17.1:

$$TC\% = \left(\frac{Implemented\ capabilities}{Required\ capabilities}\right) \times \left(\frac{Program\ primitives\ tested}{Total\ program\ primitives}\right) \times 100 \qquad (17.1)$$

In this study, the definition of TC has been modified for the following two reasons:

1. The percentage of requirement primitives implemented in the source code has been obtained from the RT measurement results, as discussed in chapter 16.
2. Since the program primitives are implemented in the format of code, the percentage of primitives executed during a set of tests is actually the coverage of code tested by test data. The software engineering literature defines multiple code coverage measures such as block (also called statement) coverage, branch coverage, and data flow coverage [Malaiya, 1993]. In this research, statement coverage was selected because it is the most

popular test coverage metric and has been embedded in many integrated development environments, such as Keil μVision2 and IAR EWZ80 used in this research.

Therefore, TC can be modified to be the requirements traceability multiplied by the fraction of the total number of statements that have been executed by the test data [Malaiya, 1996]. The concept can be shown in the following equation:

$$C_1 = \left(\frac{R_1}{R_R}\right) \times \left(\frac{LOC_{Tested}}{LOC_{Total}}\right) \times 100 \quad (17.2)$$

where

C_1	The value of the test coverage
R_1	The number of requirements implemented
R_R	The total number of required requirements including the number of original requirements specified in the SRS, derived requirements specified in the SDD and requirements implemented in code but not specified in either SRS or SDD
LOC_{Tested}	The number of lines of code that are being executed by the test data listed in the test plan
LOC_{Total}	The total number of lines of code

The measurement of statement coverage and the corresponding reliability prediction are discussed in the following sections.

17.2 Measurement Rules

A four-step measurement approach is introduced in this chapter to determine the test coverage (C_1). The four steps in the measurement approach are:

Step 1. Make the APP source code executable (Refer to Section 17.2.1)
Step 2. Determine the total number of executable lines of code (Refer to Section 17.2.2)
Step 3. Determine the number of tested lines of code (Refer to Section 17.2.3)
Step 4. Determine the percentage of requirement primitives implemented (Refer to Section 17.2.4)

17.2.1 Make the APP Source Code Executable

The software on the safety microprocessor 1 (μp1) and communication microprocessor (CP) were developed using the Archimedes C-51 compiler, version 4.23; the software on safety microprocessor 2 (μp2) was developed using the Softools compiler, version 1.60f. Due to the obsolescence of these tools, the software was ported to the Keil PK51 Professional Developer's Kit (μVision2 V2.40a) and IAR EWZ80, version 4.06a-2, respectively. The major modifications

are the replacement of some obsolete keywords with their equivalents in the new compilers. Consequently, the porting did not change the results.

Table 17.1 lists the compilers used in this research and the number of errors and warnings observed before modification of the original APP source code.

Table 17.1 Original Source Code Information with Compilers Used in This Research

Microprocessor	Compiler	Number of Errors/Warnings
μp1	Keil μVision2 V2.40a	122/1
μp2	IAR EWZ80 V4.06a-2	1345/33
CP	Keil μVision2 V2.40a	36/1

The errors and warnings mainly are to the result of the following differences between the compilers used in this study and those used by the APP developers:

1. Different keyword used;
2. Different definition of special function registers used;
3. Different interrupt definitions used;
4. Different data type used;

Several modification examples are shown in Table 17.2.

Table 17.2 APP Source Code Modification Examples

Reason	Type	Original Source Code	Modified Source Code
1	Different assembly keyword	Module VCopy	Name VCopy
2	Different bit definition	bit EA = 0xAF;	sbit EA = 0xAF;
3	Different interrupt definition	Interrupt [0x03] void EX0_int (void);	#define EX0_int =0;
4	Different data type	Data unsigned int unmemory_Loc	unsigned int data unmemory_Loc
5	Other Miscellaneous errors in APP	GO	GO:

As shown in Table 17.2, in the Archimedes C-51 compiler "Module" is the keyword used to define an assembly function while in Keil μVision2 "Name" is the correct keyword performing the same function. The Archimedes C-51 compiler uses "bit" to define a bit in a special function register and Keil μVision2 uses "sbit." The ways in which the interrupt function is defined are different in these two compilers. How to define a data type is another problem in these compilers, as the fourth example shows. Other miscellaneous syntax errors, such as a missing colon (refer to the fifth example), needed to be corrected.

17.2.2 Determine the Total Lines of Code

As specified in Section 17.1, test coverage indicates the number of executable statements encountered.

The total number of executable lines of code (eLOC) are provided by the compilers. The results are shown in Table 17.3.

Table 17.3 Total Number of Executable Lines of Code Results

	Module	eLOC	Total Number of eLOC
μp1	SF1APP	249	1537
	SF1CALTN	238	
	SF1FUNCT	353	
	SF1PROG	246	
	SF1TEST1	184	
	SF1TEST2	267	
μp2	APP1	269	1409
	CAL_TUNE	392	
	MAIN	488	
	ON_LINE	75	
	POWER_ON	185	

Table 17.3 Total Number of Executable Lines of Code Results (continued)

	Module	eLOC	Total Number of eLOC
CP	COMMONLI	116	811
	COMMPOW	183	
	COMMPROC	132	
	COMMSER	380	
Total			3757

17.2.3 Determine the Number of Tested Lines of Code

According to the original APP test plan [APP, Y6], in order to perform the tests, the following requirements should be met:

1. The software to be tested must be available in PROM and installed in an operational APP module.
2. An appropriate power supply for the module must be available.
3. In most cases, an emulator for the microprocessor and its associated software is required.
4. A compatible PC is required to monitor and control the emulator.

It should be noted that a modification of the test cases was necessary in this research. Mainly, this was due to the following reasons:

1. In this study, software testing was performed based on a real-time simulation environment and not the actual APP system. The software was not available in PROM, and debuggers were used to monitor the execution of the source code.
2. The emulator was not available. Thus, all the functions performed by the emulator were modified.
3. The main purpose of the testing in this study was different from the original purpose of the testing. The original test cases were used to test the program and check the functionality of the program. The execution of the test cases in this study was to determine the code coverage. Thus, only the input specifications sections needed to be considered. The output specifications did not need to be verified.

After step 1, the APP source code was successfully compiled either using KEIL PK51 Professional Developer's Kit (for μp1 and CP) or using IAR EWZ80 workbench (for μp2). The compiler debugger tools were used to determine the percentage of code that had been executed, denoted as C_1'. Therefore, the number of tested lines of code can be calculated by:

$$LOC_{Tested} = LOC_{Total} \times C_1' \quad (17.3)$$

The general procedure used to conduct each test case is given below:

1. Set breakpoint to halt the execution at certain desired points.
2. Check and change memory or variable values according to the input specifications described in the test plan.
3. Allow the program to proceed to the next breakpoint where additional checks may occur.
4. Record the code coverage given by the debugger.

The following subsections show how to use the debugger tools to record the code coverage.

17.2.3.1 Keil µVision2 Debugger

The µVision2 debugger offers a feature called "Code Coverage Analysis" that helps to ensure the application has been thoroughly tested. The Code Coverage Window shows the percentage of code for each module (according to the level 2 module definition in Chapter 6) in the program that has been executed. Code Coverage aids in debugging and testing the application by allowing users to easily distinguish the parts of the program that have been executed from the parts that have not.

In µVision2, colors displayed on the left of the assembly window indicate the status of the corresponding instruction.

1. Dark Grey: Indicates that the line of code has not yet been executed.
2. Green: Indicates that the instruction has been executed. In the case of a conditional branch, the condition has tested true and false at least once.

17.2.3.2 IAR EWZ80 Debugger

Similar to the Keil µVision2 Debugger, the IAR EWZ80 debugger also can provide code coverage information. The Code Coverage Window shows the percentage of code in the program module that has been executed. The untested lines of code (line number) also are shown in the window.

17.2.4 Determine the Percentage of Requirement Primitives Implemented

Chapter 16 described how to obtain the requirements traceability in details. Consult that chapter for the measurement rules.

17.3 Measurement Results

17.3.1 Determine the Required Documents

As described Section 17.4, the value of code coverage can be used to estimate the value of defect coverage. The number of defects remaining in the APP can then be estimated from the defect coverage and the number of defects found by test cases provided in the test plan. As stated earlier, the testing performed on the APP was not intended to test the program and check the functionality of the program as the original testing did. The number of defects was obtained by counting the number of defects identified in the original test reports. The reports distinguish five levels of test results:

1. Test completed successfully;
2. Test resulted in discrepancies that were resolved by Test Plan deviation;
3. Test resulted in discrepancies that required modifications to the Test Plan;
4. Test resulted in discrepancies that required modifications to the requirements specifications, design description, or code;
5. Incorrect execution of the test which resulted in a discrepancy. The correct execution of the test resolved the discrepancy.

Obviously, only level-four discrepancies were considered defects found by testing. Since there exist many versions of source code, test plans and test reports, one needs to determine which version is to be used for test coverage measurement. Table 17.4 shows this information.

Therefore, the following documents were used to measure the test coverage:

- APP Module μp1 System Software Code Revision 1.03 [APP, Y1]
- APP μp1 Flux/Delta Flux/Flow Application Software Source Code Revision 1.03 [APP, Y2]
- APP Module μp2 System Software Source Code Revision 1.02 [APP, Y3]
- APP μp2 Flux/Delta Flux/Flow Application Software Source Code Revision 1.02 [APP, Y4]
- APP Communication Processor Source Code Revision 1.04 [APP, Y5]
- APP Test Plan for μp1 Software [APP, Y6]
- APP Test Report for μp1 Software [APP, Y9]
- APP Test Plan for μp2 Software [APP, Y7]
- APP Test Report for μp2 Software [APP, Y10]
- APP Test Plan for CP Software [APP, Y8]
- APP Test Report for CP Software [APP, Y11]

Table 17.4 Testing Information for µp1

	Test Report Revision	Applicable Code Revision	Test Plan Used	Critical + Significant Defects	Test Report Revision
µp1	#00	1.03	0	4	1. Not all the address lines tested 2. All inputs boards missing in Power-on without indicating fatal error 3. Discrete inputs tripped condition 4. Detect module ID with DPR
	#01	1.03	1	0	N/A
	#02	1.04	2	0	N/A
	#03	1.07	3	0	N/A
	#04	1.08	4	0	N/A
µp2	#00	1.02	0	2	1. Online RAM test not complete 2. Online EEPROM failure is not identified as fatal failure
	#01	1.03	0	0	N/A
	#02	1.04	2	0	N/A
	#03	1.05	4	0	N/A
	#04	1.06	5	0	N/A
CP	#00	1.04	0	1	Initialize variable problem
	#01	1.04	1	0	N/A
	#02	1.04	02	0	N/A

17.3.2 Test Coverage Results

Table 17.5 shows the statement coverage results.

Table 17.5 Statement Coverage Results

Microprocessor	LOC_{Total}	C_1'	LOC_{Tested}
µp1	1537	0.886	1362
µp2	1409	0.939	1324
CP	811	0.898	729
Total	3757	0.908	3379

From the measurement results of Chapter 16, the total number of implemented requirements of the APP system, R_1, is 1,146. The number of original requirements specified in the SRS and derived requirements specified in the SDD is 1,150. There is one requirement that is implemented in code but not specified in either SRS or SDD. Thus the total number of the requirements, R_R, is 1,151. Therefore, the test coverage for APP is:

$$C_1 = \left(\frac{R_1}{R_R}\right)\left(\frac{LOC_{Tested}}{LOC_{Total}}\right) = \left(\frac{1{,}146}{1{,}151}\right)\left(\frac{3{,}379}{3{,}757}\right) = 0.8955 \quad (17.4)$$

17.3.3 Linear Execution Time Per Demand Results

The linear-execution time, T_L, is used in different RePSs. The linear-execution time is defined as the product of the number of lines of code per demand and the average execution time of each line [Malaiya, 1993].

APP linear-execution time is calculated by executing a segment of linear code (code without a loop) in the Keil-simulation environment. This segment contains seventy-four lines of code. The measurement procedure is described as follows:

1. Set the clock frequency to 12 MHz for the Intel 80C32 microprocessor and 16 MHz for the Z180 microprocessor;
2. Set breakpoints at the beginning of the code and the end of the code;
3. Execute the code and record the execution time in seconds at the start and end breakpoints (T_{start} and T_{end} respectively). This information is available in the "secs" item in the register window;
4. Calculate the difference between the breakpoints to obtain the execution time for the 74 lines of code ($T_{AE} = T_{end} - T_{start}$).

As such, the linear-execution time T_L for the given software is:

$$T_L = T_{AE} \times \frac{LOC}{74} \quad (17.5)$$

where LOC is the size in lines of code for the given software.[39]

Table 17.6 summarizes the results of this experiment.

[39] All 74 LOC are executable.

Table 17.6 Linear Execution Time for Each Microprocessor in the APP System

	µp1	**µp2**	**CP**
T_{start} **(seconds)**	0.000389	0.00029175	0.000389
T_{end} **(seconds)**	0.002844	0.002133	0.002844
T_{AE} **(seconds)**	0.0000332	0.0000249	0.0000332
LOC/demand with cycles disabled	554	346	619
T_L **(seconds)**	0.018	0.009	0.021

17.3.4 Average Execution-Time-Per-Demand Results

Similar to the linear-execution time, τ also is used in many RePSs. The value of τ can be determined during testing by recording the actual execution time. The approaches for determining τ in the simulation environments are not the same.

For µp1 and CP, because these source codes are executed in the Keil µVision2 environment, source code execution time is shown by the system register in the watch window. From the test-coverage experiment, the average execution time for µp1 is 0.082 seconds/demand and the average execution time for CP is 0.016 seconds/demand;

For µp2, the execution time is not directly given by the simulation environment IAR EWZ80; but the number of cycles (processor clock cycles) is provided. The execution time can be calculated by

$$\tau = \frac{N_C}{f} \tag{17.6}$$

where:

N_C the number of cycles given by the simulation environment
f the µp2 clock frequency (16 MHz)

From the test coverage experiment, the average number of cycles was 2,064,135, so the average execution time of µp2 is:

$$\tau = \frac{2{,}064{,}135}{1.6\times10^7} = 0.129 \text{ second} \tag{17.7}$$

17.4 RePS Construction from Test Coverage

17.4.1 Determination of the Defect Coverage

Malaiya et al. investigated the relationship between defect coverage, C_0, and statement coverage, C_1. In [Malaiya, 1996], the following relationship was proposed:

$$C_0 = a_0 \ln\{1 + a_1[exp(a_2 C_1) - 1]\} \tag{17.8}$$

where a_0, a_1, and a_2 are coefficients and C_1 is the statement coverage [Malaiya, 1996]. The coefficients were estimated from field data. Figure 17.1 depicts the behavior of C_0 for data sets two, three, and four given in [Malaiya, 1996].

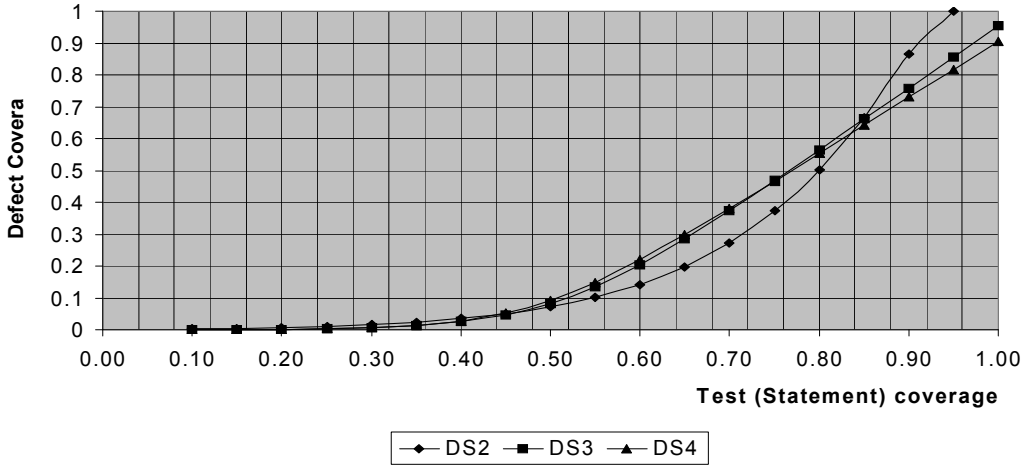


Figure 17.1 Defect Coverage vs. Test Coverage

17.4.2 Determination of the Number of Defects Remaining in APP

According to Malaiya [Malaiya, 1993], the number of defects remaining in software, N, is:

$$N = \frac{N_0}{C_0} \tag{17.9}$$

where

N	number of defects remaining in the software
N_0	number of defects found by test cases provided in the test plan
C_0	defect coverage

From Table 17.4, the total number of defects found by test cases provided in the test plan is

$$N_0 = 4 + 2 + 1 = 7$$

Table 17.7 provides the defect coverage and the corresponding total number of defects remaining in APP given the parameters in [Malaiya, 1996] for three data sets. Since all three data sets are applicable to the APP case, the total number of defects remaining is estimated by an average value: 9 defects.

Table 17.7 Defects Remaining, *N*, as a Function of TC and Defects Found for Three Malaiya Data Sets

$C_0 = a_0 \ln\{1 + a_1[exp(a_2 C_1) - 1]\}$ $C_1 = 0.896 \quad N_0 = 7$					
Data Set	a_0	a_1	a_2	C_0	$N(= N_0/C_0)$
DS2	1.31	1.80E-03	6.95	0.847	8 (8.3)
DS3	0.139	7.00E-04	14.13	0.751	9 (9.3)
DS4	0.116	6.00E-04	15.23	0.723	10 (9.7)

17.4.3 Reliability Estimation

Malaiya [Malaiya, 1993] also suggested the following expression for the failure intensity

$$\lambda = \frac{K}{T_L} N \tag{17.10}$$

where

K the value of the fault exposure ratio during the *n*-th execution
T_L the linear execution time

and the probability of *n* successful demands $p_s(n)$ is given as:

$$p_s(n) = e^{-\lambda T(n)} \tag{17.11}$$

where *T(n)* is the duration of *n* demands. It is given by:

$$T(n) = \tau \times n \tag{17.12}$$

where

τ the average execution-time-per-demand.
n the number of demands.

Replacing λ and $T(n)$ in Equation 17.11 with Equation 17.10 and Equation 17.12:

$$p_s(n) = e^{-\lambda T(n)} = e^{-\frac{K}{T_L} N\tau n} \tag{17.13}$$

The fault-exposure ratio for the seven defects identified during testing can be precisely estimated using the EFSM described in chapter 5. Using Equation 17.13:

$$K = \frac{-\ln(1-p_f)}{N\tau n} T_L \tag{17.14}$$

where

p_f the probability of failure-per-demand corresponding to the known defects. This value is given by the APP EFSM and is 5.8×10^{-10}.

Table 17.8 lists the probability of success-per-demand.

Table 17.8 Probability of Success-Per-Demand Based On Test Coverage

N	9
p_s	0.99999999942

The *linear execution time*, T_L, for each of the three subsystems (μp1, μp2, and CP) of APP has been identified in Section 17.3.3. There are several ways to estimate the linear-execution time for the entire APP system, such as using the average value of these three subsystems. For a safety-critical application, such as the APP system, the UMD research team suggests making a conservative estimation of T_L by using the minimum of these three subsystems. Namely,

$$\begin{aligned} T_L &= \min\{T_L(\mu p1), T_L(\mu p2), T_L(CP)\} \\ &= \min\{0.018, 0.009, 0.021\} \\ &= 0.009 \text{ seconds} \end{aligned} \tag{17.15}$$

where

$T_L(\mu p1)$ Linear execution time of Microprocessor 1 (μp1) of the APP system. T_L *(μp1)* = 0.018 seconds;

$T_L(\mu p2)$ Linear execution time of Microprocessor 2 (μp2) of the APP system. T_L *(μp2)* = 0.009 seconds;

$T_L(CP)$ Linear execution time of Communication Microprocessor (CP) of the APP system. T_L *(CP)* = 0.021 seconds.

Similarly, the *average execution-time-per-demand*, τ, for each subsystem has been identified in section 17.3.4. To make a conservative estimation, the average execution-time-per-demand for the entire APP system is the maximum of the three subsystems. Namely,

$$\begin{aligned} \tau &= \max\{\tau(\mu p1), \tau(\mu p2), \tau(CP)\} \\ &= \max\{0.082, 0.129, 0.016\} \\ &= 0.129 \text{ seconds/demand} \end{aligned} \tag{17.16}$$

where

$\tau(\mu P1)$ Average execution-time-per-demand of Microprocessor 1 (μp1) of the APP system. *τ(μp1)* = 0.082 seconds/demand;

$\tau(\mu P2)$ Average execution-time-per-demand of Microprocessor 2 (μp2) of the APP system. *τ(μp2)* = 0.129 seconds/demand;

$\tau(CP)$ Average execution-time-per-demand of Communication Microprocessor (CP) of the APP system. *τ(CP)* = 0.016 seconds/demand.

17.5 Lessons Learned

Normally, the measurement of TC should be completed efficiently with the help of automation tools. In this research, the time required for the measurement was excessive: a great deal of time was devoted to modifying the original APP source code so that it could be compiled successfully by current compilers. In addition, a great deal of time was spent modifying the original test cases for the current simulation environments. If no such compatibility problems existed, the measurements would have been completed faster.

17.6 References

[APP, Y1] "APP Module SF1 System Software code," Year Y1.

[APP, Y2] "APP SF1 Flux/Delta Flux/Flow Application code," Year Y2.

[APP, Y3] "APP Module µp2 System Software Source Code Listing," Year Y3.

[APP, Y4] "APP µp2 Flux/Delta Flux/Flow Application Software Source Code Listing," Year Y4.

[APP, Y5] "APP Communication Processor Source Code," Year Y5.

[APP, Y6] "APP Test Plan for µp1 Software," Year Y6.

[APP, Y7] "APP Test Plan for µp2 Software," Year Y7.

[APP, Y8] "APP Test Plan for CP Software," Year Y8.

[APP, Y9] "APP Test Report for µp1 Software," Year Y9.

[APP, Y10] "APP Test Report for µp2 Software," Year Y10.

[APP, Y11] "APP Test Report for CP Software," Year Y11.

[IEEE, 1988] "IEEE Guide for the use of Standard Dictionary of Measures to Produce Reliable Software," IEEE Std. 982.2-1988, 1988.

[Malaiya, 1993] Y. Malaiya, A.V. Mayrhauser and P. Srimani. "An Examination of Fault Exposure Ratio," *IEEE Transactions on Software Engineering*, vol. 19, pp. 1087–94, 1993.

[Malaiya, 1996] Y. Malaiya et al. "Software Test Coverage and Reliability," Colorado State University, Fort Collins, CO, 1996.

[Musa, 1987] J.D. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Applications*. New York: McGraw-Hill, 1987.

18. REAL RELIABILITY ASSESSMENT

18.1 Definition

In this study, "reliability" is defined as "the probability that the APP software (both system software and application software) functions normally within a one demand performance period."

