NUREG/GR-0019
UMD-RE-2000-23

Software Engineering Measures for Predicting Software Reliability in Safety Critical Digital Systems

University of Maryland

U.S. Nuclear Regulatory Commission
Office of Nuclear Regulatory Research
Washington, DC 20555-0001

NUREG/GR-0019
UMD-RE-2000-23

Software Engineering Measures for Predicting Software Reliability in Safety Critical Digital Systems

Manuscript Completed: October 2000
Date Published: November 2000

Prepared by
C. Smidts, M. Li

University of Maryland
College Park, MD 20742

R. Brill, NRC Project Manager

Prepared for
Division of Engineering Technology
Office of Nuclear Regulatory Research
U.S. Nuclear Regulatory Commission
Washington, DC 20555-0001
NRC Job Code K6007

ABSTRACT

This report presents the University of Maryland (UMD) research to identify measures and families for the prediction and assessment of the reliability of software-based digital systems.

A set of software engineering measures from which the potential reliability of a digital I&C system can be predicted is developed from a set of 30 pre-selected software engineering measures. These measures are derived from a pool of 78 software engineering measures identified by Lawrence Livermore National Laboratory (LLNL). The concepts of structural classification, software development life-cycle classification, and family are presented. These 30 measures are categorized using these concepts. The concept of RPS and an extended structural representation are introduced to bridge the gap between software engineering measures and reliability. Expert opinion is elicited as the input in ranking the pre-selected 30 measures in terms of software reliability prediction. 10 missing measures are identified and ranked. The potential impact of these 10 missing measures on the ranking of the pre-selected 30 measures is analyzed. The top-ranked measures and families are presented in this report. Use of the families of measures in each software development phase can lead to a quantitative prediction of software reliability.

This study is the first step towards a systematic approach predicting the reliability of a real-time I&C software using RPSs established from the top-ranked measures and families. However, current knowledge prevents the quantitative estimation of the accuracy of such prediction. Further experiments are required to investigate the quantitative reliability as a function of the RPS measures.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

EXECUTIVE SUMMARY

This report summarizes the results of a Cooperative Research Agreement between the University of Maryland (UMD) and the Nuclear Regulatory Commission (NRC) to identify quantitative measures for the prediction and assessment of the reliability of software-based digital systems.

UMD based its study on previous research carried out by Lawrence Livermore National Laboratory (LLNL). In that study, LLNL identified a pool of 78 software engineering measures related to software reliability and established a set of software engineering ranking criteria used to rank assess the measures potential as software reliability indicators. The 78 measures were evaluated by two members of the LLNL research staff.

UMD scrutinized the ranking criteria, their corresponding levels and the ranking procedure used in the LLNL study. One ranking criterion was discarded and several ranking criteria were revised. The set of 78 measures was reduced by eliminating several (5) software reliability models which were mistakenly considered as measures. This set was then further reduced to 30 using importance considerations. These resulting 30 software engineering measures constitute the basis of the UMD study.

Realizing that the LLNL study did not examine the nature of the relationships between software engineering measures and software reliability prediction, UMD investigated the existence of a bridge between these notions. In particular, the concept of software Reliability Prediction System (RPS) was introduced. A RPS is a complete set of software engineering measures from which software reliability can be predicted. The key question is **"What RPS is best for predicting the reliability of a real-time I&C software?"** However, directly answering this question is impractical to date. The simplified question, "what are the best software engineering measures candidates for a RPS?" was found to be an appropriate substitute to this complex issue. This is the question being answered in this report.

A three-dimensional classification important to the analysis of the measures was introduced in this study. The axes of classification are structural[1], life-cycle based, and semantic. Structural classification, along with its graphical representation, assists in the establishment of RPSs. Life-cycle based classification helps define whether or not a measure is applicable in a particular software development phase. Semantic classification leads to the introduction of the concept of family. Families contain measures that estimate the same quantity using alternate means of evaluation. This concept helps further identify the relationships among measures. It also significantly improves the stability of the results presented in this study. Although new measures will appear as the development of software engineering techniques continues, the number of families will not significantly vary.

Expert opinion was elicited as input to the UMD ranking process. A number of field experts were selected from the nuclear and aerospace domain. They covered the following areas of knowledge: software development, software engineering, software engineering measurement, software reliability engineering, software reliability modeling, software safety, digital I&C design. A questionnaire was designed to pool expert opinion. The experts were then convened in a workshop where they summarized their evaluations of the measures and provided feedback on the ranking methodology being used.

Multi-attribute utility theory[2] was then used to rank the experts' inputs. The aggregated results were analyzed. The top-ranked measures were identified phase per phase. The families were also ranked and the top-ranked families identified.

[1] Structural classification establishes the relative position of a software engineering measure on a scale that ranges from physical reality to an indicator used for decision-making, namely reliability.

[2] A theory that aggregates the multiple attributes of interest into a scalar, which guides decision-making.

For purposes of this study, the software development is categorized into the following phases: requirements, design, implementation, testing, and operation. The aim is to predict the reliability of the operational phase. The other four phases are the periods during which software engineering measures are gathered. The top-ranked families and measures are listed below.

During the requirements phase, the top-ranked families are "Fault detected per unit of size", "Requirements specification change requests", and "Error distribution". The top-ranked measures are "Fault density", "Requirements specification change requests", and "Error distribution".

During the design phase, the top-ranked families are "Fault detected per unit of size", "Module structural complexity", and "Time taken to detect and remove faults". The top 3 measures are "Design defect density", "Fault density", and "Cyclomatic complexity".

During the implementation phase, the top-ranked families are "Fault detected per unit of size", "Module structural complexity", and "Time taken to detect and remove faults". The top-ranked measures are "Code defect density", "Design defect density", and "Cyclomatic complexity".

During the test phase, the top-ranked families are "Failure rate", "Fault detected per unit of size", and "Module structural complexity". The top-ranked measures are "Failure rate", "Code defect density", and "Mean time to failure".

This study is the first step towards a systematic approach predicting the reliability of real-time I&C software using RPSs established from the top-ranked measures and families. However, current knowledge prevents the quantitative estimation of the accuracy of such prediction. Further experiments are required to investigate the quantitative reliability as a function of the RPS measures.

A sensitivity analysis was performed. The sensitivity analysis proves that the results obtained remain valid for a wide spectrum of aggregation schemes.

Ten measures which have been omitted by LLNL were suggested by the experts. They reflect the advances of software engineering. They cover the fault-tolerant computing environment, the mutation testing technique, the object-oriented development method, and one adaptation of "Function point". The ranking criteria levels of these 10 measures were assessed by UMD research team members using: (1) the fact that analogies between measures existed; (2) the software engineering literature; (3) field expert inputs. The aggregated rates[3] were calculated. The impact of the 10 measures on the ranking of the 30 pre-selected measures was analyzed. The analysis shows that the OO measures do not significantly influence the rankings. On the other hand, the "Coverage factor" and "Mutation score" play more important roles.

This study is the first step towards a systematic approach predicting the reliability of a real-time I&C software using RPSs established from the top-ranked measures and families. However, current knowledge prevents the quantitative estimation of the accuracy of such prediction. Further experiments are required to investigate the quantitative reliability as a function of the RPS measures.

[3] An aggregated rate (also called measure's rate, or rate) is a real value ranging from 0 to 1. The rate is an indicator of the measure's capability to predict software reliability. The higher the aggregated rate value, the more capable the measure is of predicting software reliability.

ACKNOWLEGEMENTS

We wish to acknowledge the support and interest of the U.S. Nuclear Regulatory Commission (NRC) Office of Research and in particular, that of Mr. Robert W. Brill, the project manager. Mr. Brill helped refine the research direction, contributed in the identification of experts, as well as in the organization of the workshop on software engineering measures. Finally we should thank him for his persistent editorial review. Mr. John Calvert, Joel Kramer and Terry Jackson are to be thanked for the feed-back they provided throughout this work.

We owe special thanks to the contributions of Sandra George, Administrative Assistant in the Department of Materials and Nuclear Engineering, University of Maryland (UMD), College Park. Ms. George was our point-of-contact with the experts, the workshop facilities and the hotel facilities. She handled single-handedly all practical aspects of the workshop and we would have been at a loss without her.

Also to be thanked is Mr. Gary Johnson of Lawrence Livermore National Laboratory for answering questions on related research conducted under his direction.

We are fortunate to have been able to work on this project with a talented and dedicated team of UMD researchers consisting of Nihal Kececi, S. Swaminathan, Jenny Li, Yu Wang, Yining Wang, Bin Li, and Zhihong Qin. Special thanks are presented to them for their help and the support they provided to this project.

Special thanks to the following experts for their completion of the questionnaire and additional input during the expert panel:

Dr. A. Abran, Software Engineering Management Research Laboratory, Canada.

Dr. David Card, Software Productivity Consortium, for the introduction of structural classification.

Dr. William Everett, SPRE, Inc

Dr. Jon Hagar, Lockheed Martin Astronautics.

Dr. Herb Hecht, SoHar Inc.

Dr. Watts Humphrey, Software Engineering Institute, Carnegie Mellon University.

Dr. Jean-Claude Laprie, LAAS-CNRS, France.

Dr. Michael Lyu, Chinese University of Hong Kong.

Dr. Allen Nikora, Autonomy and Control Section at JPL

Mr. Bill Petrick, Capri Technology, Inc.

LIST OF ACRONYMS

FFP	Full Function Point
FPA	Function Point Analysis
IEEE	Institute of Electrical and Electronics Engineers
JM	Jelinski Moranda model
KLOC	Kilo-LOC
LCOM	Lack of Cohesion of Methods
LOC	Line of Code
LLNL	Lawrence Livermore National Laboratory
NOC	Number of Children
NRC	Nuclear Regulatory Commission
OO	Object-Oriented
PDL	Pseudo Design Language
PRA	Probabilistic Risk Analysis
RPS	Reliability Prediction System
UMD	University of Maryland
WMC	Weighted Method per Class

CHAPTER 1 INTRODUCTION AND SUMMARY

1.1 Research Objective

The objective of this study is to identify a set of software engineering measures from which the potential reliability of a digital I&C system can be predicted. This study commences a long-term research effort for developing a method to obtain a quantitative estimate of the reliability of a digital system. For the purposes of the project, *reliability* is defined to be the probability that the digital system will successfully perform its intended safety function (for all conditions under which it is expected to respond) upon demand with no unintended functions that might affect system safety.

The ultimate purpose of the method developed is:

- To provide a set of information sufficient and necessary to estimate reliability along with its associated uncertainty that could be used in probabilistic risk assessment (PRA).

- To provide information that can be used to supplement existing review methods in an assessment of software-based digital systems which have high degree of safety requirements.

1.2 Statement of the Problem

Software-based digital systems are progressively replacing analog systems in safety-critical applications like nuclear power plants. However the ability to predict their failure rate is still not well understood and requires further study.

The modern software-based digital systems are composed of hardware devices and software embedded in the hardware. Reliability prediction for digital systems must account for the failure of hardware, the failure of software, and effects of hardware and software failures on each other and the effect that such failures have upon the function they are intended to perform.

Hardware reliability calculation usually is predicted upon the assumption that the device design is fundamentally correct. Failures are assumed to be caused by random variability in the physical stresses seen by the device and by variability in the devices' ability to withstand these stresses. Consequently, the failure rate of a hardware device can be calculated from the failures observed in similar devices subject to similar stresses.

Unlike hardware, software is a logical set of instructions, not a physical, component of a system. Its success or failure is not affected by physical stress, which is the primary cause of hardware failure. Failures of the software stem mostly from fundamental errors in the design that cause the system to fail under certain combinations of system states and input trajectories. Current qualitative methods for assuring reliability are process oriented. There is no universally accepted qualitative methods for assuring that safety critical software is free of defects to a desired reliability level.

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

However, each group of models has its inherent flaws when applying them to safety critical real-time systems. Safety critical real-time systems are characterized by the following features:

1. The probability of failure needs to be less than 10^{-6} per one year of execution[1] [Butl93].

2. The input rate is really fast, generally at the speed of 100 inputs per second.

Given such characteristics of safety critical real-time systems, applying reliability growth models is infeasible because of the exorbitant amounts of testing that would be required [Butl93].

The traditional input domain model, Nelson model [Nels78], is as simple as a point estimate of failure rate based on the number of failures and number of total test cases. According to Butler and Finelli [Butl93], this model needs exorbitant amounts of testing for safety critical real-time systems. Nelson and other researchers introduced the concept of equivalence class which significantly reduces the amount of testing required. These models started with a problematic assumption that the input domain can be thoroughly identified and classified into equivalence classes. Extreme caution should be exercised when applying this model to real-time safety critical systems: the inputs of such a system are generally infinite and unpredictable. A mechanism which is not yet established is required to guarantee the exhaustive classification of such inputs.

The structural models are widely used in fault-tolerant systems [Duga95] [Scot87]. The failure rate (or transition rate) from the normal state to the abnormal state (or vice versa) is assumed to be available. However, how to estimate this parameter is not known. In essence, this rate is the failure rate of a sub-system in the fault-tolerant system. The estimation of the rate requires the failure data to be available.

Several early prediction models exist [Gaff88] [RADC92] [Stut98]. The Gaffney model is based on the assumption that the size of the system in LOC is available (or predictable) at the time the prediction is made. Then the number of discovered faults is given by an empirical relationship. Unfortunately there is still a long way to go from the number of discovered faults to reliability prediction. The RADC model derives reliability from copious data sources by means of some unexplainable empirical relationships. No research shows that this model is applicable to real-time safety critical systems.

Therefore a new research effort was initiated in this study to understand the relationship between the characteristics of the development process, the product itself, the operational environment and software reliability.

1.3 Background

One industry which requires high integrity safety critical software is the nuclear industry. The nuclear industry usually uses IEEE Std 7-4.3.2-1993, "Standard Criteria for Digital Computers in Safety Systems of Nuclear Power Generating Stations." While the Nuclear Regulatory Commission (NRC) endorsed this standard in Regulatory Guide 1.152, Revision 1 (January, 1996), it did not endorse Section 5.15, "Reliability" as a sole means of meeting the Commission's regulations for reliability of digital equipment used in safety systems. The applicable Section 5.15 of the standard states "when qualitative or quantitative reliability goals are required, the proof of meeting the goals shall include software used with hardware." The NRC staff did not endorse that section because there is no general agreement that a measurement methodology currently exists that provides a credible method to measure software reliability.

[1] The experts claim that a failure probability lesser than 10^{-3} can not be measured in practice.

The aircraft industry standard for software is RTCA/DO-178B, "Software Considerations in Airborne Systems and Equipment Certification." It states in paragraph 12.3.4 Software Reliability methods "that currently available methods do not provide results in which confidence can be placed to the level required for this purpose." Hence this document does not provide guidance for software error rates.

During the last several years, both the NRC and the nuclear industry have recognized that PRA analysis has evolved to the point where it can be used as a tool for assisting in regulatory decision-making. In 1995, the NRC adopted a policy regarding expanded NRC use of PRA. Following publication of the Commission policy, the Commission directed the NRC staff to develop a regulatory framework that incorporates risk insights. Recently, the NRC staff has developed risk-informed regulatory guides to meet this directive. PRAs require a value in terms of failure rate per demand for the digital system to perform its intended function. The lack of a credible measurement methodology that assesses this value precludes meaningful consideration of digital systems in the development of these risk insights. The development of objective estimates of digital system software reliability which could be used in place of subjective estimates, where feasible, in PRAs will help the NRC make better risk-informed decisions.

1.4 Approach

The research presented in this report is a continuation of the work performed by Lawrence Livermore National Laboratory (LLNL) *CASE Tools for Software Reliability Measurement* [LLNL98]. In that study, LLNL reviewed worldwide literature to determine the state-of-the-art in measuring software reliability. 78 basic measurements were found to exist. LLNL then designed a set of ranking criteria in order to rank each of the measures[2]. The 78 measures were then ranked by two laboratory personnel.

Although LLNL identified and ranked 78 measures, the relationship between the top-ranked measures and software reliability had not been investigated. The University of Maryland (UMD) study started from the establishment of the theoretical bridge between software engineering measures and software reliability. The concept "software reliability prediction system (RPS)"[3] was introduced for this purpose.

The key question is **"What RPS is best for predicting the reliability of a real-time I&C software?"** However, directly answering this question is impractical to date. The simplified question, "what are the best software engineering measures candidates for a RPS?" was found to be an appropriate substitute to this complex issue. This is the question being answered in this report.

Using the 78 Lawrence Livermore National Laboratory measures as starting point UMD reduced it to thirty using structural considerations (such as the fact that software reliability models are not measures and should not be part of such a study) as well as importance considerations. A group of experts was convened in a workshop where they summarized their evaluations of the measures and provided feedback on the ranking methodology being used. The ranking of these 30 measures was then performed based upon the input provided by the group of experts. A sensitivity analysis was performed to verify the validity of the aggregation approach. During the workshop, the experts described and recommended a set of measures to be added to the pool of thirty measures selected. Ten missing measures were then rated by researchers at UMD. The rates of these missing measures were then aggregated with the initial thirty, and the final set of top-ranked measures were identified.

[2] In this report the term "measures" refers to software engineering measures. By measure we mean the degree to which a software system, component, or process possesses a given software attribute. For instance, the measure "Line of Code (LOC)" assesses the physical size of a code, the measure "Function Point" evaluates the functional size of a system.

[3] The RPS is a complete set of software engineering measures which can be used to predict software reliability.

1.5 Contents of This Report

Chapter 2 investigates the relationships between measures and reliability prediction. It introduces three axes of classification important to the analysis of the measures. These axes are structural, life-cycle based and, semantic. Chapter 2 also defines the concept of a Software Reliability Prediction System, which is a complete set of measures by which software reliability can be predicted.

Chapter 3 presents the methodology used to rank the pre-selected set of 30 software engineering measures discussed in Section 1.4. The methodology involves the elicitation of expert opinion regarding the scores of software engineering measures. The scoring is performed with respect to seven ranking criteria: *Credibility, Repeatability, Cost, Benefit, Experience, Validation, and Relevance to Reliability,* and in terms of letter grades. A letter-conversion scheme translates the letter values to real numbers between 0 and 1. These numbers are then aggregated using an aggregation equation and a weighting scheme for the seven ranking criteria. The aggregated number serves as the indicator of the "goodness" of the measure. A sensitivity analysis is further performed on all components of the analysis, i.e. letter-real conversion scheme, aggregation function form, weighting scheme, since, *a priori*, one can not assess which aggregation scheme is correct. The sensitivity analysis proves that the results obtained remain valid for a wide spectrum of aggregation schemes. Results of aggregation rates, rankings, and sensitivity analysis for the thirty pre-selected measures are presented in Chapter 4. Some potential inconsistencies are also examined and explained.

The discussion provided in Chapter 5 is designed to incorporate the latest new measures generated by the advances of software engineering into the study. The missing measures discussed in Chapter 5 were identified by experts. The measures cover the fault-tolerant computing environment, the mutation testing technique, the object-oriented development method, and one adaptation of "Function point". The ranking criteria levels were assessed by UMD research team members using: the fact that analogies between measures existed, the software engineering literature, and field expert inputs. The aggregation rates were calculated by applying the aggregation theory discussed in Chapter 3.

Chapter 6 provides a summary of our research and discusses future research.

Appendix A provides descriptions for the forty software engineering measures used in this study. This includes the description of the pre-selected 30 measures and the description of the 10 missing measures.

Appendix B presents the questionnaire used to elicit expert opinion.

Appendix C presents the aggregation results and sensitivity analysis for the 30 pre-selected measures.

Appendix D provides the input data for the missing measures.

In Appendix E a glossary of terms used in this report is provided.

1.6 Final Results

Forty software engineering measures were ranked in terms of their capability to predict software reliability. The ranking was performed phase by phase due to the fact that this capability varies in different phases. In addition, some measures applicable to non-object-oriented systems are not applicable to object-oriented (OO) systems. This leads to separate rankings performed for both OO and non-OO systems. The top-ranked measures are provided in the following:

Table 1-1 and Table 1-2 provide the top 3 measures, phase by phase, for non-object-oriented systems and object-oriented systems, respectively. These top-ranked measures are the possible roots of a complete set of measures from which software reliability can be predicted.

Requirements	Design	Implementation	Testing
Fault density	Design defect density	Code defect density	Failure rate
Requirements specification change requests	Cyclomatic complexity	Design defect density	Code defect density
Error distribution	Fault density	Cyclomatic complexity	Coverage factor

Table 1-1 Top 3 Measures per Phase for non-OO Systems

Requirements	Design	Implementation	Testing
Fault density	Design defect density	Code defect density	Failure rate
Requirements specification change requests	Fault density	Design defect density	Code defect density
Error distribution	Fault-days number	Fault density	Coverage factor

Table 1-2 Top 3 Measures per Phase for OO Systems

The concept of *family* is introduced in this study to reflect a many-to-one relationship between measures and primary attributes, such as functional size of a system, complexity of a piece of code, etc. The introduction of this concept reduces the 40 measures to 20 families for non-OO systems and 22 for OO systems. Although new measures would appear as the development of software engineering techniques, the number of families might not significantly vary. Even more encouraging is the fact that the ranking of the families is more stable than that of the measures during different software development phases.

The top 3 families for non-OO and OO systems are provided in Table 1-3 and Table 1-4 below.

Requirements	Design	Implementation	Testing
Fault detected per unit of size	Fault detected per unit of size	Fault detected per unit of size	Failure Rate
Requirements specification change requests	Module structural complexity	Module structural complexity	Fault Detected per Unit of Size
Error distribution	Time taken to detect and remove faults	Test adequacy	Fault-tolerant Coverage Factor

Table 1-3 Top 3 Families per Phase for non-OO Systems

Requirements	Design	Implementation	Testing
Fault detected per unit of size	Fault detected per unit of size	Fault detected per unit of size	Failure rate
Requirements specification change requests	Time taken to detect and remove faults	Test adequacy	Fault detected per unit of size
Error distribution	Requirements specification change requests	Time taken to detect and remove faults	Fault-tolerant coverage factor

Table 1-4 Top 3 Families per Phase for OO Systems

1.7 Significance

Currently the NRC's Standard Review Plan, Chapter 7 and BTP-14 and aircraft industry standard for software (RTCA/DO) do not contain quantitative acceptance reliability requirements. This is because there has been no technical basis for utilizing the measures currently in use by industry. University of Maryland's work provides a technical basis for utilizing measures currently in use.

This study recommends a set of software engineering measures to the software industry for better management and quality control of the software development process. In particular, this study provides the theoretical basis to support the selection of such a set of measures. Also the results of this study might help the reviewers to better understand the results of an inspection of the digital system.

Information has been identified to perform reliability prediction in the early stages of the development. This prediction along with the measures can signal potential problems in the development. According to such information, either the quality controller warns the development team to improve the quality of the development, or the decision-maker decides to terminate the development.

Another significant contribution of the study is the introduction of the concept of software reliability prediction system (RPS). This concept helps identifying support measures that need to be collected if one wants to obtain a credible reliability prediction.

This study is the first step towards a systematic approach predicting the reliability of a real-time I&C software using RPSs established from the top-ranked measures and families. However, current knowledge prevents the quantitative estimation of the accuracy of such prediction. Further experiments are required to investigate the quantitative reliability as a function of the RPS measures.

[Butl93] R. W. Butler, G. B. Finelli, The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software, *IEEE Transactions on Software Engineering*, vol. 19, no. 1, Jan., 1993.

[Nels78] E. Nelson, Estimating Software Reliability from Test Data, *Microelectronics and Reliability,* Vol 17, pp. 67 – 74, Pergamon, New York, 1978.

[Duga95] J.B.Dugan, M.Lyu. Dependability Modeling for Fault Tolerant Software and Systems, *Software Fault Tolerance*, John Wiley & Sons Ltd., 1995.

[Scot87] R.K. Scott, J. W. Gault, D. F. McAllister, Fault-Tolerant Software Reliability Modeling, *IEEE Transactions on Software Engineering*, vol. SE-13, No.5, May 1987.

[Gaff88] J. E. Gaffney, C. F. Davis, An approach to estimating software errors and availability, *SPC-TR-88-077*, version 1.0 March 1988.

[RADC92] Rome Laboratory (RL), *Methodology for Software Reliability Prediction ad Assessment,* Technical Report RL-TR-92-52, Vol. 1 and 2, 1992.

[Stut98] M. Stutzke, C. Smidts, A Stochastic Model of Fault Introduction and Removal during Software Development, *Probabilistic Safety Assessment and Management – PSAM'4,* Vol 1, Springer, pp 1111 – 1116, 1998.

[LLNL98] J. D. Lawrence, et al., *Assessment of Software Reliability Measurement Methods for Use in Probabilistic Risk Assessment,* FESSP, Lawrence Livermore National Laboratory. 1998.

CHAPTER 2 ON THE NATURE OF RELATIONSHIPS BETWEEN MEASURES AND RELIABILITY

Software engineering measures are essential not only to good software engineering practice, but also for the thorough understanding of software failure behaviors and reliability prediction [Fent97]. Software engineering measures address multiple aspects of the software development process and of the product itself. For instance, one finds measures associated with estimation and/or prediction of cost and schedule of software development, measures involving organizations, staff, number of lines in a software module, logical complexity of a module, etc.

Software development organizations typically elect to select a small number of such software engineering measures to manage, predict, and assess the quality of their development processes and products. The purpose of this research is to determine whether these measures are suitable for the prediction of software reliability, and if so, to what extent. In other words, can these measures be used as reliability predictors?

The scientific literature reveals that limited research efforts have been undertaken to answer this question [RL92] [Khos90] [LLNL98] [Evan99] [Smid98][Stut98]. However, the research has not yet reached maturity and more research is definitely required in this domain [Lyu95]. The research presented in this report builds on these prior efforts.

To assess whether a software engineering measure can serve as a reliability predictor it is necessary to assess both the intrinsic characteristics of the measure (how good the measure is in itself, the measure's purpose, how much it costs) as well as its extrinsic characteristics (how it relates to reliability). In this chapter, a classification scheme is introduced which helps classify measures structurally, with respect to the software development life-cycle and semantically. Classification of a measure is an important step in the analysis of a measure because it helps in more objectively assessing the intrinsic and extrinsic characteristics of the measure and also because it helps understand relationships between various measures. Three axes of classification are introduced: 1) structural, 2) life-cycle based and 3) semantic. The chapter begins with the definition and explanation of the structural classification (Section 2.1). Deriving the particular class to which a measure belongs involves an analysis process leading to a graphical representation of the structure of the measure. The "Gaffney estimate of the bugs per Line of Code"[1] is given as an example. Section 2.2 is dedicated to the life-cycle based classification. Section 2.3 discusses semantic classification and the notion of families.

Section 2.4 presents an extension of the structural representation to capture the layer that is missing between the software engineering measure and reliability. This more detailed representation depicts the direct tie between the measure and reliability.

From this extension, one derives the notion of a system of measures (Section2.5) and the general problem of finding a system of measures which could help predict reliability is introduced. The problem of finding the degree to which a measure is a valid indicator of reliability is shown to be a simplified version of the general problem. This simplified problem is solved in later chapters using elicited expert opinion.

2.1 Structural classification

David Card advocated an information model in [ISO15939] that defines the increasing structural levels of measures. A revised information model entitled as "structural classification" is presented in this section. Structural classification establishes the relative position of a software engineering measure on a scale that goes from physical reality to an indicator used for decision-making, namely reliability. More precisely, Figure 2-1 shows how the software engineering measure is derived from data (the physical reality) using typically simple mathematical operations. While building the chart, a determination is made as to whether the measure belongs to the following

[1] A detailed description of the measure is given in Appendix A.

structural levels: primitive measure, derived measure or indicator. Figure 2-1 depicts these different levels. Attributes (see Figure 2-1) are not measures, however they delimit the structural scale at one end and take root in physical reality. Indicators are measures and define the other end of the scale, namely reliability[2]. A detailed explanation of this terminology is given in the remainder of this paragraph. The classification is such that a primitive measure is "further away" from the indicator than a derived measure. Hence the classification helps evaluate the "conceptual distance" between the measure and the indicator. The classification also attempts to characterize the transformations used to derive the measure, i.e. a determination is made whether rules, models or algorithms are used to derive the measure. Such information is important since models are based on assumptions which may or may not be valid whereas rules or algorithms are not based on assumptions and should therefore always be valid. Hence the nature of the transformation provides a direct indication as to the extent to which the measure is valid[3].

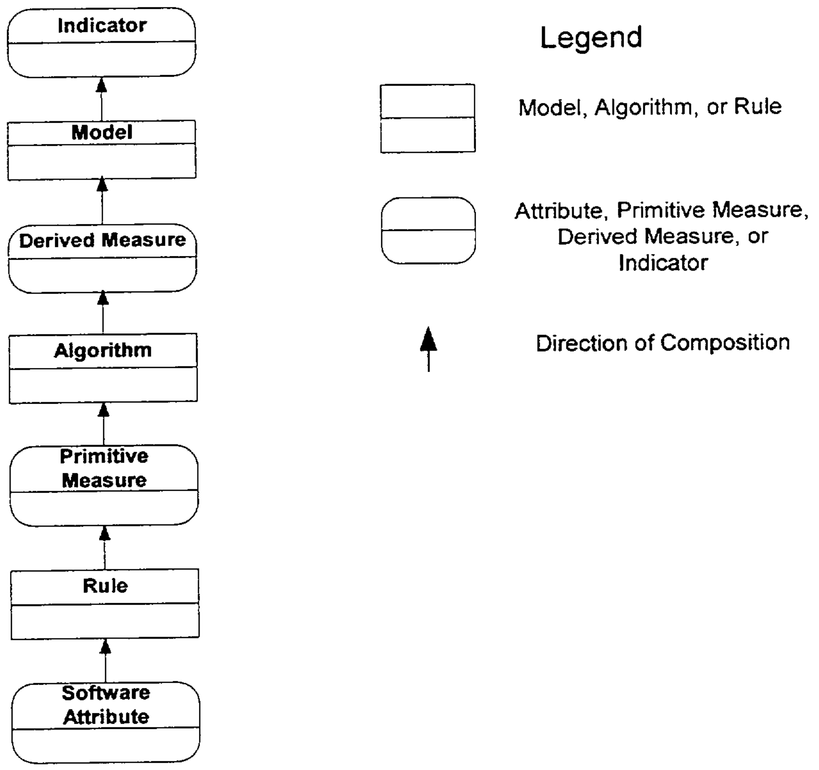


Figure 2-1 Structural Representation of a Measure

2.1.1 Definitions

The classification uses the following seven terms: indicators, derived measures, primitive measures, software attributes, models, algorithms and rules. Each term is defined in turn in the remainder of this section.

Indicators are estimates or evaluations that provide a basis for decision-making. In this particular study, reliability is deemed an appropriate indicator for decision making. Measures structurally less complex than reliability, and which when combined with other measures or parameters, can yield reliability estimates are deemed unfit to be indicators.

[2] Reliability is defined as the probability of successfully performing the safety function on demand with no unintended functions that might affect safety.

[3] By valid the authors mean how useful a measure is in predicting reliability.

Derived measures are any intermediate values which are neither indicators nor primitive measures.

Primitive measures are values resulting from the application of rules to software attributes.

Software attributes are properties of the software. Software attributes are not measures and hence do not constitute a structural level in the measurement framework. But they are needed to ascertain the source of the data, i.e. requirements documents or code as well as the type of data, for example, failures.

To go from one structural level[4] to another, the concepts of models, algorithms and rules are needed. Models, rules, and algorithms are typically simple transformations which allow the combination of several lower level measures to create a hierarchically higher measure.

Models are procedures for combining measures to produce an estimate or evaluation based on a series of assumptions. Each assumption is an idealization of reality. The procedure is logically deduced from the assumptions.

An *Algorithm* is a straightforward procedure for combining two or more measures. The output of the algorithm represents one or more characteristics of the software product under study.

A *Rule* is a mapping of the attribute to a subset of the field of real or integer numbers.

To illustrate these concepts, the next paragraph provides an example of a primitive measure, a derived measure, and an indicator.

2.1.2 Examples

2.1.2.1 Example of Primitive Measure

An example of a primitive measure is the Line Of Code (LOC) measure. A Line Of Code (LOC)[5] [Fent97] is any line that is not a comment or blank line regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements. According to this definition, LOCs are calculated through application of a simple counting rule to the source code. The software attribute here is "code size" of the physical entity "source code statements".

2.1.2.2 Example of Derived Measure

Derived measures are intermediary levels between primitive measures and indicators. As an example, consider Cyclomatic Complexity, V(G). This measure is defined as follows:

$$V(G) = e - n + 2p \qquad \text{Equation 2-1}$$

where

n is the number of nodes, i.e., the number of sequential groups of program statements,

e is the number of edges, i.e., the number of program flows between nodes,

p is the number of connected components in the control flowgraph established from the source code or PDL[6] of the software.

[4] A structural level is any of the following: software attribute, primitive measure, derived measure or indicator.

[5] There are multiple competing definitions of Lines of Code. The definition selected is probably the most commonly accepted interpretation of Line of Code.

[6] PDL stands for pseudo design language. This language can be used to specify the detailed design of software.

Variables *n, e, p* are primitive measures that can be obtained directly from the program's control flowgraph. V(G) is a derived measure obtained by application of the algorithm defined above.

2.1.2.3 Example of Indicator

The reliability estimate produced by the Jelinski and Moranda (JM) model can be transformed into an indicator. The assumptions on which the model is based are:

1. The product under study has N faults at the beginning of testing.
2. All failures are similar. In other words, all failures contribute in the same way to the final software reliability estimation.
3. All failure occurrences are independent.
4. The time interval between the i^{th} and $(i+1)^{th}$ failure, t_i, follows the exponential distribution:

$$f(t_i) = \delta_i \exp(-\delta_i t_i) \qquad \text{Equation 2-2}$$

where

f is the failure density function.

5. The failure intensity (rate) δ_i is taken proportional to the number of faults remaining in the program.

$$\delta_i = k(N - i) \qquad \text{Equation 2-3}$$

where

k is a constant of proportionality.

6. Each time a failure is observed, the corresponding fault is removed immediately. This assumes that faults are independent of each other and equally likely to be detected.
7. N, k can be estimated by the Maximum Likelihood (ML) method. Predictions are made by substitution of these ML estimates into the appropriate model equations.

Reliability at time t is then given by:

$$R(t) = e^{-\int_0 f(\tau)d\tau} \qquad \text{Equation 2-4}$$

The model directly produces a reliability figure dependent on time. However , the indicator defined for this project is a probability of success per demand. A strict adherence to the definition of the indicator requires a transformation of the time dependent function into a probability of success on demand. Such a transformation involves a conversion factor. The conversion factor is the number of safety related demands per unit of time; ρ.

The number of safety related demands n over time t is given by:

$$n = \rho \cdot t \qquad \text{Equation 2-5}$$

And reliability at time t given that the last failure was at time t_i, or reliability at demand n given that the last failure was at demand n_i is given by:

$$R(t \mid t_i) = R(n \mid n_i) = e^{-\sum_{m=n_i}^{n} \frac{k(N-i)}{\rho}} = e^{-\frac{k(N-i)}{\rho}(n-n_i)} \qquad \text{Equation 2-6}$$

Hence the probability of a successful demand of the safety function is constant between failures and given by

$$p(n \mid n_i) = p(n_i) = e^{\frac{k(N-i)}{\rho}}$$ Equation 2-7

2.1.3 Structural Representation and Analysis

The analysis of a measure's description such as, for instance, a description abstracted from [IEEE88] can help identify the structural level to which a measure belongs. As support to such analysis, the representation given in Figure 2-1 can be used. An illustration of this graphical technique on the Gaffney estimate of Bugs per Line of Code[7] is given in Figure 2-3.

The Gaffney estimate of bugs per line of code is meant to give a crude estimate of the number of faults per lines of code. The estimate is based on code size. More specifically, if there are *N* modules in a program, one estimates the number of faults, F_i, in each of the *N* modules using the following formula $F_i = 4.2 + 0.0015\, S_i^{4/3}$. The number of faults in the complete program F can be estimated as the sum of all F_i's. The formula used to derive F_i is clearly an empirical model (a correlation) based on a series of non documented assumptions. A question which the analysis of the measure and, in particular, the nature of the mathematical transformations (i.e., model) used to obtain the bug per line of code estimate raises, is the questions of validity, how well validated is the model, what is its range of applicability, etc.

To obtain F_i, the measure "S_i", number of executable source statements in the module *i*, is required. In this case "S_i" is measured directly from the existing code in the module *i*. In other words, "S_i" is a primitive measure directly related to a physical entity through a rule. The rule explains how the line count should be made. The physical entity in question, or software attribute, is the size of the existing code. An analysis of the measure description presented in Appendix A shows that the rule is not specified. Since multiple approaches for line count have been documented in the software engineering literature, multiple interpretations of the rule can thus occur. This of course may present a problem : different companies may use different interpretations of the rule and if one does not pay attention to this particular issue, incorrect conclusions can be drawn. This concern is highlighted in the structural representation (Figure 2-3) and constitutes a part of the analysis which can be made during the construction of the structural representation. The same discussion applies to the second attribute, namely, the number of modules in the code.

In conclusion, the structural analysis determines whether a measure is a primitive measure, a derived measure or an indicator. Such classification provides a preliminary assessment of a "conceptual distance" between the measure and the indicator. The structural analysis also determines whether models, algorithms or rules are involved in the transformations applied to the software attributes to yield the measure. This information is important since it gives a preliminary feel of the extent of the validity of a measure. Through the analysis process, preciseness of the definitions of the different concepts involved (for example, rules) is examined. The potential multiple interpretations of such concepts might arise if these concepts have not been precisely defined. From this knowledge, one is able to assess the degree to which a measure is repeatable. Finally, the structural representation helps identify the cost of a measure since it clearly establishes the software attributes it builds upon and hence defines the data that should be collected.

[7] For a detailed description of the measure see Appendix A.

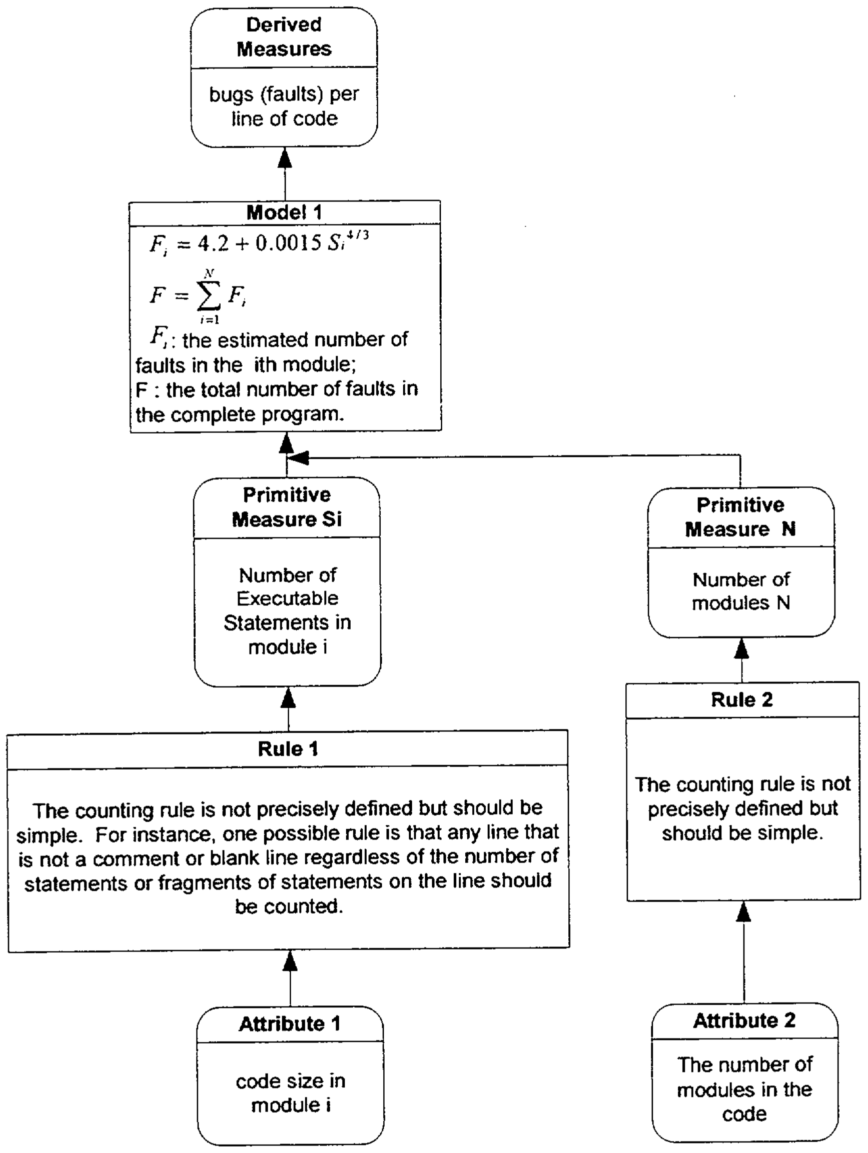

Derived Measures
bugs (faults) per line of code
Model 1
$F_i = 4.2 + 0.0015\, S_i^{4/3}$
$F = \sum_{i=1}^{N} F_i$
F_i: the estimated number of faults in the ith module;
F : the total number of faults in the complete program.
Primitive Measure Si
Number of Executable Statements in module i
Primitive Measure N
Number of modules N
Rule 1
The counting rule is not precisely defined but should be simple. For instance, one possible rule is that any line that is not a comment or blank line regardless of the number of statements or fragments of statements on the line should be counted.
Rule 2
The counting rule is not precisely defined but should be simple.
Attribute 1
code size in module i
Attribute 2
The number of modules in the code

Figure 2-2 Structural Representation and Analysis of the Gaffney Estimate of Bugs per Line of Code

2.2 Lifecycle-based classification

The second classification of interest is entitled "life-cycle coverage". This classification is used to describe to which phases of the life-cycle the measure applies.

2.2.1 Lifecycle Definition

In this study, the life-cycle of software development is represented by the following five phases: requirements, design, implementation, testing, and operation (see Figure 2-4). These are the typical high-level phases that can be found in the development of most software. Note however that the number of software life-cycle phases considered by different software development organizations tends to vary. The [IEEE610] considers eight different phases. Table 2-1 is a mapping of the phases used in this study to the [IEEE610] phases.

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

Table 2-1 Lifecycle in this Study versus Recommended Life-cycle in [IEEE610].

It should also be noted that an inherent assumption of the study is that the software described follows a waterfall life-cycle [Scha93]. A waterfall life-cycle is typically characterized by the succession of the phases from requirements to operation without too many backwards steps such as for instance the fact of going back from design to requirements. Other software development lifecycles exist such as for instance the spiral model [Boeh88], a life-cycle where development is driven by perceived risk areas and the resolution of these risks in an iterative fashion. Spiral development makes heavy use of prototyping and is typically used for software with a strong user interface component. Waterfall development on the other hand is recommended for programs with strong algorithmic component such as the software used in safety applications. However, a case for the development of safety-critical applications using Spiral development can probably be made for the real-time component of the software.

To extend the study presented to a Spiral development, one should consider adding a prototyping phase (for which it is unclear if measures exist and would be recorded in any consistent fashion) and one would need to account for the repetition of the phases such as requirements during the iterative cycles of development and the incompleteness of the products and artifacts produced.

2.2.2 Classification

A life-cycle phase can, in practice, be characterized by the process followed and by the life-cycle artifact produced. As an example, the requirement phase uses a process such as requirements review and produces a software requirements specification, the artifact. All measures can fall into at least one of the five waterfall development model phases, some measures can fall into more than one phase.

A measure falls either into the process domain or into the artifact domain. The question is why would such classification be of interest. What does it indicate? First of all, one needs to remember that the indicator of interest is the value of reliability in operation. Reliability is a property of the product and not of the process. Hence a measure of the process will tend to be "further away" from the objective of the research than a measure of the product. Furthermore, a measure of a product or of an artifact that belongs to a phase distant from the operation phase will be less appealing to the analyst than a measure which is closer to the operation phase. Consequently, analyzing a measure with respect to the life-cycle produces valuable information about the relationship that exists between the measure and reliability.

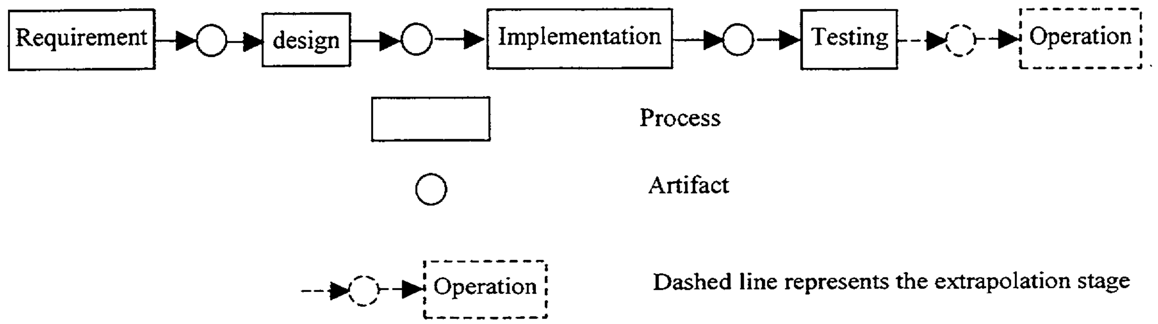


Figure 2-3 Life-Cycle Coverage

2.3 Semantic Classification and the Concept of Family

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" (Please refer to Appendix A). 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 [Alb79] [Jone86] [Jone91].

The implications of the grouping of measures into families will be examined in detail in Chapter 4. Suffice it to say that the concept of family of measures is more robust than that of single measure.

2.4 Extended Structural Representation

In Section 2.1.3, a structural representation of the "bugs per line of code" was presented. Figure 2-5 is an expansion of Figure 2-1, Structural Representation of a measure, which adds other measures to bridge the gap between a measure and reliability, the indicator. This extended representation provides a visualization of a measure's relevance to reliability, the cost entailed in bridging the gap between reliability and the measure, the degree of subjectivity involved, etc. The following example clearly illustrates this expansion.

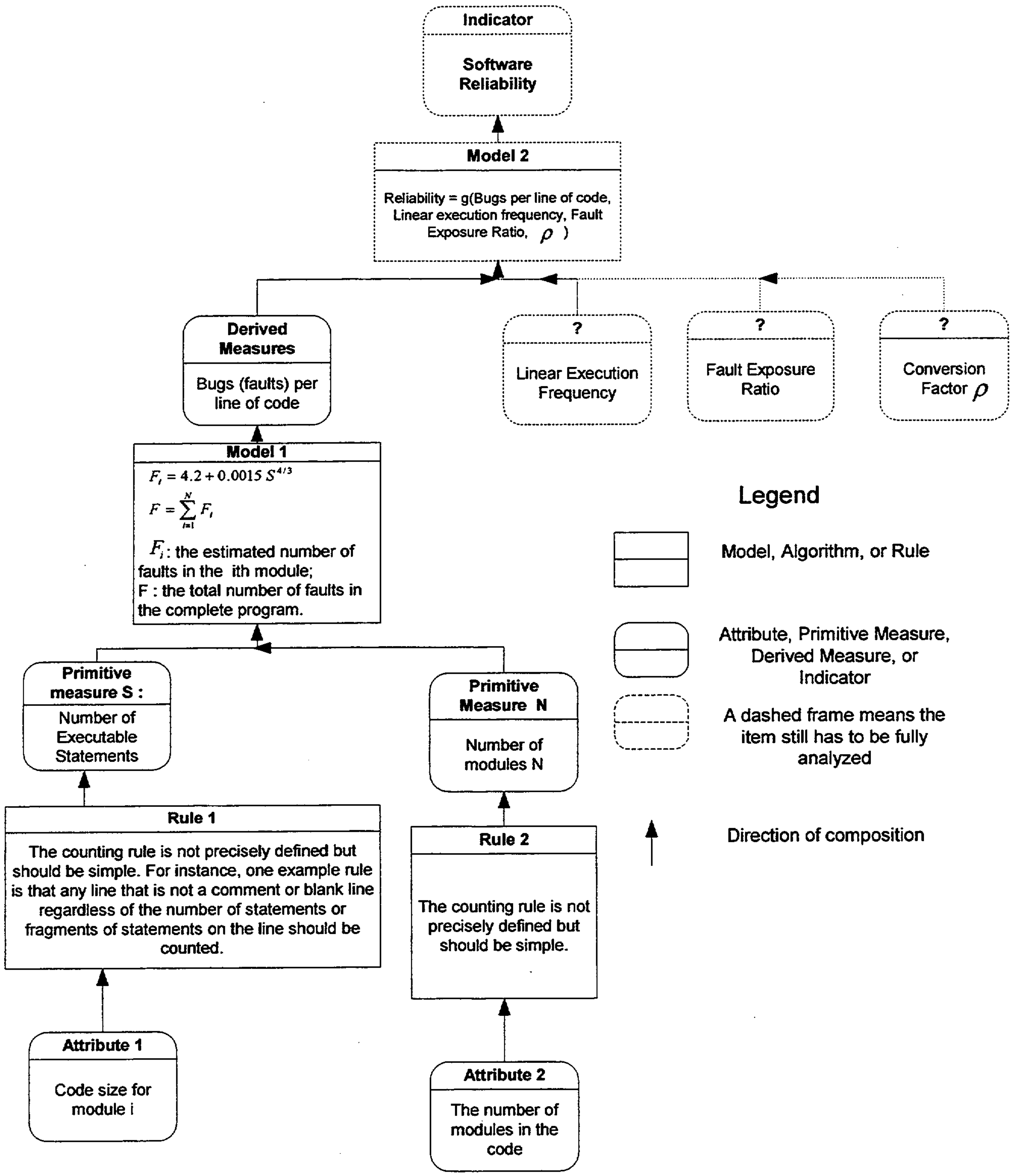

Indicator
Software Reliability
Model 2
Reliability = g(Bugs per line of code, Linear execution frequency, Fault Exposure Ratio, ρ)
Derived Measures
Bugs (faults) per line of code
?
Linear Execution Frequency
?
Fault Exposure Ratio
?
Conversion Factor ρ
Model 1
$F_i = 4.2 + 0.0015\, S^{4/3}$
$F = \sum_{i=1}^{N} F_i$
F_i: the estimated number of faults in the ith module;
F : the total number of faults in the complete program.
Legend
Model, Algorithm, or Rule
Attribute, Primitive Measure, Derived Measure, or Indicator
A dashed frame means the item still has to be fully analyzed
Direction of composition
Primitive measure S :
Number of Executable Statements
Primitive Measure N
Number of modules N
Rule 1
The counting rule is not precisely defined but should be simple. For instance, one example rule is that any line that is not a comment or blank line regardless of the number of statements or fragments of statements on the line should be counted.
Rule 2
The counting rule is not precisely defined but should be simple.
Attribute 1
Code size for module i
Attribute 2
The number of modules in the code

Figure 2-4 Structural Analysis of the Bug Per Line of Code (Gaffney Estimate) Measure

Figure 2-5's bottom-left part is the structural representation of the "bugs per line of code" and hence is an exact replica of Figure 2-3. The top part of Figure 2-5 on the other hand (displayed in dashed lines) was added to "connect" the derived measure "bugs per line of code" to the indicator of interest, reliability. Measures such as the "linear execution frequency[8]" or the "fault exposure ratio[9]" can be considered as support measures to the software engineering measure under study. The support measures could themselves be analyzed to determine whether they should be classified as derived measures or primitive measures. This would help understand their validity as well as the amount of effort (cost) involved in their evaluation. Such decomposition would parallel and complete the decomposition and analysis of the software engineering measure under study. Another element of the extended representation is function "g". This function has intentionally been left unspecified. In fact, several software reliability models can be used in place of "g". An example of such model is the "Musa Basic Execution model" [Musa87]. Function "g" is another contributor to the validity and cost of the prediction of reliability. If "g" has been extensively validated, and/or if an extensive body of experience exists then the prediction is more credible.

In conclusion, a reliability prediction for the software can be derived from the measures "bugs per line of code", "linear execution frequency" [Musa87] [Fent97], "fault exposure ratio" [Musa87], and "conversion factor ρ" (see Section 2.1.2.3). These four software engineering measures constitute a complete set (from the indicator's viewpoint), which in this document is named *Reliability Prediction System.*

2.5 Software Reliability Prediction System

A short discussion of the software reliability prediction system is necessary to fully appreciate the impact of the existence of such a set of measures.