Traditionally, the reliability of a system is estimated from failure data. The failure data is obtained either from operational failures or failures discovered during testing. In NUREG/CR-6848 [Smidts, 2004], an automatic testing environment was established and the software under study was tested using that test-bed. In this study, operational failures will be used to quantify the APP reliability.

Let us assume *r* failures are observed in *T* years of operating time. The maximum likelihood and unbiased estimate of the failure rate $\hat{\lambda}$ is given as [Ireson, 1966]:

$$\hat{\lambda} = \frac{r}{T} \tag{18.1}$$

18.2 APP Testing

During the early stages of this research, UMD was unable to obtain operational data from the plant. Thus, UMD initiated its reliability-estimation effort using testing. A test-bed was established to conduct reliability testing for the APP safety module (see Figure 18.1).

The test-bed was composed of a testing computer, which executed the testing software and provided inputs to the APP module and accepted outputs from the APP module. One PCI A/D card was installed for accepting the analog APP outputs and converting them to digital values; one D/A PCI card was installed for generating APP analog inputs; and one digital Input/Output (I/O) card was installed to establish bi-directional communication between the testing computer and the APP module. The wiring between the safety module and the testing computer was designed and implemented by the APP manufacturer.

Interface software also was developed to generate and provide inputs into the APP module and to accept and display APP module outputs. Figure 18.2 depicts this testing software interface. The user can enter analog input values and digital input values in the two left columns. After pressing

the Start button, the values in these text boxes were sent to the APP. The outputs from the APP were retrieved and appeared in the two right columns.

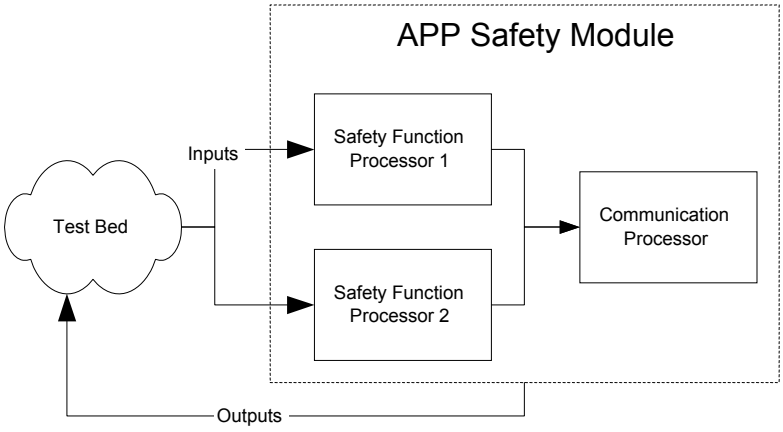


Figure 18.1 APP Reliability Testing Environment

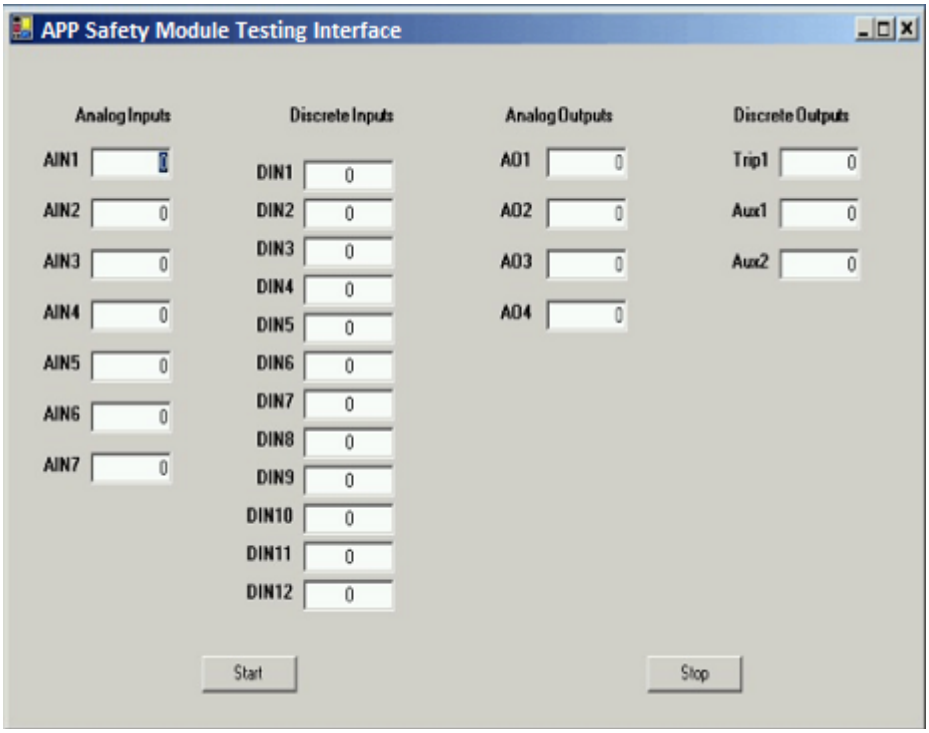


Figure 18.2 Testing Software

UMD did not follow the procedures described in NUREG/CR-6848 Chapter 4 [Smidts, 2004] to conduct APP reliability testing. In particular, generation of test cases from the TestMaster model was not performed because UMD was in the possession of a large amount of actual operational input data. The TestMaster model should be used when actual input information is unavailable, and to generate inputs that represent operational use using the operational profile.

WinRunner was used as described in NUREG/CR-6848 Chapter 4 [Smidts, 2004] to harness the testing automatically.

Within the test environment, one method to speed up the failure process was to use accelerated testing techniques. The principle of accelerated testing is to challenge a system under high stress. For mechanical components, meaningful high stresses include higher temperatures, higher voltages, higher speed of operation, etc. This technique forces the component into conditions rarely attained during normal operation. For software components, the same principle applies. High stress conditions correspond to inputs that rarely appear in normal operations.

Two types of high-stress conditions were identified for the APP. The first set of high-stress conditions is related to the application software, i.e., the inputs around the trip conditions (points around the "barn shape"). UMD identified this set of conditions as the least important because these conditions largely challenge the application software (the predicates that judge whether a trip occurs). This part of the software is relatively simple and typically less problematic than the remaining system software.

Another set of high-stress conditions was identified for the system software. The APP system software was designed to assure that the safety module (both software and hardware) was in healthy condition. Since hardware failures have a low likelihood of occurrence, it is important to fictitiously increase these likelihoods, in other words, to accelerate them, to observe module behavior under this type of high-stress condition.

However, the current testing configuration depicted in Figure 18.1 was not suitable for accelerating such stress (hardware failure). A simulation-based accelerated testing was considered a possible approach in future testing.

In summary, it was not possible to derive APPs failure rates from testing. Therefore, UMD contacted a plant using similar software to acquire failure information from operational data.

18.3 APP Operational Data

The APP had been deployed in a nuclear power plant and had been functioning for 10 years at the start of this research project. Per UMD's request, the plant sent UMD copies of the plant maintenance Work Packages addressing APP failures. The Work Packages included 14 Problem Records that were related to the APP module. Each Problem Record consisted mainly of a

detailed problem description and a corresponding set of corrective actions. Table 18.1 summarizes the 14 Problem Records. The table includes the date at which the Problem Record originated, the Problem Record Number, a determination of whether the problem was related to an APP failure, a determination of whether it was an APP software failure, the failure type (i.e., Type I or II), and the cause of the failure. From Table 18.1, several conclusions can be drawn as follows:

1. A new version of the APP software was installed in the plant. The implementation was completed on December 2003 for one unit, and the completion dates for the other units were March 2004 and October 2004, respectively. A new EEPROM on a computer card was installed into the applicable RPS APP module. The software update was not the result of a failure. The existing Flux Imbalance/Flow trip limits were determined to perform their intended functions appropriately. A maximum power trip set-point was added so that the module would trip the plant at a predefined fixed, set power level if the measured power level exceeded the fixed power level due to increased flow. This Problem Record recorded the fact that several tuning parameters were set to new values due to a change of the core design. This thus defined the end point of the window of analysis.

2. Among the 14 Problem Records there were 5 APP system failures (Numbers 6, 8, 9, 10, and 12). Not all APP system failures were related to the APP software. Three of the failures (Numbers 8, 9, and 10) were identified by the system developer as hardware failures.

3. For the sixth Problem Record, the system developer could not determine the cause of failure. None of the testing or other diagnostic efforts identified a failed component or any other problem. As a consequence, the plant owner did not upgrade the APP software but replaced some critical hardware components, such as a voltage regulator. UMD conservatively considers this Problem Record as an APP software failure. This failure was a Type II failure because it produced a trip signal although plant parameters were normal.

4. For the twelfth Problem Record, the system developer could not determine the cause of failure. UMD conservatively considers this Problem Record as an APP software failure. This failure was a Type II failure because it sent out a trip signal although plant parameters were normal.

5. One more APP failure besides those specified in the 14 Problem Records was identified by UMD after a thorough analysis of Problem Record O-02-00463. An AVIM (Analog Voltage Isolation Module) failure caused a failure of the APP system. Therefore, it was concluded that this Problem Record was not due to an APP software failure.

6. Obviously, throughout the total deployment time of the APP software version 1 there was no software reliability growth because software defects could not be located.

7. On the other hand, some APP module hardware was replaced when those modules were sent back to the system developer. Thus, the operational profile for the APP infrastructure inputs needed to be updated to reflect these changes (see Chapter 4).

Table 18.1 Summary of Problem Records

NO.	Problem Record #	APP Failure?	APP Software Failure?	Failure Type	Reason
1	O-98-00932	No	No	N/A	There was a connection problem when attempting to reinsert APP module into the cabinet.
2	O-98-02070	No	No	N/A	The instrumentation is capable of performing its intended function and there was no operability issue.
3	O-98-03661	No	No	N/A	Nothing abnormal happened and no Problem Record should be written.
4	O-99-05230	No	No	N/A	No equipment failure or loss of system/component function is involved.
5	O-00-01770	No	No	N/A	Developer believed that it was impossible to have a particular common element failed because no failure of any components in any of the modules was observed in the plant. This repeatedly out of tolerance calibration problem would due to the problem in CTC.
6	O-01-03095	Yes	Yes	II	APP module tripped while it should not. The failure cause was not able to be determined. None of the testing or other diagnostic efforts performed by developer identified a failed component or any other problem.

Table 18.1 Summary of Plant X's Problem Records (continued)

NO.	Problem Record #	APP Failure?	APP Software Failure?	Failure Type	Reason
7	O-01-03118	No	No	N/A	Module was not seated well due to loose connection.
8	Unknown	Yes	No	N/A	AVIM (analog voltage isolation module) failure.
9	O-02-00463	Yes	No	N/A	Loss of RC (Runs Commands) flow indication due to a failed AVIM (analog voltage isolation module). The cause of the AVIM failure is unknown.
10	O-02-01360	Yes	No	N/A	APP module tripped while it should not. The evaluation identified a failed 5V DC regulator as the failed component.
11	O-03-02646	No	No	N/A	Not an actual failure. Several tuning parameters were set to new values due to the change of the core design.
12	O-03-08237	Yes	Yes	II	APP module tripped while it should not. No reason was identified.
13	O-04-01439	No	No	N/A	CTC need to be calibrated or caused by a calculating rounding problem according to developer's answer.

Table 18.1 Summary of Plant X's Problem Records (continued)

NO.	Problem Record #	APP Failure?	APP Software Failure?	Failure Type	Reason
14	3/25/2005	No	No	N/A	Company Y identified a software error. The error results in a 0.2% FP non-conservative trip setpoint for the Flux/Flow/Imbalance and only one of the modules is affected. However, this error is within the hardware tolerance and has not shown up in required testing performed routinely in Plant X. Plant X did not implement the changes to their APP modules.
15	4/1/2005	No	No	N/A	Same as Problem Record #14.

The power plant control logic was comprised of three independent control units. Each unit contained four channels; each channel contained one APP safety module. The following table shows the deployment of the APP modules in the plant.

Table 18.2 Deployment of APP Modules in Plant

Unit Number	Deployed in Plant From	End Deployment Date in this Study	Total Deployment Time
1	December 1997	December 2003	73 months
2	May 1996	March 2004	95 months
3	June 1995	October 2004	113 months

The number of demands over the deployment period for the APP modules in the plant is:

$$D = \frac{T}{\tau} = \frac{(73 + 95 + 113)(30)(24)(3600)}{0.129} = 5.646 \times 10^{9} \text{ demands}$$

Where τ is the average execution-time-per-demand determined through the simulation environment. Its value is 0.129 s. For additional detail refer to Section 17.3.4.

The probability of failure per demand ($\hat{\lambda}_{Trial}$) for the APP system can be estimated using Equation 18.1:

$$\hat{\lambda}_{Trial} = \frac{r}{D} = \frac{2 \text{ failures}}{5.646 \times 10^{9} \text{ demands}} = 3.542 \times 10^{-10} \text{ failure/demand} \qquad (18.2)$$

Because $\hat{\lambda}_{Trial} = \hat{\lambda} \times \tau$, the APP system's failure rate ($\hat{\lambda}$) can be estimated using the following:

$$\hat{\lambda} = \frac{r}{T} = \frac{2 \text{ failures}}{(73 + 95 + 113 \text{ months})(30 \text{ days/month})(24 \text{ hr/day})(3600 \text{ s/hr})} = (2.746 \times 10^{-9})^{s^{-1}}$$

The failure types of the APP software failures described in the Problem Records can be identified. As shown in Table 18.1, the failures were determined as Type II failures. Thus, the APP Type II rate of failure is also 2.746×10^{-9} failure per second and the probability of a type II failure-per-demand is 3.542×10^{-10} failure per demand.

However, no Type I failure was observed during the period of investigation. Thus, Equation 18.1 does not apply for failure-rate estimation of APP Type I failures. UMD opted for a statistical approach to estimate the failure rate based on field data knowing that no type I failures had been observed.

A common solution to failure-rate estimation when no failure event has been observed is to take one half as the numerator (r) in Equation 18.1 [Welker, 1974].

The APP failure rate is thus given by:

$$\lambda = \frac{r}{T} = \frac{0.5}{(73 + 95 + 113)(30)(24)} = 2.471 \times 10^{-6} \text{ failure/hour}$$

The probability that a trip actuation will be required can be estimated using Equation 4.6:

$$\lambda = \frac{r}{T} = \frac{10}{72511} = 1.38 \times 10^{-4} \ \frac{\text{trip}}{\text{hr}}$$

Therefore, the APP type I failure-rate estimate is given by:

$$P(Type\ I\ Failure | Trip\ is\ Required) = \frac{P(Type\ I\ Failure\ and\ Trip\ is\ Required)}{P(Trip\ is\ Required)}$$

$$= \frac{2.471 \times 10^{-6} \text{ failure/hr}}{1.38 \times 10^{-4} \text{ trip/hr}} = 0.01792 \frac{Type\ I\ Failure}{Trip\ Required}$$

18.4 References

[Ireson, 1966] W.G. Ireson. *Reliability Handbook*. New York, NY: McGraw Hill, Inc., 1966.

[Smidts, 2004] C. Smidts and M. Li, "Validation of a Methodology for Assessing Software Quality," NRC, Office of Nuclear Regulatory Research, Washington DC NUREG/CR-6848, 2004.

[Welker, 1974] E.L. Welker and M. Lipow. "Estimating the Exponential Failure Rate from Data with No Failure Events," in *Proc. Annual Reliability and Maintainability Conference*, 1974.

19. RESULTS

The motivation of this project was to validate the RePS theory and the rankings presented in NUREG/GR-0019.

In previous research, as shown in NUREG/CR-6848, a first set of six RePS models was constructed for the following six root measures: Requirements Traceability (RT), Mean Time to Failure (MTTF), Defect Density (DD), Bugs per Line of Code (BLOC), Function Point (FP), and Test Coverage (TC). These models were applied to two small scale systems, the Personnel Access Control System, PACS1 and PACS2, and it was found that the results of the assessment were consistent with the ranking of the measures.

The research described in this report is a continuation of NUREG/CR-6848. Seven more RePS models were developed and were applied to a nuclear safety critical system, the APP. New RePSs were built for the measures: Cyclomatic Complexity (CC), Cause and Effect Graphing (CEG), Requirements Specification Change Requests (RSCR), Fault-days Number (FDN), Capability and Maturity Model (CMM), Completeness (COM), and Coverage Factor (CF).

It should be pointed out that it is not necessary to validate the methodology using all 40 measures identified in NUREG/GR-0019. Based on the methodology provided in this NUREG/CR report, projects have the flexibility to select their own measures for software-reliability prediction. The selection criteria include the measure's prediction ability, the measure's availability to the specific project considering cost and schedule constraints. For example, when reviewing new nuclear reactor applications, reviewers may select measures with higher prediction ability dependent upon their reviewing schedule.

In the current study, the MTTF measure was not applied to the APP and an alternative approach for assessing the failure rate was introduced in Chapter 18. As described in Section 18.3, APP failures were identified from the Problem Investigation Process (Problem Records) and the failure rate of the APP was assessed as the identified number of APP failures divided by the total APP deployment time. The other twelve RePSs are used to predict the software reliability of the APP system.

A summary description of the twelve measures is provided in Section 19.1. The results of the RePS software reliability predictions are displayed and analyzed in Section 19.2. These predictions are then validated by a comparison to the "real" software reliability obtained from operational data and statistical inference. The comparison between the NUREG/GR-0019 ratings and the RePS prediction error is also made in this section, and the efficacy of the proposed methodology for predicting software quality is determined.

Further discussion about the measurement process for the twelve measures used in this research is provided in Section 19.3. 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.

Section 19.4 discusses the difficulties encountered during the measurement process as well as the possible solutions. Conclusions, a list of follow-on issues, and their priorities ranked by an expert panel composed of field experts are presented in Section 19.5 and 19.6, respectively.

19.1 Summary of the Measures and RePSs

Twelve measures were selected and their associated RePSs were created. A summary description of the measures and RePSs are presented in Section 19.1.1 and 19.1.2, respectively.

19.1.1 Summary Description of the Measures

Table 19.1 presents a summary of the twelve measures with their applicable life cycle phases and the phases for which they were applied to the APP system. The specific documents required to perform the measurements are also specified.

Table 19.1 A Summary of Measures Used

Family	Measures	Applicable Life Cycle Phases[40]	Applied Phases for APP[41]	Required Documents
Estimate of Faults Remaining per Unit of Size	BLOC	IM, TE, Operation	Operation	Code
Cause and Effect Graphing	CEG	RQ, DE, IM, TE, Operation	Operation	SRS, Code
Software Development Maturity	CMM	RQ, DE, IM, TE, Operation	Operation	SRS, SDD, Code
Completeness	COM	RQ, DE, IM, TE, Operation	Operation	SRS, Code
Fault-Tolerant Coverage Factor	CF	TE, Operation	Operation	Code

[40] RQ, DE, IM, and TE stand for Requirements phase, Design phase, Coding phase, and Testing phase respectively.
[41] It is assumed that the version used during operation is the version that was delivered at the end of the testing phase.

Table 19.1 A Summary of Measures Used (continued)

Family	Measures	Applicable Life Cycle Phases[42]	Applied Phases for APP[43]	Required Documents
Module Structural Complexity	CC	DE, IM, TE, Operation	Operation	Code
Time Taken to Detect and Remove Faults	FDN	RQ, DE, IM, TE, Operation	Operation	SRS, SDD, Code
Functional Size	FP	RQ, DE, IM, TE, Operation	Operation	SRS
Faults Detected per Unit of Size	DD	TE, Operation	Operation	SRS, SDD, Code
Requirements Specification Change Request	RSCR	RQ, DE, IM, TE, Operation	Operation	SRS, Code
Requirement Traceability	RT	DE, IM, TE, Operation	Operation	SRS, Code
Test Coverage	TC	TE, Operation	Operation	Code

As shown in Table 19.1, all measurements are performed during the APP operation phase. Focus on the operation phase is driven by the time elapsed between delivery of the APP system and the consequent unavailability of important historical information that could have characterized the software-development process. For example, one can measure the FP count in the Requirement phase using an early version of the SRS. This would yield an estimate of reliability based on FP early in the development life-cycle. Unfortunately, these early versions of the APP SRS are no longer available. The only SRS version available is the final version, i.e., the version that was delivered at the end of the testing phase.

According to the properties of the defects found by different families/measures, the above 12 families/measures can be categorized into three groups.

Group-I Families/Measures:

- Estimate of Faults Remaining per Unit of Size/Bugs per Line of Code (BLOC)
- Software Development Maturity/Capability Maturity Model (CMM)

[42] RQ, DE, IM, and TE stand for Requirements phase, Design phase, Coding phase, and Testing phase respectively.
[43] It is assumed that the version used during operation is the version that was delivered at the end of the testing phase.

- Module Structural Complexity/Cyclomatic Complexity (CC)
- Functional Size/Function Point (FP)
- Requirements Specification Change Request/Requirements Specification Change Request (RSCR)

Group-II Families/Measures:

- Cause and Effect Graphing/Cause-effect Graphing (CEG)
- Completeness/Completeness (COM)
- Faults Detected per Unit of Size/Defect Density (DD)
- Requirement Traceability/Requirements Traceability (RT)
- Fault-Tolerant Coverage Factor/Coverage Factor (CF)

Group-III Family/Measure:

- Time Taken to Detect and Remove Faults/Fault Days Number (FDN)
- Test Coverage/Test Coverage (TC)

In the case of the first group of families/measures, only the number of defects can be obtained. Their location is unknown. The RePSs for these measures are based on Musa's estimation model. Families/measures in the second group correspond to cases where actual defects are obtained through inspections or testing. Thus, the exact location of the defects and their number is known. Extended Finite State Machine Models (see Chapter 5) or Markov Chain Models (see Chapter 10) are used to assess reliability. The measures in the third group have the combinational characteristics of the first two groups. The exact locations of defects in an earlier version are used to build the fault location models to obtain a software-specific fault exposure ratio and the final reliability estimation is based on Musa's estimation model.

It should be noted that seven out of the twelve measures are unique measures in their families while UMD selected one of the measures in the other five families. This information is presented in Table 19.2.

Table 19.2 Family/Measure Information

Family	Measure(s) in This Family	UMD Selected Measure
Estimate of Faults Remaining per Unit of Size	BLOC	BLOC
Cause and Effect Graphing	CEG	CEG
Software Development Maturity	CMM	CMM

Table 19.2 Family/Measure Information (continued)

Family	Measure(s) in This Family	UMD Selected Measure
Completeness	COM	COM
Module Structural Complexity	Cyclomatic Complexity (CC)	CC
	Minimal Unit Test Case Determination	
Faults Detected per Unit of Size	Code Defect Density (DD)	DD
	Design Defect Density	
	Fault Density	
Fault-Tolerant Coverage Factor	CF	CF
Time Taken to Detect and Remove Faults	Fault Days Number (FDN)	FDN
	Man Hours per Major Defect Detected	
Functional Size	Function Point (FP)	FP
	Feature Point Analysis	
	Full Function Point	
Requirements Specification Change Request	RSCR	RSCR
Requirement Traceability	RT	RT
Test Coverage	Test Coverage (TC)	TC
	Functional Test Coverage	
	Modular Test Coverage	

Since BLOC, CEG, CMM, COM, CF, RSCR, and RT are the only measure in their respective families, these were automatically selected. The selection of the other measures among the members of their families is based on Table 19.3.

Table 19.3 Information about Families Containing More Than One Measure

Family	Measures	Experts Rate
Module Structural Complexity	Cyclomatic Complexity (CC)	0.72
	Minimal Unit Test Case Determination	0.7
Faults Detected per Unit of Size	Code Defect Density (DD)	0.83
	Design Defect Density	0.75
	Fault Density	0.75
Time Taken to Detect and Remove Faults	Fault Days Number (FDN)	0.72
	Man Hours per Major Defect Detected	0.63
Functional Size	Function Point (FP)	0.5
	Feature Point Analysis	0.45
	Full Function Point	0.48
Test Coverage	Test Coverage (TC)	0.68
	Functional Test Coverage	0.62
	Modular Test Coverage (MTC)	0.7

As seen in the above table, the experts' rates for each measure in a family do not vary much. Basically, either measure can be chosen to represent its family. The fundamental criterion that UMD used was to choose the measures with the highest rates. Thus, CC, DD, FDN, FP, and MTC are automatically chosen. However, considering the difficulties of their RePS constructions, TC was selected to replace the MTC measure because the RePS for TC is available while no RePS exists for the MTC measure. Therefore, UMD selected CC, DD, FDN, FP, and TC to represent their families.

19.1.2 Summary Description of the RePSs

Twelve RePSs were used to predict the software reliability of the APP system; they are summarized in Table 19.4. Multiple key elements are involved in the evaluation of each RePS.

Some can be measured directly using available documents, others can only be estimated. Table 19.4 also lists these key elements and whether they can be measured or should be estimated.

Table 19.4 Summary of the RePSs

Measure	Estimation of Probability of Failure	Key Elements	Measured or Estimated
BLOC	$P_f = 1 - e^{-\frac{K}{T_L}\tau\left(\sum_{i=1}^{M}\left(4.2+0.0015S_i^{4/3}\right)\times SL - N_{found}\right)}$	K: fault exposure ratio	Estimated
		M: the number of modules	Measured
		S_i: the number of lines of code (LOC) for each module	Measured
		N_{found}: the number of known defects found by inspection and testing	Measured
		T_L: linear execution time	Measured
		τ: the average execution-time-per-demand	Measured
		SL: Severity level	Estimated
CEG	$P_f = \oint_{OP,N_{CEG}} P(i)\cdot I(i)\cdot E(i)$	N_{CEG}: the number and locations of defects found by the CEG measure	Measured
		OP: Operational profile	Measured
		$P(i)$: the propagation probability for the *i*-th defect	Measured
		$I(i)$: the infection probability for the *i*-th defect	Measured
		$E(i)$: the execution probability for the *i*-th defect	Measured
CMM	$P_f = 1 - e^{-\frac{K}{T_L}\tau N_{CMM}}$	K: fault exposure ratio	Estimated
		N_{CMM}: the number of defects estimated by the CMM measure	Estimated
		T_L: linear execution time	Measured
		τ: the average execution-time-per-demand	Measured
COM	$P_f = \oint_{OP,N_{COM}} P(i)\cdot I(i)\cdot E(i)$	N_{COM}: the number and locations of defects found by the COM measure	Measured
		OP: Operational profile	Measured
		$P(i)$: the propagation probability for the *i*-th defect	Measured
		$I(i)$: the infection probability for the *i*-th defect	Measured
		$E(i)$: the execution probability for the *i*-th defect	Measured

Table 19.4 Summary of the RePSs (continued)

Measure	Estimation of Probability of Failure	Key Elements	Measured or Estimated
CF	$P_f = 1 - \sum_{i=1}^{3} p_i(t)$	$P_i(t)$: the probability that the system remains in the i-th reliable state. i = 1, 2, and 3, corresponding to the Normal, the Recoverable, and the Fail-safe states.	Measured
CC	$P_f = 1 - e^{-\frac{K}{T_L}\tau\left(k \cdot A \cdot F^{1-2SLI_{CC}}\right)}$	K: fault exposure ratio	Estimated
		A: the size of the delivered source code in terms of LOC	Measured
		k: a universal constant	Estimated
		F: a universal constant	Estimated
		SLI_{CC}: the Success Likelihood Index for the CC measure	Estimated
		T_L: linear execution time	Measured
		τ: the average execution-time-per-demand	Measured
DD	$P_f = \oint_{OP,N_{DD}} P(i) \cdot I(i) \cdot E(i)$	N_{DD}: the number and locations of defects found by the DD measure	Measured
		OP: Operational profile	Measured
		$P(i)$: the propagation probability for the i-th defect	Measured
		$I(i)$: the infection probability for the i-th defect	Measured
		$E(i)$: the execution probability for the i-th defect	Measured
FDN	$P_f = 1 - e^{-\frac{K}{T_L}\tau N_{FDN}}$	K: fault exposure ratio	Estimated
		N_{FDN}: the number of defects estimated by the FDN measure	Estimated
		T_L: linear execution time	Measured
		τ: the average execution-time-per-demand	Measured
FP	$P_f = 1 - e^{-\frac{K}{T_L}\tau N_{FP}}$	K: fault exposure ratio	Estimated
		N_{FP}: the number of defects estimated by the FP measure	Estimated
		T_L: linear execution time	Measured
		τ: the average execution-time-per-demand	Measured

Table 19.4 Summary of the RePSs (continued)

Measure	Estimation of Probability of Failure	Key Elements	Measured or Estimated
RSCR	$P_f = 1 - e^{-\frac{K}{T_L}\tau\left(k \cdot A \cdot F^{1-2SLI_{RSCR}}\right)}$	*K*: fault exposure ratio	Estimated
		A: the size of the delivered source code in terms of LOC	Measured
		k: a universal constant	Estimated
		F: a universal constant	Estimated
		SLI_{RSCR}: the Success Likelihood Index for the RSCR measure	Estimated
		T_L: linear execution time	Measured
		τ: the average execution-time-per-demand	Measured
RT	$P_f = \oint_{OP,N_{RT}} P(i) \cdot I(i) \cdot E(i)$	N_{RT}: the number and locations of defects found by the RT measure	Measured
		OP: Operational profile	Measured
		P(i): the propagation probability for the *i*-th defect	Measured
		I(i): the infection probability for the *i*-th defect	Measured
		E(i): the execution probability for the *i*-th defect	Measured
TC	$vK = \frac{\oint_{OP,N_0} P(i) \cdot I(i) \cdot E(i)}{N_0}$ $P_f = 1 - e^{-\frac{vK \times N_0}{a_0 \ln[1 + a_1(e^{a_2 c_1} - 1)]}}$	vK: fault exposure ratio	Measured
		N_0: the number and locations of defects found by testing in earlier version of code	Measured
		OP: Operational profile	Measured
		P(i): the propagation probability for the *i*-th defect	Measured
		I(i): the infection probability for the *i*-th defect	Measured
		E(i): the execution probability for the *i*-th defect	Measured
		a_0, a_1, a_2: coefficients	Estimated
		C_1: test coverage	Measured

The current regulatory review process does not use metrics to assess the potential reliability of digital instrumentation and control systems in quantitative terms. The goal of the research described in this report was to identify methods that could improve the regulatory review process by giving it a more objective technical basis. While some of the models in this report use generic industry data, experimental data, and subjective assessments, much of the modeling is based on direct measurements of the application under study and, as such, is purely objective in nature.

Thus, the use of the proposed RePSs models (i.e., of the highly accurate RePSs) could potentially yield better results than what can be obtained from the current review process.

19.2 Results Analysis

This section presents a detailed analysis of the results, which includes an analysis of the number of defects estimated or measured by the twelve software engineering measures and an analysis of the reliability predictions.

19.2.1 Defects Comparison

The total number of Level-1 and Level-2 defects remaining in the APP source code according to the twelve measures is shown in Table 19.5 and also illustrated in Figure 19.1.

Table 19.5 Number of Defects Remaining in the Code

Measure	Number of Defects Found
BLOC	14
CEG	1
CMM	19
COM	1
CF	6
CC[44]	29
DD	4
FDN	1
FP	10
RSCR[45]	12
RT	5
TC	9

[44] The Number of Defects Found for CC is the number of faults remaining obtained without the use of support measures.
[45] The Number of Defects Found for RSCR is the number of faults remaining obtained without the use of support measures.

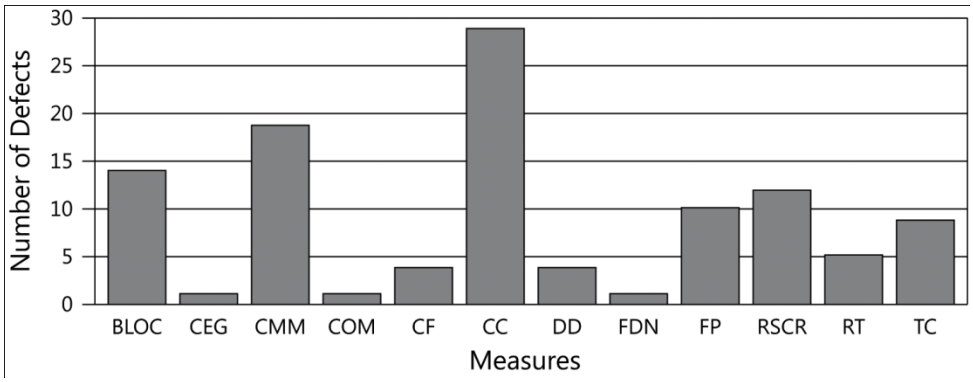


Figure 19.1 Number of Defects Remaining in the Code Per Measure

If the measures are re-ordered according to the groups defined in Section 19.1, Figure 19.1 becomes Figure 19.2. One can see from Figure 19.2 that the "number of defects remaining" estimated using Group-I measures is much larger than the actual number of defects remaining using Group-II measures. As discussed before, only an estimated number of defects can be obtained by Group-I measures. These estimated numbers are mainly based on Capers Jones' historical data. However, Jones' data does not cover the entire spectrum of system types. That is, there is no specific data available for safety critical software. In this research, data in the general "system software" category was used. As a consequence, the number of defects will be overestimated. In the case of Group-II measures, defects are uncovered through inspection. It is clear that the inspector will not uncover all the defects in the software. Thus, the number of defects is a lower bound estimate of the actual number of defects. These reasons explain why the number of defects predicted by Group-I measures is generally much larger than the number found by Group-II measures.

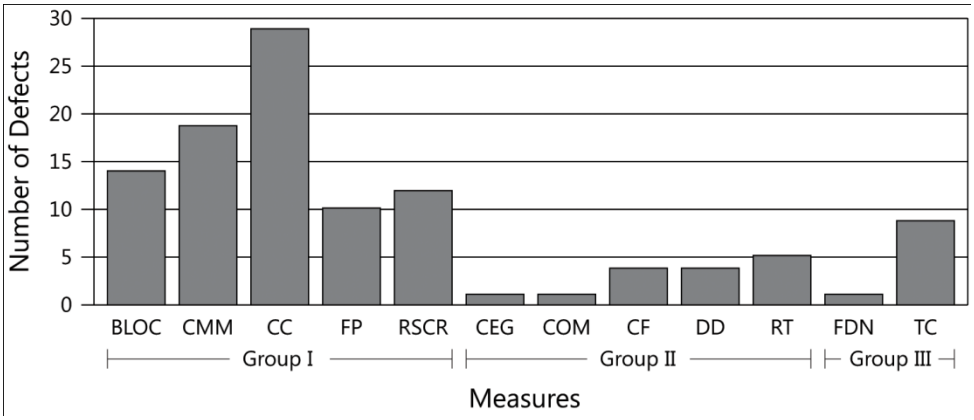


Figure 19.2 Number of Defects Remaining Per Measure Per Group

An analysis of defect characteristics is provided in Section 19.2.1.1. The actual number of defects can be estimated using a capture-recapture model. The derivation is shown in 19.2.1.2.

19.2.1.1 Discussion about Measures in the Second Group

The obvious advantage of the measures in the second group is that the exact location of the defects can be determined. Table 19.6 provides the number of the defects found by measures in Group-II.

Table 19.6 Defects Found by the Measures in the Second Group

Measures	Total Number of Defects	Number of Level-1 and Level-2 Defects Only	Number of Defects Remaining in the Code (Level-1 and Level-2 only)
CEG	7	7	1
COM	113	29	1
DD	11	6	4
RT	7	5	5
CF	6	6	6

It should be noted that only the defects remaining in the code need to be considered to predict the reliability of the software system. A detailed description of all Level-1 and Level-2 defects remaining in the code is provided in Table 19.7.

Table 19.7 Detailed Description of Defects Found by the Second Group of Measures (Still Remaining in the Code)

No.	Defect Description	CEG	COM	DD	RT	CF
1	The check algorithm of μp1 cannot detect coupling failures between address lines.			X	X	
2	The function "Copy the contents of the table to the Dual Port RAM" is not implemented in μp1 source code.				X	
3	The function "Give up the semaphore" is not implemented in μp1 source code.				X	

Table 19.7 Detailed Description of Defects Found by the Second Group of Measures (Still Remaining in the Code) (continued)

No.	Defect Description	CEG	COM	DD	RT	CF
4	The check algorithm of µp2 cannot detect coupling failures between address lines.				X	
5	The address lines test does not cover all 16 address lines of µp2.			X		
6	The application program of µp2 has a logic problem.			X		
7	The check algorithm of CP cannot detect coupling failures between address lines.			X	X	
8	The logic to enter the CP diagnostics test is problematic.	X	X			
9	Cannot detect incorrect value of the variable SA_TRIP_1_DEENRGZE.					X
10	Cannot detect incorrect value of the variable fAnalog_Input_6.					X
11	Cannot detect incorrect value of the variable Trip_condition.					X
12	Cannot detect incorrect value of the variable AIN[4].					X
13	Cannot detect incorrect value of the variable chLEDs_Outputs.					X
14	Cannot detect incorrect value of the variable have_dpm.					X

Table 19.8 presents the exact location of each defect in the delivered source code. From Table 19.6 and 19.7, it is obvious that each measure discovered almost totally different defects. Only three out of the fourteen defects were simultaneously found by more than one measure. This implies the objectives of the measures are different and can be used to find different types of defects in the SRS, SDD, and code. It is also possible that a defect could not be found using only

these measures but the likelihood of this is very low, although it may be impossible to discover all existing defects through these measures.

19.2.1.2 Obtaining the Actual Number of Defects Remaining in the APP

Defects in the APP source code were identified through the Group-II measures. Unknown remaining defects in the APP system may still contribute to failure—ignoring them will result in an overestimation of reliability. The use of Capture/Recapture (C/R) models has been proposed to estimate the number of defects remaining in a software engineering artifact after inspection [Briand, 1997]. To determine the number of remaining defects, it is necessary to discuss C/R models, their use in software engineering, and their application specifically to the APP system.

The five measures in Group-II were assigned to five inspectors whose abilities to detect defects were different. In addition, different defects have different detection probabilities. The C/R model introduced in NUREG/CR-6848 was applied to estimate the number of defects remaining in the APP.

Table 19.8 Detailed Description of the Defects

No.	Defect Description	Micro-processor	Modes	Module
1	The check algorithm of μp1 cannot detect coupling failures between address lines.	μp1	Power-on Normal	VAddr_Lines_Test()
2	The function "Copy the contents of the table to the Dual Port RAM" is not implemented in μp1 source code.	μp1	Calibration Tuning	VCalibrate_Tune()
3	The function "Give up the semaphore" is not implemented in μp1 source code.	μp1	Calibration Tuning	VCalibrate_Tune()
4	The check algorithm of μp2 cannot detect coupling failures between address lines.	μp2	Power-on Normal	address_line_test()

Table 19.8 Detailed Description of the Defects (continued)

No.	Defect Description	Micro-processor	Modes	Module
5	The address lines test does not cover all 16 address lines of μp2.	μp2	Power-on, Normal	address_line_test()
6	The application program of μp2 has a logic problem.	μp2	Normal	update_application()
7	The check algorithm of CP cannot detect coupling failures between address lines.	CP	Power-on Calibration Tuning	Addr_Line_Test()
8	The loop condition of CP's PROM test is problematic.	CP	Power-on Calibration Tuning	Chksum_Proc()
9	Cannot detect incorrect value of the variable SA_TRIP_1_DEENRGZE.	μp1	Normal	serial interrupt function
10	Cannot detect incorrect value of the variable fAnalog_Input_6.	μp1	Normal	application program
11	Cannot detect incorrect value of the variable Trip_condition.	μp2	Normal	application_function
12	Cannot detect incorrect value of the variable AIN[4].	μp2	Normal	application_function
13	Cannot detect incorrect value of the variable chLEDs_Outputs.	μp1	Normal	Generate front panel LEDs output signals function
14	Cannot detect incorrect value of the variable have_dpm.	μp2	Normal	get_Semaphone

Defects found by the Coverage Factor measure have different characteristics than defects found by the four other Group-II measures: their detectability does not depend on the inspector's ability. Thus, the C/R model was only applied to the four other measures (CEG, COM, DD, and RT) to obtain the actual number of defects remaining in the APP system.

In the NUREG/CR-6848 study, the C/R model was applied only to the results of the Defect Density measurement that was performed by multiple inspectors. The defects were at the same level of detail. However, in this study, UMD attempted to apply the C/R model in the case of multiple-measurement approaches and the defects discovered may not be at the same level of detail. For example, Defect Density should discover defects more detailed than those discovered by Requirement Traceability. Yet it is necessary to maintain all defects at the same level of detail. That is, each defect represents only one functional problem, which is a numbered item specified in the SRS. Applying this new criterion to the defects found by the Group-II measures, Table 19.7 is modified as shown in Table 19.9. For example, the second and third defect were discovered using the Requirement Traceability measure. These two defects were affecting two sub-functions in the Calibration function of μp1. The Calibration function is a numbered specification in μp1 SRS. Thus, these two defects should be grouped together.

Table 19.9 Modified Defects Description

No.	Defect Description	CEG	COM	DD	RT
1	The check algorithm of μp1 cannot detect coupling failures between address lines.			X	X
2	The Calibration function of μp1 is not correctly implemented in the source code.				X
3	The check algorithm of μp2 cannot detect coupling failures between address lines.				X
4	The address lines test does not cover all 16 address lines of μp2.			X	
5	The application program of μp2 has a logic problem.			X	
6	The check algorithm of CP cannot detect coupling failures between address lines.			X	X
7	The logic to enter the CP diagnostics test is problematic.	X	X		

The defect population size is given as:

$$\widehat{N}_i = \frac{D}{\hat{C}_i} + \frac{f_1}{\hat{C}_i}\hat{\bar{\gamma}}_i^2 \qquad i = 1, 2, 3 \tag{19.1}$$

Where

$\widehat{N}_i$ the i-th defect population size estimator

D the number of distinct defects found by t inspectors

f_1 the number of defects found by exactly one inspector

The term $\hat{C}_i$ in Equation 19.1 is given as:

$$\hat{C}_1 = 1 - \frac{f_1}{\sum_{k=1}^{t} k f_k} \tag{19.2}$$

$$\hat{C}_2 = 1 - \frac{f_1 - 2f_2\left(\frac{1}{t-1}\right)}{\sum_{k=1}^{t} k f_k} \tag{19.3}$$

$$\hat{C}_3 = 1 - \frac{f_1 - 2f_2\left(\frac{1}{t-1}\right) + 6f_3\left[\frac{1}{(t-1)(t-2)}\right]}{\sum_{k=1}^{t} k f_k} \tag{19.4}$$

and $\hat{\bar{\gamma}}_i^2$ is given as:

$$\hat{\bar{\gamma}}_i^2 = \max\left[\frac{\widehat{N}_{0,i}\sum_k k\,(k-1)f_k}{2\sum\sum_{j<k} n_j n_k} - 1{,}0\right] \qquad i = 1, 2, 3 \tag{19.5}$$

where

$$\widehat{N}_{0,i} = \frac{D}{\hat{C}_i} \qquad i = 1, 2, 3 \tag{19.6}$$

and

t the number of inspectors ($t = 4$)

n_j the number of defects found by the j-th inspector

f_k the number of defects found by exactly k inspectors, $k = 1, \ldots, t$

Table 19.10 shows the inspection information for the APP system:

Table 19.10 Inspection Results for the APP System

Measures	CEG	COM	DD	RT
Inspector	1	3	4	5
n_j	1	1	4	4

For the APP system:

The total number of distinct defects is $D = 7$
The number of defects found by one inspector is $f_1 = 4$
The number of defects found by two inspectors is $f_2 = 3$
The number of defects found by three inspectors is $f_3 = 0$
The number of defects found by four inspectors is $f_4 = 0$

The results of Equations 19.2 to 19.5 are shown in Table 19.11.

Table 19.11 Capture/Recapture Model Results for the APP System

i	**1**	**2**	**3**
$\hat{C}_i$	0.6	0.8	0.8
$\hat{\gamma}_i^2$	0.061	0	0
$\widehat{N}_i$	11.67	8.75	8.75

Sample coverage (SC), defined as the fraction of the detected defects, is calculated as follows:

$$SC_1 = \frac{D}{\widehat{N}_1} = \frac{7}{11.67} = 0.60 = 60.0\%$$

$$SC_2 = \frac{D}{\widehat{N}_2} = \frac{7}{8.75} = 0.80 = 80.0\%$$

$$SC_3 = \frac{D}{\widehat{N}_3} = \frac{7}{8.75} = 0.80 = 80.0\%$$

The point estimate of the defect-detection probability is given in Table 19.12.

Table 19.12 Defects Discovery Probability

Defect No.	Detection Probability (*Pi*)
1	0.5
2	0.25
3	0.25
4	0.25
5	0.25
6	0.5
7	0.5

From Table 19.12, the coefficient of variation (CV)—defined as the standard deviation of *p* over the arithmetic mean of *p*—is 0.374. A Jackknife model [Otis, 1978] is appropriate when CV < 0.4 and the sample coverage is greater than 0.50. This is the case for the APP system. By using a second-order Jackknife model, the result is:

$$\widehat{N} = \sum f_i + \frac{2t-3}{t} f_1 - \left[\frac{(t-2)^2}{t(t-1)}\right] f_2 = 4 + 3 + \frac{5}{4} \times 4 - \left[\frac{4}{12}\right] \times 3 = 11$$

Therefore, the best estimation of the number of remaining defects, $\widehat{N}$, for APP is 11. As addressed earlier in this section, the C/R model was applied to the seven defects identified through CEG, COM, DD, and RT-related inspections. These seven defects are listed in Table 19.9. Since the application of CF to APP was incomplete (due to time and resource constraints), the defects identified through CF were not included in the C/R analysis. Thus, the remaining APP defects estimated through C/R should not be compared with the 14 defects listed in Table 19.8, but with the seven defects listed in Table 19.9.