First, as discussed earlier in this chapter, a point was made that software engineering companies typically select a few measures which will then be used to evaluate the quality of the software development process, make decisions to change the functionality developed, adjust the schedule, and field or not to field the product. The following questions must be answered in order to be able to predict the final product's reliability:

- Does the set of measures selected by the company contain at least one complete software reliability prediction system?
- If the selection was such that it does not, can one reconstruct such a system or is the data necessary unrecoverable? If the answer to both questions is negative, predicting software reliability is impossible.

Second, it is necessary to examine the relationships between measures in a software reliability prediction system. This consideration is used to validate the selection of the measures.

The issue relates to the existence of redundant measures in a set. For instance, assume $S_1(b, c, d)$ is a system, and b, c, d are measures in system S_1. If any measure among b, c, d can be derived from any other among (b, c, d) then this measure is redundant and can safely be eliminated from the set S_1. The software reliability prediction system thus obtained is minimal. Hence in a minimal system, measures must be independent[10] of one another. Note that adding redundant measures to a system is unnecessary. It artificially increases the data collection effort for no apparent gain.

Third, it is worthwhile mentioning the existence of reliability prediction systems that are hierarchically related to each other. These systems have the interesting characteristic of being equally predictive if the hierarchical relationship between them is unique. Consider the two systems depicted in Figure 2-6. The first, S_1, is composed of

[8] The linear execution frequency is the number of times the program would be executed per unit time if it had no branches or loops.

[9] The fault exposure ratio represents the fraction of time that the "passage" results in a failure.

[10] Independent here means that one measure can not be inferred from another. For example, the cohesion measure (see Appendix A) for a software can not be derived from the coupling measure for the same software, and vice versa.

measures *(a, b, c, d)*, the second S_2 is composed of measures *(a, β, c, γ, δ)*. The two systems are hierarchically related. Examining the two systems, one sees that *β* depends on *b*, and *γ* and *δ* depend on *d*. But the dependency is of a particular type: structural. Indeed, measure *b* is a parent[11] of *β*, and *d* is a parent of *γ* and *δ*. Other measures contained in S_1 and S_2 are identical. Since all measures in S_1 are parents or equals to measures in S_2 the systems are declared hierarchically related with S_1 a parent of S_2. This hierarchical relationship opens the door to one of the fallacies of classification of individual measures with respect to an indicator. Indeed, it is true that if one were to identify the measures which are the most relevant to reliability, one would find that measures *a, b ,c, d* are more relevant to reliability than measures *β, γ* and *δ*. However, interestingly enough, the set (*a, b, c, d*) and the set (*a, β, γ, c, δ, γ*) have exactly identical predictive value if there exists only one unique function to compose (*a, b, c, d*) from set (*a, β, γ, c, δ, γ*). Although this point is made very clear in Figure 2-6, it can be easily forgotten. This comment will explain some of the findings in Chapter 4.

[11] A measure a is a parent of a measure b if a is hierarchically closer to the indicator than b and :

1. either a relationship exists between *a* and *b* such that $a=g(b)$.
2. or support measures $\underline{b}$ exist such that $a=g(b, \underline{b})$

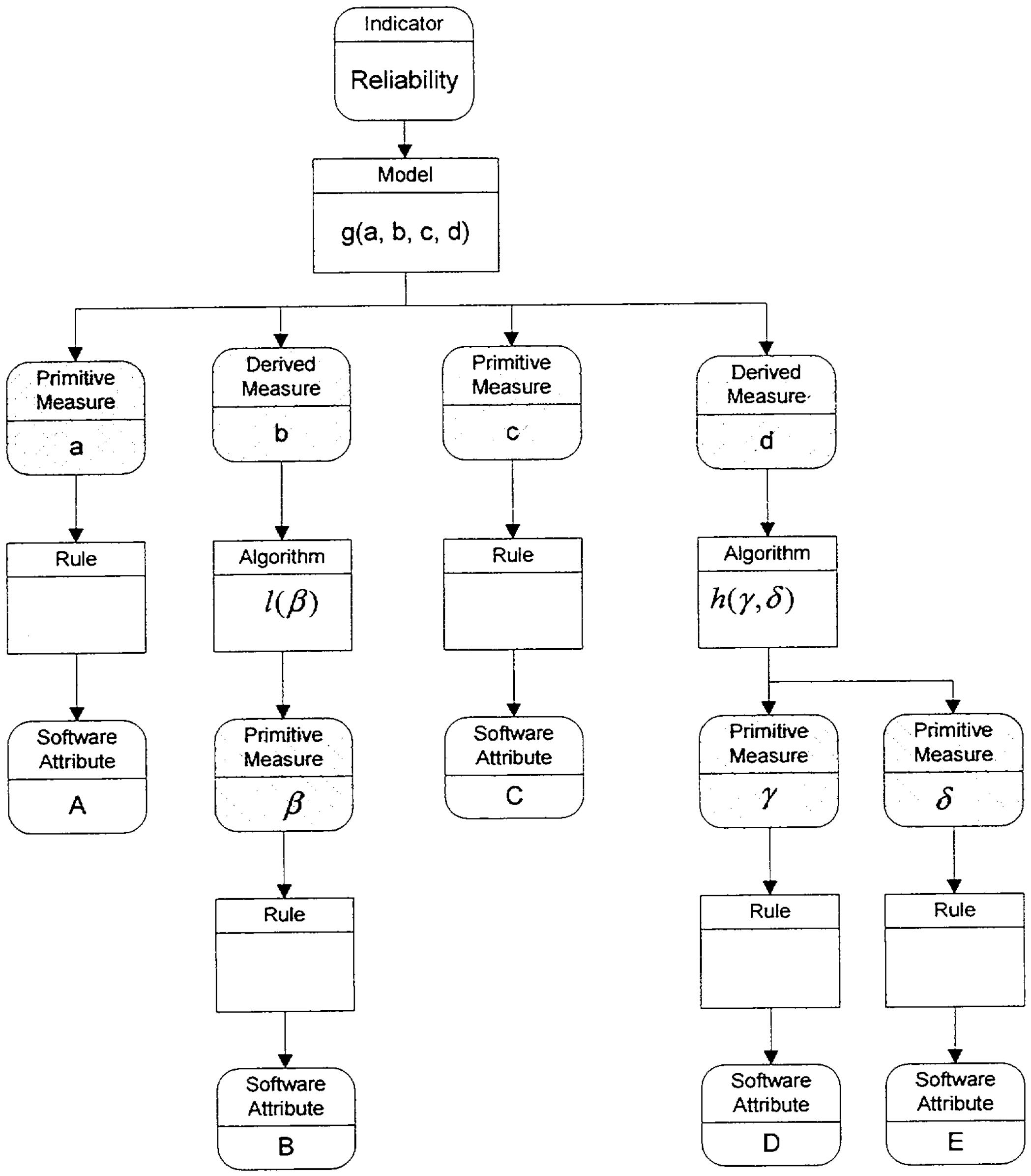


Figure 2-5 Hierarchical relationships between software reliability prediction systems

2.6 Selecting a Reliability Prediction System

The next step is to select a software reliability prediction system. To explain the elements at play in the selection process, Figure 2-6, an idealized version of Figure 2-5 which abstracts the various concepts found in the extended graphical representation, is used.

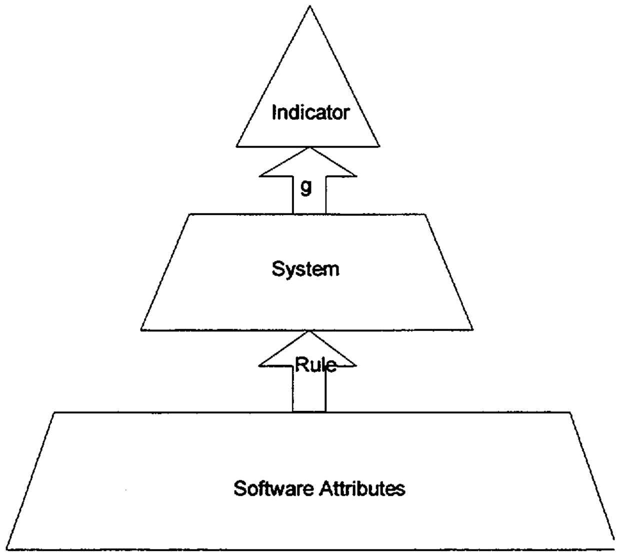


Figure 2-6 Extended Representation and Software Reliability Prediction System

The top triangle represents the indicator, reliability. The middle layer is the reliability prediction system, i.e., a system that contains the measures that can completely assess reliability. The bottom layer contains the attributes supporting the measures in the second layer. The arrows between layers represent the rules, algorithms and models required to establish the prediction system and the indicator's expression. In this context, reliability is then represented by:

$$R = g(S) \qquad \text{Equation 2-8}$$

where

R	Reliability
g	The model function
S	The system

The remainder of this section will examine how one can use Equation 2-8 and the elements of the extended graphical representation to rank a specific system with respect to other systems or to rank a specific combination *(S, g)* with respect to other combinations *(S', g')*.

To make a selection between systems a ranking framework is needed. The following section establishes the elements of a preliminary ranking framework.

2.6.1 A Ranking Framework for Software Reliability Prediction Systems

To establish a ranking framework, one need initially identify a preliminary set of criteria which define preference. This set is composed of the following criteria:

Cost	An estimation of how much would be spent in applying the system to predict reliability.
Benefit	The net gain that ensues from the use of the prediction system.
Validity	The degree of confidence in the accuracy of the reliability prediction.
Credibility	The degree to which the system supports the specified goals.
Experience	How widely the system has been used as a whole.
Repeatability	The degree of similarity of the results obtained by the repeated application of the system by the same or different people.

The ranking of a system can be obtained by aggregating evaluations of all criteria of the system. That is,

$$R(S) = \underset{g\in G(S)}{Max}\{f(C(S), B(S), V(S), E(S), Cr(S), Rp(S),\ C(g), B(g), V(g), E(g), Cr(g), Rp(g))\} \quad \text{Equation 2-9}$$

where

S	A specific system
$R(S)$	The ranking of the system S
$f()$	The aggregation function
$G(S)$	The set of software reliability models which can predict reliability based on the set of software engineering measures S
$C(S)$	The evaluation[12] of the cost criterion for the system of measures
$B(S)$	The evaluation of the benefit criterion for the system of measures
$V(S)$	The evaluation of the validity criterion for the system of measures
$E(S)$	The evaluation of the experience criterion for the system of measures
$Cr(S)$	The evaluation of the credibility criterion for the system of measures
$Rp(S)$	The evaluation of the repeatability criterion for the system of measures
$C(g)$	The evaluation of the cost criterion for function g
$B(g)$	The evaluation of the benefit criterion for function g
$V(g)$	The evaluation of the validity of function g
$E(g)$	The evaluation of the experience with function g
$Cr(g)$	The evaluation of the credibility of function g

[12] An evaluation is an assessment of the criterion on a scale. Typically the scale is discrete and counts only a few levels. For instance, the cost criterion could count three levels: high, medium and low. The role of the assessor is then to evaluate the cost incurred by the measure on this scale.

Rp(g)	The evaluation of the repeatability of function *g*

The benefit obtained through use of *g* should be identical for all function *g's* and all systems *S* since this is the benefit incurred from the prediction of reliability. Consequently one can remove the contribution *B(g)* in Equation 2-9. This yields:

$$R(S) = \underset{g \in G(S)}{Max}\{f(C(S), B(S), V(S), E(S), Cr(S), Rp(S), C(g), V(g), E(g), Cr(g), Rp(g))\} \quad \text{Equation 2-10}$$

If one is interested in ranking sets *(S, g)* instead of solely *S*, it can be done using the following expression:

$$R(S,g) = f(C(S), B(S), V(S), E(S), Cr(S), Rp(S), C(g), V(g), E(g), Cr(g), Rp(g)) \quad \text{Equation 2-11}$$

Where *R(S, g)* is now the rank of a particular combination of software engineering measures and software reliability model.

2.6.2 Ranking Based on the Measures Composing the Software Reliability Prediction System

If the values in Equation 2-10 are not directly available, one then can turn towards the measures that make up the software reliability prediction system and estimate the system values from the measure values. This ranking approach is introduced in this paragraph.

Let *C(m), B(m), V(m)* and *E(m)* be the evaluation of the cost, benefit, validity and experience criterion for a measure *m* of *S*. Then possible expressions of *C(S), B(S), V(S)* and *E(S)* are given by:

$$C(S) = \sum_{m \in s} C(m) \quad \text{Equation 2-12}$$

$$V(S) = \underset{m \in s}{Min}(V(m)) \quad \text{Equation 2-13}$$

$$B(S) = h(B(m)) \qquad \textit{for all } m \in S \quad \text{Equation 2-14}$$

$$E(S) = \underset{m \in s}{Min}(E(m)) \quad \text{Equation 2-15}$$

$$Cr(S) = \underset{m \in s}{Min}(Cr(m)) \quad \text{Equation 2-16}$$

$$Rp(S) = \underset{m \in s}{Min}(Rp(m)) \quad \text{Equation 2-17}$$

where

h()	An Under-determined benefit function over all measures in *S*

Each expression (Equation 2-5 to Equation 2-10) is justified in turn in the remainder of this paragraph.

The major cost contributor to a measure is the cost incurred by the data collection effort and training of the analyst collecting the data. The training effort relates to the understanding and mastering of the rules which allow the establishment of primitive measures. The mathematical transformations involved in the implementation of models and algorithms used in the realm of software engineering is certainly marginal compared to the expenditures related to data collection and training. Hence there is reason to believe that the cost function is additive as long as two measures do not share identical data (software attributes) or similar rules, a phenomenon which is rare in practice.

The nature of the benefit function is more difficult to identify and will be the object of future research.

As for validity, the reasoning follows the "weakest link principle", well known to system designers, i.e., it is reasonable to consider that the trust one puts in a system is only as good as the trust one puts in its weakest component. The same reasoning holds for the experience, credibility, and repeatability criteria.

2.7 Ranking Individual Measures

The previous paragraph establishes a framework for the ranking of a system of measures *S* as well as for the ranking of a combination *(S, g)* . In the previous sections of this chapter, we established that it is not possible to consider measures in isolation but that measures must be considered in sets named software reliability prediction systems. The ranking of such systems is an extremely difficult problem. It requires identification of measures which make up a system, determination of all systems, and identification of all possible functions *g*. This task is impractical at this stage because of the number of software engineering measures in existence today and the fact that the number of possible systems grows exponentially with the number of existing measures. Therefore, a more progressive approach needs to be taken. Consequently, a simplified problem is investigated in the remainder of this report. A solution to this simplified problem constitutes a first step in answering the more difficult problem posed by Equation 2-9 and Equation 2-10.

This simplified problem consists of ranking single measures instead of systems. The selection process begins by establishing the measures rankings with respect to ranking criteria establishing the intrinsic validity of the measure and its relevance to the determination of reliability (extrinsic validity[13]). Relevance is somewhat similar to the notion of distance between reliability and the measure. This integrates the distance as measured through a hierarchical decomposition, through an examination of life-cycle coverage and through a semantic mapping of the measure. The criteria selected will be explained in detail in chapter 3. But mainly, the ranking criteria will include *C(m), B(m), V(m),E(m), Rp(m),* and *Cr(m)* as well as relevance to reliability *Rel(m),* where *m* is the measure. Note that the extraction of this data is a step towards answering the question underlying Equation 2-9 and Equation 2-10 since *C(m), B(m),V(m), E(m), Rp(m),* and *Cr(m)* are an integral part of these equations. Relevance to reliability allows selection of the measures which are most closely related to reliability and which, as such, lead to the simplest reliability models. Figure 2-7 describes how the two problems are related.

[13] The term validity here means as an expression of the utility of the measure.

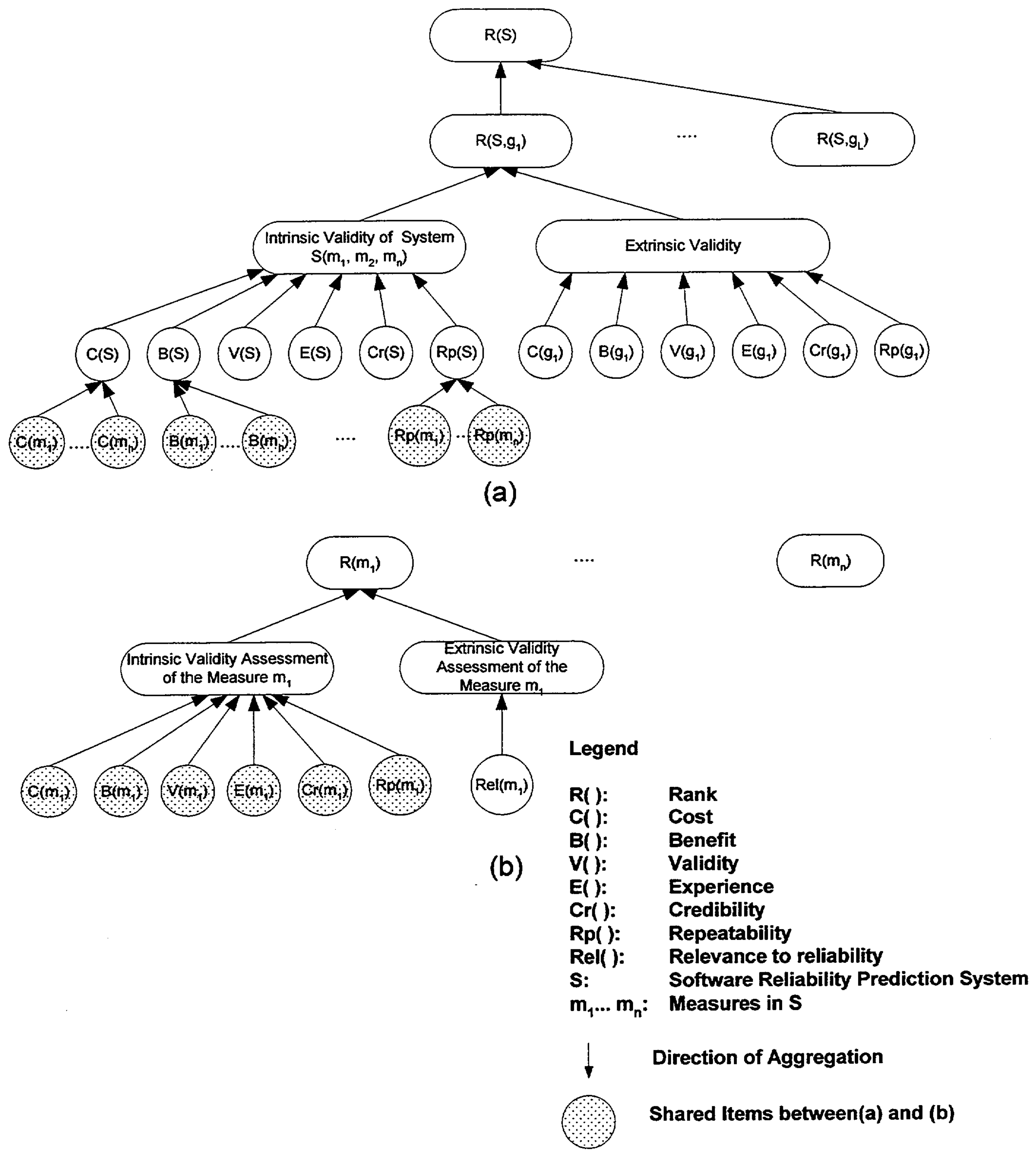


Figure 2-7 Ranking Problems: a) Ranking S, b) Problem being solved

2.8 Summary and Conclusions

This chapter investigated the relationships between measures and reliability prediction. It introduced three axes of classification important to the analysis of the measures. These axes are structural, life-cycle based and semantic. Semantic classification lead to the introduction of the concept of Family. A graphical method was described for the purpose of structural representation. The creation of this graphical representation was shown to be another valuable tool in the analysis of a software engineering measure.

The chapter also defined the concept of a Software Reliability Prediction System. The issue of selecting a software reliability prediction system was examined and a possible selection process suggested through Equation 2-8 to Equation 2-11.

The point was made that the selection of a software reliability prediction system is a difficult task and that a simpler but related problem should be examined first: the problem of selecting single software engineering measures of high degree of validity which would be most relevant to reliability. The criteria for selection of the measures contain relevance, cost, benefit, validity, experience, credibility, and repeatability. This set must be examined to determine whether it is complete. And furthermore, research must be carried out to determine the aggregation function *f*.

The chapter also showed that measures and interrelationships between measures need to be well understood before they are used. One should avoid the use of redundant measures and one should make sure that the set of measures at hand is complete from a software reliability prediction stand-point. Finally, once measures have been ranked separately, they need to be reinterpreted in the context of other measures.

[Fent97] Fenton, N. E., Pfleeger, S. L., *Software Metrics, A Rigorous & Practical Approach*, International Thomson Computer Press, 1997, 2nd Edition.

[RL92] Rome Laboratory (RL), *Methodology for Software reliability Prediction and Assessment,* Technical report RL-TR-92-52, volumes 1 and 2, 1992.

[Khos90] Khoshgoftaar, T., Munson, J., *Predicting Software Development Errors using Software complexity Metrics*, IEEE Journal on Selected Areas in Communications, Vol. 8, No. 2, Feb. 1990

[LLNL98] Lawrence, J. D., et al., *Assessment of Software Reliability Measurement Methods for Use in Probabilistic Risk Assessment*, Technical report UCRL-ID-136035, FESSP, Lawrence Livermore National Laboratory. 1998

[Evan99] Evanco, W. M., *Using a proportional hazards model to analyze software reliability*, STEP'99. Proceedings Ninth International Workshop Software Technology and Engineering Practice p. xvii+187, 134-41

[Smid98] Smidts, C., Stutzke, M., Stoddard, R. W., Software Reliability Modeling: An Approach to Early Reliability Prediction, *IEEE Transactions on Reliability* , Vol. 47, no.3, May 1998 September

[Stut98] Stutzke, M., Smidts, C., A Stochastic Model of Fault Introduction and Removal during Software Development, *Probabilistic Safety Assessment and Management – PSAM'4,* Vol 1, Springer, pp 1111 – 1116, 1998

[Lyu95] Lyu, M. R., editor, *Handbook of Software Reliability Engineering*, computing McGraw-Hill, 1995

[ISO15939] Emam, K., Card, D., *ISO/IEC Standard 15939, Software Measurement Process,* International Organization for Standardization, October, 1999 (draft).

[IEEE88] *IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software*, IEEE, 1988.

[IEEE610] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE, 1990.

[Scha93] Schach, S. R., *Software Engineering* 2nd Edition, Richard D. Irwin, Inc., and Aksen Associates, Inc., Boston, 1993.

[Boeh88] Boehm, B. W., "A Spiral Model of Software Development and Enhancement," *IEEE Computer* 21 (May 1988), pp. 61-72.

[Alb79] Albrecht, A. J., "Measuring application development", *Proceedings of IBM Applications Development Joint SHARE/GUIDE Symposium,* Monterey, CA, pp. 83-92, 1979

[Jone86] Jones, C., *Programming Productivity,* McGraw-Hill, Inc., 1986.

[Jone91] Jones, C., *Applied Software Measurement,* McGraw-Hill, Inc., 1991.

[Musa87] Musa, J. D., Iannino, A., Okumoto, K., *Software Reliability, Measurement, Prediction, Application,* McGraw-Hill Book Company, New York, 1987.

CHAPTER 3 RANKING METHODOLOGY

In Chapter 2, we investigated the relationships between software measures and reliability. The concept of a reliability prediction system (RPS) was introduced and it was shown that the problem of ranking reliability prediction systems is a difficult task. The simplified problem of identifying single measures to be used (per life-cycle phase) to characterize reliability was introduced as a preliminary step towards a resolution of the ranking of RPSs. In Chapter 3 we examine the process used to select single measures. Figure 3-1 depicts the problem solved. In Figure 3-1, the ranking of the measure is determined by its intrinsic[1] and extrinsic validity[2].

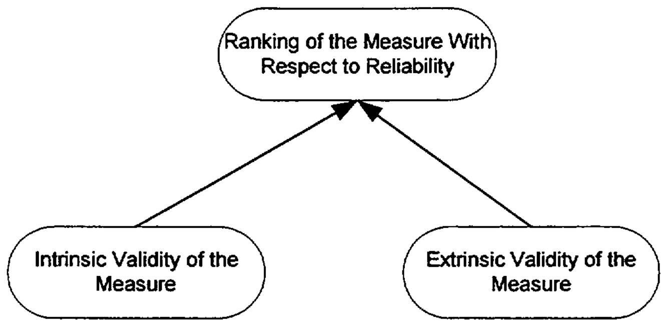

Ranking of the Measure With Respect to Reliability
Intrinsic Validity of the Measure
Extrinsic Validity of the Measure

Figure 3-1 Ranking of a Measure

The software measures used in this study were selected from a pool of software measures identified in an earlier Lawrence Livermore National Laboratory report (LLNL) [LLNL98]. This report contains a list of 78 measures believed to be relevant to software reliability prediction.

Most measures in the LLNL report were extracted from the IEEE standard [IEEE982] and the remainder from the recent software engineering literature. These measures were ranked by two resident LLNL experts. However, the ranking obtained was not peer reviewed. Under a separate contract, the University of Maryland organized a peer review of the measures. In the process, the University of Maryland identified the thirty top measures. These were derived by first conducting a thorough critical review of the measures themselves.

This review effort indicated a number of discrepancies which are listed below:

- The list intermingled software engineering measures and software reliability models. Software reliability models, however, are not measures *per se*. Rather they constitute a means of predicting reliability from measures. As such they have no place in an attempt of classifying and ranking software engineering measures and were eliminated from the list of potential candidate measures. The shortened list contained 73 measures.

- The list contained a number of duplicate measures which were removed.

- The list did not account in any particular manner for fault tolerant systems or for new technologies. This fact alerted us to the need to involve in the peer review effort experts knowledgeable in these

[1] The intrinsic validity depends how well a measure performs with respect to quality ranking criteria (defined later) and cost effectiveness ranking criteria (defined later).

[2] The extrinsic validity of the measure is defined by its degree of relevance to reliability.

areas which could supplement the measures provided. As a consequence we also introduced a mechanism by which experts could provide additional measures.

- The list did not account for support measures in a reliability prediction system such as operational profile. After a long discussion with the NRC representatives, it was decided that this support measure deserved special consideration and should be examined separately during discussions with the experts.

- Relationships between measures were not investigated; Understanding of the differences between ranking of a single measure and the larger problem of ranking of reliability prediction system were not addressed. This led the University of Maryland team to the writing of Chapter 2 of this report.

- Some measures presented little impact on reliability. These measures were removed from the list and thirty measures remained.

Apart from these apparent discrepancies, the list of measures provided withstood scrutiny and constituted a valuable preliminary set of measures.

This chapter provides a description of the methodology used in ranking the software engineering measures.

3.1 Overview of the Methodology

In chapter 2 we established that it is not possible to consider measures in isolation but that measures must be considered in sets named software reliability prediction systems. Rather than performing the extremely difficult task of ranking RPSs, we took a more progressive approach which identifies single measures that can be used (per life-cycle phase) to characterize reliability. An overview of this methodology is given in this chapter. The identification process begins by establishing the measure rankings with respect to ranking criteria establishing the intrinsic validity of the measure and its relevance to the determination of reliability (extrinsic validity). Then a questionnaire was designed to elicit expert opinion on rankings of the measures. Expert opinions were then aggregated into one single comparable number. Sensitivity analysis was performed to validate the aggregation framework.

The methodology followed during this research consisted of the eight steps shown in Figure 3-2. Each step is briefly described in the remainder of this paragraph. The details of each step are examined in Section 3.2 through Section 3.8. Section 3.10 provides a brief summary of the chapter and conclusions.

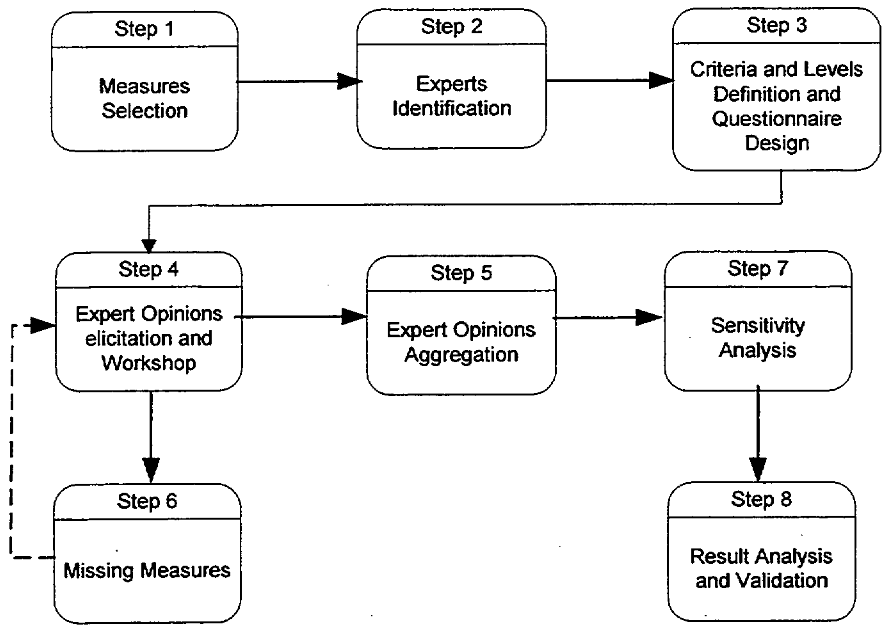

Step 1
Measures Selection
Step 2
Experts Identification
Step 3
Criteria and Levels Definition and Questionnaire Design
Step 4
Expert Opinions elicitation and Workshop
Step 5
Expert Opinions Aggregation
Step 7
Sensitivity Analysis
Step 6
Missing Measures
Step 8
Result Analysis and Validation

Figure 3-2 Steps followed in ranking the software engineering measures

Step 1: Measure selection. Measure selection is the first step in this eight-step methodology. Measure selection consisted in identifying measures that would serve in the analysis, in other words, measures that may be relevant to reliability and have a high ranking criteria level of intrinsic validity. The 30 measures used in this study were selected from the pool of measures identified in [LLNL98]. This set of thirty measures is the set used in the remainder of the study.

Step 2: Expert Identification. Quantitative information that would help support ranking does not exist in the current software engineering literature. This quantitative information may on the other hand exist in industry but is jealously protected through proprietary clauses. Consequently, reliance on expert opinion is currently the optimal approach to the problem of collecting ranking data. The expert selection process conducted in this study was based on ranking criteria such as research field, organization, background, engineering domain and occupation. Namely, the experts were selected from the software reliability, software safety, safety critical digital control communities, and from the aerospace and nuclear domains. A mix of practitioners, researchers, industry and government agencies was also achieved in the process.

Step 3: Ranking criteria and Ranking Criteria Levels[3] Definition and Questionnaire Design. A set of ranking criteria and corresponding ranking criteria levels is necessary for the ranking. The ranking criteria must epitomize intrinsic validity and relevance to reliability. The set of ranking criteria developed also included aspects of cost-benefit that are important factors in the selection of measures by companies and can not be neglected in any ranking analysis. The ranking criteria levels help define the ranking criterion scale such as, for instance, the cost scale which can vary between a staff-person week and three staff-person years. In preparation for the workshop, a questionnaire was developed to constitute the basis for expert opinion elicitation. During this step,

[3] Each ranking criterion discussed in this report is quantified into levels. Level is the position on the scale of a ranking criterion's quantity, strength, value, etc.

ranking criteria and ranking criteria levels which were developed during the LLNL study were revisited and revised to better suit the objectives of the study.

Step 4: Expert Opinion Elicitation and Workshop. Once designed, the questionnaires were distributed to the experts with the request that they'd be completed within a few weeks. The experts were then convened in a workshop where they were to summarize their evaluations of the measures and provide feedback on the ranking methodology being used. During the workshop, the experts described the measures they typically used in their research or in their respective industrial settings, commented on the ranking criteria, ranking criteria levels and on the documentation provided in support of the measures. They also described and recommended a set of measures to be added to the pool of thirty measures selected. This feedback is an integral part of the current report and the methodology and approach to the ranking problem were revised to incorporate the expert's comments.

Step 5: Expert Opinion Aggregation. This step was devoted to the analysis of the data collected through the expert opinion elicitation. It consisted of reviewing the data collected and reducing (aggregating) the data collected for each measure into a single number on a scale from 0 to 1 that symbolizes the ranking of the measure. Review of the data was conducted in order to determine and eliminate any inconsistencies, differences in interpretation, and whether the aggregation scheme needed to be modified to account for missing data. Multi-attribute theory [Keen76] was used in this study to aggregate expert input as it was in the LLNL study. A simple additive equation with equal weights was used as the basis of the aggregation procedure[4]. The equations used are given in Section 3.6.2.

Step 6. Missing Measures. As described in step 1, the measures selected in this study are based on a LLNL study and the ranking of the measures performed by two of the LLNL experts. To handle the possible omission of measures deemed worthy of consideration, elicitation of missing measures was built into the methodology as a separate step. In other words the questionnaire was designed with cells to provide the experts a method for defining a set of "missing measures", i.e. measures which were not part of the initial set of thirty measures but should have been. These measures would be ranked by the experts. Unfortunately given the limitations in time and effort and the already considerable effort exerted by the experts, this step was not performed. Since some of these measures might have a considerable impact on the final findings, it was decided however that the University of Maryland team would substitute itself for the experts and rank the measures. These rankings are provided in a separate chapter (Chapter 5) to clearly distinguish them from the measures ranked by the experts.

Step 7: Sensitivity Analysis. To understand the possible impact on the ranking of the form of the aggregation equation, of its parameters (the weights), and of the scheme by which the qualitative ranking criteria levels were transformed into real values, a sensitivity analysis study was performed. Ranking criteria, ranking criteria levels, aggregation weights, and aggregation formulae were varied and the impact of these variations assessed. The results of the sensitivity analysis show that little variation in the ranking of measures is observed and that for all practical purposes the additive aggregation formula with equal weights can be considered sufficient for the ranking of software engineering measures with respect to reliability.

Step 8: Result Analysis and Validation. This step draws conclusions from the analysis. Top-ranked measures are defined. The road from the top-ranked measures to reliability is under investigation. Lessons obtained during this research are summarized.

[4] The additive equation is typically selected as a first choice in most ranking approaches since it is definitely the simplest and it uses no preconceived notions as to the way the different ranking criteria impact the final ranking.

3.2 Measures Selection

As explained earlier the purpose of performing these steps is to determine a set of measures that have a high degree of intrinsic and extrinsic validity. The following section provides a detailed description of steps 1 through 8.

3.2.1 The Lawrence Livermore National Laboratory Study

As noted in Section 3.1, the study presented in this report is based on prior work performed by a LLNL research team. The LLNL study report " Assessment of Software Reliability Measurement Methods for Use in Probabilistic Risk Assessment" [LLNL98] identified 78 software engineering measures related either directly or indirectly to software reliability and that might be appropriate to the study of digital I&C systems. It also documented a set of ranking criteria developed by LLNL and the laboratories initial ranking of the measures. As explained in Section 3.1, this set of 78 measures was reduced to thirty using structural considerations (such as the fact that software reliability models are not measures and should not be part of such a study) as well as importance considerations. This new set of thirty measures served as the basis for an expert opinion elicitation effort described in the remainder of this chapter. The expert opinion elicitation effort is based on the revised set of ranking criteria and ranking criteria levels defined in Section 3.4.1 and on the software engineering measures questionnaire described in Section 3.4.2 and Appendix B.

3.2.2 Thirty Software Engineering Measures

The set of thirty measures considered in the study is listed below. A concise description of each measure can be found in Appendix A. This Appendix also lists additional references to which the reader is referred for a full understanding and appreciation of the measures.

- Bugs per line of code (Gaffney estimate)
- Cause & effect graphing
- Code defect density
- Cohesion
- Completeness
- Cumulative failure profile
- Cyclomatic complexity
- Data flow complexity
- Design defect density
- Error distribution
- Failure rate
- Fault density
- Fault-days number
- Feature point analysis
- Function point analysis
- Functional test coverage
- Graph-theoretic static architecture complexity
- Man hours per major defect detected
- Mean time to failure
- Minimal unit test case determination
- Modular test coverage
- Mutation testing (error seeding)
- Number of faults remaining (error seeding)
- Requirements compliance
- Requirements specification change requests
- Requirements traceability
- Reviews, inspections and walkthroughs
- Software capability maturity model
- System design complexity
- Test coverage

These thirty measures serve as the basis for the expert opinion elicitation, aggregation and ranking. The ranking criteria and their levels are defined in Section 3.4.1.

3.3 Experts Identification

Experts were selected who together covered the following areas of knowledge: software development, software engineering, software engineering measurement, software reliability engineering, software reliability modeling, software safety, digital I&C design. These diverse sources were defined as the base of knowledge necessary for resolving the problem of identifying single measures to be used (per life-cycle phase) to characterize reliability. Many of the experts, who took part in the study presented in this report, are knowledgeable in more than one of these areas. Furthermore a conscious effort was made to select experts from diverse engineering domains. The final selection includes experts from the nuclear and aerospace domain. This achieves a better representation of diverse engineering fields since the problem posed in this study applies to more than the nuclear engineering field. Finally an effort was made to obtain representation from academia as well as from industry. Again some of the experts have been both in academia and in the industry. This mix takes into consideration the fact that members of industry may have better insights into issues of cost and benefit whereas academia may have better knowledge of measures in experimental development and is at the edge of technological advances.

The final list of experts is given below.

Name	Occupation	Area of Expertise
Alain Abran	Director of the Research Laboratory in Software Engineering Management and a Professor at Universite du Quebec a Montreal (Canada)	Software measurement and management
David Card	Researcher and consultant in software measurement and process improvement in the Software Productivity Consortium	Software measurement and process improvement
William Everett	Consultant and owner of SPRE, Inc	Software reliability
Jon Hagar	Senior staff software engineer and group leader at Lockheed Martin Astronautics in Denver, Co.	Software reliability and testing
Herbert Hecht	Chairman of the Board of SoHaR Incorporated, Beverly Hills, California	Software reliability, safety and digital I&C system
Watts Humphrey	IBM's Director of Programming Quality and Process	Software engineering
Michael Lyu	Associate Professor at the Computer Science and Engineering department of the Chinese University of Hong Kong.	Software reliability/safety
Jean-Claude Laprie	"Directeur de Recherche" of CNRS, the French National Organization of Scientific Research	Software reliability/safety
William Petrick	President of Capri Technology Inc	Software safety
Allen Nikora	Senior member of the Information Systems and Computer Science staff in the Autonomy and Control Section at JPL	Software reliability modeling

3.4 Ranking Criteria, Ranking Criteria Levels Definition and Questionnaire Design

3.4.1 Ranking Criteria and Ranking Criteria Levels Definition

Software engineering measures can be compared by means of several attributes, collectively termed *ranking criteria*. For the problem of identifying a single measure which can be used (per life-cycle phase) to characterize reliability, seven ranking criteria were selected. Each of these ranking criteria evaluates some particular aspect of the measures that were considered important to the objectives of the study. The ranking criteria are grouped into three sets: quality (intrinsic validity of the measure), cost effectiveness (inherent cost of the use of the software engineering measure), and relevance (relevance to reliability and strength of the relationship between measure and reliability). Ranking criteria within a set judge particular aspects of the set.

The following is the list of ranking criteria and their assignment to one of the three sets, namely, quality, cost effectiveness and relevance.

Ranking criterion[5]	Set
Benefit	Cost effectiveness
Cost	Cost effectiveness
Credibility	Quality
Experience	Quality
Repeatability	Quality
Validation	Quality
Relevance to Reliability	Relevance

Each measure was evaluated according to ranking criteria levels. These ranking criteria levels provide a qualitative estimate of the "goodness" of a measure with respect to the ranking criterion. Several ranking criteria levels were defined for each ranking criterion and a single intermediate interpolation was permitted between each pair of defined ranking criteria levels.

The ranking criteria are described in detail in Sections 3.4.1.1 – 3.4.1.3.

The ranking criteria identified seem to be adequate for measures that have appeared to date. No guarantee exists however that the set of ranking criteria constitutes a complete set.

3.4.1.1 Quality Set

The following ranking criteria belong to the quality set: credibility, repeatability, experience, and validation. Each ranking criterion is defined in turn in the remainder of this section.

Credibility

The documentation[6] given for each measure claims that it measures some aspect of software development or software. A measure is considered to be *credible* if we judge it likely to support the specified goals. For example, if the measure is supposed to estimate software defects then it was deemed credible if it was judged that it did indeed measure software defects. This ranking criterion is internal to the measure in the sense that it rates the measure only in terms of its documented goals, not in terms of the project goals. Six ranking criteria levels were defined for credibility as follows.

A The measure directly evaluates or estimates the stated goal.

B The measure uses one or more quantities from which the stated goal can be derived using an algorithm.

[5] The initial list of ranking criteria was designed by Lawrence Livermore National Laboratory and was extracted from LLNL [LLNL98]. The LLNL list contained the following ranking criteria: benefit, cost, directness, timeliness, credibility, experience and repeatability. The University of Maryland team reviewed the ranking criteria and modified them to better fit the objectives of the study. In the following, we briefly review the changes made. The timeliness ranking criterion favored measures which were available early rather than late in the software development process. This ranking criterion was eliminated since all rankings are performed on a phase by phase basis. The directness ranking criterion was semantically close to relevance to reliability. Relevance to reliability and (the corresponding ranking criteria levels) was introduced in its place to clarify the meaning. This ranking criterion was also initially misplaced in the quality category in the materials for the workshop and replaced correctly in a semantically different category in this report. Repeatability, credibility and validation were reused entirely as defined by LLNL. Definitions for the cost and benefit ranking criteria were expanded and clarified from [LLNL98].

[6] By documentation, we mean the brief description of the measure provided to the experts to help them in their ranking of the measures. This documentation is reproduced in Appendix A.

C The measure uses one or more quantities from which the stated goal can be derived, but the definition or derivation is not precise.

D The measure uses one or more quantities from which the stated goal may be inferred, but the inference is indirect.

E The measure is not formalized with an algorithm, but a defined method of evaluating the measure does exist.

F The measure uses a quantity from which the stated goal may be inferred, but the inference is not plausible.

Repeatability

A measure is considered *repeatable* if the repeated application of the measure by the same or different people result in similar results. Five ranking criteria levels were defined for repeatability, as follows.

A The calculation uses a formula that requires no judgment on the part of the user.

B The calculation uses a formula; expert judgment is required to specify one or more inputs to the formula, but no judgment is required to perform the calculation or interpret the results.

C The calculation uses a formula: (a) no expert judgment is required to perform the calculation but judgment is required to interpret the results, or (b) no expert judgment is required to interpret the results but judgment is required to perform the calculation.

D The calculation uses a formula, but expert judgment is required both to perform the calculation and to interpret the result.

E The calculation is completely ad-hoc.

Experience

Commercial *experience* in using the different measures varies widely. Measures that are in wide use were judged more acceptable than those that are not widely used. This addresses a different aspect than the other quality ranking criteria since a measure might be widely used but still not technically useful in judging software reliability. For instance, although the measure LOC had been widely accepted by the industry for decades, no definite relationship between the LOC and the reliability has been established [Jone96].

Five ranking criteria levels were defined, as follows.

W Measure is in wide commercial use (i.e., hundreds of companies).

M Measure has had a modest amount of commercial use (i.e., dozens of companies).

L Measure has had little commercial use (i.e., a few companies at most).

E Measure has received some reported experimental use, but no commercial use.

N Measure has not been used.

Validation

Measures that have been extensively *validated* by the software community should be more acceptable than those that have not been validated. Five ranking criteria levels were defined as follows.

A The measure has been formally validated by persons other than the inventors of the measure.

B The measure has been formally validated only by the inventors (i.e., the validation is based on many sets of data).

C The measure has been informally validated by the inventors (i.e., it has been used on a few sets of data).

D The measure has not been validated (i.e., there exist theoretical studies, but no experimental results).

E The measure has been invalidated (i.e., it does not work in practice, or data used in validation is erroneous).

3.4.1.2 Relevance Set

The relevance set only contains the *relevance to reliability ranking criterion.* This ranking criterion measures the strength of the relationship existing between the measure and reliability.

Relevance to Reliability

This ranking criterion identifies relevant measures for predicting/estimating software reliability of safety critical digital systems during the various phases of their life cycles. Six ranking criteria levels were defined for each phase.

A Most of the models that currently estimate software reliability incorporate this measure, and any model assessing reliability of safety critical digital systems should incorporate this measure.

B. Some models that currently estimate software reliability incorporate this measure, and any model assessing reliability of safety critical digital systems should incorporate this measure.

C. Most of the models that currently estimate software reliability incorporate this measure, and this measure can be useful in any model assessing reliability of safety critical digital systems.

D. Some models incorporate this measure and this measure can be useful in any model assessing reliability of safety critical digital systems.

E. Some models incorporate this measure, but this measure would not be useful in any model assessing reliability of safety critical digital systems.

F. Very few or none of the models include this measure, and this measure is not of relevance.

3.4.1.3 Cost Effectiveness

There are two aspects to cost effectiveness—the cost of using the measure, and the benefits gained by using the measure. Benefits can be thought of as avoidance of costs. The two ranking criteria are defined below.

Cost

The *cost* ranking criterion concentrates on the effort required to implement and use the measure. Cost includes collection of data that is not normally immediately available from the standard development activity, the recording of such data, the calculation of results and the interpretation of results. Cost includes training time and tool acquisition (converting acquisition dollars into equivalent staff time). Cost does not include management use of the results.

For example, suppose a measure is associated with software reviews. Then the cost of the reviews is not included, but the cost of collecting, analyzing, and reporting data from the reviews is included. The cost of learning how to use the measure is part of the measure as well as the cost of acquiring, setting up and learning to use any tools peculiar to the measure.

Because cost is subject to great differences among actual development organizations, a model of a developer was created. It was assumed that the development organization is a small business with a staff of about twenty software engineers. The engineers were assumed to be well-trained in software development, but not in the particular measure under evaluation. The organization was assumed to have competent management that understands software development. If tool support is required for a measure, it was assumed that adequate tools exist but have not yet been acquired. It was assumed that all costs of using the measure must be included in a single project's cost. The following cost considerations are based on this company's typical one-year production.

Based upon this model, the possible cost ranking criteria levels range from one staff week to twenty staff years. Therefore it is reasonable and appropriate to design ranking criteria levels in log scale as follows in order to cover this wide range of values. Note however that the last cost ranking criteria level is 3 staff-person years since none of the measures currently available in the literature require an amount of resources superior to this.

Five ranking criteria levels were defined, as follows.

W	Use of the measure will require about a staff-person week.
M	Use of the measure will require about a staff-person month.
Q	Use of the measure will require about a staff-person quarter (three months).
Y	Use of the measure will require about a staff-person year.
T	Use of the measure will require about three staff-person years.

Benefits

Benefits are the other aspect of cost effectiveness. Benefits are defined to be the avoidance of costs that would be incurred if the measure was not used. All benefits accrued to the specific project should be included. This may be a reduction in technical effort, a reduction in management effort, or a reduction in the maintenance effort required to repair a fielded version of the software. Reliable software may lead to additional sales, and the benefits from those sales should be included. Dollar costs were converted to staff weeks to facilitate inclusion.

For example, assume that a measure is associated with software reviews. If a design review is held, and faults are found in the software design that would not otherwise be found until coding is complete and testing has taken place, then the reduced testing time is considered a benefit. Greater benefits accrue when faults are discovered early in the development life cycle.

The same developer model that was used for estimating cost was used to estimate benefits.

Based upon this model, the possible benefit ranking criteria levels range from one staff week to twenty staff years. Therefore it is reasonable and appropriate to design ranking criteria levels in log scale as follows in order to cover this wide range of values.

Six ranking criteria levels were defined as follows.

A	Use of the measure will reduce staff time by twenty or more staff years.
B	Use of the measure will reduce staff time by about ten staff years.
C	Use of the measure will reduce staff time by about five staff years.
D	Use of the measure will reduce staff time by about two years.
E	Use of the measure will reduce staff time by about one year.
F	Use of the measure will reduce staff time by less than 1 year.

3.4.2 Questionnaire Design

The following section briefly describes the questionnaire designed for expert opinion elicitation. Data elicited relates to the experts' evaluation of the ranking criteria levels for each measure and their ranking criteria level of confidence in this assessment. The detailed questionnaire can be found in Appendix B. The questionnaire was structured in five major sub-sections as follows:

The first sub-section establishes the ranking criteria level of expertise with respect to the different measures. The expert is requested to outline his exposure to the measure. Different types of exposure are defined in this subsection. They read as follows:

- Are you (the expert) the inventor of this measure?
- Have you (the expert) used this measure on different projects or experiments?
- Have you (the expert) been exposed to this measure through readings or through workshops/conferences?

If the expert has used the measure on a project, additional information (such as number of projects, type of projects, size of projects) is gathered. Using the data collected, the credibility of the expert can be assessed in an objective manner. In case of marked inconsistency in the data, this information can be cross-referenced with the degree of confidence and eventually used to reject the data or request further information. Possible biases due to the authorship of a measure can also be flagged.

The second sub-section elicits the ranking criteria levels for each ranking criterion defined in Section 3.4.1 and a degree of confidence in these estimates. Close examination of the ranking criteria indicated that the ranking criterion, "Relevance to Reliability", may depend heavily on the life-cycle phase. Consequently, relevance to reliability levels and degrees of confidence were solicited for each life-cycle phase. The degree of confidence is a subjective measure of the expert's confidence in the ranking criteria level given. This measure varies between 0 and 1. For each measure, the expert is to define a degree of confidence per ranking criterion or a global degree of confidence. A high degree of correlation should exist between degree of confidence and the factual data relating to the credibility of the expert.

The third sub-section investigates dependencies between measures. A table was given where the experts are requested to identify measures with high to medium levels of dependency (correlation) with other measures. Recognizing dependencies is important if one wishes to understand whether two measures are strongly coupled and thus provide inherently redundant information.

The fourth sub-section elicits new measures that have not been included in the set of thirty measures selected for the study and that the expert considers worthy of attention. A description of the additional measures is requested. Pertinent data, such as sources of knowledge, level and degree of confidence, level of correlation with respect to other measures, are solicited (Appendix B Tables B-6 to B-10).

The last sub-section is devoted to comments on the ranking criteria.

Examples of how to fill in the tables are provided in the questionnaire.

Roughly there are 1500 cells in the questionnaire. The effort required to complete the questionnaire was estimated to take one staff-week.

3.5 Expert Opinions Elicitation and Workshop

The workshop was set up to understand the expert's answers to the data collection effort and to obtain feedback on the general methodology as it pertained to ranking of measures with respect to reliability.

This two-day workshop was divided into:

- Individual presentations of each expert followed by questions and answers from the audience;
- Group discussion on software engineering measures;
- A concluding session with final presentations from the software measures working group, and questions and answers from the audience;

The group discussion on software measure classification led to the introduction of the structural classification in indicators, derived measures, primitive measures and attributes which was further refined after the workshop and is integrated in Chapter 2 Section 2.1. As part of the lessons learned from the workshop, it appears that in the future a process of this type would become even more effective if one could clarify the description of the measures in [IEEE982] and [LLNL98] which served as the basis for Appendix A; if a clearer definition of the ranking criteria and their levels could be given; or even better if one could set-up the workshop in such way that preliminary discussions with the experts are held to help them better understand the background of the study, the definitions, etc.

3.6 Expert Opinions Aggregation (Ranking Methodology)

To transform the experts' multidimensional ranking of a measure into a single number, which can then be used to rank the measures: 1) an equivalence between the letter grade scale and the set of real numbers belonging to [0, 1] was introduced; and 2) an aggregation scheme for the different numbers obtained was defined.

[7] This count does not include data related to the missing measures.

3.6.1 Ranking Criteria Levels' Quantification

This section explains how to convert the letter grades into real numbers. The conversion of the letter grade scale into [0, 1] is given in Table 3-1 for each ranking criterion and it's corresponding ranking criteria level.

The quantification of ranking criteria levels is predicated on the following principles:

- A value of 1 is assigned to the first ranking criteria level of a ranking criterion since it represents the best possible situation and hence is the situation of greatest utility;
- A value of 0 is assigned to the last ranking criteria level since it represents the worst possible situation and has the lowest possible utility;
- A ranking criteria level lying between the first and the last ranking criteria levels takes values between 0 and 1. Values taken depend on the relative utility of the ranking criteria level considered.