19.2.2 Reliability Estimation Comparison

As stated in Chapter 4, there are four operational modes in the APP system: power-on, normal, calibration, and tuning. The failure mechanisms in the power-on, calibration, and tuning modes are simple: any failure in these modes is considered a failure of the APP system. Unfortunately, the detailed, actual failure information in each mode is not available to UMD. Also, it is

unimportant to individually consider these modes because during its actual usage, APP will be bypassed during these modes. The most important mode is the normal operation mode. All data in the normal operational mode is available to UMD. The true failure probability was estimated successfully in Chapter 18 and will be used in the following to validate the RePSs and rankings.

The probability of failure and reliability estimation results from the twelve measures is shown in Table 19.13.

Table 19.13 Reliability Estimation Results

Measure	Probability of Failure (per demand)	Reliability (per demand)
BLOC	0.0000843	0.9999157
CEG	6.732×10^{-13}	0.999999999999327
CMM	0.0001144	0.9998856
COM	6.683×10^{-13}	0.999999999999332
CF	1.018×10^{-11}	0.9999999999898
CC	0.0001746	0.9998254
DD	2.312×10^{-10}	0.9999999997688
FDN	6.450×10^{-11}	0.9999999999355
FP	0.0000602	0.9999398
RSCR	0.0000722	0.9999278
RT	3.280×10^{-10}	0.9999999996720
TC	5.805×10^{-10}	0.9999999994195

It should be noted that the probabilities of failure obtained from the Group-I are much larger than those obtained from Group-II. This is because:

1. The Extended Finite State Machine (EFSM) can model the actual structure of the APP system. For instance, during normal operation, μp1 and μp2 work redundantly for safety concerns. If either of the microprocessors calculates a trip condition, the APP system will send out a trip signal. However, the actual structure of the system may be very difficult to

take into account in Musa's exponential model because it is difficult to separate the number of defects per processor and know what type of failure will occur.

2. ESFM models simulate the actual fault exposure information of the system while the fault exposure ratio is estimated as 4.2×10^{-7} in Musa's model. This number is outdated and incorrect by orders of magnitude for safety critical systems.
3. The average number of defects found by Group-I is 17 and the average number of defects found by Group-II is only three.

A more-detailed discussion of the reliability estimation results is provided in the following subsections.

19.2.2.1 Reliability Estimation from Group-I Measures

The results from Group-I are shown in Table 19.14. The reliability-estimation results are still very low compared with the measures in Group-II. This is mainly because the high-level structure of the APP system and the defect type cannot be taken into account in the reliability-estimation process for the following measurements: BLOC, CMM, CC, and RSCR.

Table 19.14 Failure Probability Results for Measures in the First Group

Measure	Number of Defects	Probability of Failure (per demand)	Reliability (per demand)
BLOC	14	0.0000843	0.9999157
CMM	19	0.0001144	0.9998856
CC	29	0.0001746	0.9998254
FP	10	0.0000602	0.9999398
RSCR	12	0.0000722	0.9999278

19.2.2.2 Reliability Estimation from Group II Measures

Because the exact location and nature of the defects found by the second group of measures could be determined, the EFSM model, the ROBDD program, or the Markov Chain Model for the four distinct operational modes could be built based on this information.

The mechanisms of failure should also be incorporated into the models. As stated in Chapter 4, there are four operational modes in the APP system: power-on, normal, calibration, and tuning.

During the normal operational mode, defects can trigger two basic types of failures. In the case of the APP system, the failures are defined as follows:

Type I: The APP system should send out a TRIP signal but it did not;
Type II: The APP system should not send out a TRIP signal but did.

The failure mechanism in power-on, calibration, and tuning modes are simple: any failure in these modes is considered a failure of the APP system.

From the safety point-of-view, only a Type-I failure is critical and this concern was applied to the design of the system by using redundant microprocessors. Type-II failures should also be considered because there could be significant expenditures as a consequence of an unnecessary TRIP. It should be noted that only these two types of failure are considered. The third failure type, which is identified in Chapter 10 (Coverage Factor), is neglected in the discussion because it relates only to auxiliary failures.

The failure probability estimates obtained using measures in Group-II are shown in Table 19.15 and illustrated in Figure 19.3. The Coverage Factor was excluded from this table and figure because it only focused on the normal operational mode.

Table 19.15 Failure Probability Results in Each Mode for Measures in the Second Group

Measures	Probability of Failure (per demand)							
	Power-on		Normal		Calibration		Tuning	
	Type I	Type II	Type I	Type II	Type I	Type II	Type I	Type II
CEG	0	0	0	0	2.81e-11	2.81e-11	2.81e-11	2.81e-11
COM	0	0	0	0	2.78e-11	2.78e-11	2.78e-11	2.78e-11
DD	8.45e-11	8.45e-11	1.17e-10	1.17e-10	0	0	0	0
RT	1.03e-10	1.03e-10	1.64e-10	1.64e-10	3.36e-13	3.36e-13	0	0

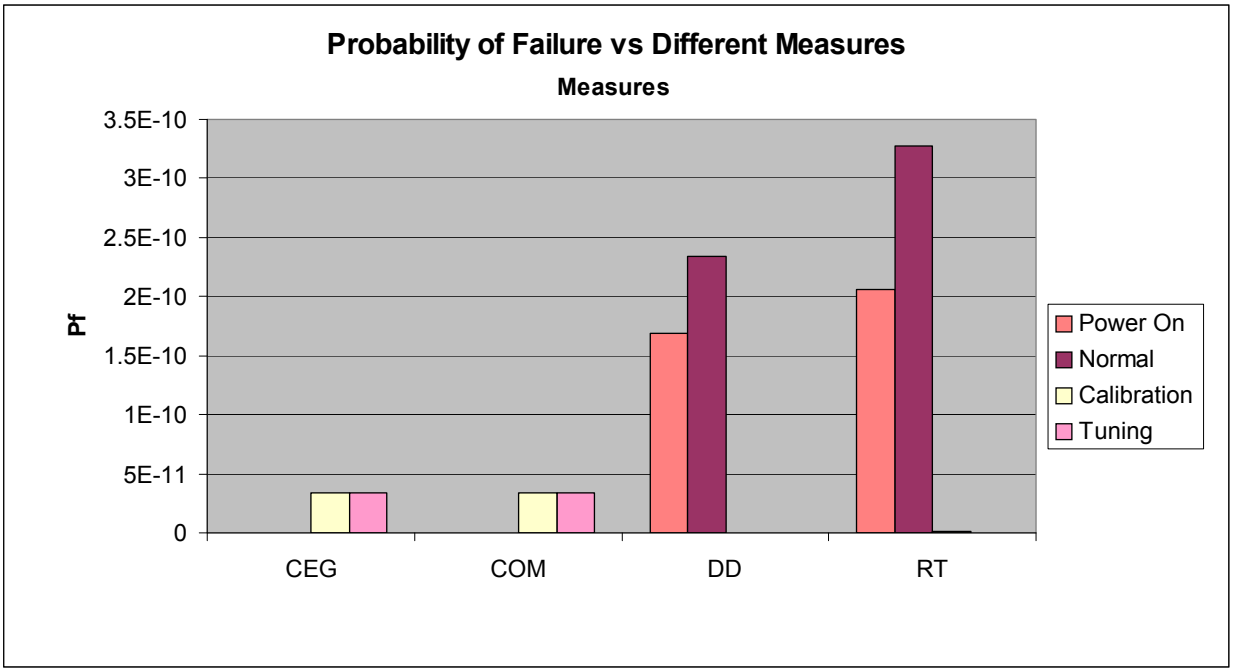


Figure 19.3 Failure Probability Estimates for Measures in the Second Group

The failure probability of the APP system can be calculated using the failure probability results shown in Table 19.16 and the operational-mode profile presented in chapter 4. The results are shown below:

Table 19.16 Failure Probability Results for Measures in the Second Group

Measure	Number of Remaining Defects	Probability of Failure (per demand)	Reliability (per demand)
CEG	1	6.732×10^{-13}	1
CF	6	1.018×10^{-11}	0.9999999999898
COM	1	6.683×10^{-13}	1
DD	4	2.312×10^{-10}	0.99999999977
RT	5	3.280×10^{-10}	0.99999999967

As shown in Figure 19.3, the probabilities of failure obtained from CEG and COM are very close. This is because their failures are caused by the same defects in both calibration and tuning

modes. No defects were found in the power-on and normal operation modes. It should be noted that only the defects remaining in the code were considered and used to obtain the probability of failure of the APP system.

Since both measures are specifically designed to discover requirement faults, their focus is not defect identification in code, and their performance for reliability evaluation is low. CEG is used to check logical relationships between inputs and outputs while completeness, COM, is used to check the completeness of the specifications. Table 19.17 shows the original defects found in the requirements specifications.

Table 19.17 Original Defects Found in the APP Requirement Specification

Modes	CEG[46]	COM
	Number of Defects	Number of Defects
Power-On	3	14
Normal	3	9
Calibration	1	6
Tuning	1	6

It is noticed that both CEG and COM are powerful tools to find imperfections in the SRS. However, most of the defects found in the SRS have been fixed later, such as in the design-development phase and the coding phase. Therefore, the reliability estimation based on the original results is inappropriate. To get better reliability estimation, checking if the defects are still in the code is necessary. Further research can focus on applying CEG and COM measures directly to the SDD and the source code.

Defect density is the most powerful measure to discover the defects remaining in the code by checking all the SRS, SDD documents, and source code. As already shown in Table 19.6, four out of 11 defects were found by the defect-density measure. Application of this measure, however, requires more software engineering experience than that which is required to implement measures like CEG and COMs. To obtain a better result, it is recommended that the checking speed of the documents should not exceed two pages per hour.

For the requirements traceability measure, the defects were found in power-on, normal operation, and calibration mode. No defect was found in the tuning mode.

[46] CEG defects documented in Chapter 7 were not partitioned per mode. Thus the total number of defects in Chapter 7 and in Table 19.17 may not necessarily be equal. One of the defects was common to calibration and tuning.

Since all of the above four measures need to examine the SRS, SDD, and source code carefully, the measurements are time-consuming. Constructing a corresponding EFSM model is also time-consuming.

Coverage factor is a special measure in this group. A Markov Chain model is used to estimate the reliability. In this study, because of several technical limitations, the complete fault injection experiments could not be conducted for the three microprocessors in the APP system. Only two safety function microprocessors were studied and the communication microprocessor was not subjected to fault injection. Thus the probability of failure in the power-on, calibration, and tuning mode could be obtained. Only the reliability in the normal operation mode was calculated. This is why the probability of failure from the coverage factor measure is so low. Also the faults were only injected in the RAM, PROM, and registers. It is obvious that the reliability has been overestimated.

19.2.2.3 Reliability Estimation from Group-III Measures

In the case of Test Coverage, the fault-exposure ratio, K, can be updated using the extended finite state machine models and defects found during testing. However, if no defect is found during the testing, then the method is not applicable. The problem can be circumvented by considering the last version with faults.

From the results, it is found that for different subsystem structures, there are distinct fault exposure ratios. The seven defects used for estimating the new K (vK) were only in power-on and normal operation mode. The failure probability in power-on made a very small contribution to the total failure probability of the APP system, so only the fault exposure ratios in normal operation mode are considered. Therefore, from the test coverage measure, the vK was obtained from the extended finite state machine results and shown in Table 19.18.

Table 19.18 Fault Exposure Ratio Results

	Fault Exposure Ratio
Musa's K	4.2×10^{-7}
New ratio $K^*(vK)$	4.5×10^{-12}

19.2.2.4 Applying the vK to the Twelve Measures

As shown in Table 19.18, the actual fault exposure ratio for the APP is much less than 4.2×10^{-7}. It is proved that Musa's K is no longer suitable for safety critical systems. By applying the new fault-exposure ratio, the reliability for the APP system from the Test Coverage and Fault-days Number measure are:

$$R_{TC} = e^{-\left(N \cdot vK \cdot \tau / T_L\right)} = e^{-\left(9 \times 4.5 \times 10^{-12} \times 0.129 / 0.009\right)} = 0.9999999994195/demand$$

$$R_{FDN} = e^{-\left(N \cdot vK \cdot \tau / T_L\right)} = e^{-\left(1 \times 4.5 \times 10^{-12} \times 0.129 / 0.009\right)} = 0.9999999999355/demand$$

By applying this new fault-exposure ratio to Musa's model, the results from the Group-I measures are very close to those calculated using the measures in Group-II. Table 19.19 shows the results if this vK (fault-exposure ratio) obtained from the Test Coverage measure is applied to the measures in Group-I.

Table 19.19 Updated Results if vK is Applied to Group-II Measures

Measure	Number of Defects	Probability of Failure with Old K (per demand)	Probability of Failure with vK (per demand)
BLOC	14	0.0000843	9.03×10^{-10}
CMM	19	0.0001144	1.23×10^{-09}
CC	29	0.0001746	1.87×10^{-09}
FP	10	0.0000602	6.45×10^{-10}
RSCR	12	0.0000722	7.74×10^{-10}

In conclusion, there are three approaches to update the results for the Group-I measures:

1. Considering the high-level system structure;
2. Using the new fault-exposure ratio (vK) that can be obtained using the Test Coverage ESFM model.
3. Obtaining the exact K for each subsystem in each mode based on the number of defects found using fault-location models [Nejad, 2002].

It is obvious that the third approach is the strongest, although it is time-consuming and not always applicable. If the structure of the subsystem is unknown or the system cannot be divided into separate modes, then the third approach cannot be applied. Therefore, for most systems, it is recommended to use the second approach.

19.2.2.5 Validate the Ranking by Reliability Comparison

Having obtained reliability predictions based on each of the twelve measures, the estimations obtained were compared and contrasted to each other and to the rankings established in NUREG/GR-0019.

First, the inaccuracy ratio (ρ) is defined to quantify the quality of the software prediction:

$$\rho_{(RePS)} = \left| \log \frac{P^*_{f(RePS)}}{P_f} \right|$$

where

$\rho_{(RePS)}$ is the inaccuracy ratio for a particular RePS;
P_f is the probability of failure-per-demand from reliability testing or operational data;
$P^*_{f(RePS)}$ 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. Table 19.20 provides the inaccuracy ratio for each of the 12 measures. The rankings based on the calculated inaccuracy ratio and the experts' rankings obtained in NUREG/GR-0019 are also provided in Table 19.20. The rates of these 12 measures during the testing phase are shown as the last column of Table 19.20.

Table 19.20 Inaccuracy Ratio Results and Rankings for Each RePS

Measure	Probability of Failure /demand	ρ_{RePS}	Rankings based on ρ_{RePS}	Experts' Rankings	Rate
BLOC	0.0000843	5.3764	10	11	0.4
CEG	6.732×10^{-13}	2.7243	7	10	0.44
CMM	0.0001144	5.5091	11	7	0.6
COM	6.683×10^{-13}	2.7211	6	12	0.36
CF	1.018×10^{-11}	1.5416	5	2	0.81
CC	0.0001746	5.6927	12	3	0.72
DD	2.312×10^{-10}	0.1853	2	1	0.83

Table 19.20 Inaccuracy Ratio Results and Rankings for each RePS (continued)

Measure	Probability of Failure /demand	ρ_{RePS}	Rankings based on ρ_{RePS}	Experts' Rankings	Rate
FDN	6.450×10^{-11}	0.7397	4	4	0.72
FP	0.0000602	5.2303	8	9	0.5
RSCR	0.0000722	5.3095	9	5	0.69
RT	3.280×10^{-10}	0.0334	1	8	0.55
TC	5.805×10^{-10}	0.2146	3	6	0.68

Several conclusions can be drawn from these results as follows:

1. From the table, it is clear that RePSs that use structural information and actual defects (Group-II RePSs) are clearly superior to RePSs that do not use structural information or actual defects (Group-I RePSs). The rankings based on the inaccuracy ratio appear not to be consistent with the expert-opinion rankings established in NUREG/GR-0019. UMD concludes that this is due to the fact Group-I RePSs use an exponential reliability-prediction model with a fault-exposure ratio parameter set to 4.2×10^{-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. Further development effort could focus on creating better prediction models from these measures or as suggested in Section 19.2.2.4 on experimentally obtaining a more accurate fault exposure ratio for the application instead of using a universal parameter such as the value 4.2×10^{-7}.

2. The rest of the section validates the rankings within Group-II RePSs.

In Group-II CF could not be used in the validation of the rankings because the fault-injection experiments were not complete (see Table 19.21). Thus, UMD only compared the other four measures in this group to the experts' ranking. DD remains a highly rated measure while CEG and COM are still rated low. RT is ranked higher than it should (i.e., it is found here that RT is better than DD). UMD carefully studied the reasons for this inversion as shown below:

1) A formal approach for measuring RT can be easily established. Indeed, in the case of RT one needs only to verify whether an item is present in the requirements documents and the code. Figure 19.4 illustrates how a simple measurement matrix can be built to systematically trace the requirements.

Table 19.21 Validation Results for Group II RePSs

Measure	Rankings based on ρ_{RePS}	Experts' Rankings
CEG	4	3
COM	3	4
CF	N/A	N/A
DD	2	1
RT	1	2

2) In the case of DD, checklists are available to guide the inspection process. However, the process remains difficult to execute for the following reasons:

- For a single segment of requirement or design specification, or source code module, a large number of items need to be verified (see the Table 19.22 extract from Ebenau [Ebenau, 1993]).

Table 19.22 DD Measure Checklist Information

Inspection of	Number of Items That Need to be Checked in the Checklist
Software Requirements	12
Detailed Design	16
Code	46

- Some of the items are high level and cannot be verified systematically nor answered objectively. For instance, the checklist does not provide a clear definition of "complete," "correct," and "unambiguous" for an item such as: "Are the requirements complete, correct, and unambiguous?"
- The larger the application, the more difficult a complete measurement of defect density becomes.

Sl. No	Requirement Identifier	Requirement Description	Is it traceable to the code and back to requirement?	Code Identifier	Code Details	Remarks
Specific Requirements – Initialization						
	3.1.1.3	The external interrupts of the Z180 micro-processor are defeated so that no external communications will occur to interrupt the safety function processor.	Yes	16.26	`outport(ITC, 0x00); /** defeat external interrupts **/`	

	3.1.1.3	The discrete outputs used by the safety function processor are all initialized to "off."	Yes	16.29–17.4	`outport(0xc100,0x00); /** Trip relay 1 to tripped state **/`
					`outport(0xc300,0x00); /** Trip relay 2 to tripped state **/`
					`outport(0xc500,0x00); /** Status relay 1 to tripped state **/`
					`outport(0xc600,0x00); /** Output relay 1 to tripped state **/`
					`outport(0xc800,0x00); /** Output relay 2 to tripped state **/`
					`outport(0xca00,0x00); /** Output relay 3 to tripped state **/`
					`outport(0xd100,0x00); /** Trip relay 3 to tripped state **/`
					`outport(0xd300,0x00); /** Trip relay 4 to tripped state **/`
					`outport(0xd500,0x00); /** Status relay 2 to tripped state **/`
					`outport(0xd600,0x00); /** Output relay 4 to tripped state **/`
					`outport(0xd800,0x00); /** Output relay 5 to tripped state **/`
					`outport(0xda00,0x00); /** Output relay 6 to tripped state **/`	

Figure 19.4 Requirements Traceability Measurement Matrix

3. As for Group-III RePSs, one cannot conclude FDN was ranked higher than TC in the case of the APP system but the actual experts' ratings are close.

4. As shown in column 4 and 5 of Table 19.20, the APP results only partially confirm the experts' rankings obtained in NUREG/GR-0019. This may be due to the following reasons:

    1) It has been 10 years since the experts ranked the measures. During the past 10 years, new tools, techniques, and methodologies have been created or proposed. Additional experiments have been run for safety-critical and non-safety-critical systems. Our research has capitalized on these new developments while the experts did not have access to this extra knowledge. The experts' ranking on the measures may thus not be in par with the current state of the art and probably need to be updated.

    2) The experts ranked the measures and not the RePSs. It may be that our modeling effort has, in some instances, involuntarily created stronger RePSs than in other instances. In some cases (e.g., Test Coverage), we have increased the reliability-prediction potential by adopting strong support measures. For example, the precise definition of Test Coverage is "the percentage of the source code covered during testing." In this study, we have taken advantage not only of the Test Coverage value but of the number and location of defects found during testing.

In conclusion, the experts' rankings could and should be updated by using the Bayesian theory so as to reflect the strength of the measure as well as the strength of the RePS. The original experts' rankings can serve as prior information and the APP results are evidence that can be used to update this prior information. Further validation of this point could be obtained by collecting more data points as evidence to further update the experts' rankings.

The remainder of this section compares the results obtained in this study with results obtained in the preliminary validation report (NUREG/CR-6848). The application considered in NUREG/CR-6848 was PACS, the control software activating a secure gate.

In NUREG/CR-6848, five measures (DD, TC, RT, FP, and BLOC) were ranked with respect to their prediction error (p_e) defined as:

$$p_e = \frac{|p_s(real) - p_s(est)|}{1 - p_s(real)}$$

where

$p_s(real)$ the probability of success-per-demand obtained from reliability testing
$p_s(est)$ the probability of success-per-demand obtained from the RePS

To be consistent with the method followed in this report, the five measures are re-ranked using the inaccuracy ratio proposed in this section. The vales of ρ and ranking results are presented in

Table 19.23. The validation rankings (and ρ) for these five measures on APP system are also listed in Table 19.23.

As one can conclude from Table 19.23, RT ranks better than DD for APP while DD and RT ranked identically in NUREG/CR-6848. The reasons have been examined earlier in this section.

Table 19.23 Comparison of the Rankings with Results in NUREG/CR-6848

Measures	Rankings in this study (APP)	ρ (APP)	Rankings in NUREG/CR-6848 (PACS 1)	ρ (PACS 1)
DD	2	0.1853	1	0.0345855
TC	3	0.2146	3	0.0395085
RT	1	0.0334	1	0.0345855
FP	4	5.2303	4	1.631691
BLOC	5	5.3764	5	3.4771213

19.3 Discussion about the Measurement Process

An estimate of the time for training, performing the different measurements, and calculating the values of predictions given by each corresponding RePSs is given in Table 19.24. Training here is defined as becoming familiar with the required tools prior to performing measurements. Some measurements are very time consuming. Table 19.24 shows the total time spent for the 12 RePSs.