For example, the first ranking criteria level of the "Repeatability" ranking criterion is labeled (graded) A and defined as "The calculation uses a formula that requires no judgment on the part of the user". The last ranking criteria level is labeled E and defined as "The calculation is completely ad-hoc." Three other ranking criteria levels are recognized: "expert judgment is required to specify the inputs," "expert judgment is required to perform the calculation or to interpret the results," "expert judgment is required to both perform the calculation and interpret the result." Five possible calculation schemes thus exist which help determine the value of the software engineering measure. Different degrees of preference can be assigned to each scheme. The degree of preference is a function of the calculation scheme's capability to produce an identical value for the software engineering measure through repeated iterations of the formula by different or identical analysts. The quantification table given in Table 3-1 suggests that a scheme such as "expert judgment is required to perform the calculation or to interpret the results" is roughly three times less valuable than a scheme where "expert judgment is required to specify the inputs". In other words, repeating the calculation identically is three times more difficult. A scheme such that "expert judgment is required in calculation and the result interpretation" is roughly five times more difficult to repeat than the scheme "expert judgment is required to specify the inputs" (see the definitions of the criterion Repeatability and its levels, and the quantification of each level in Table 3-1).

Table 3-1 is based on the opinions of the UMD analysts and as such should be considered qualitative in essence. Since major differences might be expected if other analysts were to quantify the ranking criteria levels, sensitivity analysis was later performed to confirm that the rankings were independent of the choice of ranking criteria levels.

Cost		Experience	
Ranking criteria level	Value	Ranking criteria level	Value
W	1	W	1
M	0.9	M	0.55
Q	0.75	L	0.2
Y	0.3	E	0.15
T	0	N	0

Ranking criteria level	Benefits	Credibility	Repeatability	Validation	Relevance to Reliability
A	1	1	1	1	1
B	0.9	0.9	0.85	0.85	0.9
C	0.6	0.7	0.45	0.4	0.8
D	0.3	0.6	0.25	0.25	0.75
E	0.1	0.35	0	0	0.2
F	0	0			0

Table 3-1 Ranking criteria Level Values

3.6.2 Data Analysis Methodology

Once the conversion of letter graded ranking criteria levels into real values is final, these values need to be aggregated using an aggregation equation. The basic aggregation equation selected for this study and which serves as a reference is the linear additive equation with equal weights. In this aggregation scheme, each ranking criterion is assigned an equal weight (or importance) and the real values are combined linearly using a scheme of the type defined in Equation 3-1. A real value between 0 and 1 is obtained for each measure per expert and per phase. The per phase component of the analysis refers to an earlier comment which expressed the fact that the relevance to reliability ranking criterion might actually vary per phase. The ranking criteria levels for other ranking criteria are assumed independent of the software development phase.

Using such aggregation scheme, the rate of a measure for a given expert and a given phase *Rate(i, j, ϕ)* is :

$$Rate(i,j,\phi) = \sum_{k \in S_{cr}} r(i,j,k)w_k + r_\phi(i,j)w_\phi \qquad \text{Equation 3-1}$$

where

Rate(i, j, φ) :	The rate of measure *i* given by the *jth* expert in phase φ.
i:	Measure index. The range is from 1 to the number of measures under study.
j:	Expert index. The range is from 1 to *N*, where is *N* is the number of experts.
k:	Ranking criterion index.
S_{cr}:	The set *{Cost, Benefit, Credibility, Repeatability, Experience, Validation}*.
φ:	Development phase index. The range is from 1 to 4. A value of 1 stands for the requirements' phase, 2 for the design phase, 3 for the implementation phase, 4 for the testing phase.
r(i, j, k):	The real value equivalent of the *kth* ranking criterion level for measure *i* given by the *jth* expert.
$r_\phi(i, j)$:	The real value equivalent of the relevance to reliability ranking criterion for the *ith* measure in the *φth* phase given by the *jth* expert.
w_k:	The weight for each ranking criterion in set S_{cr}.
w_ϕ:	The weight of the relevance to reliability ranking criterion for the *φth* phase. Generally, this weight is a constant regarding different phase.

So the overall rate for a specific measure *i* in a given phase *φ* is given by:

$$Rate(i,\phi) = \frac{1}{N}\sum_{j=1}^{N} Rate(i,j,\phi) \qquad \text{Equation 3-2}$$

where

N	The total number of valid experts' inputs.
Rate(i, φ)	The combined rate over all experts for a specific measure *i* during a given phase *φ*.

In reality, the data analysis approach followed in this study differed somewhat from the one prescribed in Equation 3-1 and Equation 3-2, because, as we anticipated, many cells in the experts' questionnaires were left blank. Hence, Equation 3-1 and Equation 3-2 could not be used directly. To overcome this vacuum, UMD combined all of the experts inputs together before performing any further analysis using the following equations:

$$R(i,k) = \frac{1}{N(i,k)} \sum_{j \in S(i,k)} r(i,j,k)$$ Equation 3-3

$$R_\phi(i) = \frac{1}{N_\phi(i)} \sum_{j \in S_\phi(i)} r_\phi(i,j)$$ Equation 3-4

where

R(i, k):	The combined real value equivalent for the *kth* ranking criterion of measure *i*.
r(i, j, k):	The combined real value equivalent for the *kth* ranking criterion of measure *i* given by the *jth* expert.
S(i, k):	The set of valid[8] inputs for the *kth* ranking criterion of the *ith* measure.
N(i, k):	The number of elements in set *S(i, k)*.
k:	Ranking criterion index. $k \in$ {*Cost, Benefits, Credibility, Repeatability, Experience, Validation*}.
$R_\phi(i)$:	The combined level of ranking criterion relevance to reliability for the *ith* measure in phase ϕ.
$r_\phi(i, j)$:	The level of the ranking criterion relevance to reliability for the *ith* measure provided by the *jth* expert in the *ϕth* phase.
$S_\phi(i)$:	The set of valid[8] inputs for relevance to reliability for the *ith* measure in the *ϕth* phase.
$N_\phi(i)$:	The number of elements in set $S_\phi(i)$.

After processing the data using Equation 3-3 and Equation 3-4, two kinds of combined inputs are obtained. One is the combined (over all experts involved in the analysis) real-value-equivalent level for all ranking criteria of a measure with the exception of relevance to reliability; the other is the combined (over all experts involved in the analysis) real-value-equivalent level of a measure's relevance to reliability ranking criterion. To obtain the overall rate of a specific measure *i* in a specific phase ϕ the following equation is used:

$$Rate(i,\phi) = \sum_k R(i,k)w(k) + R_\phi(i)w_\phi$$ Equation 3-5

where

Rate(i, ϕ):	The overall rank for measure *i* in phase ϕ.
w(k):	Weight for ranking criterion *k*.

[8] A value is called valid if it is not empty after the processing from level to real number described in Section 3.6.1

w_ϕ:	Weight for the relevance to reliability ranking criterion in phase ϕ.
k:	Measures index. Please refer to notations for Equation 3-3 and Equation 3-4.
ϕ:	Phase index. Please refer to notations for Equation 3-1.

Weights in Equation 3-5 should verify Equation 3-6 for $\phi = 1, 2, 3, 4$ respectively.

$$\sum_k w(k) + w_\phi = 1$$ Equation 3-6

In the reference (base) case all weights are equal, that is, $w(k) = w_\phi = 1/7$ for any $k \in S_{Cr}$ and any phase ϕ.

3.6.3 Phase-Based Measures' Availability

Not all software engineering measures are applicable to a development phase. For instance, the measure "Cyclomatic complexity" can not be calculated until the design phase since the primitives used to calculate cyclomatic complexity are not defined until that phase. Once the primitives are available, they will remain available in the later phases of the life-cycle. Hence, a measure is defined as applicable to a phase if the primitives required to calculate the measure are available in the specific phase.

Availability information is used in the phase–to-phase calculation of the rate of a measure. Indeed, rates provided by the experts need to be filtered to reflect the availability of the measure during the phase.

Availability information is displayed in Table 3-2. A value of 1 corresponds to a measure that is available and a value of 0 to a measure which is not. This information is used as a multiplicative filter to eliminate measures from the corresponding phases or this information is used as a multiplicative filter to retain measures in the corresponding phases. For example, the availability of the "Failure Rate" is 0 during the "Requirement" phase. Thus the rate of this measure during the requirements phase is the value given in Equation 3-1 multiplied by the filter value 0 and will thus be equal to 0.

A note is necessary here to better understand the notion of availability. A measure specifically defined to capture the software's design characteristics is available from the design phase on till the end of the software's life, and not just during the design phase. One should also note that most of the phase-based availability information presented in Table 3-2 is extracted from IEEE Std 982.2 [IEEE982]. Where conflicts appear between the IEEE interpretation and this study's interpretation, these have been carefully noted in footnotes.

Table 3-2 lists the phase-by-phase availability of the thirty measures selected.

Measure	Phase-Based Availability			
	Requirement	Design	Implementation	Testing
Bugs per line of code (Gaffney estimate)	0	0	1	1
Cause & effect graphing	1	1	1	1
Code defect density	0	0	1	1
Cohesion	0	1	1	1
Completeness	1	1	1	1
Cumulative failure profile	0	0	0	1[*]

[*] In the IEEE Standard [IEEE982], this measure is considered available from the requirement phase.

Measure	Phase-Based Availability			
	Requirement	Design	Implementation	Testing
Cyclomatic complexity	0	1	1	1
Data flow complexity	0	1	1	1
Design defect density	0	1	1	1
Error distribution	1	1	1	1
Failure rate	0	0	0	1
Fault density	1	1	1	1
Fault-days number	1	1	1	1
Feature point analysis	1	1	1	1
Function point analysis	1	1	1	1
Functional test coverage	0	0	0	1
Graph-theoretic static architecture complexity	0	1	1	1
Man hours per major defect detected	0	1	1	1
Mean time to failure	0	0	0	1
Minimal unit test case determination	0	1[9]	1	1
Modular test coverage	0	0	0	1
Mutation testing (error seeding)	0	0	0	1
Number of faults remaining (error seeding)	1	1	1	1
Requirements compliance	1	1	1	1
Requirements specification change requests	1	1	1	1
Requirements traceability	0[10]	1	1	1
Reviews, inspections and walkthroughs	1	1	1	1
Software capability maturity model	1	1	1	1
System design complexity	0	1	1	1
Test coverage	0	0	0	1

Table 3-2 Phase-Based Measure Availability

3.7 Missing Measures

The experts identified, per step 4, a set of measures or categories of measures that should be added to the analysis. Specifically the following set of measures and measure categories were identified: {*Coverage Measure, Test Mutation Score, Full Function Point, Measures for Object Oriented (OO) Technologies and other Modern Technologies*}[11].

[9] In IEEE Standard [IEEE982] this measure is considered not available during the requirements phase. UMD changed it to available.

[10] In IEEE Standard [IEEE982] this measure is considered available during the requirements phase. UMD changed it to not available.

[11] These measures were not part of the initial 78 measures described in the Lawrence Livermore National Laboratory's study.

A brief description of: *{Coverage Measure, Test Mutation Score, Full Function Point}* is proposed in Table 3-3. A more detailed description has also been developed and is given in Appendix A.

Measures	Definition
Coverage Measure	Coverage = Probability [system recovers| fault occurs], which means the probability that the system can recover from an error. The goal of this measure is to reflect the ability of the system to automatically recover from the occurrence of a fault during normal system operation.
Test Mutation Score	A mutation is a single-point, syntactically correct change, introduced in the program P to be tested. The mutation score, denoted *ms*, is the ratio of the non-equivalent mutants of P (i.e. those which are distinguishable from P under at least one data item from the input domain) which are killed (distinguished from P) by a specific test data set T. It is a number in the interval [0,1]. The goal of this measure is to provide a measure of the efficiency of the test data set T. A high score indicates that T is very efficient for the program P with respect to mutation fault exposure.
Full Function Point (FFP) measurement	FFP is an adaptation of Function Point Analysis (FPA) techniques to the functional characteristics of real-time software. FFP measurement involves applying a set of rules and procedures to a given piece of software, as it is perceived from the perspective of its inherent functional user requirements. FFP, like FPA, measures functional size by evaluating transactional processes and logical groups of data. Full Function Point Analysis is a functional size measure for real-time control software.

Table 3-3 Missing Measures

The experts did not specify precisely what they meant by "Measures for OO Technologies and other Modern Technologies", or more specifically which of the many OO measures or of the measures for "modern technologies" should be used.

Measures for OO technology as well as for web-based systems are beginning to emerge. However, the current digital technology does not use web-based technology. Furthermore digital controllers will not be developed using web-technology in the foreseeable future and measurement within this area is still in its infancy. Based on these arguments, no measure was identified to reflect these.

On the other hand there certainly is no reason to rule out the use of OO in embedded systems. Hence, a preliminary set of OO measures [Chid94] [Lore94] [Hend96] for ranking was identified, which is presented in Table 3-4. The rationale for selecting this preliminary set of software measures is explored below.

The key component of a system implemented using OO technology is the concept of class. A class is an abstraction of an existing entity, such as a sensor or a control panel in a control system. Each class possesses a set of attributes, which represents the state of the class, and a set of methods, a concept equivalent to the concept of functions, which can act on the class. Classes are combined together by mechanisms called messages to construct a system. Consequently, the resulting system is characterized by the attributes of its classes, class attributes, class methods and the messages among the classes. The OO measures selected and displayed in Table 3-4 reflect these notions. Next to each measure is listed the attribute of the system it measures. Although these measures serve as a suitable initial set of measures, no guarantee is given that these measures constitute a complete set of OO measures adequate for reliability prediction. They need further validation.

OO Measure	**Attribute**
Class Coupling	Coupling among classes. This is one of the measures characterizing message complexity
Class Hierarchy Nesting Level	The level of inheritance and information reuse
Lack of Cohesion in Methods	The cohesion of the class
Number of Children	The number of immediate subclasses derived from the class
Number of Class Methods in a Class	The number of methods in each class
Number of Key Classes	Size of the system (in class)
Weighted Method per Class	Complexity of the class

Table 3-4 OO Measures and Corresponding Attributes

A brief description of the OO measures is given in Table 3-5. The full description of the measures can be found in Appendix A.

Measures	Definition
Class coupling	Coupling is defined as: when one object depends implicitly on another, they are tightly coupled. Object instances are tightly coupled with their classes. When one object depends directly on the visibility of another, they are closely coupled. When one object references another only indirectly through the other's public interface, they are loosely coupled. The goal of this measure is to examine how the class relates to other classes, subsystems, users, and so on. In practice, we want to build systems that get their work done by requesting services from other objects, which means we want to leverage the other classes' services, but we want to have services available at the right level, which means we want to keep the amount of coupling to a limited number.

Measures	Definition
Class hierarchy nesting level	Classes are organized for inheritance purposes hierarchically in a tree structure, with the base or the topmost class called the root. The further down from the root that a class exists in this hierarchy is called its nesting level. The goal of this measure is to identify the quality of the classes' use of inheritance. Large nesting numbers indicate a design problem, where developers are overly zealous in finding and creating objects. This will usually result in subclasses that are not specialization of all the super-classes. A subclass should ideally extend the functionality of the super-classes.
Lack of Cohesion in Methods (LCOM)	This measure is a relative indicator of cohesion of a class. The "relative" originates from the fact that this measure is the subtraction of the number of related method pairs from the number of unrelated method pairs within the class under measurement. Therefore the value of LCOM is a comparison between the number of correlated methods and the number of irrelevant method from design perspective (because whether two methods are correlated is determined by whether there is any instance variable shared by both of them. This criterion is based on the design consideration).
Number of Children	NOC is the count of the immediate subclasses of the class being measured. NOC was presented by Chidamber and Kemerer s a measure of complexity.
Number of Class methods In a Class	Classes are objects that can provide services (and state data) that are global to their instances. The number of methods available to the class and not its instances affects the size of the class. The number of class methods can indicate the amount of commonality being handled for all instances. It can also indicate poor design if services, better handled by individual instances, are handled by the class itself.
Number of Key Classes	The number of key classes is an indicator of the volume of work involved in developing an application. It is also an indication of the number of long-term reusable objects that will be developed as a part of this effort for applications dealing with the same or similar problem domain. Key classes are central to the business domain being developed. A key class can be related to a subset of an entity class or a model class. The number of key classes is a count of identified classes that are deemed to be of critical importance to the business.

Measures	Definition
Weighted Method per Class (WMC)	This measure deals with the internal characteristics of the classes' methods. The method measured can be a function or an action. There has historically been a lot of work done in the area of code complexity, but the fact that there are some basic differences between OO design and sequential design (1. OO codes are more compact and 2. OO design does not use case statement) makes these measurements less useful. This measure considers the number and types of messages sent by a method as being the basic measurement of complexity. It uses some assigned weights to compute method complexity. WMC is the sum of weighted methods in a class. Each method within the class is weighted by the complexity defined above and this weight is summed to arrive at WMC.

Table 3-5 Brief Descriptions of Missing OO Measures

For lack of available time during the workshop and due to the fact that not all missing measures had been precisely identified at that time, these additional measures could not be ranked by the experts during the workshop. Such ranking is however a necessary step for the completion of the analysis. Hence it was decided that the University of Maryland would perform a preliminary ranking of the additional measures and that this ranking would be validated by the workshop experts. This ranking effort is discussed in Chapter 5.

3.8 Sensitivity Analysis

The ranking methodology presented in Section 3.6 (and part of step 5) is based on a given equivalence between letter-graded ranking criteria levels and the set of real numbers, on the additive aggregation equation and equal weights. This reference or base needed to be validated by sensitivity analysis since:

- It could not be ensured that the letter/real equivalence was correctly inferred. The levels of ranking criteria were identified in the light of an assumed "distance" between adjacent ranking criteria levels. However, the quantification of this distance was essentially subjective, thus results needed to be validated by varying the ranking criteria level values. This variation needed to be comprehensive to cover all the possibilities.

- It was unclear which of the ranking criteria most influenced our ability to predict reliability. Such inadequate understanding prevents determination of a so-called correct set of weights and thus leads to the need for the analysis of the impact of variations in the weights used in the aggregation equation.

- It was unclear how the ranking criteria's influences should be combined. Unfortunately, the way in which ranking criteria interact defines the form of the aggregation function. Hence, the lack of understanding of this interaction mechanism impedes the construction of a so-called correct aggregation function. The only remedy to this obstacle was to explore alternate forms of the aggregation function and determine whether the aggregated rate[12] of a measure is sensitive to the form of the equation.

Hence, sensitivity analysis is performed by varying: 1) the function used to transform a measure's alphanumerical ranking criteria levels into real numbers, 2) the weights used for aggregating ranking criteria levels into a single real number in [0, 1] used to rank the measure, and 3) the equations used for aggregating ranking criteria levels into a single real number in [0, 1] used to rank the measures.

The result of the sensitivity analysis is an understanding of the specific impact of each of these elements on the ranking of the measures. The following paragraphs describe the different types of sensitivity analysis performed and provide the rationale for the selection of these particular schemes.

The results of the sensitivity analysis show that little variation in the ranking of measures is observed and that for all practical purposes the additive aggregation formula with equal weights can be considered sufficient for the ranking of software engineering measures with respect to reliability.

3.8.1 Sensitivity Analysis with Respect to Ranking Criteria Levels Quantification

Sensitivity analysis was first done with respect to the conversion of the letter scale used for ranking criteria levels into real numbers. Five different transformations were selected. The five transformations are described in Appendix C.

3.8.2 Sensitivity With Respect to the Weights Used in Aggregation Schemes

Sensitivity was then done by varying the weights appearing in the reference aggregation scheme. Four weighting schemes were considered. Their description follows:

Scheme 1 Each ranking criterion was assumed to have the same weight. This scheme modeled the case where no single ranking criterion is more important than any other.

Scheme 2 Scheme 2 distinguished four groups of ranking criteria: {Cost, Benefits}, {Credibility, Repeatability}, {Experience, Validation}, {Relevance to Reliability}.

The motivation for considering scheme 2 in this way was as follows. The four groups each represent distinctly different aspects of a measure. The group {Cost, Benefits} is related to its financial impact. The group {Credibility, Repeatability} is related to the theoretical validity of a measure. The group {Experience, Validation} is related to the experimental validity of a measure. The group {Relevance to Reliability} evaluates the relationship between the measure and the goal of the study: reliability. Since *a priori* each of these aspects will contribute equally to the ranking, these four groups can be assigned equal weights. And, furthermore, equal weights can be given within each group if no further knowledge is available.

In practice, since there are four distinct groups of ranking criteria, each group is assigned a weight of 1/4. Since each ranking criterion in a group receives the same weight, this translates for instance for the group {Cost, Benefits} into a weight of 1/8 for the Cost ranking criterion as well as for the Benefits ranking criterion. **On the other hand, the group {Relevance to Reliability} has only one criterion, therefore the criterion Relevance to Reliability receives a weight of 1/4.**

[12] An aggregated rate is a real value ranging from 0 to 1. It indicates the capability of the measure to predict software reliability. The higher aggregated rate value, the more capable the measure is of predicting software reliability.

Scheme 3 Scheme 3 uses the same groups as scheme 2 but varies the contributions of each group. The weighting scheme for the groups is as follows: {Credibility, Repeatability} = 1/3, Relevance to Reliability = 1/3, and {Cost, Benefits} = 1/6, and {Experience, Validation} = 1/6. The weights are distributed equally between ranking criteria in a group. Scheme 3 implies that factors such as theoretical validity and relevance to reliability contribute equally to the aggregated rates and are twice as important as {Experience, Validation} and financial considerations. This scheme reflects a weighting scheme that gives more value to the theoretical validity of a measure than to the practical aspects of applying the measure.

Scheme 4 -5 Schemes 1 to 3 are based on reasonableness assumptions. Two weighting schemes were added to the set to allow for errors in the reasoning leading to schemes 1 to 3. These schemes were obtained by random selection of the weights with the constraint that the sum of all weights equals to 1.

The reader should be reminded of the fact that the aim of the schemes is not to represent reality but to cover the space of possible weighting schemes.

3.8.3 Sensitivity with Respect to the Aggregation Equation

Finally, a sensitivity analysis was performed with respect to the aggregation equation. Two equations were considered. The first equation is a simple linear weighted sum which has already served as the reference equation for the analysis. The underlying concept for this equation is that every ranking criterion contributes to the aggregated rate of a measure independently, or in other words, ranking criteria are orthogonal or linearly independent. The second form is derived from the consideration that quality ranking criteria and relevance to reliability may interact with each other. The part of the equation that relates to quality ranking criteria and relevance to reliability is thus multiplicative [Keen76].

$$R = \sum_k w_k v_k$$ Equation Form 1

where

R:	The rate of the measure.
k:	Ranking criterion index, $k \in$ {*Cost, Benefits, Credibility, Repeatability, Experience, Validation, Relevance to Reliability*}.
w_k:	The weight for the *kth* ranking criterion. $\forall k,\ w_k = 1/7$.
v_k:	The real-value-equivalent for *kth* 's ranking criterion level.

$$R = w_{\bar{C}} v_{\bar{C}} + w_C v_C$$ Equation Form 2

where

$$v_{\bar{C}} = \frac{1}{K}[\prod_l (1 + K w_l) - 1]$$

$$K = \prod_{l}(1 + Kw_l) - 1$$

$w_{\bar{C}}$	Weights for all ranking criteria except the cost/benefit ranking criteria.
w_C	Weights for the cost set with $w_{\bar{C}} + w_C = 1$
l:	Running index referencing one of the ranking criteria within the following set {*Credibility, Repeatability, Experience, Validation, Relevance to Reliability*}
w_l:	Weight for the lth ranking criterion. $\sum_l w_l$ is not required to be 1.

Credibility	*Repeatability*	*Experience*	*Validation*	*Relevance to Reliability*	*K*	$w_{\bar{C}}$	w_C
0.4	0.4	0.2	0.2	0.8	-0.9346	0.5	0.5

Table 4 Weights used in Equation Form 2

3.9 Possible Limitations of the Methodology

The methodology proposed in this chapter is based on expert opinion elicitation. Consequently, the correctness and accuracy of the results depends heavily on the experts.

Software engineering technology is continually evolving and concurrently measures are being developed to reflect these changes. In the volatile nature of software engineering lies a weakness of the study presented: the need to modify the set of measures as the technology evolves

3.10 Summary and Conclusions

This chapter presents the methodology used to rank software engineering measures. The methodology is based on the use of expert opinion elicitation to solicit the scores of software engineering measures. The scoring is performed with respect to seven ranking criteria: credibility, repeatability, cost, benefit, experience, validation and relevance to reliability. The scoring is performed in terms of letter grades. A letter-conversion scheme translates the letter values to real numbers between 0 and 1. These numbers are then aggregated using an aggregation equation and a weighting scheme for the seven ranking criteria. The aggregated number serves as the indicator of the "goodness" of the measure. A sensitivity analysis is further performed on all components of the analysis: letter-real conversion scheme, aggregation function form, weighting scheme. Since *a priori* one can not assess which aggregation scheme is correct, the purpose of such sensitivity analysis is to prove that the results obtained remain valid for a wide spectrum of aggregation schemes.

The next chapter, Chapter 4, will provide the results obtained by application of this methodology to the initial set of measures. Chapter 5 will discuss the ranking of the missing measures.

[LLNL98] Lawrence, J. D., et al., *Assessment of Software Reliability Measurement Methods for Use in Probabilistic Risk Assessment*, Technical report UCRL-ID-136035, FESSP, Lawrence Livermore National Laboratory. 1998.

[Keen76] Keeney, R. L., Raiffa, H., *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*, John Wiley & Sons, New York, 1976.

[Jone96] Jones, C., *Applied Software Measurement: Assuring Productivity and Quality*, 2nd Edition, McGraw-Hill, New York, 1996.

[IEEE982] *IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software*, IEEE, 1988.

[Mill72] Mills, H. D., "On the Statistical Validation of Computer Programs", IBM Federal Systems Division, Gaithersburg, MD, Red. 72-6015, 1972.

[Chid94] Chidamber, S. R., Kemerer, F., A Metrics Suite for Object Oriented Design, *IEEE Transactions on Software Engineering,* Vol. 20, No. 6, June 1994.

[Lore94] Lorenz, M., Kidd, J., *Object-Oriented Software Metrics: A Practical Guide*, Prentice Hall, Inc. New Jersey, 1994.