Table 19.24 Total Time Spent for the Twelve RePSs

Measure	Total Time Spent	Duration
Bugs Per Line of Code	160 hrs (20 days)	Short
Cause-effect Graphing	350 hrs (44 days)	Medium
Capability Maturity Model	120 hrs (15 days)	Short
Completeness	512 hrs (64 days)	Medium
Coverage Factor	752 hrs (94 days)	Long
Cyclomatic Complexity	360 hrs (45 days)	Medium
Defect Density	704 hrs (88 days)	Long
Fault Days Number	240 hrs (30 days)	Short
Function Point	128 hrs (16 days)	Short

Table 19.24 Total Time Spent for the 12 RePSs (continued)

Measure	Total Time Spent	Duration
RSCR	360 hrs (45 days)	Medium
Requirements Traceability	640 hrs (80 days)	Long
Test Coverage	904 hrs (113 days)	Long

The duration is defined as follows:

1. **Short:** 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
3. **Long:** 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 because there is no need to inspect the SRS, SDD, and code. Measurements and calculations related to CEG, Completeness, CC, and RSCR require careful inspections of the SRS or the code and therefore require more time. Measurements related to DD and RT require inspections of all the related documents. As a result, the RePSs measurement process for these two measures is slow. The time required for the measurement and calculations related to coverage factor and Test Coverage were excessive. This is because a great deal of time was spent on modifying the original APP source code so that it could be compiled successfully by current compilers. In addition, for the measurement of Test Coverage, a great deal of time was spent modifying the original test cases for the current simulation environments. If there were no such compatibility problems, the measurements would have been completed much faster.

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

The effort includes the time spent for tool acquisition, comparison between possible tools, training to become familiar with the identified tools, and an initial upfront cost that would remain identical whether small or large applications are considered and would disappear for routine applications of the methodology. The effort also specifically includes measurement costs that may already be part of a routine development process. Measurements and RePSs construction were performed by graduate students that were implementing and refining the methodology as they applied it. It is expected that a routine application of the methodology would be less time-consuming. Finally, the APP was developed more than 10 years ago and the development process did not benefit from current tools and methods (e.g., the effort devoted to RT measurement could have been improved with current traceability tools).

Some measurements also are quite costly. Table 19.25 shows the required tools and corresponding cost for performing measurements for these twelve measures.

Table 19.25 Cost of the Supporting Tools

RePSs	Required Tools	Cost
Bugs Per Line of Code	RSM Software	Free
Cause-effect Graphing	UMD Software 1 (CEGPT)	$750
Capability Maturity Model	CMM Formal Assessment	$50,000
Completeness	TestMaster	$50,000
Coverage Factor	Keil μVision 2	$320
	IAR EWZ80	$900
Cyclomatic Complexity	RSM Software	$Free
Defect Density	TestMaster	$50,000
Fault Days Number	UMD Software 2 (FDNPT)	$750
Function Point	FP Inspection	$7,000
RSCR	No	$0
Requirements Traceability	TestMaster	$50,000
Test Coverage	TestMaster	$50,000
	Keil μVision 2	$320
	IAR EWZ80	$900

For three of these 12 RePSs, corresponding measurements have to be performed by experts. Table 19.26 presents related information.

Table 19.26 Experts Required

Measure	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

As shown in Table 19.26, some tasks must be performed by senior-level software- or system-engineers with 10 years training. This requirement may vary depending on the talent of the engineer, but it is clear that experience in software engineering and nuclear systems will be necessary to find defects in nuclear power plant safety system software source code.

19.4 Difficulties Encountered during the Measurement Process

This section describes the experience with collecting and analyzing data during the measurement process and discusses the issues encountered. Possible solutions are briefly addressed.

Two types of data were collected and analyzed: 1) data used to predict the reliability and 2) data used to estimate the reliability. The remainder of this section is organized as follows: Section 19.4.1 discusses the study and problems encountered with the data collection and analysis for the reliability prediction; Section 19.4.2 discusses the study and problems encountered with the data collection and analysis for the reliability estimation. Possible solutions to the encountered problems are briefly addressed in Section 19.4.3.

19.4.1 Data Collection and Analysis for Reliability Prediction

For the 12 measures, detailed measurement rules should be provided to measure each primitive. Unfortunately, these rules are imprecisely defined. As an example, in the case of the BLOC measure, problems were encountered with the definition of a "module." A "module" is defined as "an independent piece of code with a well-defined interface to the rest of the product" in [Schach, 1993]. IEEE [IEEE, 1990] defines "module" in the following two ways: "A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading," or "A logically separable part of a program." Gaffney, author of BLOC [Gaffney, 1984], however, did not provide a clear definition of "module" but only mentioned it as a "functional group." The existence of multiple definitions of the module concept and the lack of consensus make its measurement problematic. The same endemic problem reoccurs for most of the measures considered.

For measures such as CEG, COM, DD, and RT, which need the direct inspection of the software requirement specifications (SRS), we encountered difficulties collecting and analyzing the data mainly because of issues with the clarity of the documents. Because the APP was developed more than ten years prior to the research, some of the documents did not follow or only partially followed the IEEE standards. More specifically, the first step of the inspection is typically to identify the "functional requirements" defined in the SRS. The IEEE standards mentioned that the keyword "shall" should be used to indicate a functional requirement. However, many segments of specifications that used this keyword were not functional requirements. Sometimes "shall" indicated "descriptive requirements." Also, the SRSs under study failed to be unambiguous. For example, it was difficult to count the number of unique cause/effect pairs for the CEG measure since the authors of the SRS repeated themselves frequently. It should be noted that if the CEG measurement is done manually, the results are highly dependent on the ability of the analyst. This is mainly because:

1) It is not very easy to differentiate the prime effects from the intermediate effects if the analyzer does not have a comprehensive knowledge of the system.

2) It is not easy to identify the true logical relationship between the causes and the constraints since the relationships are usually implied and not specified explicitly using keywords like "and," "or," "either," etc.

It also should be mentioned that for the above four measures, the measurement process was time consuming. A considerable amount of time was spent to manually parse the natural language SRS. There were 289 pages of SRS to be inspected and the total measurement time for the COM measure was 512 hours (64 work days) and 350 hours for the CEG measure. In the case of DD, for a single segment of requirement or design specification, or source code module, a large number of items need to be verified (12 items for SRS, 16 for SDD, and 46 for code). Some of the items are high level and cannot be verified systematically nor answered objectively. For instance, checklists available for DD do not provide a clear definition of "complete," "correct," and "unambiguous" for an item such as: "Are the requirements complete, correct, and unambiguous?"

In the case of the CMM measure, a standard CMM-level assessment had not been performed for the company that developed the software module. Furthermore, the software module was ten years old and most of the development team members were no longer working with the company. The CMM assessment could only be conducted based on the available team member's answers to the Maturity Questionnaire. As a consequence, any results of an assessment are post-mortem and, as such, do not qualify for a formal assessment.

For measures that require the collection of the software-development process data, we were unable to collect the exact required data since those data were not documented or clearly documented in the software development documents (SRS, SDD code, and V&V). For instance, in the FDN measure, the exact effort for each development phase could not be obtained. This is because the development effort for each team member was not recorded during the original development because the original development had not envisioned the measurement of FDN. Even if these data had been recorded, the

exact effort for each phase would have been hard to measure since the development did not precisely follow a waterfall development model. Developers did go back to work on the SRS after the code was written.

Once the indirect indicators are measured, they are linked to reliability prediction models. Some of the RePS models are based on the PIE [Voas, 1992] technique and require the actual operational profile (OP). OP is used to measure software reliability by testing the software in a manner that represents actual use or it is used to quantify the propagation of defects (or unreliability) through EFSM models. However, determining the OP of the system is a difficult part of software reliability assessment in general [Musa, 1992].

We assessed the infrastructure inputs related OP by inspecting the software requirement specification and relied on a related hardware component failure database to quantify portions of the OP. Unfortunately, some of the hardware-failure-rate information was not available in the database, i.e., the address line failure rate. In addition, we discovered that the information contained in the database was typically too generic. For instance, we were looking for the failure rate for an 8 kB RAM. However, only the general failure rate of the RAM was given without mention of the size. Also, obtaining such information from the manufacturer revealed itself as being impossible because of the obvious business implications.

The plant-inputs-related OP was assessed by examining the operational data. The problem encountered here was the need to interpret the operational data available. We were not able to analyze the data correctly without the help of the plant experts. According to their opinions, the following three categories of data should not be considered part of the operational data:

1) Outage data: Data recorded during plant outages cannot be considered an integral part of the normal operational data set. Indeed, data recorded during these time periods is out-of-range and basically meaningless;
2) Missing data: Some data is missing from the data set. This data is typically labeled: “bad input,” “shutdown,” and “under range;”
3) Aberrant data: There were several strange records either with a negative reactor flow value or an extremely large flow value (of the order of 10^{26}, which far exceeds normal values that are typically of the order of 10^{5}).

Once the valid operational data was identified, a statistical extrapolation method was used to estimate the trip condition probability due to the rarity of the events. However, the accuracy of the extrapolation should be further validated and may jeopardize the validity of the profile.

19.4.2 Data Collection and Analysis for the Reliability Estimation

The quality of the safety-critical system under study is measured in terms of its reliability estimate. Reliability was estimated through operational data. This type of operational data was obtained mainly based on the problem records provided by the nuclear power plant that utilized the system under study for ten years.

The main problem encountered was the analysis and interpretation of the problem records. More specifically, since the records documented all the problems experienced with the reactor protection system, it included the actual failures/false alarms of the entire protection system and the actual failures/false alarms of the digital system itself. The first step of the analysis required distinguishing actual failures of the digital system from the others. For example, one problem record described the digital system as working improperly due to a connection problem when attempting to reinsert the digital system into the cabinet. This problem, apparently, was not an actual digital system failure.

The second step of the analysis required distinguishing software module failures from the hardware failures of the digital system. For example, one of the problem records documented that a software module was sent back to its manufacturer since a trip signal was sent out when it should not have been. However, none of the testing or other diagnostic efforts performed by the manufacturer identified a failed component or any other problem.

Another problem encountered was the identification of the actual usage duration of each software module. A typical safety-critical system possesses redundant units. Thus, multiple digital systems and correspondingly, multiple software modules, were installed to monitor one nuclear reactor unit. The actual usage duration for each such module differed. Some of the modules were in active use, others were kept as cold spares while others might have been sent back to their manufacturer for repair or diagnostics. The exact usage duration was difficult to determine since part of the information necessary to determine usage was kept at the plant while other information was kept by the manufacturer under different denominations. Sometimes the information provided by these two organizations was not consistent.

19.4.3 Possible Solutions

As discussed in Section 19.4.1, the measurement process can be extremely time-consuming, error prone, and highly dependent on the qualification of the inspectors involved. Two solutions to these problems are possible: 1) Training and certification of inspectors; 2) Automation of the measurement process.

For TC and RT, training would focus on how to trace requirements forward to the source code and from the source code back to the requirements. For DD, trainees should understand how to inspect different software system artifacts. For CEG and COM, trainees should know how to distinguish the functional requirements from the descriptive requirements. For any of these measures, trainees

should already have some experience in developing software systems. They also should have had at least an introductory course on software engineering.

For the measures under study, much of the measurement is manually accomplished, so training is probably the largest factor for ensuring repeatability. The measures should be further formalized and industry-wide standard definitions also might improve the current situation, especially if the measurement rules that support the definition can be embedded in tools. As such, developing tools for automatic extraction of semantic content from the different artifacts created during the development process is one of the possible solutions.

Audit of the data collection process also should be made part of an organization's processes. There should be an independent evaluation of the quality of the data collected, to ascertain compliance to standards, guidelines, specifications and procedures.

Since data collected by different companies may not always be consistent as discussed in Section 19.4.2, when multiple companies enter an interaction, sharing of information standards and tools or data repositories between the companies should be defined.

19.5 Recommended Measures and RePSs

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

- David N. Card, Fellow, Software Productivity Consortium
- J. Dennis Lawrence, Partner, Computer Dependability Associates, LLC
- Michael R. Lyu, Professor, Chinese University of Hong Kong, and
- Allen P. Nikora, Principal Member, Jet Propulsion Laboratory

As an integral part of their review of this document and based on the results of this research, the experts recommended a subset of the measures and corresponding RePS for use. The experts elected to recommend a measure if the prediction error, ρ, of its related RePS was less than 1 (see Table 19.27).

Table 19.27 Recommended Measures

Measure	Probability of Failure/demand	ρ	Recommended? (Yes/No)
BLOC	0.0000843	5.3765	No
CEG	6.732×10^{-13}	2.7243	No

Table 19.27 Recommended Measures (continued)

Measure	Probability of Failure/demand	ρ	Recommended? (Yes/No)
CMM	0.0001144	5.5091	No
COM	6.683×10^{-13}	2.7211	No
CF	1.018×10^{-11}	1.5416	No
CC	0.0001746	5.6927	No
DD	2.312×10^{-10}	0.1853	Yes
FDN	6.450×10^{-11}	0.7397	Yes
FP	0.0000602	5.2303	No
RSCR	0.0000722	5.3095	No
RT	3.280×10^{-10}	0.0334	Yes
TC	5.805×10^{-10}	0.2146	Yes

19.5.1 Recommended Use of this Methodology in Regulatory Reviews

This section discusses the recommended use of the RePS theory for nuclear regulatory review.

IEEE Std 7-4.3.2 clause 5.3.1.1 [IEEE, 2003] specifies:

> The use of software quality metrics shall be considered throughout the software life cycle to assess whether software quality requirements are being met. When software quality metrics are used, the following life cycle phase characteristics should be considered:
>
> — Correctness/Completeness (Requirements phase)
> — Compliance with requirements (Design phase)
> — Compliance with design (Coding phase)
> — Functional compliance with requirements (Test and Integration phase)
> — On-site functional compliance with requirements (Installation and Checkout phase)
> — Performance history (Operation and Maintenance phase)

Table 19.28 describes how each measure supports these six characteristics and therefore supports the regulatory review process. Symbol "√" in the table indicates that a measure supports a specific

characteristic. "N/A" is used when a measure is not applicable to a specific lifecycle phase. Symbol "◊" indicates that a measure does not directly support a specific characteristic but could assist the review process, i.e., serve as a general indicator. Group-I measures fall into the "◊" category. These measures cannot tell us the exact nature of problems encountered. For example, a high value of CC cannot tell us whether the application contains a large number of functional compliance issues. However, if one compares multiple modules whose values of CC have been assessed, a high CC for one module while another is small may indicate that the latter module would be less likely to contain compliance with design issues. In essence, these measures should only be used as general "indicators" of fault proneness. But in order to use these indicators, one will need to define acceptable and unacceptable ranges of values for these indicators. For this, the reader is referred to some of the efforts made in the software engineering literature.

Table 19.28 Measures and Life-Cycle Phase Characteristics

Measures	Correctness Completeness (Requirement phase)	Compliance with requirements (Design phase)	Compliance with design (Coding phase)	Functional compliance with requirements (Test and Integration phase)	On-site functional compliance with requirements (Installation and Checkout phase)	Performance history (Operation and Maintenance phase)
BLOC	**N/A**	**N/A**	◊	◊	**N/A**	◊
CEG	√	√	√	√	**N/A**	√
CMM	◊	◊	◊	◊	**N/A**	◊
COM	√	√	√	√	**N/A**	√
CF	**N/A**	**N/A**	**N/A**	√	**N/A**	√
CC	**N/A**	◊	◊	◊	**N/A**	◊
DD	**N/A**	**N/A**	**N/A**	√	**N/A**	√
FDN	√	√	√	√	**N/A**	√
FP	◊	◊	◊	◊	**N/A**	◊
RSCR	◊	◊	◊	◊	**N/A**	◊
RT	**N/A**	√	√	√	**N/A**	√
TC	**N/A**	**N/A**	**N/A**	√	**N/A**	√

19.6 Follow-On Issues

This section discusses the follow-on issues raised as a consequence of performing this study. The issues are first listed and briefly discussed. A prioritization of the issues based on recommendations of three field experts is provided at the end of this section.

19.6.1 Defect Density Robustness

Defect density is the root measure of one of the highest ranked RePSs. As such, it is the primary element of one of the most important 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 an 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. On the other hand, if the inspector detects defects that have a high likelihood of occurrence, then reliability estimation may be precise even if the defect-detection efficiency is low. For a safety-critical system, one would in addition want the inspector to detect defects that are safety-relevant.

19.6.2 Test Coverage Repair

The Test Coverage (TC) RePS relies on the assumption that the number of defects found during the testing is not zero. This assumption may not hold for safety critical software, and this was the case for the APP system. Multiple versions of the APP test plans and source code exist. The testing of the final version did not reveal any failures. However, the version before the final version discovered defects. The approach currently followed by UMD uses this earlier version of the source code and test plan to conduct the TC measurement and RePS calculation. This approach introduced errors as it is either:

a. Too conservative if the defects found are actually fixed; this is the most likely case; or
b. Incorrect if new defects are introduced during repair and not detected by the new test cases.

The approach can be improved by considering the defect introduction and removal mechanisms in the testing stage. More specifically, one could calculate a repair rate for the APP using the available

life-cycle data. One could also calculate a probability for introducing new defects due to repair using this same life-cycle information. This would reduce the errors discussed [Shi, 2010] [Smidts, 2011].

19.6.3 Issues with the Fault Exposure Ratio

The fault exposure ratio *K* is used in the RePSs for several measurements (CC, RSCR, CMM, BLOC, FP, FDN, and TC). This parameter is currently extracted from the literature. Experience from this study has shown that:

1) *K* is a critical parameter for reliability estimation;
2) The values of *K* proposed in the literature are outdated and incorrect by orders of magnitude, in particular for safety critical applications.

Thus, a follow on issue is to examine how to obtain an accurate value of *K* for each system under study.

19.6.4 CC, RSCR, and FDN Models

Chapters 7, 13, and 15 introduced new RePSs for CC, FDN, and RSCR, respectively. These RePSs have not been validated on other applications. A follow on issue is to validate these models on additional applications (especially FDN since this is a highly ranked measure).

19.6.5 Cases Where No Defects Are Found

As can be seen in Table 19.17, high-ranked measures do not always detect defects in all modes of operation. The smaller the partitioning of the application under study, the more likely it becomes that defects are not found. This may require conducting multiple measurements in parallel or else involve a group of inspectors.

19.6.6 Issues with Repeatability and Scalability

As has been shown in Table 19.24, 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. 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. A follow on issue is to examine each of these avenues and how the solutions should be implemented.

19.6.7 Issues with Common-Cause Failures

At this point, none of the measures considered include a measurement of common cause failure potential. This may lead to an underestimation of the probability of failure at the software system level since it is currently assumed there is independence between the versions. This underestimation may be several orders of magnitude low. For measures such as Cyclomatic Complexity, Function Point, Bugs Per Line of Code, and Requirements Specification Change Request, a CCF correction factor will need to be investigated. This factor would represent the fraction of CCF which will be observed. For measures such as Defect Density and Requirements Traceability, the EFSM propagation technique will need to be modified to account for similar defects in multiple versions.

19.6.8 Issues with Uncertainty and Sensitivity

Software reliability prediction is subject to uncertainty. The sources of uncertainties in software reliability prediction can be divided into two general categories: measurement uncertainty and model uncertainty. The 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.

An initial qualitative sensitivity analysis that accounts for measurement and parameter uncertainty was conducted. The results are shown in Table 19.29 (note: effect of parameter uncertainty is limited to K and T_L; CF sensitivity equations are grouped together with Group II measures as a first approximation of CF behavior). For each measure, quantities that drive the uncertainty are identified. A follow on issue is to perform a quantitative sensitivity analysis for inclusion of model uncertainty.

Table 19.29 Initial Sensitivity Analysis Results

Group	Measure	Sensitivity Equations
I	BLOC	$$dR_{BLOC} = \left(\frac{-K \cdot t}{T_L}\right) \cdot R_{BLOC} \cdot N_{BLOC} \cdot \left\{\frac{dK}{K} - \frac{dT_L}{T_L} + \frac{\sum_{i=1}^{M} 0.002067 \cdot S_i^{1/3}}{\left[\sum_{i=1}^{M} g(S_i) - N_{found}\right]} dS_i + \frac{\left(4.2 + 0.00155 \cdot S_{M+1}^{4/3}\right)}{\left[\sum_{i=1}^{M} g(S_i) - N_{found}\right]} dM + \frac{1}{SL(FP)} \cdot \frac{\partial SL(FP)}{\partial FP} dFP\right\}$$ Where: $$R_{BLOC}(t) = \exp\left(\frac{-K \cdot t}{T_L} \cdot N_{BLOC}\right)$$ $$N_{BLOC} = \left(F - N_{found}\right) \cdot SL(FP)$$ $$F = \sum_{i=1}^{M} g(S_i)$$ $$g(S_i) = 4.2 + 0.00155 \cdot S_i^{4/3}$$
I	CMM	$$dR_{CMM} = \left(\frac{-K \cdot t}{T_L}\right) \cdot R_{CMM} \cdot N_{CMM} \cdot \left\{\frac{dK}{K} - \frac{dT_L}{T_L} + \left[\frac{f_{CMM}(CMM+1)}{f_{CMM}(CMM)} - 1\right] dCMM + \left[\frac{1}{FP} + \frac{1}{SL(FP)} \cdot \frac{\partial SL(FP)}{\partial FP}\right] dFP\right\}$$ Where $f_{CMM}(CMM)$ is given by Table 8.3 and: $$R_{CMM}(t) = \exp\left(\frac{-K \cdot t}{T_L} \cdot N_{CMM}\right)$$ $$N_{CMM} = FP \cdot f(CMM) \cdot SL(FP)$$

Table 19.29 Initial Sensitivity Analysis Results (Continued)

Group	Measure	Sensitivity Equations
I	CC	$$dR_{CC}(t) = \frac{-K \cdot t}{T_L} \cdot R_{CC} \cdot N_{CC} \cdot \left(\frac{dK}{K} - \frac{dT_L}{T_L} + \frac{dSIZE}{SIZE}\right) - \frac{K \cdot t}{T_L} \cdot R_{CC} \cdot \alpha \cdot SIZE \cdot \gamma \cdot \beta^{\gamma-1}$$ $$\cdot \left\{\left(\frac{1}{M}\sum_{i=1}^{g}\sum_{j=1}^{M}\left[H(b_i - CC_j) - H(a_i - CC_j)\right]\right) df_i$$ $$+ \left(-\frac{1}{M}\gamma + \frac{1}{M}\sum_{i=1}^{g} f_i \cdot [H(b_i - CC_{M+1}) - H(a_i - CC_{M+1})]\right) dM$$ $$+ \left(\frac{1}{M}\sum_{j=1}^{M}\sum_{i=1}^{g} f_i \cdot \left[\delta(a_i - CC_j) - \delta(b_i - CC_j)\right]\right) dCC_j\Bigg\}$$ Where: $$\gamma = \frac{1}{M}\sum_{i=1}^{g} f_i \sum_{j=1}^{M}\left[H(b_i - CC_j) - H(a_i - CC_j)\right]$$ $$N_{CC} = \alpha \cdot SIZE \cdot \beta^{\gamma}$$ $$R_{CC}(t) = \exp\left(\frac{-K \cdot t}{T_L} \cdot N_{CC}\right)$$

Table 19.29 Initial Sensitivity Analysis Results (Continued)