[Hend96] Henderson-Sellers, B., *Object-oriented metrics: measures of complexity*, Prentice Hall, New Jersey, 1996.

CHAPTER 4 RESULTS AND ANALYSIS

Chapter 2 examined the problem of selecting software engineering measures to predict software reliability and designed a methodology which would help achieve this objective. The methodology is based on a rating and the consequent ranking of the measures. The rating of a measure is derived from the intrinsic and extrinsic characteristics of the measure. Chapter 3 outlined a full description of the rating process. Chapter 4 provides the rates and ranks obtained when applying the rating methodology to the measures selected. As discussed in chapter 2, measures can be classified into categories named "families". Hence, results are also presented by family. Results of the sensitivity analysis on letter-real conversion schemes, aggregation weights, and aggregation functions are also provided. These allow an assessment of the stability of the rankings and rates under various ranking schemes. A systematic analysis of the importance of ranking criteria was also performed and is presented at the end of the chapter.

4.1 Measures' Rates and Rankings

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 may vary from one development phase to another since relevance to reliability may vary between phases. Hence the methodology described in Chapter 2 and 3 aggregates the rates of measures phase by phase. Four development phases are of interest: Requirements, Design, Implementation, and Testing. Later phases are ruled out of the analysis since the software is already in operation. Table 4-1 lists the rates of the thirty measures in the four phases.

The following sections (Section 4.1.1 to Section 4.1.4) will discuss in greater detail the rates and rankings of the measures phase by phase. The emphasis will be placed on noteworthy trends and results.

Measure[1]	Development Phase			
	Requirement	Design	Implementation	Testing
Bugs per line of code (Gaffney estimate)			0.46	0.40
Cause & effect graphing	0.45	0.43	0.40	0.44
Code defect density			0.83	0.83
Cohesion		0.42	0.36	0.36
Completeness	0.42	0.36	0.36	0.36
Cumulative failure profile				0.76
Cyclomatic complexity		0.73	0.74	0.72
Data flow complexity		0.62	0.59	0.59
Design defect density		0.75	0.75	0.75
Error distribution	0.68	0.68	0.65	0.66
Failure rate				0.83
Fault density	0.71	0.73	0.73	0.75
Fault-days number	0.60	0.71	0.71	0.72
Feature point analysis	0.46	0.50	0.50	0.45
Function point analysis	0.51	0.54	0.55	0.50
Functional test coverage				0.62
Graph-theoretic static architecture complexity		0.52	0.46	0.46
Man hours per major defect detected		0.63	0.61	0.63
Mean time to failure				0.79
Minimal unit test case determination		0.59	0.64	0.70
Modular test coverage				0.70
Mutation testing (error seeding)				0.50
Number of faults remaining (error seeding)	0.46	0.46	0.47	0.51
Requirements compliance	0.50	0.49	0.50	0.50
Requirements specification change requests	0.70	0.69	0.69	0.69
Requirements traceability		0.56	0.56	0.55
Reviews, inspections and walkthroughs	0.61	0.61	0.61	0.61
Software capability maturity model	0.60	0.60	0.60	0.60
System design complexity		0.53	0.53	0.53
Test coverage				0.68

Table 4-1 Rates for the Different Software Engineering Measures Studied

1An empty cell in the table denotes that the corresponding measure is not applicable in the corresponding phase. For instance, the measure "Code defect density" is not available in the Requirements and Design phases. The same comment applies to all tables throughout the chapter.

4.1.1 Results in the Requirements Phase

Table 4-2 lists rates and rankings for the twelve measures available in the requirements phase[2]. The three top measures are "Fault density", "Requirements specification change requests", and "Error distribution". It implies that these measures are prime candidates as roots[3] of a software reliability prediction system in the requirements phase.

Other results worth commenting about are discussed below. The measure "Completeness" has the lowest ranking, a result due to the fact that the measure scores low in the *Credibility* criterion. This is because the measure attempts to assess completeness of the requirements, an objective that cannot theoretically be achieved. Furthermore the measure uses several primitives which are related to the design phase rather than to the requirements phase. This leads to confusion since the analyst is led to believe that the measure cannot be completely assessed during the requirements phase.

The measures "Feature point analysis" and "Function point analysis" belong to a family of measures dedicated to the evaluation of system functional size. The measure "Feature point analysis" is a variant of "Function point analysis" dedicated to real-time embedded systems. One would thus expect that these two measures would score identically or even that feature point analysis would score higher than function point analysis since it is devoted to the objective of this particular study, i.e. embedded systems for safety critical applications. However, the results in Table 4-2 indicate otherwise. The inconsistency originates from the difference observed in the scores of the *Experience* criterion: there is less industrial experience with "Feature point analysis" than with "Function point analysis".

Measure	Rate	Rank
Fault density	0.71	1
Requirements specification change requests	0.70	2
Error distribution	0.68	3
Reviews, inspections and walkthroughs	0.61	4
Fault-days number	0.60	5
Software capability maturity model	0.60	6
Function point analysis	0.51	7
Requirements compliance	0.50	8
Feature point analysis	0.46	9
Number of faults remaining (error seeding)	0.46	10
Cause & effect graphing	0.45	11
Completeness	0.42	12

Table 4-2 Rates and Rankings in the Requirements Phase

4.1.2 Results in the Design Phase

Table 4-3 lists rates and rankings for the twenty measures available in the design phase. The top ranked measures are "Design defect density", "Fault density", and "Cyclomatic complexity". Two of these three measures were not available in the requirements phase and are therefore new. The three measures are prime candidates as roots[3] of a software reliability prediction system in the design phase. But it should be noted

[2] For the list of measures available in a particular phase, the reader is referred to Table 3-2 in Chapter 3.

[3] A root of a software reliability prediction system is a measure that constitutes the starting point of a system and should be supplemented by additional measures which will complete the system.

that the number of measures above and close to the arbitrary threshold value of 0.7[4] has increased from 3 to 6 (0.68 and 0.69 are very close to 0.70) if compared with the requirements phase. This means that the number of measures considered valid for reliability prediction has increased and that the size of the underlining reliability prediction system has increased except some of the measures are redundant. Another interesting fact is that the highest rate encountered has increased between requirements and design phase from 0.71 to 0.75. This trend is to be expected since measures reflect artifacts and process phases which are closer to the final code being delivered. Hence the general "quality" of the measures should improve.

Measure	Rate	Rank
Design defect density	0.75	1
Fault density	0.73	2
Cyclomatic complexity	0.73	3
Fault-days number	0.71	4
Requirements specification change requests	0.69	5
Error distribution	0.68	6
Man hours per major defect detected	0.63	7
Data flow complexity	0.62	8
Reviews, inspections and walkthroughs	0.61	9
Software capability maturity model	0.60	10
Minimal unit test case determination	0.59	11
Requirements traceability	0.56	12
Function point analysis	0.54	13
System design complexity	0.53	14
Graph-theoretic static architecture complexity	0.52	15
Feature point analysis	0.50	16
Requirements compliance	0.49	17
Number of faults remaining (error seeding)	0.46	18
Cause & effect graphing	0.43	19
Cohesion	0.42	20
Completeness	0.36	21

Table 4-3 Rates and Rankings in the Design Phase

4.1.3 Results during the Implementation Phase

Table 4-4 lists rates and rankings for the 23 measures available in the implementation phase. The top ranked measures are "Code defect density", "Design defect density ", "Cyclomatic complexity", and "Fault density", respectively. The number of measures above (including one is very close to 0.70) the arbitrary level of 0.7 is still 6. The top ranked measure in this phase has a rating of 0.83 instead of 0.75 which was the value of the top-ranked measure in the design phase, reflecting once again a general improvement in the "quality" of the measures.

The rankings obtained seem reasonable but for two exceptions: the fact that the measure "Code defect density" and "Cyclomatic complexity" rank higher than the measure "Fault density", and that the measure "Number of faults remaining (error seeding)" only ranks number 18.

[4] This number is an arbitrary value selected by the University of Maryland. It is a threshold that allows to distinguish a "good" measure from a "bad" one.

The relative ranking of "Code defect density" versus "Fault density" is due to differences in the criteria *Repeatability*, *Experience*, and *Relevance to Reliability*. These scores reveal that the measure "Fault density" is more difficult to reproduce consistently, and is less widely used in industry than "Code defect density". Moreover, the measure "Fault density" is less relevant to reliability prediction than the measure "Code defect density". This is puzzling since one would think that the measure "Code defect density" is a specific instance of the measure "Fault density" particular to the implementation phase. This quagmire can be avoided by remembering that reliability is not solely determined by the fault content, but also by how frequently the faults manifest themselves as failures. Manifestation as a failure is determined by the position of the faults as explained in the example below.

Module M_1 is characterized by fault density FD_1, and a frequency of execution of ρ_1. Module M_2 's fault density is FD_2, and its frequency of execution is ρ_2. If one assumes that FD_1 is greater than FD_2, and ρ_1 is much smaller than ρ_2, then the statement " module M_1 impacts the reliability of the system more significantly than module M_2" does not hold. In other words, the reliability of a system is determined not only by the number of faults residing in the system (or fault density), but also by the frequency at which these faults are encountered.

As shown in Appendix A, the process involved in measuring "Fault density" is that of tracing back from failures observed to the faults that caused the failure. However, the faults counted in the measure "Code defect density" are observed directly through the code inspection and walkthrough process. Hence fault location information for "Code defect density" is more reliable than for "Fault density". Therefore the measure "Code defect density" was assessed more relevant to reliability than the measure "Fault density".

The similar analysis can resolve the puzzle that the measure "Cyclomatic complexity" ranks higher than the measure "Fault density". Although the scores of ranking criteria *Benefit, Credibility, and Relevance to Reliability* of the measure "Fault density" are higher than those of the measure "Cyclomatic complexity" (0.23 against 0.15, 0.82 against 0.76, 0.62 against 0.46, respectively), the scores of ranking criteria *Cost, Repeatability, Experience* are lower (0.88 against 0.92, 0.68 against 0.90, 0.91 against 1.00, respectively). Therefore the aggregation results are 0.74 against 0.73.

The second puzzling result is the low ranking achieved by the "Number of faults remaining", a result seemingly counterintuitive. The criteria *Benefit*, *Experience*, and *Validation*, were ranked low, i.e. the experts believe the application of the measure not to be practical. *Relevance to Reliability* received an equally low rate. This can be explained in the same manner as above, i.e. the measure does not provide location information on the faults that together with the number of faults remaining could be used for reliability estimation.

Measure	Rate	Rank
Code defect density	0.83	1
Design defect density	0.75	2
Cyclomatic complexity	0.74	3
Fault density	0.73	4
Fault-days number	0.71	5
Requirements specification change requests	0.69	6
Error distribution	0.65	7
Minimal unit test case determination	0.64	8
Reviews, inspections and walkthroughs	0.61	9
Man hours per major defect detected	0.61	10
Software capability maturity model	0.60	11
Data flow complexity	0.59	12
Requirements traceability	0.56	13
Function point analysis	0.55	14
System design complexity	0.53	15
Requirements compliance	0.50	16
Feature point analysis	0.50	17
Number of faults remaining (error seeding)	0.47	18
Bugs per line of code (Gaffney estimate)	0.46	19
Graph-theoretic static architecture complexity	0.46	20
Cause & effect graphing	0.40	21
Cohesion	0.36	22
Completeness	0.36	23

Table 4-4 Rates and Rankings in the Implementation Phase

4.1.4 Results during the Testing Phase

All thirty measures are applicable in this phase. The top ranked measures are "Failure rate", "Code defect density", and "Mean time to failure", respectively. The number of measures above the 0.7 threshold has increased to 10 as expected, and so has the rate of the top ranked measure, now equal to 0.83.

Worthy of comment is the observed difference between "Failure rate" and "Mean time to failure", and the relative rating of "Mean time to failure" and "Code defect density". These are discussed in turn.

Going back to the raw data provided by the experts, the difference in rankings between the measures "Failure rate" and "Mean time to failure" can be attributed to differences in the *Experience* criterion.

Note also that one would typically expect measures such as "Failure rate", "Mean time to failure", and "Cumulative failure profile" which all relate to the notion of failure (a notion conceptually close to reliability since reliability is inherently the study of software failures) to be better rated than the measure "Code defect density" which conceptually relates to faults. However, the rate of "Code defect density" is higher than the rate of "Mean time to failure" and "Cumulative failure profile" and distinguished from the measure "Failure rate" only by the third digit after the point (0.828 against 0.833). This can be explained by differences in the rate given to the criterion *Experience*. There is indeed much more reported industrial experience with "Code defect density" than with the other two measures.

Finally, differences between "Code defect density " and "Fault density", and the apparently surprising result of the low rates and rankings achieved by the "Number of faults remaining (error seeding)" in all four development phases can be explained in the same manner as in Section 4.1.3.

Measure	Rate	Rank
Failure rate	0.83	1
Code defect density	0.83	2
Mean time to failure	0.79	3
Cumulative failure profile	0.76	4
Fault density	0.75	5
Design defect density	0.75	6
Cyclomatic complexity	0.72	7
Fault-days number	0.72	8
Modular test coverage	0.70	9
Minimal unit test case determination	0.70	10
Requirements specification change requests	0.69	11
Test coverage	0.68	12
Error distribution	0.66	13
Man hours per major defect detected	0.63	14
Functional test coverage	0.62	15
Reviews, inspections and walkthroughs	0.61	16
Software capability maturity model	0.60	17
Data flow complexity	0.59	18
Requirements traceability	0.55	19
System design complexity	0.53	20
Number of faults remaining (error seeding)	0.51	21
Requirements compliance	0.50	22
Function point analysis	0.50	23
Mutation testing (error seeding)	0.50	24
Graph-theoretic static architecture complexity	0.46	25
Feature point analysis	0.45	26
Cause & effect graphing	0.44	27
Bugs per line of code (Gaffney estimate)	0.40	28
Cohesion	0.36	29
Completeness	0.36	30

Table 4-5 Rates and Rankings in the Testing Phase

4.2 Aggregation Results by Family

Section 4.1 provides rates by measure in each development phase. Section 4.2 provides rates per *family* of measures per phase. The concept of family has been explained in Chapter 2 as a grouping of measures which evaluates the same underlying concept. The thirty measures extracted from the LLNL study can be classified into eighteen families defined in Table 4-6.

Family	Measure
Fault detected per unit of size	*Code defect density*
	Design defect density
	Fault density
Functional size	*Feature point analysis*
	Function point analysis
Estimate of faults remaining in code	*Mutation testing (error seeding)*
	Number of faults remaining (error seeding)
Estimate of faults remaining per unit of size	*Bugs per line of code (Gaffney estimate)*
Failure rate	*Cumulative failure profile*
	Failure rate
	Mean time to failure
System architectural complexity	*Data flow complexity*
	Graph-theoretic static architecture complexity
	System design complexity
Module structural complexity	*Cyclomatic complexity*
	Minimal unit test case determination
Cohesion	*Cohesion*
Time taken to detect and remove faults	*Fault-days number*
	Man hours per major defect detected
Test coverage	*Functional test coverage*
	Modular test coverage
	Test coverage
Error distribution	*Error distribution*
Software development maturity	*Software capability maturity model*
Reviews, inspections and walkthroughs	*Reviews, inspections and walkthroughs*
Cause & effect graphing	*Cause & effect graphing*
Requirements compliance	*Requirements compliance*
Requirements traceability	*Requirements traceability*
Requirements specification change requests	*Requirements specification change requests*
Completeness	*Completeness*

Table 4-6 The Definition of the Family

The introduction of the concept of family results in the following benefits:

1. Conceptual redundancies among measures are eliminated and hence the population of measures is dramatically reduced. Less effort is required to reach meaningful conclusions. One can finally "see the forest for the trees." In the specific case examined here the number of measures was brought from 30 down to 18, a considerable reduction.

2. Families are more stable than measures. A distressing phenomenon plaguing the software measurement community is the continuous advent of new software engineering measures. Many of these measures constitute minute variations of known measures. Others originate from the advent of new development techniques. However, software as an artifact and software development in its wake is defined by a small number of fundamental characteristics. These characteristics are independent of a particular language, and are not likely to evolve in the very near future. Measures that relate to these particular characteristics are called families. Because all characteristics of software can be thoroughly identified and classified, so can families.

3. The concept of family eliminates the possible noise in the expert's inputs and hence improves the robustness of the ranking of measures. For instance, the ranking of "Code defect density" in the testing phase is higher than the ranking of measures such as "Mean time to failure" and "Cumulative failure profile". This is contradictory to intuition as explained in Section 4.1.4. However, "Mean time to failure" and "Cumulative failure profile" belong to the family "Failure rate". By choosing to represent the family by its highest ranked member, here "Failure rate", one reestablishes the natural ordering of measures: the family "Failure rate" ranks higher than the family "Fault detected per unit of size".

4. Each family reflects at least one software characteristic. Software reliability is ultimately determined by the interaction of such characteristics. Hence, the task of selecting measures to constitute a software reliability prediction system becomes tantamount to the task of picking one measure from each family.

The data in this section is taken directly from the data in Section 4.1. The only difference comes from the fact that a family is a set of related measures, and the rates and rankings in Section 4.1 are replaced by the values of the minimum, median, and maximum statistics of the rate in a family. The maximum rate represents the family as a whole. It tells the analyst how good a family can be in practice. This value however is still different from the value that could be attained by the concept underlying the family since the physical implementation is a "degraded" version of the concept itself. The median value is the most likely observation of the concept. As for the minimum value, it represents the worst known (in this study) implementation of the concept.

This section will examine the rates of the eighteen families in each development phase.

4.2.1 Results during the Requirements Phase

Family rates in the requirements phase are summarized in Table 4-7. The top ranked families are "Fault detected per unit of size", "Requirements specification change requests", and "Error distribution". The only difference between Table 4-7 and Table 4-2 is the fact that the twelve measures in Table 4-2 are regrouped into the eleven families in Table 4-7. All other comments in Section 4.1.1 are applicable to this section.

Family	Min	Median	Max
Fault Detected per Unit of Size	0.71	0.71	0.71[5]
Requirements specification change requests	0.70	0.70	0.70
Error Distribution	0.68	0.68	0.68
Reviews, inspections and walkthroughs	0.61	0.61	0.61
Time Taken to Detect and Remove Faults	0.60	0.60	0.60
Software Development Maturity	0.60	0.60	0.60
Functional Size	0.46	0.48	0.51
Requirements compliance	0.50	0.50	0.50
Estimate of Faults Remaining in Code	0.46	0.46	0.46
Cause & effect graphing	0.45	0.45	0.45
Completeness	0.42	0.42	0.42

Table 4-7 Rates by Family in the Requirements Phase

[5] Graying is used to depict the rates of the families that have crossed the arbitrary 0.7 threshold.

4.2.2 Results during the Design Phase

Family	Min	Median	Max
Fault Detected per Unit of Size	0.73	0.74	0.75
Module Structural Complexity	0.59	0.66	0.73
Time Taken to Detect and Remove Faults	0.63	0.67	0.71
Requirements specification change requests[6]	0.69	0.69	0.69
Error Distribution	0.68	0.68	0.68
System Architectural Complexity	0.52	0.53	0.62
Reviews, inspections and walkthroughs	0.61	0.61	0.61
Software Development Maturity	0.60	0.60	0.60
Requirements traceability	0.56	0.56	0.56
Functional Size	0.50	0.52	0.54
Requirements compliance	0.49	0.49	0.49
Estimate of Faults Remaining in Code	0.46	0.46	0.46
Cause & effect graphing	0.43	0.43	0.43
Cohesion	0.42	0.42	0.42
Completeness	0.36	0.36	0.36

Table 4-8 Rates by Family in the Design Phase

Table 4-8 lists family rates in the design phase. Families are ordered by their "Max" values. Therefore the ranking of the family "Requirements specification change requests" is lower than that of the family "Time taken to detect and remove faults". The potential measures for a software reliability prediction should be chosen from the families "Fault detected per unit of size", "Module structural complexity", and "Time taken to detect and remove faults". Families "Requirements specification change requests" and "Error distribution" should also be considered because their rates are close to the top ranking families.

4.2.3 Results during the Implementation Phase

Table 4-9 lists rates by family in the implementation phase. The top ranked families (top four) are identical to the top ranked families found in the design phase. This implies that no significant changes occur between design and implementation, due to the probable similarity between the phases (detailed design closely approximates coding).

Family	Min	Median	Max
Fault Detected per Unit of Size	0.73	0.75	0.83
Module Structural Complexity	0.64	0.69	0.74
Time Taken to Detect and Remove Faults	0.61	0.66	0.71
Requirements specification change requests	0.69	0.69	0.69
Error Distribution	0.65	0.65	0.65
Reviews, inspections and walkthroughs	0.61	0.61	0.61
Software Development Maturity	0.60	0.60	0.60
System Architectural Complexity	0.46	0.53	0.59
Requirements traceability	0.56	0.56	0.56
Functional Size	0.50	0.53	0.55

[6] "Requirements specification change requests" and "Error distribution" are included in the set of top ranking families since they so closely follows "Time Taken to Detect and Remove Faults" in Table 4-8.

Requirements compliance	0.50	0.50	0.50
Estimate of Faults Remaining in Code	0.47	0.47	0.47
Estimate of Faults Remaining per Unit of Size	0.46	0.46	0.46
Cause & effect graphing	0.40	0.40	0.40
Cohesion	0.36	0.36	0.36
Completeness	0.36	0.36	0.36

Table 4-9 Rates by Family in the Implementation Phase

4.2.4 Results during the Testing Phase

Table 4-10 lists rates by family in the testing phase. As expected, the family "Failure rate" is ranked as No. 1. The family "Fault detected per unit of size" falls in the second position. The next 4 families are rated very closely and above or barely below the "good enough" threshold of 0.7. Therefore these families should all be candidates in the software reliability prediction system.

Family	Min	Median	Max
Failure Rate	0.76	0.79	0.83
Fault Detected per Unit of Size	0.75	0.75	0.83
Module Structural Complexity	0.70	0.71	0.72
Time Taken to Detect and Remove Faults	0.63	0.67	0.72
Test Coverage	0.62	0.68	0.70
Requirements specification change requests	0.69	0.69	0.69
Error Distribution	0.66	0.66	0.66
Reviews, inspections and walkthroughs	0.61	0.61	0.61
Software Development Maturity	0.60	0.60	0.60
System Architectural Complexity	0.46	0.53	0.59
Requirements traceability	0.55	0.55	0.55
Estimate of Faults Remaining in Code	0.50	0.50	0.51
Requirements compliance	0.50	0.50	0.50
Functional Size	0.45	0.47	0.50
Cause & effect graphing	0.44	0.44	0.44
Estimate of Faults Remaining per Unit of Size	0.40	0.40	0.40
Cohesion	0.36	0.36	0.36
Completeness	0.36	0.36	0.36

Table 4-10 Rates by Family in the Testing Phase

The reader may find interest in noting that the number of families above the arbitrary limit of 0.7 is almost constant reflecting the stability throughout the software development lifecycle of the software characteristics of importance. This number takes values "3", "5", "4" and "6" respectively.

This section (Section 4.2) provides rates for all measures and families. These results are calculated based on a specific aggregation framework that includes a letter-real conversion scheme, an aggregation weighting scheme, and an aggregation function form. This specific aggregation framework needs to be validated by means of sensitivity analysis. Section 4.3 discusses this validation effort.

4.3 Sensitivity Analysis

The analysis presented in this section examines whether the results obtained (rates and ranks of the measures studied) depend on the aggregation framework[7] or, whether or not the present aggregation framework is "good enough" to reproduce the ranking and rates that would be observed in the most likely situations one can encounter.

In this section, the letter-real conversion scheme, the aggregation weighting scheme, and the aggregation function form are varied, and, one computes the new rates and rankings accordingly. The correlation coefficients between rates and rankings are then calculated.

Correlation coefficients characterize the existence (or non-existence) of a "linear relationship" [Sinc92] between two random variables, namely y_1 and y_2. If y_2 increases as y_1 increases, and y_2 decreases as y_1 decreases, then the relationship between y_2 and y_1 is "linear". The correlation coefficient between y_1 and y_2 tends to be 1 if y_2 increases as y_1 increases, and 0 if y_2 randomly increases or decreases as y_1 increases.

The rates obtained under an aggregation framework denoted 1 can be considered as the random variable y_1, and the rates obtained under an aggregation framework denoted 2 can be considered as the second random variable y_2. If the correlation coefficient of y_1 and y_2 equals 1, which means y_2 increases as y_1 increases, and vice versa, then the rankings related to y_2 should be identical to the rankings related to y_1. Hence, in the case of a high correlation coefficient between y_1 and y_2 one can conclude that the use of aggregation framework 2 instead of 1 will not impact the rankings. If one can prove that the correlation coefficient remains high when framework 2 varies to cover the entire valid aggregation framework space, then rankings are invariant for all valid frameworks in this valid framework space. The next three sub-sections (Section 4.3.1 to Section 4.3.3) discuss the correlation analysis for variations of the letter-real conversion scheme, the aggregation weighting scheme, and the aggregation function form, respectively.

4.3.1 Sensitivity Analysis on the Letter-Real Conversion

In this section, five letter-real conversion schemes are examined. Each scheme is fully described in Appendix C. The selection of these different conversion schemes is based on two considerations: complete coverage of all potential conversion-curve shapes and realism.

For instance, Figure 4-1 displays five different conversion schemes. "Scheme 1" is characterized by a high density at the high end of the conversion scheme. This means that most letters transform into values close to 1. "Scheme 2" emphasizes the middle of the conversion-curve, that is, the levels B[8], C, D, and E are all converted to a value close to 0.5, the level F is converted into 0.0 and the level A is converted into 1.0. "Scheme 3" equally distributes the values along the [0,1] interval. "Scheme 4" is the opposite image of "Scheme 2", and puts emphasis at the each end of the conversion-curve. Levels B and C convert into values close to 1.0, however D and E convert into values close to 0.0. Level A is converted into 1.0 and level F is converted into 0.0. "Scheme 5" is characterized by a high density at the low end of the conversion-curve. This means that most letters transform into values close to zero. These five schemes cover the space of possible conversion schemes.

[7] An aggregation framework is defined as the set {aggregation equation, weights, a letter-real conversion scheme}.

[8] The letter levels are not constrained to the range from A to F. Some criterion, like the *Cost*, utilizes letters T, Y, Q, M, and W which are equivalent to E, D, C, B, and A, respectively.