<table>
<tr><th>Group</th><th>Measure</th><th>Sensitivity Equations</th></tr>
<tr><td>I</td><td>FP</td><td>
$$dR_{FP}(t) = \left(\frac{-K \cdot t}{T_L}\right) \cdot R_{FP} \cdot N_{FP} \cdot \left\{\frac{dK}{K} - \frac{dT_L}{T_L} + \left[\frac{a_{ACAT}}{FP \cdot f_{FP}(FP, ACAT)} + \frac{1}{SL(FP)} \cdot \frac{\partial SL(FP)}{\partial FP}\right] dFP\right\}$$
Where:
$$R_{FP}(t) = \exp\left(\frac{-K \cdot t}{T_L} \cdot N_{FP}\right)$$
$$N_{FP} = f_{FP}(FP, ACAT) \cdot SL(FP)$$
a_{ACAT} is the slope of f_{FP} and b_{ACAT} is the intercept of f_{FP}. The values of a_{ACAT} and b_{ACAT} can be determined using the following table:
<table>
<tr><th>ln(FP)</th><th>End User</th><th>MIS</th><th>Outsource</th><th>Commercial</th><th>Systems</th><th>Military</th><th>Average</th></tr>
<tr><td>0</td><td>0.05</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0.01</td></tr>
<tr><td>2.302585093</td><td>0.25</td><td>0.1</td><td>0.02</td><td>0.05</td><td>0.02</td><td>0.03</td><td>0.07</td></tr>
<tr><td>4.605170186</td><td>1.05</td><td>0.4</td><td>0.18</td><td>0.2</td><td>0.1</td><td>0.22</td><td>0.39</td></tr>
<tr><td>6.907755279</td><td>N/A</td><td>0.85</td><td>0.59</td><td>0.4</td><td>0.36</td><td>0.47</td><td>0.56</td></tr>
<tr><td>9.210340372</td><td>N/A</td><td>1.5</td><td>0.83</td><td>0.6</td><td>0.49</td><td>0.68</td><td>0.84</td></tr>
<tr><td>11.51292546</td><td>N/A</td><td>2.54</td><td>1.3</td><td>0.9</td><td>0.8</td><td>0.94</td><td>1.33</td></tr>
<tr><td>Slope = a_{ACAT}</td><td>0.217147</td><td>0.215286</td><td>0.1158946</td><td>0.07879343</td><td>0.070356</td><td>0.085618</td><td>0.112668</td></tr>
<tr><td>Intercept = b_{ACAT}</td><td>-0.05</td><td>-0.34095</td><td>-0.180476</td><td>-0.0952381</td><td>-0.11</td><td>-0.10286</td><td>-0.11524</td></tr>
</table>
ACAT is an index that specifies the category of the application. The values of ACAT are specified in the first row of the table (i.e., ACAT takes the values End User, MIS, etc.).
</td></tr>
</table>

Table 19.29 Initial Sensitivity Analysis Results (Continued)

<table>
<tr><th>Group</th><th>Measure</th><th>Sensitivity Equations</th></tr>
<tr><td>I</td><td>RSCR</td><td>
$$dR_{RSCR} = N\left(\frac{-Kt}{T_L}\right)R_{RSCR} \cdot \left[\frac{2a\ln(20)\,(W-1)}{W^2 SIZE_{delivered}}dSIZE_{changed} + \left(\frac{1}{SIZE_{delivered}} - \frac{2\ln(20)\,a(W-1)SIZE_{changed}}{W^2 SIZE_{delivered}^2}\right)dSIZE_{delivered} + \frac{dK}{K} - \frac{dT_L}{T_L}\right]$$
Where:
$$W = 1 + e^{a\left(\frac{SIZE_{changed}}{SIZE_{delivered}} - 35\right)}$$
$$R_{RSCR} = \exp\left[0.036 \cdot SIZE_{delivered} \cdot 20^{1-\frac{2}{W}} \cdot \left(\frac{-K \cdot t}{T_L}\right)\right]$$
and a is obtained from the curve-fitting process. The value of a and the residuals from the fitted model are given below:
<table>
<tr><th>$REVL = \frac{SIZE_{changed}}{SIZE_{delivered}} \times 100$</th><th>$SLI_{RSCR} = \frac{1}{W}$</th><th>$Prediction$</th><th>$(SLI_{RSCR} - Prediction)^2$</th></tr>
<tr><td>0</td><td>1</td><td>0.932589367</td><td>0.004544193</td></tr>
<tr><td>5</td><td>1</td><td>0.904810279</td><td>0.009061083</td></tr>
<tr><td>20</td><td>0.75</td><td>0.755086396</td><td>2.5714×10^{-5}</td></tr>
<tr><td>35</td><td>0.5</td><td>0.5</td><td>0</td></tr>
<tr><td>50</td><td>0.34</td><td>0.244913604</td><td>0.009041423</td></tr>
<tr><td>65</td><td>0.16</td><td>0.095189721</td><td>0.004200372</td></tr>
<tr><td>80</td><td>0</td><td>0.032997158</td><td>0.001088812</td></tr>
<tr><td>100</td><td>0</td><td>0.007547105</td><td>5.69588×10^{-5}</td></tr>
</table>
<table>
<tr><td>$\sum(SLI_{RSCR} - Prediction)^2$</td><td>0.028018714</td></tr>
<tr><td>a</td><td>0.075061777</td></tr>
</table>
</td></tr>
</table>

Table 19.29 Initial Sensitivity Analysis Results (Continued)

Group	Measure	Sensitivity Equations
II	CEG	$$dR_{SW} = \sum_{j=1}^{t/\tau}\sum_{i=1}^{N}\left\{\left[\frac{\partial(P(i))}{\partial(L_i)}\cdot I(i)\cdot E(i) + P(i)\cdot\frac{\partial(I(i))}{\partial(L_i)}\cdot E(i)\right]d(L_i)\right.$$ $$\left. + \left[\frac{\partial(P(i))}{\partial(Ty_i)}\cdot I(i)\cdot E(i) + P(i)\cdot\frac{\partial(I(i))}{\partial(Ty_i)}\cdot E(i) + P(i)\cdot I(i)\cdot\frac{\partial(E(i))}{\partial(Ty_i)}\right]d(Ty_i)\right\}$$ $$+ \sum_{j=1}^{t/\tau}[P(N+1)\cdot I(N+1)\cdot E(N+1)]dN - \frac{t}{\tau^2}\cdot\sum_{i=1}^{N}[P(i)\cdot I(i)\cdot E(i)]d\tau$$ Where SW represents CEG, COM, DD, or RT. $E(i)$ depends only on the location of the defect; $P(i)$ depends on the location L_i and on the type of the defect Ty_i; $I(i)$ depends on the location and type of the defect; various iterations in time are independent; defects are independent. Note that the $\oint$ used in previous chapters becomes a simple Σ when the defects are independent.
II	COM	
II	DD	
II	RT	
II	CF	

Table 19.29 Initial Sensitivity Analysis Results (Continued)

Group	Measure	Sensitivity Equations
III	TC	$$dR_{TC} = \frac{1}{C_0} \cdot \left[1 - \sum_{j=1}^{t/\tau} \sum_{i=1}^{N_0} [P(i) \cdot I(i) \cdot E(i)]\right]^{\frac{1}{C_0}-1} \cdot \left\{ \sum_{j=1}^{t/\tau} \sum_{i=1}^{N_0} \left\{ -\left[\frac{\partial(P(i))}{\partial(L_i)} \cdot I(i) \cdot E(i) + P(i) \cdot \frac{\partial(I(i))}{\partial(L_i)} \cdot E(i)\right] d(L_i) \right.\right.$$ $$\left. - \left[\frac{\partial(P(i))}{\partial(Ty_i)} \cdot I(i) \cdot E(i) + P(i) \cdot \frac{\partial(I(i))}{\partial(Ty_i)} \cdot E(i) + P(i) \cdot I(i) \cdot \frac{\partial(E(i))}{\partial(Ty_i)}\right] d(Ty_i) \right\}$$ $$\left. - \sum_{j=1}^{t/\tau} [P(N_0+1) \cdot I(N_0+1) \cdot E(N_0+1)] dN_0 + \frac{t}{\tau^2} \cdot \sum_{i=1}^{N_0} [P(i) \cdot I(i) \cdot E(i)] d\tau \right\}$$ $$+ \left[1 - \sum_{j=1}^{t/\tau} \sum_{i=1}^{N_0} [P(i) \cdot I(i) \cdot E(i)]\right]^{\frac{1}{C_0}} \cdot \ln\left[1 - \sum_{j=1}^{t/\tau} \sum_{i=1}^{N_0} [P(i) \cdot I(i) \cdot E(i)]\right] \left(-\frac{1}{C_0^2}\right) \left(\frac{a_0 a_1 a_2 e^{a_2 C_1}}{1 + a_1[\exp(a_2 C_1) - 1]}\right)$$ $$\cdot C_1 \left(\frac{dR_1}{R_1} - \frac{dR_R}{R_R} + \frac{dLOC_{tested}}{LOC_{tested}} - \frac{dLOC_{total}}{LOC_{total}}\right)$$ Where: $P(i)$ is a function of L_i and Ty_i; $I(i)$ is a function of L_i and Ty_i; $E(i)$ is a function of L_i. Various iterations in time are independent; defects are independent. Note that the $\oint$ used in previous chapters becomes a simple Σ when the defects are independent.

Table 19.29 Initial Sensitivity Analysis Results (Continued)

Group	Measure	Sensitivity Equations
III	**FDN**	$$dR_{FDN} = \frac{\mu_U(t_{OP})}{N_A} \cdot \left[1 - \sum_{j=1}^{t/\tau}\sum_{i=1}^{N_A}[P(i)\cdot I(i)\cdot E(i)]\right]^{\frac{\mu_U(t_{OP})}{N_A}-1} \cdot \left\{\sum_{j=1}^{t/\tau}\sum_{i=1}^{N_A}\left\{-\left[\frac{\partial(P(i))}{\partial(L_i)}\cdot I(i)\cdot E(i) + P(i)\cdot\frac{\partial(I(i))}{\partial(L_i)}\cdot E(i)\right]d(L_i) - \left[\frac{\partial(P(i))}{\partial(Ty_i)}\cdot I(i)\cdot E(i) + P(i)\cdot\frac{\partial(I(i))}{\partial(Ty_i)}\cdot E(i) + P(i)\cdot I(i)\cdot\frac{\partial(E(i))}{\partial(Ty_i)}\right]d(Ty_i)\right\} + \frac{t}{\tau^2}\cdot\sum_{i=1}^{N_A}[P(i)\cdot I(i)\cdot E(i)]d\tau\right\} + R_{FDN}\cdot\mu_U(t_{OP})\cdot\left(-\frac{\ln\left[1-\sum_{j=1}^{t/\tau}\sum_{i=1}^{N_A}[P(i)\cdot I(i)\cdot E(i)]\right]}{N_A^2} - \frac{\sum_{j=1}^{t/\tau}[P(N_A+1)\cdot I(N_A+1)\cdot E(N_A+1)]}{N_A\cdot\left[1-\sum_{j=1}^{t/\tau}\sum_{i=1}^{N_A}[P(i)\cdot I(i)\cdot E(i)]\right]}\right)dN_A + \frac{R_{FDN}}{N_A}\cdot\ln\left[1-\sum_{j=1}^{t/\tau}\sum_{i=1}^{N_A}[P(i)\cdot I(i)\cdot E(i)]\right] \cdot\left[-\frac{\mu_U(t_{OP})}{\gamma}\cdot\left(\frac{\partial\gamma}{\partial\upsilon\mu_H}\cdot d\upsilon\mu_H + \frac{\partial\gamma}{\partial\mu_R}\cdot d\mu_R + \frac{\partial\gamma}{\partial z_a}\cdot dz_a + \frac{\partial\gamma}{\partial t_{OP}}\cdot dt_{OP}\right) + \frac{d^2(FDN_A)}{dt_{OP}}\cdot\frac{\mu_U(t_{OP})}{\gamma}\right]$$ Where: N_A is the number of faults identified in the last version of development code. The last version is defined as the version right before the released version and the last version which still contains defects which will be further corrected in the released version; t_{OP}, is the time at which the operational phase starts (it is also the time at which the code is frozen); $P(i)$ is a function of L_i and Ty_i (for $i = 1$ to N_A); $I(i)$ is a function of L_i and Ty_i (for $i = 1$ to N_A); $E(i)$ is a function of L_i (for $i = 1$ to N_A) and the form of $\gamma(...)$ is used symbolically. Note that the $\oint$ used in previous chapters becomes a simple Σ when the defects are independent.

19.6.9 Data Collection and Analysis

As discussed in Section 19.4, a follow on issue is to define a data collection and analysis process based on ISO 15939 [ISO, 2007].

19.6.10 Combining Measures

A follow on effort could determine how to down-select to a smaller number of measures that can be combined to yield a more accurate reliability estimation—an estimation that would be better than any single measure alone.

19.6.11 Automation Tools

As shown in Table 19.24, performing some of the measurements is time consuming. It would be helpful if automation tools were developed to assist the measurement process. However, the development of automation tools is out of the scope of this particular research. Tools have been used to evaluate the number of lines of code in BLOC, code cyclomatic complexity in CC, and test coverage in TC. For other measures, i.e., CEG, COM, DD, RSCR, and RT, the measurement process was conducted manually. No validated tools with the ability to replace humans in the inspection of natural language-based requirements and design documents currently exist. The development of such natural-language processing tools was not the objective of this research and should be the focus of a follow on effort. Reliance on such tools would significantly reduce the time necessary to apply the methods discussed in the report and would, in addition, increase the repeatability of the measurement process. Approaches to automation will be discussed in Chapter 20.

19.6.12 Priority Ranking of the Follow-On Issues

The experts provided a ranking of the follow-on issues displayed in Table 19.30 and identified possible solutions to each of the high-priority issues.

Table 19.30 Priority Ranking for Follow-On Issues

Follow on Issue	Priority	Overall Rankings	Recommendations
Repeatability	H	1	Perform Requirements Review studies;
Data collection process and Data Analysis (Detailed Guidelines)	H	1	Define a data collection and analysis process based on ISO 15939 for each of the measures selected (i.e., Measures Recommended in Table 19.27). This standard provides a detailed process to ensure the quality of the data collection; Draft piloted;
Uncertainty (failure probability distribution for predictions based on different measures)	H	1	Consider both Measurement Uncertainty and Model Uncertainty; The uncertainty can be reduced by ensuring the quality of the data collection and repeatability; Reduce the model uncertainty from OP, EFSM and Parameters.
Combining measures	M	5	
Additional applications	M	4	
Common Cause Failures	M	6	
Cases with no defect	M	6	
Tools/Automation	L	8	
Old Parameters	L	8	

19.7 References

[Briand, 1997] L.C. Briand et al. "Quantitative Evaluation of Capture-Recapture Models to Control Software Inspections," presented at *The 8th International Symposium on Software Reliability Engineering*, Albuquerque, NM, USA, 1997.

[Chao, 1992] A. Chao, S.M. Lee and S.L. Jeng. "Estimating Population Size for Capture-Recapture Data When Capture Probabilities Vary by Time and Individual Animal," *Biometrics*, vol. 48, pp. 201–16, 1992.

[Ebenau, 1993] R.G. Ebenau and S.H. Strauss. *Software Inspection Process*. McGraw-Hill, 1993.

[Gaffney, 1984] J.E. Gaffney. "Estimating the Number of Faults in Code." *IEEE Transactions on Software Engineering*, vol. 10, pp. 459–64, 1984.

[IEEE, 1990] "IEEE Standard Glossary of Software Engineering Terminology," IEEE Std. 610, 1990.

[IEEE, 2003] "IEEE Standard Criteria for Digital Computers in Safety Systems of Nuclear Power Generating Stations," IEEE Std. 7-4.3.2, 2003.

[ISO, 2007] ISO/IEC 15939:2007, "Systems and software engineering – Measurement process," ISO, 2007.

[Musa, 1992] J. Musa. "The Operational Profile in Software Reliability Engineering: An Overview," presented at *3rd International Symposium on Software Reliability Engineering*, 1992.

[Nejad, 2002] H.S. Nejad, M. Li and C. Smidts. "On the Location of Faults in a Software System," Master's Thesis, University of Maryland, College Park, 2002.

[Otis, 1978] D.L. Otis et al. "Statistical Inference from Capture Data on Closed Animal Populations," *Wildlife Monographs*, vol. 62, pp. 1–135, 1978.

[Schach, 1993] S.R. Schach. *Software Engineering*. 2nd ed., Homewood, IL: Aksen Associates Inc., 1993.

[Shi, 2010] 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 Systems Assurance Engineering and Management*. vol. 1, pp. 36–43, 2010.

[Smidts, 2011] C. Smidts and Y. Shi. "Predicting Residual Software Fault Content and their Location during Multi-Phase Functional Testing Using Test Coverage," *International Journal of Reliability and Safety*. 2011.

[Voas, 1992] J.M. Voas. "PIE: A Dynamic Failure-Based Technique," *IEEE Transactions on Software Engineering*, vol. 18, pp. 717–27, 1992.

20. DEVELOPMENT AND USE OF AUTOMATION TOOLS

This chapter discusses the development or increased use of automation tools as discussed in Section 19.6.11.

The RePS methodology has been validated on PACS 1, PACS 2, and the APP. However, many of the measurements and processes required to predict reliability were manually performed or used limited automation. As addressed in chapter 19, 10 out of the 12 measurements and related reliability predictions required more than 30 days effort to complete.

Measurements related to Defect Density (DD), Requirements Traceability (RT), and Test Coverage (TC), which were identified in chapter 19 as the best candidates for reliability prediction, cannot be fully automated. More specifically, current tools for inspecting requirements documents have not been validated. Also, currently there is no available tool support for inspecting design documents. A number of tools exist claiming the ability to perform automated requirements traceability and test-coverage analysis. A follow on issue is to evaluate existing tools.

Construction of the Extended Finite State Machine (EFSM), which is used to propagate the defects uncovered by various measurement processes, is time-consuming. Current tools used for EFSM construction provide only limited support in automatically propagating the identified defects. Further development is required to automate straightforward but tedious activities.

The purpose of future efforts would be to develop an automated reliability prediction tool. This CASE tool should provide for:

1. Construction of the EFSM from requirements documents
2. Building the operational profile (OP)
3. Mapping the defects uncovered by different measurement processes to the constructed EFSM
4. Mapping the OP to the EFSM
5. Running the modified EFSM and obtaining reliability predictions

This follow on development effort should first evaluate existing tools that were designed to aid the measurements process for DD, RT, and TC and determine whether these tools implement the claimed functionalities as well as assess their efficiency and effectiveness. A new tool for assessing the quality of software code and documents would be based on the most efficient and effective of these tools. New functionalities should be developed as required.

To meet the above objectives, the following activities should be performed:

1. Construct the EFSM semi-automatically based on the requirements documents and the procedure, which will be described in detail in Appendix A.

2. Obtain the OP (operational profile) using the following possible approaches:

   2.1 If some operational data is available, develop a function that could either automatically or semi-automatically transform the information to a format that can be mapped into the constructed EFSM

   2.2 If PRA records are available, develop a function that could either automatically or semi-automatically transfer and transform the information into a form that can be interpreted by the EFSM

   2.3 If hardware-failure information is available, also develop a function that could either automatically or semi-automatically transfer the information to the EFSM

3. Develop a function that could either automatically or semi-automatically map the uncovered defects into the EFSM.

4. Connect the obtained OP, uncovered defects, and the constructed EFSM and create a function for reliability prediction. The entire process is illustrated in Figure 20.1.

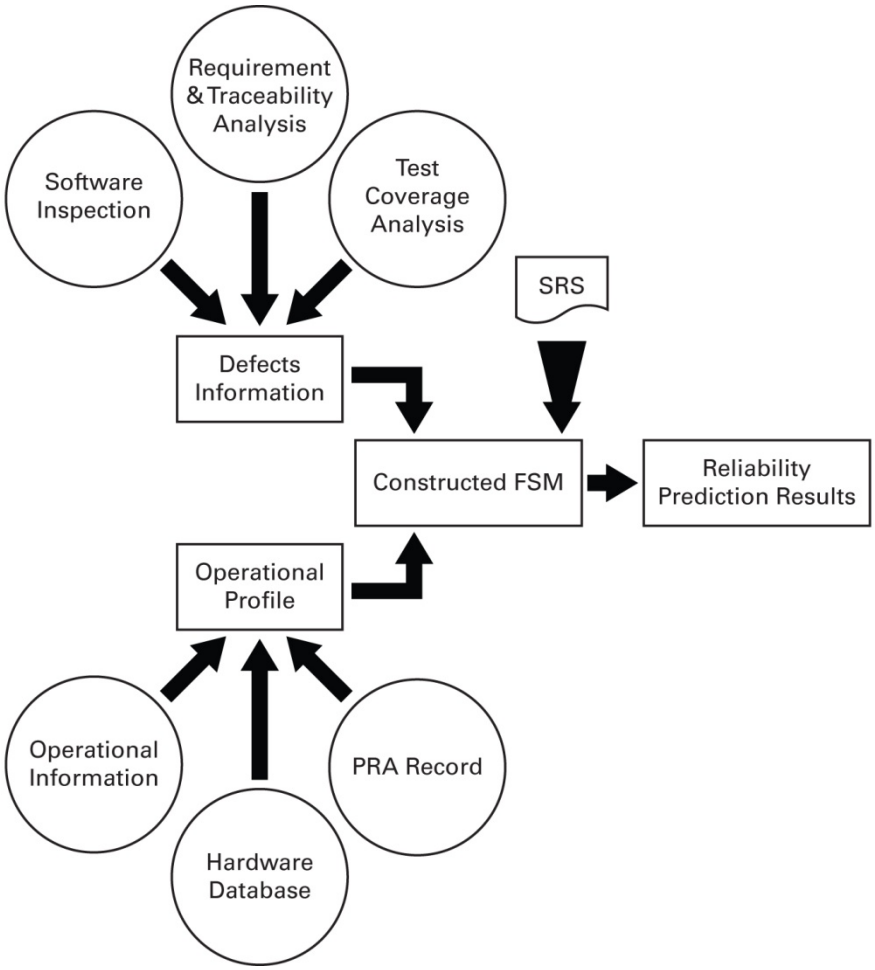


Figure 20.1 Structure of the Automated Reliability Prediction System

5. Systematically evaluate current tools used for measurement processes.

    5.1 Evaluate the tools for requirements analysis. Example tools are the NASA ARM (Automated Requirements Measurement, 1997) and the SEI QuARS (Quality Analyzer for Requirements Specifications 2005) [Lami, 2005].
    5.2 Evaluate currently available code inspection tools.
    5.3 Evaluate currently available requirements traceability analysis tools.
    5.4 Evaluate currently available code coverage tools.
    5.5 Select efficient and effective tools.

20.1 References

[Lami, 2005] G. Lami. “QuARS: A Tool for Analyzing Requirements.” Technical Report, CMU/SEI-2005-TR-014, 2005.

APPENDIX A: EXTENDED FINITE STATE MACHINE AND ITS CONSTRUCTION PROCEDURES[47]

As specified in Section 5.1, the PIE concept was introduced to describe the software failure mechanism if one knows the location of the defects. How to implement the PIE concept for reliability quantification is discussed in this appendix. [Shi, 2009]

In the original assessment method, P, I, and E are quantified statistically using mutation [Voas, 1992]. 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 for complex systems.

In this appendix, a simple, convenient, and effective method to solve this problem using an Extended Finite State Machine (EFSM) [Wang, 1993] model 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 new system 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 software input variables. These variables cross the boundary of the application.
- Γ is the set of software output variables. These variables cross 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.

[47] Extract from "On the Use of Extended Finite State Machine Models for Software Fault Propagation and Software Reliability Estimation," by Ying Shi, et al. Published in the International Topical Meeting on Nuclear Plant Instrumentation Control, and Human-Machine Interface Technologies, Knoxville, TN, March 5-9, 2009. Copyright 2009 by the American Nuclear Society, La Grange Park, Illinois.

The method proposed for assessing software reliability based on an 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; and
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 for assessing software reliability based on an EFSM is discussed in turn in the following sections.

A.1 Step 1: Construct of a High-Level EFSM Based On the SRS

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. The HLEFSM will be manually constructed based on the SRS. Figure A.1 shows a typical prototype outline for an SRS [IEEE, 1998].

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 A.1 Typical Prototype Outline for SRS

The general procedure to be followed for constructing a HLEFSM is illustrated in Figure A.2.

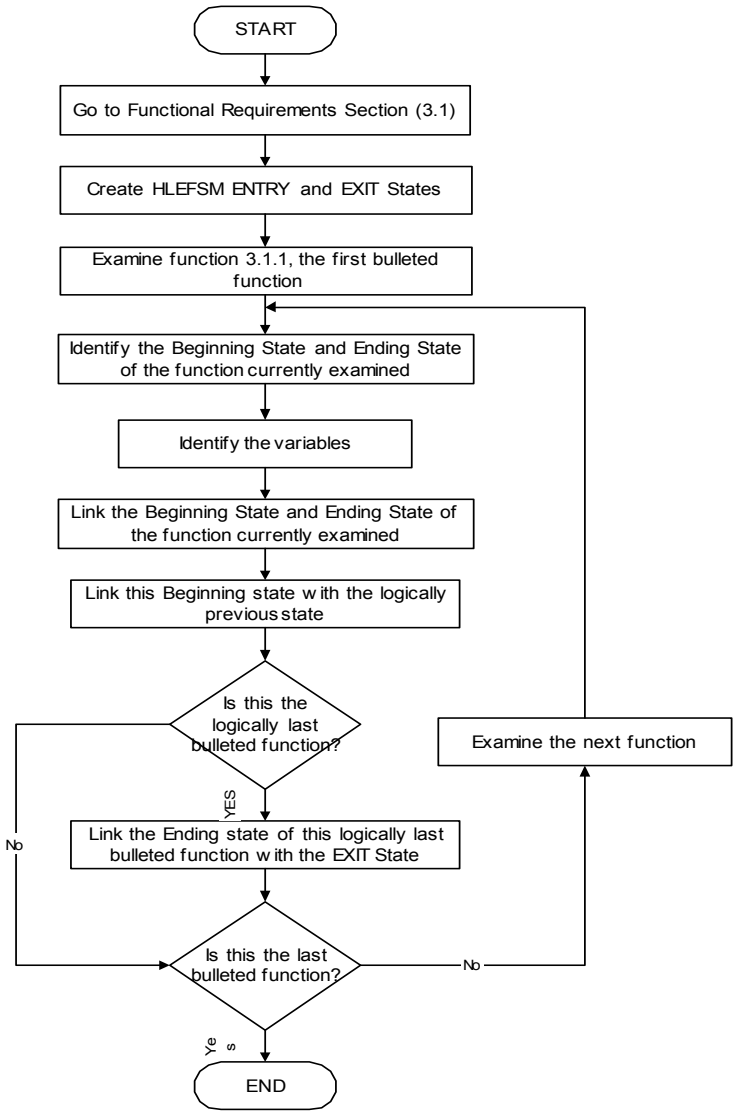


Figure A.2 SRS-Based HLEFSM Construction

The general construction procedure includes:

a) Study the SRS and focus on the Functional Requirements section (Section 3.1 in Figure A.1). It should be noted that there exists several other SRS prototypes [IEEE, 1998]. For those prototypes, one can still find a section similar to the Functional Requirements section that describes the functions of the software system.
b) Create an ENTRY state and an EXIT state for the entire application.
c) Examine the first bulleted[48] function f_1 defined under 3.1.1 in Figure A.1.

[48] A bulleted function is a function explicitly documented using a bullet in the SRS document for distinguishing it from other functions.

d) Define the corresponding states of the function f_1 (normally it is logically the first function of the software system): the starting state $S_i(f_1)$: $S_i(f_1) \in S$ and the ending state $S_0(f_1)$: $S_0(f_1) \in S$ of the function f_1.

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 $S_0(f_1)$, denoted as $V_{S_0(f_1)}$, since a state is the condition of an extended 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 $V_{S_0(f_1)} = ov(f_1) \cup nu(f_1)$.

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 $S_0(f_1)$.

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 $S_i(f_j)$ 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, $V_{S_0(f_{j-1})}$ plus some inputs from Σ. That is, $V_{S_i(f_j)} = V_{S_0(f_{j-1})} \cup v_j$, 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 A.1 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[49]) SRS and its associated EFSM elements identifications are shown in Table A.1.

Table A.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	$V_{S_o(f_2)} = ov(f_2)$

A.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 [Fagan, 1976] [Beizer, 1990]. All the defects identified through inspection or testing should be recorded properly for further references and examinations. Table A.2 or similar table should be generated.

[49] 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 A.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 A.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, Software Design Documents (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 A.2.

Table A.3 Possible Instances or Further Description for Each Field in Table A.2

Item	Possible Instances of Each Field in Table A.2	
Defect Description	Plain English sentence.	
Defect Location	SRS; 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 A.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}

A.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 A.3. The following subsections will describe how to localize the defects in the HLEFSM and how to modify a HLEFSM and the obtained low-level EFSM (LLEFSM).

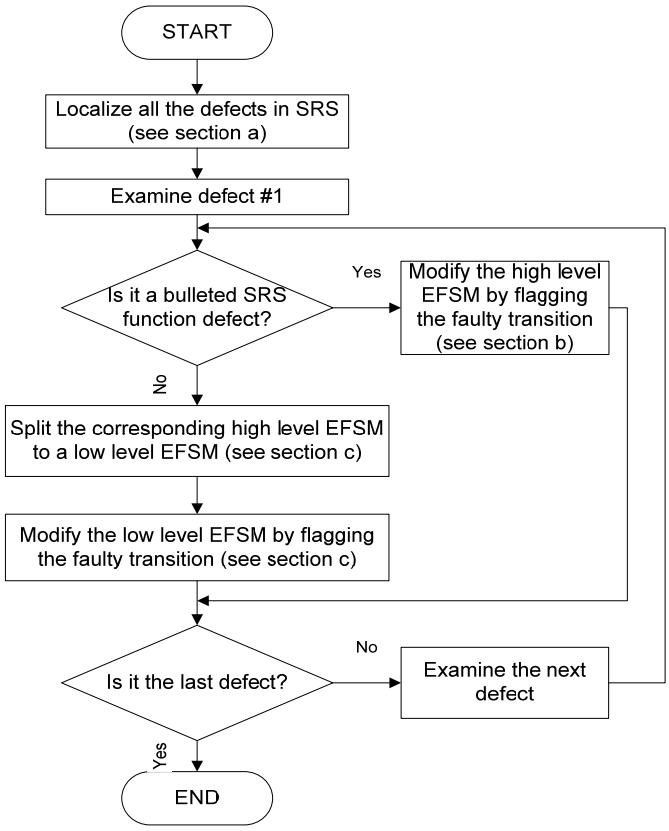


Figure A.3 General Procedures for Defect Mapping

A.3.1 Section A: Localize the Defects in the HLEFSM:

One must know the exact locations of the defects to correctly modify the HLEFSM. The localization of the defects is based on tracing among the development documents: SRS, SDD, and code that have been inspected. Figure A.4 illustrates the detailed tracing procedures.

A.3.2 Section B: Modify the HLEFSM:

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_i have now changed from $t_i = \{p_i, f_i(iv)\}$ to $t_i = \{p_i, f_i(iv), d_i\}$.

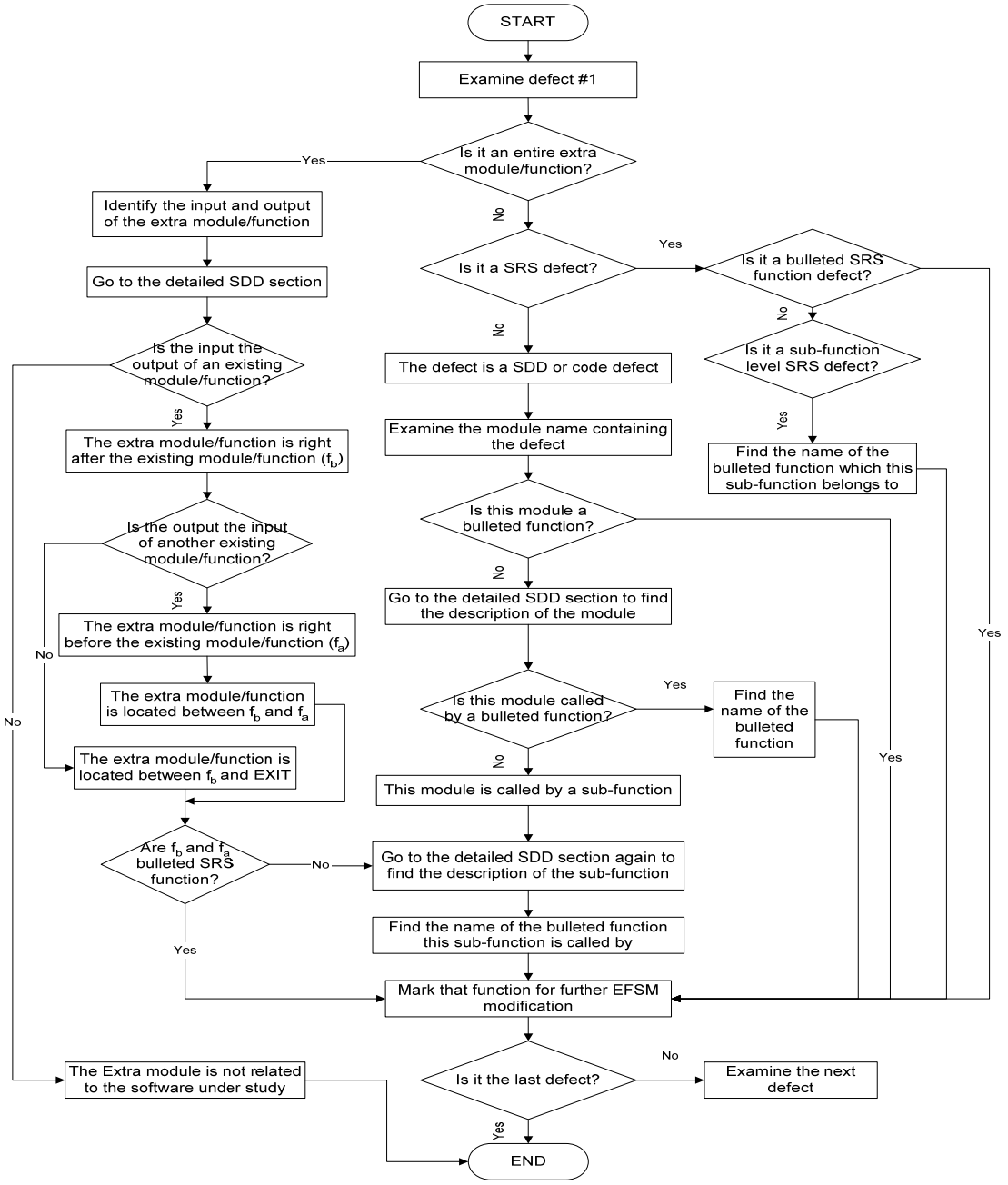


Figure A.4 Flowchart for Localizing the Defects

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

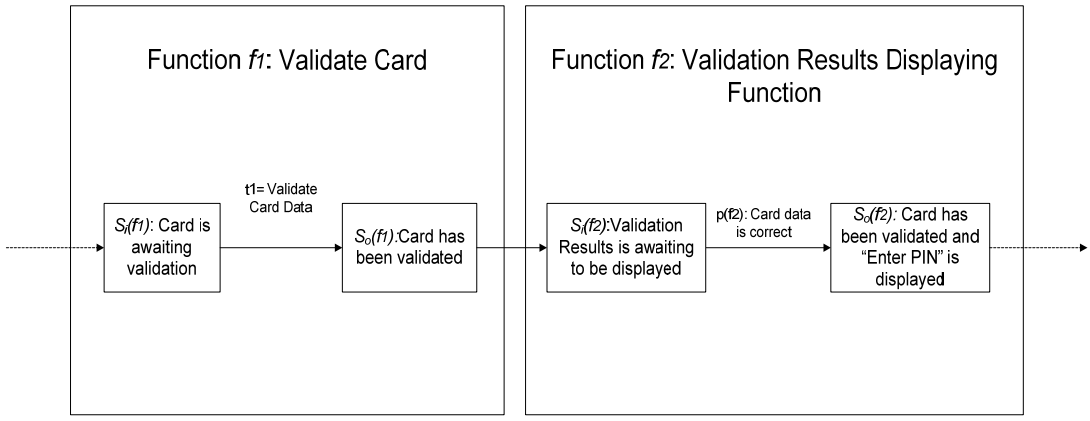


Figure A.5 Original EFSM for Example 1

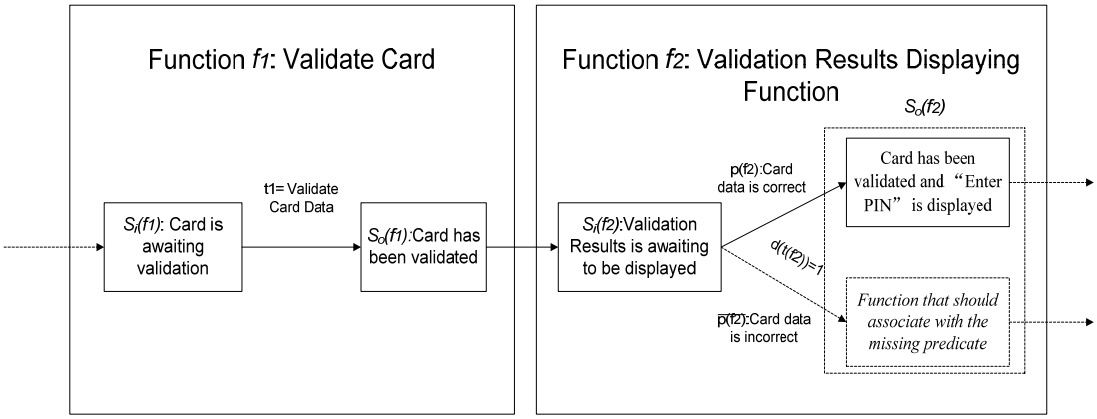


Figure A.6 Modified EFSM for Example 1

A.3.3 Section C: Split the HLEFSM to a LLEFSM

If a defect is 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[50].

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 $f_{ij}, j = 1,2, \dots, n$. These identified sub-functions act 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 directly linking the ending state $S_0(f_{in})$ of the last child function with the ending state of its parent function $S_0(f_i)$.
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.

A.3.4 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 because 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, that is, $y = f(x)$. 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 [Garret, 1995] should be

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

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.

A.3.5 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 used 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. 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 extended 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. The probability of failure is:

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

where

p_f the probability of failure
p_{path_i} the probability of execution of the *i*-th path with defect(s)

A.4 References

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

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

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

[Garrett, 1995] 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.

[Shi, 2009] 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 *Proc. 6th American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Controls, and Human Machine Interface Technology*, 2009.

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

[Wang, 1993] C.J. Wang and M.T. Liu. "Generating Test Cases for EFSM with Given Fault Models," in *Proc. 12th Annual Joint Conference of the IEEE Computer and Communications Societies*, 1993.

APPENDIX B: LIST OF SYMBOLS

Chapter 4

O	Operational profile
O_{s1}	Operational profile for subsystem 1
O_{s2}	Operational profile for subsystem 2
O_{sn}	Operational profile for subsystem n
O_v	Operational profile for the voter
O_{S_i}	Operational profile for subsystem i
$O_{S_1M_1}$	Operational profile for the first system mode of subsystem i
$O_{S_iM_n}$	Operational profile for the n-th system mode of subsystem i
$O_{S_ipi_j}$	Operational profile for the plant inputs
$O_{S_iii_j}$	Operational profile for the infrastructure inputs
O_{APP}	Operational profile for the APP system
$O_{\mu p1}$	Operational profile for µp1
$O_{\mu p2}$	Operational profile for µp2
O_{CP}	Operational profile for CP
$\Pr(EEPROM)$	Probability of failure per demand
λ_{ave}	Average failure rate
$\Pr(EEPROM)'$	Updated probability of failure per demand
$\hat{\lambda}$	Unbiased failure rate
r	Failures
T	Hours

Chapter 5

p_f	Failure probability (unreliability)
$P(i)$	Propagation probability for the *i*-th defect
$I(i)$	Infection probability for the *i*-th defect
$E(i)$	Execution probability for the *i*-th defect
$p_{d(g)}$	*g*-th input/output path
$\Pr(p_{d(g)})$	Probability of traversing the *g*-th path
p_f^*	Probability of failure caused by defect
$d_{d(g)}(q)$	Probability that the *q*-th transition is traversed in the *g*-th path
q	Transition index
g	Path index
$n(g)$	Number of transitions in the *g*-th path
λ	Failure rate
$R(t)$	Software reliability
K	Fault exposure ratio
T_L	Linear execution time
N	Number of defects
t	Execution time
vK	New fault exposure ratio

Chapter 6

F	Total number of defects in the software
i	Module index
N	Number of modules

FP	Function point count
s_i	Number of lines of code for the *i*-th module
$p_s(BLOC)$	Reliability estimation for the APP system using the BLOC measure
K	Fault exposure ratio, in failure/defect
N_{BLOC}	Number of defects estimated using the BLOC measure
τ	Number of defects estimated using the BLOC measure
T_L	Linear execution time
$T_L(\mu p1)$	Linear execution time of μp1 of the APP system
$T_L(\mu p2)$	Linear execution time of μp2 of the APP system
$T_L(CP)$	Linear execution time of CP of the APP system
$\tau(\mu p1)$	Average execution-time-per-demand of μp1
$\tau(\mu p2)$	Average execution-time-per-demand of μp2
$\tau(CP)$	Average execution-time-per-demand of CP

Chapter 7

$A_{existing}$	Ambiguities in a program remaining to be eliminated
A_{total}	Total number of ambiguities identified
$CE\%$	Percentage of ambiguities remaining over indentified
$ACEG$	Actually implemented cause-effect graph
C^A	The cause set of the ACEG
E^A	The observable effect set of the ACEG
F^A	The Boolean function set of the ACEG
CON^A	The constraint set of the ACEG
$BCEG$	Benchmark cause-effect graph

C^B	The cause set of the BCEG
E^B	The observable effect set of the BCEG
F^B	The Boolean function set of the BCEG
CON^B	The constraint set of the BCEG
e_j^A	The *j*-th distinct observable effect in the ACEG
m	The number of distinct effects in the union set $E^A \cup E^B$
e_j^B	The peer observable effect in the BCEG corresponding to e_j^A
f_j^A	A Boolean function in F^A corresponding to e_j^A
f_j^B	A Boolean function in F^B corresponding to e_j^B
C_j^A	The set of causes appearing in f_j^A
C_j^B	The set of causes appearing in f_j^B
C_j	The union set of C_j^A and C_j^B
n_j	The number of distinct causes in C_j
$\overrightarrow{C_j}$	A cause state vector, which represents a state combination of all causes
$\overrightarrow{C_j}^k$	The *k*-th vector of $\overrightarrow{c_j}$

Chapter 8

$p_s(CMM)$	Reliability estimation for the APP system using the CMM measure.
K	Fault Exposure Ratio, in failures/defect
N_{CMM}	Number of defects estimated using the CMM measure
τ	Average execution-time-per-demand, in seconds/demand
T_L	Linear execution time of a system, in seconds
$N_{CMM,critical}$	Number of delivered *critical defects* (severity 1)
$N_{CMM,significant}$	Number of delivered *significant defects* (severity 2)

$T_L(\mu p1)$	Linear execution time of μp1 of the APP system
$T_L(\mu p2)$	Linear execution time of μp2 of the APP system
$T_L(CP)$	Linear execution time of CP of the APP system
$\tau(\mu p1)$	Average execution-time-per-demand of μp1
$\tau(\mu p2)$	Average execution-time-per-demand of μp2
$\tau(CP)$	Average execution-time-per-demand of CP

Chapter 9

CM	Completeness measure
w_i	The weight of the *i*-th derived measure
D_i	The *i*-th derived measure
D_1	The fraction of functions satisfactorily defined
D_2	The fraction of data references having an origin
D_3	The fraction of defined functions used
D_4	The fraction of referenced functions defined
D_5	The fraction of decision points whose conditions and condition options are all used
D_6	The fraction of condition options having processing
D_7	The fraction of calling routines whose parameters agree with the called routines defined parameters
D_8	The fraction of condition options that are set
D_9	The fraction of set condition options processed
D_{10}	The fraction of data references having a destination
B_1	The number of functions not satisfactorily defined
B_2	The number of functions
B_3	The number of data references not having an origin

B_4	The number of data references
B_5	The number of defined functions not used
B_6	The number of defined functions
B_7	The number of referenced functions not defined
B_8	The number of referenced functions
B_9	The number of decision points missing condition(s)
B_{10}	The number of decision points
B_{11}	The number of condition options having no processing
B_{12}	The number of condition options
B_{13}	The number of calling routines whose parameters not agreeing with the called routines defined parameters
B_{14}	The number of calling routines
B_{15}	The number of condition options not set
B_{16}	The number of set condition options having no processing
B_{17}	The number of set condition options
B_{18}	The number of data references having no destination

Chapter 10

c	Coverage factor of a fault-tolerance mechanism
Pr	The probability of $H(g) = 1$ when $g \in G$
H	A variable characterizing the handling of a particular fault/activity pair
G	The global input space of a fault-tolerance mechanism
F	Fault Space
A	Activity space, or activation space
g	A fault/activity pair, or a point in space G