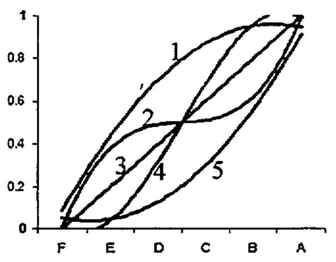


Figure 4-1 Letter-Real Conversion Schemes

The correlation coefficients for the rates and rankings corresponding to the five conversion schemes are given below phase by phase in Table 4-11 to Table 4-14. "Rate 1" denotes the aggregated rate obtained using conversion "Scheme 1". "Rank 1" denotes the aggregated ranking obtained using conversion "Scheme 1". Similarly, "Rate 2" and "Rank 2" correspond to the "Scheme 2", and so forth and so on. These notations hold for any other labels found in the following four tables. At the intersection of two rates, e.g., Rate 1 and Rate 2, one finds the correlation coefficient ρ_{12}. The same convention holds for the rankings.

	Rate 1	*Rate 2*	*Rate 3*	*Rate 4*	*Rate 5*
Rate 1	1				
Rate 2	0.985	1			
Rate 3	0.939	0.928	1		
Rate 4	0.975	0.991	0.921	1	
Rate 5	0.950	0.936	0.968	0.951	1

Table 4-11 Correlation Coefficients in the Requirements Phase

	Rate 1	*Rate 2*	*Rate 3*	*Rate 4*	*Rate 5*
Rate 1	1				
Rate 2	0.990	1			
Rate 3	0.941	0.922	1		
Rate 4	0.953	0.971	0.844	1	
Rate 5	0.962	0.960	0.924	0.956	1

Table 4-12 Correlation Coefficients in the Design Phase

	Rate1	*Rate2*	*Rate3*	*Rate4*	*Rate5*
Rate1	1				
Rate2	0.995	1			
Rate3	0.961	0.946	1		
Rate4	0.956	0.976	0.870	1	
Rate5	0.979	0.981	0.945	0.964	1

Table 4-13 Correlation Coefficients in the Implementation Phase

	Rate1	*Rate2*	*Rate3*	*Rate4*	*Rate5*
Rate1	1				
Rate2	0.992	1			
Rate3	0.955	0.939	1		
Rate4	0.947	0.966	**0.843**	1	
Rate5	0.984	0.98	0.939	0.959	1

Table 4-14 Correlation Coefficients in the Testing Phase

Observing the four tables above, one can reach the following conclusion:

Correlation coefficients between scheme 3 and scheme 4 are much lower than other coefficients. Scheme 3 and scheme 4 represent two extremely opposite situations in the letter-real conversion schemes. Hence this difference, although not significant, shows that extreme cases do impact the final rates and rankings, but do not generate drastic changes.

4.3.2 Sensitivity Analysis on Aggregation Weights

Sensitivity analysis on aggregation weights is discussed in this section. The correlation coefficients, results of this analysis, are presented in Table 4-16 to Table 4-19. A discussion of the selection of weighting schemes can be found in Section 3.8 and the detailed weighting schemes are shown in Appendix C and Table 4-15.

	Cost	Benefits	Credibility	Repeatability	Experience	Validation	Relevance to Reliability
Scheme 1	0.14	0.14	0.14	0.14	0.14	0.14	0.14
Scheme 2	0.13	0.13	0.13	0.13	0.13	0.13	0.25
Scheme 3	0.08	0.08	0.17	0.17	0.08	0.08	0.33
Scheme 4	0.245	0.045	0.088	0.036	0.130	0.239	0.216
Scheme 5	0.20	0.03	0.10	0.17	0.16	0.14	0.20
Scheme 6	0	0	0.25	0.25	0.25	0	0.25

Table 4-15 Weighting Schemes used in the Sensitivity Analysis

The first three weighting schemes are based on reasonableness considerations with respect to each criterion's contribution. Schemes 4 and 5 are obtained by random selection of the weight sets. The sixth scheme is an extreme case in which the criteria Cost, Benefit, and Validation are set to zero. In this scheme it is assumed that these criteria can be eliminated without critical impact on the final aggregation rates and rankings.

In Table 4-16, only the correlation coefficients of Scheme 1 and Scheme 3, Scheme 3 and Scheme 5, Scheme 3 and Scheme 6, and Scheme 4 and Scheme 6, are below 0.95[9], while others are above 0.95. In Table 4-17, the same thing happens to the correlation coefficients of Scheme 1 and Scheme 3, and Scheme 4 and Scheme 6. In Table 4-18 the phenomenon reoccurs with Scheme 4 and Scheme 6, and also in Table 4-19 with Scheme 4 and Scheme 6. A more in-depth study of the weighting schemes shows that Scheme 4

[9] UMD classifies the correlation coefficients as follows:

1. 0.95 – 1.0 characterizes a relationship between the two random variables labeled as "strongly linear".
2. 0.9 – 0.95 characterizes a relationship between the two random variables labeled as "satisfyingly linear".
3. 0.85 – 0.90 characterizes a relationship between the two random variables labeled as "acceptably linear".
4. below 0.85 . This value is characteristic of a relationship between the two random variables labeled as "non linear".

assigns strong weights to Cost, Validation, and Relevance to Reliability, while Scheme 6 assigns zero weights to Cost and Validation criteria.

Results presented in this sub-section show that a high degree of correlation exists between the different weighting schemes. This justifies the use of the scheme with equal weights adopted in Sections 4.1 and 4.2.

	Rate 1	*Rate 2*	*Rate 3*	*Rate 4*	*Rate 5*	*Rate 6*
Rate 1	1					
Rate 2	0.962	1				
Rate 3	0.908	0.983	1			
Rate 4	0.956	0.986	0.953	1		
Rate 5	0.984	0.978	0.939	0.984	1	
Rate 6	0.980	0.967	0.931	0.937	0.968	1

Table 4-16 Correlation Coefficients in the Requirements Phase

	Rate 1	*Rate 2*	*Rate 3*	*Rate 4*	*Rate 5*	*Rate 6*
Rate 1	1					
Rate 2	0.981	1				
Rate 3	0.947	0.988	1			
Rate 4	0.959	0.978	0.950	1		
Rate 5	0.988	0.987	0.963	0.978	1	
Rate 6	0.975	0.962	0.945	0.901	0.961	1

Table 4-17 Correlation Coefficients in the Design Phase

	Rate 1	*Rate 2*	*Rate 3*	*Rate 4*	*Rate 5*	*Rate 6*
Rate 1	1					
Rate 2	0.991	1				
Rate 3	0.973	0.993	1			
Rate 4	0.975	0.981	0.960	1		
Rate 5	0.993	0.991	0.979	0.983	1	
Rate 6	0.979	0.976	0.972	0.927	0.972	1

Table 4-18 Correlation Coefficients in the Implementation Phase

	Rate 1	*Rate 2*	*Rate 3*	*Rate 4*	*Rate 5*	*Rate 6*
Rate 1	1					
Rate 2	0.992	1				
Rate 3	0.972	0.992	1			
Rate 4	0.984	0.983	0.959	1		
Rate 5	0.995	0.991	0.974	0.988	1	
Rate 6	0.984	0.984	0.978	0.948	0.979	1

Table 4-19 Correlation Coefficients in the Testing Phase

4.3.3 Sensitivity Analysis on the Functional Form of the Aggregation Equation

In this study two forms of aggregation functions are examined. One is the simple additive function. This functional form characterizes situations where all criteria affect the aggregated result independently [Keen76]. It implies that one would be willing to give up a designated amount on criterion *i* to gain a designated amount on another criterion *j* regardless of the levels of the other criteria. For instance, one would give up 0.2 on criterion *Credibility* to gain 0.2 on the criterion *Cost* given equal weights, regardless at which levels any criterion, including *Credibility* and *Cost*, currently is.

The second functional form hypothesizes a potential influence among criteria in the quality and relevance set. Such influence can be represented by a product function. The cost/benefit set is considered independent of the quality and relevance sets, and the two groups are thus combined using an additive function. In this case a different type of dependency holds: the amount one would be willing to give up on a criterion in the quality or relevance sets to gain a designated amount on criteria *Cost* and *Benefit* depends on the current level of the other criteria in the quality or relevance sets. For instance, if one considers the trade-off between criteria *Credibility* and *Cost*, the level of the other criteria such as *Experience, Repeatability* would impact the designated trade-off value. For instance, if the value of criterion *Experience* is 0.6, one will give up an amount of 0.2 on criterion *Credibility* to gain the amount of 0.4 on criterion *Cost*; on the other hand, if the value of criterion *Experience* is very high, namely, 0.9, which means that there exists a wide commercial use of the measure, one would give up more on the criterion *Credibility* to gain the same amount on the criterion *Cost*. In this case, the relationship between *Credibility* and *Experience* is characterized by the multiplicative form.

The correlation coefficients between the two forms of aggregation functions shown in Table 4-20 to Table 4-23 demonstrate that a linear relationship exists in the Implementation and Testing Phase. However, the correlation coefficients in the Requirements phase cannot reveal a readily linear relationship. The correlation coefficients in the design phase are higher than those in Requirements phase and can demonstrate an acceptable linear relationship.

In contrast to the conversion scheme and weighting scheme, the aggregation function needs further study. The UMD research team believes that the criteria *Credibility, Repeatability, Experience, Validation,* and *Relevance to Reliability* interactively influence the rankings of a measure with respect to reliability prediction. A low level of any such criterion should impair the aggregated rankings drastically even if other criteria have relatively high levels. This dependency is better represented by a multiplicative function than by an additive function.

	Rate 1	*Rate 2*
Rate 1	1	
Rate 2	0.826905	1

Table 4-20: Correlation Coefficients in the Requirements Phase

	Rate 1	*Rate 2*
Rate 1	1	
Rate 2	0.901893	1

Table 4-21: Correlation Coefficients in the Design Phase

	Rate 1	*Rate 2*
Rate 1	1	
Rate 2	0.930295	1

Table 4-22: Correlation Coefficients in the Implementation Phase

	Rate 1	*Rate 2*
Rate 1	1	
Rate 2	0.946141	1

Table 4-23: Correlation Coefficients in the Testing Phase

The introduction of the concept of *family* significantly improves the robustness of the rankings under various function forms. For instance, the correlation coefficient of the rates in the requirements phase is 0.982 when rating is done per family against the 0.82 if the rating is done per measure. In the testing phase, the results are 0.952 and 0.94, respectively. Both cases display an improvement due to the use of families.

4.4 Criteria Analysis

An aggregation framework has been defined as a set {aggregation equation, weights, letter-real conversion scheme}. This set is defined on a set of criteria which at this point have never been challenged. To conclude this chapter, an experiment is conducted to challenge the criteria themselves. As has been repeatedly noted throughout this entire study, the set of criteria may be incomplete, incorrect or too many criteria may be involved in the analysis.

This section attempts to respond to the latter concern. In other words, "Are all seven criteria presented in this report necessary in ranking measures with regard to software reliability prediction?"

An experiment was subsequently carried out in which the weights of each criterion was varied according to the following procedure:

1. Select one of the criteria and assign a 1.0 to its weight and 0s to the weights of all remaining criteria. Repeat this step until each of the criteria has been given a weight of 1.0.

2. Select two out of seven criteria and assign each of these two criteria a weight of 0.5. Assign a weight of 0 to all other criteria. Repeat this step, selecting a new set of two, until each of the possible combinations is examined.

3. Select three out of seven criteria and assign each of these three criteria a weight of 0.33. Assign a weight of 0 to all other criteria. Repeat this step, selecting a new set of three, until each of the possible combinations is examined.

4. Select four out of seven criteria and assign each of these four criteria a weight of 0.25. Assign a weight of 0 to all other criteria. Repeat this step, selecting a new set of four, until each of the possible combinations is examined.

5. Select five out of seven criteria and assign each of these five criteria a weight of 0.2. Assign a weight of 0 to all other criteria. Repeat this step, selecting a new set of five, until each of the possible combinations is examined.

6. Select six out of seven criteria and assign each of these six criteria a weight of 0.17. Assign a weight of 0 to the remaining criteria. Repeat this step, selecting a new set of six, until each of the possible combinations is examined.

For each weighting scheme obtained by the above procedure, aggregate the rates and rankings for all measures. Perform a sensitivity analysis between this scheme and the 5 schemes described in Section

4.3.2. The results of the above experiment were sorted using the concept of "virtual distance" defined in Equation 4-1.

Equation 4-1

$$VD_i = \sum_j \frac{1}{5}(1 - \rho_{i,j})$$

where

i	The index of the weighting scheme under study. It ranges from 1 to 127 in this experiment.
VD_i	The "Virtual Distance (VD)" of the *ith* weighting scheme. The concept of VD quantifies the strength of the linear relationship between the rates under the *ith* weighting scheme and the rates under the first five weighting schemes described in Section 4.3.2.
j	The index of the five weighting schemes described in Section 4.3.2.
$\rho_{i,j}$	The correlation coefficient of the rates under the *ith* weighting scheme (I=1,127) and the rates under the *jth* reference weighting scheme (j=1,5) described in Section 4.3.2.

Results are presented in Appendix C. Observing the results, one can reach the following conclusions:

1. The larger the number of criteria included in the aggregation, the stronger the linear relationship observed between the experiment's rates and the reference rates described in Section 4.3.2. Therefore "more is better" and if possible one should include as many criteria as possible in the analysis.

2. All top-ranked weighting schemes contain the *Relevance to Reliability* criterion, implying that this criterion is significant in the determination of the rank of a measure.

3. The three criteria combinations {Cost, Credibility, Experience, Relevance to Reliability}, {Credibility, Repeatability, Experience, Validation, Relevance to Reliability}, and {Cost, Credibility, Repeatability, Experience, Validation, Relevance to Reliability} are the best possible criteria combinations obtained by selecting four out of seven, five out of seven, and six out of seven criteria, respectively. Furthermore, these combinations lead to virtual distances that are so low that the ranking would remain unchanged if one were to only know the values of these criteria.

4.5 Summary

This chapter discussed the rates and rankings (with respect to software reliability prediction) obtained for the 30 measures selected at the beginning of the study. Some potential inconsistencies are examined and explained. Sensitivity analysis with respect to the weights, the functional form of the aggregation function and with the letter-real conversion was carried out.

The top-ranked measures were aggregated by an additive function with equal weights. These measures constitute the possible roots of software reliability prediction systems.

Similar results were found for various aggregation schemes and can be found in Appendix C.

It was proven however that rates and ranks remain relatively stable for different aggregation schemes. The sole exception was the change in equation form that signals noticeable changes in the results. This indicates that a more detailed study of the form of the aggregation equation should be performed. Stability, however, is reestablished when one analyzes the results by families rather than by single measure.

A final result of interest lies in the study of the impact of the ranking criteria. The study shows that optimal combinations of four, five and six criteria exist which generate a ranking that closely approximates the ranking obtained using seven criteria. This allows ranking under partial information.

[Sinc92] Sincich, T., *Statistics for Engineering and the Sciences*, 3rd Edition, Dellen Publishing corp. New York, 1992.

[Keen76] Keeney, R. L., Raiffa, H., *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*, John Wiley & Sons, New York, 1976.

CHAPTER 5 MISSING MEASURES

The research performed by Lawrence Livermore National Laboratory identified 78 measures as discussed in Chapter 3. University of Maryland refined them into the 30 measures which were evaluated by experts. The experts identified 11 missing measures witch needed to be included into the study.

The missing measures presented in this chapter were recommended by experts who participated in the NRC workshop. The set of missing measures includes "Full function point" (FFP), which is an alternative for "Function point" for real-time systems, the "Coverage factor" to better represent fault-tolerant systems, the "Mutation score" to capture mutation testing techniques, and a bevy of measures for object-oriented (OO) techniques. The experts also suggested the addition of the "Reliability Trend Indicator". However, this is actually a reliability analysis method. Therefore it should not be considered as a missing measure.

The ranking methodology discussed in Chapter 3 was applied to the missing measures. Section 5.1 presents a brief description for each missing measure. Section 5.2 describes how the missing measures were rated. Section 5.3 presents classifications of all measures in terms of design and systems discussed in this study, and divides them into four groups. Separate family groups were identified to highlight the differences between non-OO and OO families. Section 5.4 provides the ranking results, and analyzes the impact of these missing measures on the rankings of the pre-selected measures. The analysis is performed per measure and by family. Finally Section 5.5 summarizes the contents of this chapter. All the raw input data and aggregated results discussed in this chapter are provided in Appendix D.

5.1 Introduction of the Missing Measures

The missing measures proposed by experts try to cover functional size measurement for real-time control systems, the fault-tolerant computing environment, the mutation testing technique, and the new but widely spread object-oriented development technique. A brief description of each of the ten missing measures is given below.

Full Function Point (FFP) is an adaptation of Function Point Analysis (FPA) for the counting of the real-time software's functional size [SEL]. FFP measurement applies a set of rules and procedures to a given piece of software, as it is perceived from the perspective of its inherent functional user requirements. FFP, like FPA, measures functional size by evaluating transactional processes and logical groups of data.

Coverage Factor is defined as the probability of a fault-tolerant system automatically recovering from the occurrence of a failure by the failure detection and recovery mechanism embedded in the system. The objective of this measure is to gauge the ability of the system to automatically recover from the occurrence of a failure during normal system operation [Arno73].

Mutation Score is an indicator of the efficiency of a test data set. A mutation is a single-point, syntactically correct change, introduced in the program P to be tested. The mutation score, denoted *ms*, is the ratio of the non-equivalent mutants of P (i.e. those which are distinguishable from P under at least one data item from the input domain) which are killed (distinguished from P) by a specific test data set T to the number of the non-equivalent mutants of P. The mutation score is a number in the interval [0,1] [Voas98].

The OO measures selected characterize the different aspects of OO design [Lore94]. The seven OO measures selected are described below:

Class Coupling. Coupling was proposed and defined by Myers [Myer78] as the degree of interaction between two modules. Chidamber and Kemerer [Khid94] revised this notion for OO systems as the dependability among classes. They also entitled it in [Khid94] "Coupling between Objects (CBO)".

This measure examines how the class relates to other classes. In practice, the coupling of a class to others needs to be limited to assure a good design and class reuse.

Class Hierarchy Nesting Level. Classes are organized for inheritance purposes hierarchically in a tree structure, with the base or the topmost class called the root. The number of levels from the root to a class is called its nesting level [Lore94].

This measure sheds light on the quality of the design with respect to inheritance. It is commonly understood that the deeper a class is nested in the inheritance hierarchy, the more public and protected methods there are for the class, and the more chances for method overrides or extensions. This all results in greater difficulty in testing a class.

Lack of Cohesion of Methods (LCOM). LCOM was introduced by Chidamber and Kemerer as a measure of the interaction among methods within a class [Khid94]. *Module* cohesion [Myer78] is defined as the degree of interaction within a module. LCOM, the OO version of cohesion, was defined as the difference between the number of method pairs that share at least one instance variable (also called attribute in some other literature) and the number of method pairs that do not share an instance variable.

The LCOM value provides a measure of the relatively disparate nature of methods in the class. The "relatively" here comes from the fact that although LCOM tries to quantify the strength of cohesion of a class, it does not arrive at an absolute value of the cohesion. Instead, it utilizes a value that can indirectly gauge the strength of the notion of cohesion. In other words, the value of LCOM grows as the strength of cohesion increases and vice versa. However, the potential relationship between LCOM and the absolute measure of cohesion (if it exists) is still not identified as yet.

LCOM is intimately tied to the instance variables and methods of a class, and therefore is a measure of the attributes of an object class.

Number of Children (NOC). NOC is defined as the number of immediate subclasses subordinated to a class in the class hierarchy [Khid94]. NOC measures how many subclasses are going to inherit the methods of the parent class. According to Chidamber and Kemerer, 1) the greater the number of children, the greater the inheritance and 2) the more children a parent class has, the greater the potential for improper abstraction of the parent class [Khid94]. NOC was introduced as a measure of complexity.

NOC is used as an ordinal scale of psychological complexity (understandability) [Neal96]. In other words, the understandability of a class is closely related to the number of immediate subclasses.

Number of Class Methods in a Class. Class methods are class services or behaviors. Methods are executed whenever an object receives a message. The number of methods available to the class affects the size of the class.

Number of Key Classes. A key class is a class that is requisite to the construction of a system. For instance, the calling class, connection class, and switch class are key classes of a telephony system. The number of key classes reveals the amount of total work required to complete the software. They are typically identified early in the OO analysis and design process. It is also an indication of the number of long-term reusable objects that will be developed as a part of the current effort.

Weighted Method per Class (WMC). WMC was introduced by its authors as a measure of complexity [Khid94]. It is the sum of weighted methods in a class. Each method within the class is weighted by a certain complexity value and this weight is summed to arrive at WMC. In this study we adopt a complexity value[1] defined by Lorenz and Kidd in [Lore94].

[1] A detailed description can be found in Appendix A, Section A.40.

This section provided brief descriptions of the missing measures. The next section will discuss the ratings of the measures.

5.2 Rating the Missing Measures

The rating process consists of applying the methodology described in Chapter 3 to the missing measures. The input of the rating process was provided by UMD. This section describes the approaches that UMD has taken to obtain the inputs for the 10 missing measures.

5.2.1 Using Analogy

The experts' inputs[2] for the pre-selected 30 measures are available at this point of our study. Hence the level of a criterion for a missing measure can be determined by the level of the same criterion for a related measure among the pre-selected 30 measures. Two measures A and B are "related" if, for instance, the cost estimate of applying the measure A is one month effort, and the cost of applying the measure B is three times as much as that of the measure A. A and B are then "analogous" to each other. In the following, the determination of several of FFP inputs illustrates the analogy.

FFP is a functional measure based on the standard function point analysis (FPA) technique. It was designed for both management information systems (MIS) and real-time software. Since FFP is an extension of the standard FPA, all rules of FPA are included in the FFP counting process. However, FPA rules dealing with control concepts have been expanded considerably. The analogy is explained as follows:

1. Credibility. The goals of FFP and FPA are almost identical. Therefore levels of the criterion *Credibility* should be identical (D+ for FFP).

2. The counting rules and processes are similar. Hence levels of the criterion *Repeatability* should be identical (C for FFP).

3. FFP and FPA both measure the functional size of the system. Therefore FFP is as relevant to reliability as FPA. Consequently the levels of the criterion *Relevance to Reliability* for FFP are E- for the requirements phase, D- for the design phase, D- for the implementation phase, and F+ for the testing phase.

5.2.2 Using the Scientific Literature and the Experts Opinion

An alternative approach is to refer to the literature and/or elicit expert inputs to assess the levels of ranking criteria. The approach is demonstrated by the determination of the levels of criteria *Cost, Benefit, Experience,* and *Validation* for FFP.

1. Regardless of the similarities between FFP and FPA, the difference between the levels of the criterion *Cost* cannot be determined by a simple analogy. FFP expands significantly the concepts and counting rules used in FPA. Therefore the use of FFP is more expensive than that of FPA. This is due to the increased effort and cost incurred in training and in the data collection process. However, the quantitative relationship between cost increase and the extension of rules is not simple and clear. Field expert inputs are required in this case. Unlike the approach taken during the workshop, UMD only selected a small number (less than three) of the field experts for the input of the specific criterion (or criteria) for the specific missing measures.

2. The improvement of FFP on "Function point" is the fact that applying FFP can lead to a more conceptually correct functional size count. The potential benefit generated through this improved

[2] An "input" is the level of a ranking criterion for a software engineering measure.

accuracy can only be evaluated through practical experience. Expert inputs are required to perform benefit assessment.

3. The levels for criteria *Validation* and *Experience* were culled from the scientific literature. UMD scrutinized a number of papers and articles published in journals, conferences, and laboratory reports related to FFP. All the information related to the validation and application of FFP was extracted and used to support the determination of the levels.

The methodology used for assessing the levels of the ranking criteria for the missing measure FFP is described in this section. Other missing measures were assessed using identical approaches. Table 5-1 defines the approach used for each missing measure.

Measure	Criterion Assessed Using Analogy	Criterion Assessed Using the Literature	Criterion Assessed Using Experts' Inputs
FFP	Cr., Rep., Rel.	Va., Ex.	Co., Be.
Mutation score		Cr., Rep., Rel., Va., Ex.	Co., Be.
Coverage factor		Cr., Rep., Rel., Va., Ex.	Co., Be.
Class coupling		Cr., Rep., Rel., Va., Ex., Co., Be.	
Class hierarchy nesting level		Cr., Rep., Rel., Va., Ex., Co., Be.	
Lack of cohesion in methods		Cr., Rep., Rel., Va., Ex., Co., Be.	
Number of children		Cr., Rep., Rel., Va., Ex., Co., Be.	
Number of class methods		Cr., Rep., Rel., Va., Ex., Co., Be.	
Number of key classes		Cr., Rep., Rel., Va., Ex., Co., Be.	
Weighted method per class		Cr., Rep., Rel., Va., Ex., Co., Be.	

Table 5-1 Measures vs Criterion Assessment Method

In Table 5-1, the abbreviations "Co.", "Be.", "Cr.", "Rep.", "Va.", "Ex.", and "Rel." stand for *Cost, Benefit, Credibility, Repeatability, Validation, Experience, and Relevance to Reliability*, respectively.

5.3 Revising Family Composition

The introduction of new measures inevitably influences family composition. The introduction of FFP, "Coverage factor", "Mutation score" and the six OO measures adds to the number of families and expands the sizes of some of the existing families defined in Chapter 4.

Table 5-2 lists measures classified in terms of non-OO vs OO design and fault-tolerant vs non-fault-tolerant systems.