$p(g)$	The probability of occurrence of g
$H(g)$	The value of H for a given point g
$E(H)$	$E(H)$ is the expected value of H
N_1	The number of occurrences of the Normal State for an experiment
N_2	The number of occurrences of the Fail-safe State for an experiment
N_3	The number of occurrences of the Normal State for an experiment
N_4	The number of occurrences of the Fail-safe State for an experiment
N_{t1}	The total number of experiments with analog input inside the "Barn shape"
N_{t2}	The total number of experiments with analog input outside the "Barn shape"
W_1	The weight of experiments such that the analog input is inside the "Barn shape"
W_2	The weight of experiments such that the analog input is outside the "Barn shape"
λ	The failure rate of a microprocessor
β	The failure rate of the *i*-th primary component
λ_1	The rate at which the system deals with the fault injected and generates the result
γ_1	The probability that the system is brought back to the Normal State when an erroneous state is recovered
γ_2	The probability that the system remains in the Recoverable State when an erroneous state cannot be recovered
γ_3	The probability that the system enters the Failure State 1 when an erroneous state leads to the system failure
γ_4	The probability that the system enters the Failure State 2 when an erroneous state leads to the system failure
β_1	Failure rate of RAM
β_2	Failure rate of PROM
β_3	Failure rate of EEPROM
β_4	Failure rate of DPM

β_5	Failure rate of Address Bus Line
β_6	Failure rate of CP register
N_5	The number of occurrences of the Recoverable State for an experiment such that the analog input is inside the "Barn shape"
N_6	The number of occurrences of the Recoverable State for an experiment such that the analog input is outside the "Barn shape"
N_7	The number of occurrences of the Failure State 1 for an experiment such that the analog input is outside the "Barn shape"
N_8	The number of occurrences of the Failure State 3 for an experiment such that the analog input is outside the "Barn shape"
N_9	The number of occurrences of the Failure State 2 for an experiment such that the analog input is inside the "Barn shape"
N_{10}	The number of occurrences of the Failure State 3 for an experiment such that the analog input is inside the "Barn shape"
$\overrightarrow{P(t)}$	A column vector whose elements are the system state probabilities at time t
$p_i(t)$	The probability that the system is in a state i at time t
n	A finite and countable number of states for a state space
$\bar{\bar{A}}$	The $n \times n$ matrix of the transition rates
$p_1(t)$	The probability the system is in "Normal State" at time t
$p_2(t)$	The probability that the system is in "Recoverable State" at time t
$p_3(t)$	The probability that the system is in "Fail-safe State" at time t
$p_4(t)$	the probability that the system is in "Failure State 1" at time t
$p_5(t)$	The probability that the system is in "Failure State 2" at time t
$p_6(t)$	The probability that the system is in "Failure State 3" at time t
$R(t)$	The reliability of a microprocessor
$R_s(t)$	The reliability of the whole APP system
$F_i(t)$	The probability of the *i*-th type of failure

Chapter 11

CC_i	The cyclomatic complexity measure of the *i*-th module
E_i	The number of edges of the *i*-th module
N_i	The number of nodes of the *i*-th module
$p_1\%$	Percentage of modules whose cyclomatic complexity is less than 4
$p_2\%$	Percentage of modules whose cyclomatic complexity is greater than or equal to 4 and less than 10
$p_3\%$	Percentage of modules whose cyclomatic complexity is greater than or equal to 10 and less than 16
$p_4\%$	Percentage of modules whose cyclomatic complexity is greater than or equal to 16 and less than 20
$p_5\%$	Percentage of modules whose cyclomatic complexity is greater than or equal to 20 and less than 30
$p_6\%$	Percentage of modules whose cyclomatic complexity is greater than or equal to 30 and less than 80
$p_7\%$	Percentage of modules whose cyclomatic complexity is greater than or equal to 80 and less than 100
$p_8\%$	Percentage of modules whose cyclomatic complexity is greater than or equal to 100 and less than 200
$p_9\%$	Percentage of modules whose cyclomatic complexity is greater than or equal to 200
p_i	The percentage of modules whose cyclomatic complexity belong to the *i*-th level
n_i	The number of modules whose cyclomatic complexity belong to the *i*-th level
SLI_1	The SLI value of the cyclomatic complexity factor
f_i	Failure likelihood f_i used for SLI_1 calculations
N	The number of faults remaining in the delivered source code
k	A universal constant, estimated by fitting experiment data
A	The amount of activity in developing the delivered source code
F	Universal constant, estimated by fitting experiment data
SLI	The Success Likelihood Index of the entire software product

$SIZE$	The size of the delivered source code in terms of LOC
$p_s(CC)$	Reliability estimation for the APP system accounting for the effect of Cyclomatic Complexity (CC)
K	Fault Exposure Ratio, in failure/defect
N_{CC}	Number of defects estimated using the *CC* measure
τ	Average execution-time-per-demand, in seconds/demand
T_L	Linear execution time of a system, in seconds
$T_L(\mu p1)$	Linear execution time of μp1 of the APP system
$T_L(\mu p2)$	Linear execution time of μp2 of the APP system
$T_L(CP)$	Linear execution time of CP of the APP system
$\tau(\mu p1)$	Average execution-time-per-demand of μp1 of the APP system
$\tau(\mu p2)$	Average execution-time-per-demand of μp2 of the APP system
$\tau(CP)$	Average execution-time-per-demand of CP of the APP system
$TDEV_{actual}$	Actual time to develop the software, in calendar months
$TDEV_{nominal}$	Nominal time to develop the software, in calendar months
$SIZE_{developed}$	The size of developed source code, in KLOC
$SIZE_{delivered}$	The size of finally delivered source code, in KLOC
$SIZE_{discarded}$	The size of source code discarded during development, in KLOC
$SIZE_{new}$	The size of new code developed from scratch, in KLOC
$ESIZE_{adapted}$	The equivalent size of adapted code, in KLOC
$ESIZE_{reused}$	The equivalent size of reused code, in KLOC
$ESIZE_{COTS}$	The equivalent size of off-the-shelf software, in KLOC
AA	Percentage of assessment and assimilation
AAF	Adaptation adjustment factor
AAM	Adaptation adjustment modifier

AT	Percentage of code re-engineered by automation
CM	Percentage of code modified
DM	Percentage of design modified
IM	Percentage of integration effort required for integrating adapted or reused software
SU	Percentage of software understanding
$UNFM$	Programmer unfamiliarity with software
W_i	The weight of the *i*-th influence factor
SLI_i	The SLI value of the *i*-th influence factor

Chapter 12

DD	Defect Density
i	An index reflecting the development stage. A value of 1 represents the requirements stage, a value of 2 represents the design stage and a value of 3 represents the coding stage
j	The index identifying the specific inspector
$D_{i,j}$	The number of unique defects detected by the *j*-th inspector during the *i*-th development stage in the current version of the software
$DF_{l,k}$	The number of defects found in the *l*-th stage and fixed in the *k*-th stage
DU_M	The number of defects found by exactly m inspectors and remaining in the code stage
N	Total number of inspectors
$KLOC$	The number of source lines of code (LOC) in thousands

Chapter 13

FD	Fault-days for the total system
FD_i	Fault-days for the *i*-th fault
f_{in}	Date at which the *i*-th fault was introduced into the system

f_{out}	Date at which the *i*-th fault was removed from the system
l	Total number of faults
$\{t_{end}\}_{\varphi}$	Ending date of the phase φ in which the fault was introduced/removed
$\{t_{beginning}\}_{\varphi}$	Beginning date of the phase φ in which the fault was introduced/removed
$\{\mu_u(t)\}_{j\varphi}$	Expected fault count at time t
j	A category of faults introduced during phase *j*
f	A life cycle phase
t	Life cycle time
$\{v(t)\mu_u(t)\}_{j\varphi}$	Estimate of fault introduction rate in phase f
$\{z_a(t)\}_{j\varphi}$	Intensity function of per-fault detection in phase f
$\{\mu_R(t)\}_{\varphi}$	Expected change in fault count due to each repair in phase f
$\overline{\{v(t)\mu_H(t)\}_j}$	Unadjusted estimate of the fault introduction rate of the *j*-th fault categories
F_{φ}	A constant
DP	Fault potential per function point
fd_{φ}	Fraction of faults that originated in phase f
$\bar{t}_{fp,\varphi}$	Mean effort necessary to develop a function point in phase f
$\{\mu_R(t)\}_{\varphi}$	Expected change in fault count due to 1 repair in the life cycle phase φ
φ	A life cycle phase
$\{N_{fixed}\}_{\varphi}$	Number of requested repairs that are fixed in the life cycle phase φ
$\{N_{rr}\}_{\varphi}$	Number of repairs requested in the life cycle phase φ
$z_a(t)$	The intensity function of per-fault detection
r	Fault-detection rate
x	Fault-detection efficiency
t_{fp}	Effort necessary to develop a function point

t_0	t at which the considered phase originates
$\{f_{in}\}_j$	Date at which type j faults are introduced into a system
$\{f_{out}\}_{j,\varphi}$	Date at which type j faults are removed from a system
$\Delta_{j,\varphi}$	Number of type j faults (critical and significant) removed during phase f
$\{\mu_u(t_{begin})\}_{j,\varphi}$	Expected number of type j faults at the beginning of phase f
$\{\mu_u(t_{end})\}_{j,\varphi}$	Expected number of type j faults at the end of phase f
$\{FDN_{j,\varphi}\}_{per\ fault}$	Fault-days number per fault of type j removed during phase f
$\{f_{out}\}_{remain}$	Removal date of faults remaining in the delivered source code
$\{t_{end}\}_{TE}$	Ending date of testing phase, which is the last phase in the software development life cycle of the APP system
$\{FDN_{j\,remain}\}_{perfault}$	Fault-days number per fault of type j remaining in the delivered source code
$N_{j,remain}$	Number of type j faults (critical and significant) remaining in the delivered source code
$FDN(t+\Delta t)$	The fault-days number at time $t+\Delta t$
$FDN(t)$	The fault-days number at time t
$v(t)\mu_u(t)$	Estimate of fault introduction rate
$z_a(t)$	Intensity function of per-fault detection
$\mu_R(t)$	Expected change in fault count due to each repair
$\mu_u(t)$	Expected fault count at time t
FDN_A	The apparent fault-days number
$\gamma(t; v, z_a, \mu_R, \mu_H)$	A function of v, z_a, μ_R, and μ_H which relates FDN_A to FDN
FDN	The exact fault-days number
$N_{FDN,total}$	Total number of delivered faults in APP estimated using the FDN measure
$p_s(FDN)$	Reliability estimation for the APP system using the FDN measure
K	Fault exposure ratio, in failures/fault
N_{FDN}	Number of defects in APP estimated using the FDN measure

$N_{FDN,critical}$	Number of delivered *critical defects* (severity 1) estimated using the FDN measure
$N_{FDN,significant}$	Number of delivered *significant defects* (severity 2) estimated using the FDN measure
τ	Average execution-time-per-demand, in seconds/demand
T_L	Linear execution time, in seconds
$T_L(\mu p1)$	Linear execution time of Microprocessor 1 (µp1) of the APP system
$T_L(\mu p2)$	Linear execution time of Microprocessor 2 (µp2) of the APP system
$T_L(CP)$	Linear execution time of Communication Microprocessor (CP) of the APP system
$\tau(\mu p1)$	Average execution-time-per-demand of Microprocessor 1 (µp1) of the APP system
$\tau(\mu p2)$	Average execution-time-per-demand of Microprocessor 2 (µp2) of the APP system
$\tau(CP)$	Average execution-time-per-demand of Communication Microprocessor (CP) of the APP system

Chapter 14

$DDD(APP)$	The delivered defect density for the APP system, in defects/function point
$N_{FP,total}$	The number of total delivered defects for the APP system
$FP(APP)$	The function point count for the APP system
$p_s(FP)$	Reliability estimation for the APP system using the FP measure
K	Fault Exposure Ratio, in failure/defect
N_{FP}	Number of defects estimated using the FP measure
τ	Average execution-time-per-demand, in seconds/demand
T_L	Linear execution time of a system, in second
$N_{FP,critical}$	Number of delivered *critical defects* (severity 1)
$N_{FP,significant}$	Number of delivered *significant defects* (severity 2
$T_L(\mu p1)$	Linear execution time of Microprocessor 1 (µp1) of the APP system

$T_L(\mu p2)$	Linear execution time of Microprocessor 2 (μp2) of the APP system
$T_L(CP)$	Linear execution time of Communication Microprocessor (CP) of the APP system
$\tau(\mu p1)$	Average execution-time-per-demand of Microprocessor 1 (μp1) of the APP system
$\tau(\mu p2)$	Average execution-time-per-demand of Microprocessor 2 (μp2) of the APP system
$\tau(CP)$	Average execution-time-per-demand of Communication Microprocessor (CP) of the APP system

Chapter 15

$REVL$	Measure of requirements Evolution and Volatility Factor
$SIZE_{changed\ due\ to\ REVL}$	Size of changed source code corresponding to requirements specification change requests, in Kilo Line of Code (KLOC)
$SIZE_{delivered}$	Size of the delivered source code, in KLOC
$SIZE_{added\ due\ to\ REVL}$	Size of added source code corresponding to requirements specification change requests, in KLOC
$SIZE_{deleted\ due\ to\ REVL}$	Size of deleted source code corresponding to requirements specification change requests, in KLOC
$SIZE_{modified\ due\ to\ REVL}$	Size of modified source code corresponding to requirements specification change requests, in KLOC
N	Number of faults remaining in the delivered source code
$SIZE$	Size of the delivered source code in terms of LOC
SLI	Success likelihood index of a software product
$p_s(REVL)$	Reliability estimation for the APP system based on REVL
K	Fault exposure ratio, in failure/defect
N_{REVL}	Number of defects estimated based on REVL
τ	Average execution-time-per-demand, in seconds/demand
T_L	Linear execution time of a system, in second
$T_L(\mu p1)$	Linear execution time of Microprocessor 1 (μp1) of the APP system

$T_L(\mu p2)$	Linear execution time of Microprocessor 2 (µp2) of the APP system
$T_L(CP)$	Linear execution time of Communication Microprocessor (CP) of the APP system
$\tau(\mu p1)$	Average execution-time-per-demand of Microprocessor 1 (µp1) of the APP system
$\tau(\mu p2)$	Average execution-time-per-demand of Microprocessor 2 (µp2) of the APP system
$\tau(CP)$	Average execution-time-per-demand of Communication Microprocessor (CP) of the APP system

Chapter 16

RT	The value of the measure requirements traceability
R_1	The number of requirements met by the architecture
R_2	The number of original requirements

Chapter 17

C_1	The value of the test coverage
R_1	The number of requirements implemented
R_2	The number of requirements that should have been implemented
R_R	The number of requirements that should be implemented plus the number of requirements that were added
LOC_{tested}	The number of lines of code that are being executed by the test data listed in the test plan
LOC_{total}	The total number of lines of code
N_C	The number of cycles given by the simulation environment
f	The frequency of µp2 (16 MHz)
N_0	The number of defects found by test cases provided in the test plan
C_0	The defect coverage
λ	Failure intensity

K	Value of the fault exposure ratio during the n-th execution
T_L	The linear execution time
N	The number of defects remaining in the software
n	The average execution-time-per-demand
τ	The number of demands
p_f	The probability of failure-per-demand corresponding to the known defects

Chapter 18

r	Failures
T	Years
$\hat{\lambda}$	Failure rate
τ	Average execution time per trial
$\hat{\lambda}_{trial}$	Failure rate per trial

Chapter 19

K	Fault Exposure Ratio
M	The total number of modules
S_i	The number of lines of code (LOC) for each module
N_{found}	The number of known defects found by inspection and testing
T_L	Linear execution time
τ	The average execution-time/demand
SL	Severity Level
N_{CEG}	The number and locations of defects found by the CEG measure
OP	Operational Profile

$P(i)$	The propagation probability for the *i*-th defect
$I(i)$	The infection probability for the *i*-th defect
$E(i)$	The execution probability for the *i*-th defect
N_{CMM}	The number of defects estimated by the CMM measure
N_{COM}	The number and locations of defects found by the COM measure
$P_i(t)$	The probability that the system remains in the *i*-th reliable state
A	The size of the delivered source code in terms of LOC
k	A universal constant
F	A universal constant
SLI_{CC}	The Success Likelihood Index for the CC measure
N_{DD}	The number and locations of defects found by the DD measure
N_{FDN}	The number of defects estimated by the FDN measure
N_{FP}	The number of defects estimated by the FP measure
SLI_{RSCR}	The Success Likelihood Index for the RSCR measure
N_{RT}	The number and locations of defects found by the RT measure
vK	Fault exposure ratio
N_0	The number and locations of defects found by testing in an earlier version of code
a_0, a_1, a_2	Coefficients
C_1	Test coverage
$\widehat{N}_i$	The *i*-th defect population size estimator
D	The number of distinct defects found by t inspectors
f_1	The number of defects found by exactly one inspector
t	The number of inspectors
n_j	The number of defects found by the *j*-th inspector

f_k	The number of defects found by exactly k inspectors
$\rho_{(RePS)}$	The inaccuracy ratio for a particular RePS
P_f	The probability of failure-per-demand from the reliability testing or operational data
$P^*_{f(RePS)}$	The probability of failure-per-demand predicted by the particular RePS
$p_s(real)$	The probability of success-per-demand obtained from reliability testing
$p_s(est)$	The probability of success-per-demand obtained from the RePS
$SL(FP)$	Severity level as a function of function point count
FP	Function point count
$SIZE$	Size of the delivered source code in terms of LOC
f_i	Failure likelihood used for SLI_1 calculations
$H(x)$	The Heaviside step function, where $H(x) = 0, x < 0$ and $H(x) = 1, x \geq 0$
$\delta(t)$	The Dirac delta function, where $\int_{-\infty}^{x} \delta(t)dt = H(x)$
a_i	The lower boundary of level i
b_i	The upper boundary of level i
CC_j	CC of module j
a_{ACAT}	The slope of $f_{FP}(FP, ACAT)$
b_{ACAT}	The intercept of $f_{FP}(FP, ACAT)$
$ACAT$	An index that specifies the category of the application
a	A variable obtained from the curve-fitting process
$SIZE_{changed}$	Size of changed source code
$SIZE_{delivered}$	Size of delivered source code
L_i	The location of a defect
Ty_i	The type of a defect
N_A	The number of faults identified in the last version of development code

γ	A function of v, z_a, μ_R, and μ_H which relates FDN_A to FDN
d	The number of distinct defects found by t inspectors
FDN_A	The apparent fault-days number
C_0	The defect coverage
R_1	The number of requirements implemented
R_R	The number of requirements that should be implemented plus the number of requirements that were added
LOC_{tested}	The number of lines of code that are being executed by the test data listed in the test plan
LOC_{total}	The total number of lines of code

Appendix A

Σ	The set of software input variables; these variables cross the boundary of the application
Γ	The set of software output variables; these variables cross the boundary of the application
S	A finite, non-empty set of states; a state usually corresponds to the real-world condition of the system
T	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	The set of predicates, the truth value of the predicates is attached to the relevant transition
V	The set of variables defined and used within the boundary of the application
OP	The set of probabilities of the input variables
f_1	The first explicitly documented function; logically the first function of the software system
$S_i(f_j)$	The starting state of f_j
$S_0(f_j)$	The ending state of f_j
iv	The set of input variables
p	The set of predicates

ov	The set of output variables
$V_{S_i(f_j)}$	The variables stored in $S_i(f_j)$
$V_{S_0(f_j)}$	The variables stored in $S_0(f_j)$
nu	Variables neither used as the input variables nor used as the predicates of f_1 that remain the same and are part of the variables stored in the output
t_j	A transition; the set of the function f_j and its associated predicates
d	The bulleted function-level defects
y	An internal variable that is a function of x
x	Variable from the set Σ whose OP is known either by analyzing operational data or by searching in databases
f	A complex function used to analytically calculate the variable y, given x
p_f	The probability of failure
p_{path_i}	The probability of execution of the i-th path with defect(s)

NRC FORM 335
(12-2010)
NRCMD 3.7

U.S. NUCLEAR REGULATORY COMMISSION

BIBLIOGRAPHIC DATA SHEET

(See instructions on the reverse)

1. REPORT NUMBER
(Assigned by NRC, Add Vol., Supp., Rev., and Addendum Numbers, if any.)

NUREG/CR-7042

2. TITLE AND SUBTITLE

A Large Scale Validation of a Methodology for Assessing Software Reliability

3. DATE REPORT PUBLISHED

MONTH	YEAR
July	2011

4. FIN OR GRANT NUMBER

N6878

5. AUTHOR(S)

C. S. Smidts, Y. Shi, M. Li, W. Kong, J. Dai

6. TYPE OF REPORT

Technical

7. PERIOD COVERED *(Inclusive Dates)*

8. PERFORMING ORGANIZATION - NAME AND ADDRESS *(If NRC, provide Division, Office or Region, U.S. Nuclear Regulatory Commission, and mailing address; if contractor, provide name and mailing address.)*

Reliability and Risk Laboratory
Nuclear Engineering Program
The Ohio State University
Columbus, Ohio

9. SPONSORING ORGANIZATION - NAME AND ADDRESS *(If NRC, type "Same as above"; if contractor, provide NRC Division, Office or Region, U.S. Nuclear Regulatory Commission, and mailing address.)*

Division of Engineering
Office of Nuclear Regulatory Research
U.S. Nuclear Regulatory Commission
Washington, DC 20555-0001

10. SUPPLEMENTARY NOTES

R. Shaffer, S. Arndt, N. Carte, and M. Waterman, Project Managers

11. ABSTRACT *(200 words or less)*

This report summarizes the results of a research program to validate a method for predicting software reliability using software quality metrics. The method is termed the Reliability Prediction System (RePS). The RePS methodology was initially validated on a small control system application with a set of five software quality metrics. The effort described in this report is a validation of the RePS methodology using twelve software quality metrics.

The application used to validate the RePS methodology was based on a safety-related digital module typical of what might be used in a nuclear power plant. The module contained both discrete and high-level analog input and output circuits that read signals and produced outputs for actuating system equipment, controlling processes, or providing alarms and indications. The transfer functions performed between the inputs and outputs were dependent on the software installed in the application.

The twelve RePS software quality metrics are ranked based on their prediction capabilities. The rankings are compared with those obtained through an expert opinion elicitation effort and with those obtained through the small scale validation effort. The research provides evidence that the twelve metrics used in the RePS methodology can be used to predict software reliability in safety-critical applications.

12. KEY WORDS/DESCRIPTORS *(List words or phrases that will assist researchers in locating the report.)*

RePS, software reliability, software quality, software metrics

13. AVAILABILITY STATEMENT

unlimited

14. SECURITY CLASSIFICATION

(This Page)
unclassified

(This Report)
unclassified

15. NUMBER OF PAGES

16. PRICE

Federal Recycling Program

UNITED STATES
NUCLEAR REGULATORY COMMISSION
WASHINGTON, DC 20555-0001

OFFICIAL BUSINESS

NUREG/CR-7042

A LARGE SCALE VALIDATION OF A METHODOLOGY FOR ASSESSING SOFTWARE RELIABILITY

JULY 2011