	Non-fault-tolerant	**Fault-tolerant**
Non-OO	Bugs per line of code (Gaffney estimate) Cause & effect graphing Code defect density Cohesion Completeness Cumulative failure profile Cyclomatic complexity Data flow complexity Design defect density Error distribution Failure rate Fault density Fault-days number Feature point analysis Full function point Function point analysis Functional test coverage Graph-theoretic static architecture complexity Man hours per major defect detected Mean time to failure Minimal unit test case determination Modular test coverage Mutation score Mutation testing (error seeding) Number of faults remaining (error seeding) Requirements compliance Requirements specification change requests Requirements traceability Reviews, inspections and walkthroughs Software capability maturity model System design complexity Test coverage	Bugs per line of code (Gaffney estimate) Cause & effect graphing Code defect density Cohesion Completeness Coverage factor Cumulative failure profile Cyclomatic complexity Data flow complexity Design defect density Error distribution Failure rate Fault density Fault-days number Feature point analysis Full function point Function point analysis Functional test coverage Graph-theoretic static architecture complexity Man hours per major defect detected Mean time to failure Minimal unit test case determination Modular test coverage Mutation score Mutation testing (error seeding) Number of faults remaining (error seeding) Requirements compliance Requirements specification change requests Requirements traceability Reviews, inspections and walkthroughs Software capability maturity model System design complexity Test coverage
OO	Cause & effect graphing Class coupling Class hierarchy nesting level Code defect density Completeness Cumulative failure profile Design defect density Error distribution	Cause & effect graphing Class coupling Class hierarchy nesting level Code defect density Completeness Coverage factor Cumulative failure profile Design defect density

	Failure rate Fault density Fault-days number Feature point analysis Full function point Function point analysis Functional test coverage Lack of cohesion in methods (LCOM) Man hours per major defect detected Mean time to failure Mutation score Mutation testing (error seeding) Number of children (NOC) Number of class methods Number of faults remaining (error seeding) Number of key classes Requirements compliance Requirements specification change requests Requirements traceability Reviews, inspections and walkthroughs Software capability maturity model Test coverage Weighted method per class (WMC)	Error distribution Failure rate Fault density Fault-days number Feature point analysis Full function point Function point analysis Functional test coverage Lack of cohesion in methods (LCOM) Man hours per major defect detected Mean time to failure Mutation score Mutation testing (error seeding) Number of children (NOC) Number of class methods Number of faults remaining (error seeding) Number of key classes Requirements compliance Requirements specification change requests Requirements traceability Reviews, inspections and walkthroughs Software capability maturity model Test coverage Weighted method per class (WMC)

Table 5-2 Measure Classification in terms of Design and Systems

Only the measure "Coverage factor" is fault-tolerant specific. Therefore the fault-tolerant category differs from the non-fault-tolerant category solely as indicated by this measure.

The measures "Bugs per line of code (Gaffney estimate)", "Cohesion", "Cyclomatic complexity", "Data flow complexity", "Graph-theoretic static architecture complexity", "Minimum unit test case determination", "Modular test coverage", and "System design complexity" are not applicable to OO systems.

The non-applicability of the measures is explained below.

No research to date reveals whether the relationship in "Bugs per line of code (Gaffney estimate)" still holds true for OO systems. The OO environment has its own version of cohesion, specifically, "Lack of cohesion in methods". The traditional implementation of cohesion is not applicable to OO systems. "Cyclomatic complexity" can not capture the structural complexity of a method. This is because the fact that the size of each method is limited to a small value (the LOC of a method is typically 6 to 8 for SmallTalk and 20 for C++. Please refer to [Lore94] for further discussion). "Weighted method per class" is an OO-specific implementation of complexity introduced to substitute for "Cyclomatic complexity". The measure "Minimum unit test case determination" is no longer applicable to OO systems because of its consistently small value, which prevents it from acting as a quality discriminator as it does in non-OO environments.

"Modular test coverage" was removed from the list of measures applicable to OO systems because the concept "module" does not exist in OO systems.

Measures "Data flow complexity", "Graph-theoretic static architecture complexity", and "System design complexity" demonstrate the interaction between modules developed in sequential languages, such as

FORTRAN and C. The execution of the system is driven by the data flow, namely, the state of the system. On the other hand, the OO system is composed of loosely connected classes. Each class is autonomous, having its own state variables, and behavior methods. The execution of the OO system is driven by events. The group of OO measures selected in this study reflects these characteristics of an OO system.

Family	Measure
Cause & effect graphing	*Cause & effect graphing*
Cohesion	*Cohesion*
Completeness	*Completeness*
Error distribution	*Error distribution*
Estimate of faults remaining in code	*Mutation testing (error seeding)*
	Number of faults remaining (error seeding)
Estimate of faults remaining per unit of size	*Bugs per line of code (Gaffney estimate)*
Failure rate	*Cumulative failure profile*
	Failure rate
	Mean time to failure
Fault detected per unit of size	*Code defect density*
	Design defect density
	Fault density
Fault-tolerant coverage factor	*Coverage factor*
Functional size	*Feature point analysis*
	Function point analysis
	Full function point[3]
Module structural complexity	*Cyclomatic complexity*
	Minimal unit test case determination
Requirements compliance	*Requirements compliance*
Requirements specification change requests	*Requirements specification change requests*
Requirements traceability	*Requirements traceability*
Reviews, inspections and walkthroughs	*Reviews, inspections and walkthroughs*
Software development maturity	*Software capability maturity model*
System architectural complexity	*Data flow complexity*
	Graph-theoretic static architecture complexity
	System design complexity
Test adequacy	*Mutation Score*
Test coverage	*Functional test coverage*
	Modular test coverage
	Test coverage
Time taken to detect and remove faults	*Fault-days number*
	Man hours per major defect detected

Table 5-3 non-OO Family Definitions

Some families (or elements in families) were removed from the family list because the fact that they were not applicable to OO systems. Some new families (or new elements) were introduced because the introduction of new OO measures. Table 5-3 and Table 5-4 present the revised families in non-OO and OO systems, respectively.

[3] Gray shadowing of a row is used to pinpoint missing measures, their rates, and the corresponding ranking.

In Table 5-3 a new family called "Fault-tolerant coverage factor" is created to reflect the introduction of the "Coverage factor". FFP goes into the family "Functional size". "Mutation score" introduces a family titled "Test adequacy" because this measure tests the efficiency (or adequacy) of a set of test data.

All changes in Table 5-3 also appear in Table 5-4. In addition, families titled "Class behavioral complexity", "Class inheritance breadth", "Class inheritance depth", "Class structural complexity", "Cohesion", and "Coupling" are introduced to accommodate the following new OO measures: "Number of class methods", "Number of children (NOC)", "Class hierarchy nesting level", "Weighted method per class", "Lack of cohesion in methods", and "Class coupling", respectively.

"Number of class methods" assesses the number of methods available to a class. It also indicates how many activities this class could perform. "Number of children" indicates the breadth of a class's immediate inheritance. Likewise, the "Class hierarchy nesting level" shows the depth of a class in its inheritance hierarchy. The original "Module structural complexity" is revised to "Class structural complexity" to accommodate the new measure "Weighted method per class (WMC)". "Lack of cohesion in methods (LCOM)" substitutes for the original "Cohesion" in the family "Cohesion" because the latter is not applicable to OO systems. A new family "Coupling" is introduced by the introduction of the measure "Class coupling".

"Number of key classes" goes into the family "Functional size" because this measure can indicate the functional size of an OO system in terms of the concept of key classes. However, this measure is not as mature as "Function point". This immaturity can be illustrated by the following example.

Let us assume that one measures the "Number of key classes" of two different OO systems A and B. A is an airplane system in which the key classes should be the body, the jet and its support system, and the control system. B is a car system in which the key classes are the engine, the transmission, and the body. In this example the values of the "Number of key classes" of both systems are 3. However these values are not comparable in terms of the functional size of the system (the functional size of system A must be much larger than that of the system B). This inconsistency is unavoidable from the fact that a set of counting rules, like those used for "Function point", are not precisely defined.

Family	Measure
Cause & effect graphing	*Cause & effect graphing*
Class behavioral complexity	*Number of class methods*
Class inheritance breadth	*Number of children (NOC)*
Class inheritance depth	*Class hierarchy nesting level*
Class structural complexity	*Weighted method per class (WMC)*
Cohesion	*Lack of cohesion in methods (LCOM)*
Coupling	*Class coupling*
Completeness	*Completeness*
Error distribution	*Error distribution*
Estimate of faults remaining in code	*Mutation testing (error seeding)*
	Number of faults remaining (error seeding)
Failure rate	*Cumulative failure profile*
	Failure rate
	Mean time to failure
Fault detected per unit of size	*Code defect density*
	Design defect density
	Fault density
Fault-tolerant coverage factor	*Coverage factor*
Functional size	*Feature point analysis*
	Function point analysis
	Full function point
	Number of key classes
Requirements compliance	*Requirements compliance*
Requirements specification change requests	*Requirements specification change requests*
Requirements traceability	*Requirements traceability*
Reviews, inspections and walkthroughs	*Reviews, inspections and walkthroughs*
Software development maturity	*Software capability maturity model*
Test adequacy	*Mutation score*
Test coverage	*Functional test coverage*
	Test coverage
Time taken to detect and remove faults	*Fault-days number*
	Man hours per major defect detected

Table 5-4 OO Family Definitions

5.4 Results Analysis and the Impact of the Missing Measures

The rates of the missing measures were aggregated by applying the methodology described in Chapter 3. The aggregated values are provided in Table 5-6. Availability[4] information is provided in Table 5-5.

[4] Availability is where a measure specifically defined to capture the software's characteristics is available from the phase where it is introduced until the end of the software's life, and not just during that specific phase.

	Requirements	Design	Implementation	Testing
Full function point	1	1	1	1
Coverage	0	0	0	1
Mutation score	0	0	1	1
Number of key classes	0	1	1	1
Weighted measure per class	0	1	1	1
Number of class methods	0	1	1	1
Class hierarchy nesting level	0	1	1	1
Class coupling	0	1	1	1
Number of children	0	1	1	1
Lack of cohesion in methods	0	1	1	1

Table 5-5 Availability of Missing Measures

	Rates			
	Requirements	Design	Implementation	Testing
Full function point	0.49	0.53	0.53	0.48
Coverage				0.81
Mutation score			0.71	0.71
Number of key classes		0.53	0.53	0.51
Weighted measure per class		0.67	0.67	0.65
Number of class methods		0.69	0.69	0.66
Class hierarchy nesting level		0.69	0.69	0.66
Class coupling		0.69	0.69	0.66
Number of children		0.69	0.69	0.66
Lack of cohesion in methods		0.67	0.67	0.65

Table 5-6 Rates of Missing Measures

At this point it is appropriate to make a comment regarding the availability of the measure "Mutation Score". According to Offutt [Offu95], "mutation testing is a technique for unit testing..." But unit testing is performed during the implementation phase. Therefore it follows that the "Mutation Score" is applicable to the implementation phase.

This section is composed of two subsections. The rankings of the 33 measures and 20 families applicable to non-OO systems are analyzed in Section 5.4.1. And the rankings of the 32 measures and 22 families applicable to OO systems are discussed in Section 5.4.2.

5.4.1 Measures Applicable to non-OO Systems

5.4.1.1 Requirements Phase

Only the new measure FFP is introduced in this phase. Table 5-7 lists all the measures, the corresponding rates and rankings for the requirements phase. The missing measure FFP plays a less important role than the measure "function point" but is of greater importance than the measure "feature point": the lack of experience with FFP diminishes its importance in predicting software reliability. However, the wider acceptance of FFP contributes to a higher ranking of FFP than "Feature point".

FFP belongs to the family "Functional size". The introduction of FFP does not significantly change the statistics of this family but increases the median from 0.48 to 0.49. Table 5-8 shows the family rates after the introduction of FFP. The ranking of the family "Functional size" remains the same as with the introduction of the FFP.

In summary, the introduction of the new measure FFP does not influence the rankings of families that are applicable during the requirements phase.

Measure	Rate	Ranking
Fault density	0.71	1
Requirements specification change requests	0.70	2
Error distribution	0.68	3
Reviews, inspections and walkthroughs	0.61	4
Fault-days number	0.60	5
Software capability maturity model	0.60	6
Function point analysis	0.51	7
Requirements compliance	0.50	8
Full function point[5]	0.49	9
Feature point analysis	0.46	10
Number of faults remaining (error seeding)	0.46	11
Cause & effect graphing	0.45	12
Completeness	0.42	13

Table 5-7 Rates of non-OO Measures during the Requirements Phase

Family	Rate		
	Min	Median	Max
Fault detected per unit of size	0.71	0.71	0.71[6]
Requirements specification change requests	0.70	0.70	0.70
Error distribution	0.68	0.68	0.68
Reviews, inspections and walkthroughs	0.61	0.61	0.61
Time taken to detect and remove faults	0.60	0.60	0.60
Software development maturity	0.60	0.60	0.60
Functional size	0.46	**0.49**[7]	0.51
Requirements compliance	0.50	0.50	0.50
Estimate of faults remaining in code	0.46	0.46	0.46
Cause & effect graphing	0.45	0.45	0.45
Completeness	0.42	0.42	0.42

Table 5-8 Rates of non-OO Families during the Requirements Phase

5.4.1.2 Design Phase

No missing measures were found for the design phase. Table 5-9 lists all measure rates and rankings in the design phase. Table 5-10 lists all family rates in the design phase.

[5] Gray shadowing of a row is used to pinpoint missing measures, their rates, and the corresponding ranking.

[6] Right diagonal graying is used to depict the rates of families that have crossed the arbitrary 0.7 threshold.

[7] Bolding is used to pinpoint statistics modified by the introduction of missing measures.

The rankings of FFP in this phase is higher than that in the requirements phase. Its higher degree of relevance to reliability contributes to this higher ranking. This is consistent with the experts' inputs. As a matter of fact, the level of the criterion *Relevance to Reliability* for FPA increases from the requirements phase to the implementation phase and reaches its apogee during the implementation phase. The corresponding FFP criterion follows the same trend.

The discussion presented in Chapter 4 Sections 4.1.2 and 4.2.2 is also relevant to this section.

Measure	Rate	Ranking
Design defect density	0.75	1
Cyclomatic complexity	0.73	2
Fault density	0.73	3
Fault-days number	0.71	4
Requirements specification change requests	0.69	5
Error distribution	0.68	6
Man hours per major defect detected	0.63	7
Data flow complexity	0.62	8
Reviews, inspections and walkthroughs	0.61	9
Software capability maturity model	0.60	10
Minimal unit test case determination	0.59	11
Requirements traceability	0.56	12
Function point analysis	0.54	13
Full function point	0.53	14
System design complexity	0.53	15
Graph-theoretic static architecture complexity	0.52	16
Feature point analysis	0.50	17
Requirements compliance	0.49	18
Number of faults remaining (error seeding)	0.46	19
Cause & effect graphing	0.43	20
Cohesion	0.42	21
Completeness	0.36	22

Table 5-9 Rates of non-OO Measures during the Design Phase

Family	Rate		
	Min	Median	Max
Fault detected per unit of size	0.73	0.74	0.75
Module structural complexity	0.59	0.66	0.73
Time taken to detect and remove faults	0.63	0.67	0.71
Requirements specification change requests	0.69	0.69	0.69
Error distribution	0.68	0.68	0.68
System architectural complexity	0.52	0.53	0.62
Reviews, inspections and walkthroughs	0.61	0.61	0.61
Software development maturity	0.60	0.60	0.60
Requirements traceability	0.56	0.56	0.56
Functional size	0.50	**0.53**	0.54
Requirements compliance	0.49	0.49	0.49
Estimate of faults remaining in code	0.46	0.46	0.46
Cause & effect graphing	0.43	0.43	0.43
Cohesion	0.42	0.42	0.42
Completeness	0.36	0.36	0.36

Table 5-10 Rates of non-OO Families during the Design Phase

5.4.1.3 Implementation Phase

The missing measure "Mutation score" emerged the implementation phase. "Mutation score" ranks 6th in this phase.

"Mutation score" is a support measure that helps validate the completeness of the failure data. It is a percentage of the observed failures against potential failures. Therefore the use of this measure can reveal the number of potential failures residing in the code based upon the number of observed failures.

Table 5-11 lists the measures and their corresponding rankings. Table 5-12 provides the families and their corresponding statistics during the implementation phase.

Measure	Rate	Ranking
Code defect density	0.83	1
Design defect density	0.75	2
Cyclomatic complexity	0.74	3
Fault density	0.73	4
Fault-days number	0.71	5
Mutation score	0.71	6
Requirements specification change requests	0.69	7
Error distribution	0.65	8
Minimal unit test case determination	0.64	9
Man hours per major defect detected	0.61	10
Reviews, inspections and walkthroughs	0.61	11
Software capability maturity model	0.60	12
Data flow complexity	0.59	13
Requirements traceability	0.56	14
Function point analysis	0.55	15

System design complexity	0.53	16
Full Function Point	0.53	17
Feature point analysis	0.50	18
Requirements compliance	0.50	19
Number of faults remaining (error seeding)	0.47	20
Bugs per line of code (Gaffney estimate)	0.46	21
Graph-theoretic static architecture complexity	0.46	22
Cause & effect graphing	0.40	23
Cohesion	0.36	24
Completeness	0.36	25

Table 5-11 Rates of non-OO Measures during the Implementation Phase

Family	Rate		
	Min	Median	Max
Fault detected per unit of size	0.73	0.75	0.83
Module structural complexity	0.64	0.69	0.74
Test adequacy	0.71	0.71	0.71
Time taken to detect and remove faults	0.61	0.66	0.71
Requirements specification change requests	0.69	0.69	0.69
Error distribution	0.65	0.65	0.65
Reviews, inspections and walkthroughs	0.61	0.61	0.61
Software development maturity	0.60	0.60	0.60
System architectural complexity	0.46	0.53	0.59
Requirements traceability	0.56	0.56	0.56
Functional size	0.50	0.53	0.55
Requirements compliance	0.50	0.50	0.50
Estimate of faults remaining in code	0.47	0.47	0.47
Estimate of faults remaining per unit of size	0.46	0.46	0.46
Cause & effect graphing	0.40	0.40	0.40
Cohesion	0.36	0.36	0.36
Completeness	0.36	0.36	0.36

Table 5-12 Rates of non-OO Families during the Implementation Phase

The family "Test adequacy" is applicable with the implementation phase because of the introduction of the measure "Mutation score". The introduction of FFP during this phase does not change the statistics of the family "Functional size", let alone the rankings of this family.

5.4.1.4 Testing Phase

The measure "Coverage factor" emerges in the testing phase. The measure "Mutation score" during this phase retains the same value from the previous (implementation) phase. The ranking of the measure "Coverage factor" follows immediately that of the measure "Failure rate". The ranking of "Mutation score" decreases from 6 to 10 while the rate remains the same. This change stems from the fact that other measure rates increase with this phase. The rates and rankings of all measures applicable during the testing phase are provided in Table 5-13.

Measure	Rate	Ranking
Failure rate	0.83	1
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
Cyclomatic complexity	0.72	8
Fault-days number	0.72	9
Mutation score	0.71	10
Minimal unit test case determination	0.70	11
Modular test coverage	0.70	12
Requirements specification change requests	0.69	13
Test coverage	0.68	14
Error distribution	0.66	15
Man hours per major defect detected	0.63	16
Functional test coverage	0.62	17
Reviews, inspections and walkthroughs	0.61	18
Software capability maturity model	0.60	19
Data flow complexity	0.59	20
Requirements traceability	0.55	21
System design complexity	0.53	22
Number of faults remaining (error seeding)	0.51	23
Function point analysis	0.50	24
Mutation testing (error seeding)	0.50	25
Requirements compliance	0.50	26
Full function point	0.48	27
Graph-theoretic static architecture complexity	0.46	28
Feature point analysis	0.45	29
Cause & effect graphing	0.44	30
Bugs per line of code (Gaffney estimate)	0.40	31
Cohesion	0.36	32
Completeness	0.36	33

Table 5-13 Rates of non-OO Measures during the Testing Phase

Table 5-14 provides the families and their corresponding rates. The introduction of the measure "Coverage factor" introduces a new family titled "Fault-tolerant coverage factor". As a support family of "Failure rate" in a fault-tolerant system, the family "Fault-tolerant coverage factor" ranks lower but very close to the family "Failure rate". All the families related to failure and fault information during the testing phase rank higher than other families. The only exception is the instance where the family "Module structural complexity" ranks higher than the family "Time Taken to Detect and remove Faults". This exception indicates the fact that the module structural complexity measures ("Cyclomatic complexity" and "Minimum unit test case determination") still play an important role in software reliability prediction even during the testing phase.

Family	Rate		
	Min	Median	Max
Failure Rate	0.76	0.79	0.83
Fault Detected per Unit of Size	0.75	0.75	0.83
Fault-tolerant Coverage Factor	0.81	0.81	0.81
Module Structural Complexity	0.70	0.71	0.72
Time Taken to Detect and Remove Faults	0.63	0.67	0.72
Test adequacy	0.71	0.71	0.71
Test Coverage	0.62	0.68	0.70
Requirements specification change requests	0.69	0.69	0.69
Error Distribution	0.66	0.66	0.66
Reviews, inspections and walkthroughs	0.61	0.61	0.61
Software Development Maturity	0.60	0.60	0.60
System Architectural Complexity	0.46	0.53	0.59
Requirements traceability	0.55	0.55	0.55
Estimate of Faults Remaining in Code	0.50	0.50	0.51
Requirements compliance	0.50	0.50	0.50
Functional Size	0.45	0.48	0.50
Cause & effect graphing	0.44	0.44	0.44
Estimate of Faults Remaining per Unit of Size	0.40	0.40	0.40
Cohesion	0.36	0.36	0.36
Completeness	0.36	0.36	0.36

Table 5-14 Rates of non-OO Families during the Testing Phase

5.4.2 Measures Applicable to OO Systems

5.4.2.1 Requirements Phase

The sets of measures applicable to non-OO and OO systems are identical during the requirements phase. All rates and rankings are presented in Table 5-7 and Table 5-8.

5.4.2.2 Design Phase

Table 5-15 lists the rates and rankings of OO measures which are applicable during the design phase. Table 5-16 provides the statistical record of the rates for OO families.

OO measures are introduced in this phase. The three top-ranked measures are "Design defect density", "Fault density", and "Fault-days number". The OO measures, which capture the design characteristics of OO systems, score lower than the top-ranked fault measures. This again supports the idea that fault data plays a more important role than system structural data in software reliability prediction.

Measure	Rate	Ranking
Design defect density	0.75	1
Fault density	0.73	2
Fault-days number	0.71	3
Class coupling	0.69	4

Class hierarchy nesting level	0.69	5
Number of children (NOC)	0.69	6
Number of class methods	0.69	7
Requirements specification change requests	0.69	8
Error distribution	0.68	9
Lack of cohesion in methods (LCOM)	0.67	10
Weighted method per class (WMC)	0.67	11
Man hours per major defect detected	0.63	12
Reviews, inspections and walkthroughs	0.61	13
Software capability maturity model	0.60	14
Requirements traceability	0.56	15
Function point analysis	0.54	16
Full function point	0.53	17
Number of key classes	0.53	18
Feature point analysis	0.50	19
Requirements compliance	0.49	20
Number of faults remaining (error seeding)	0.46	21
Cause & effect graphing	0.43	22
Cohesion	0.42	23
Completeness	0.36	24

Table 5-15 Rates of OO Measures during the Design Phase

Unlike other OO measures, "Number of key classes" scores much lower. As a matter of fact, the lack of precisely defined counting rules for this measure drastically lowers the level of the criterion *Repeatability*, which leads to the lower rates of the measure.

The addition of OO measures significantly favors the family "Cohesion". The higher ranking of the family "Cohesion" is due mainly to the fact that the measure LCOM, which is the OO alternative of cohesion, scores much higher than "Cohesion" does. On the other hand, the statistics of the family "Functional size" does not change with the introduction of the OO measure "Number of key classes".

Family	Rate		
	Min	Median	Max
Fault detected per unit of size	0.73	0.74	0.75
Time taken to detect and remove faults	0.63	0.67	0.71
Requirements specification change requests	0.69	0.69	0.69
Class behavioral complexity	**0.69**	**0.69**	**0.69**
Class inheritance breadth	**0.69**	**0.69**	**0.69**
Class inheritance depth	**0.69**	**0.69**	**0.69**
Coupling	**0.69**	**0.69**	**0.69**
Error distribution	0.68	0.68	0.68
Class structural complexity	**0.67**	**0.67**	**0.67**
Cohesion	**0.67**	**0.67**	**0.67**
Reviews, inspections and walkthroughs	0.61	0.61	0.61
Software development maturity	0.60	0.60	0.60
Requirements traceability	0.56	0.56	0.56
Functional size	0.50	0.53	0.54
Requirements compliance	0.49	0.49	0.49

Estimate of faults remaining in code	0.46	0.46	0.46
Cause & effect graphing	0.43	0.43	0.43
Completeness	0.36	0.36	0.36

Table 5-16 Rates of OO Families during the Design Phase

5.4.2.3 Implementation Phase

The three top-ranked OO measures during the implementation phase are "Code defect density", "Design defect density", and "Fault density". Table 5-17 provides the rates and rankings of all OO measures applicable in this phase.

Measure	Rate	Ranking
Code defect density	0.83	1
Design defect density	0.75	2
Fault density	0.73	3
Fault-days number	0.71	4
Mutation score	0.71	5
Class coupling	0.69	6
Class hierarchy nesting level	0.69	7
Number of children (NOC)	0.69	8
Number of class methods	0.69	9
Requirements specification change requests	0.69	10
Lack of cohesion in methods (LCOM)	0.67	11
Weighted method per class (WMC)	0.67	12
Error distribution	0.65	13
Man hours per major defect detected	0.61	14
Reviews, inspections and walkthroughs	0.61	15
Software capability maturity model	0.60	16
Requirements traceability	0.56	17
Function point analysis	0.55	18
Full Function Point	0.53	19
Number of key classes	0.53	20
Feature point analysis	0.50	21
Requirements compliance	0.50	22
Number of faults remaining (error seeding)	0.47	23
Cause & effect graphing	0.40	24
Completeness	0.36	25

Table 5-17 Rates of OO Measures during the Implementation Phase

Table 5-18 lists the family rates during the implementation phase. The three top-ranked families are "Fault detected per unit of size", "Test adequacy", and "Time taken to detect and remove faults". The high ranking of the new family is a strong indicator that the efficiency of the test data plays an important role in software reliability determination.

Family	Rate		
	Min	Median	Max
Fault detected per unit of size	0.73	0.75	0.83
Test adequacy	**0.71**	**0.71**	**0.71**
Time taken to detect and remove faults	0.61	0.66	0.71
Requirements specification change requests	0.69	0.69	0.69
Class behavioral complexity	**0.69**	**0.69**	**0.69**
Class inheritance breadth	**0.69**	**0.69**	**0.69**
Class inheritance width	**0.69**	**0.69**	**0.69**
Coupling	**0.69**	**0.69**	**0.69**
Class structural complexity	**0.67**	**0.67**	**0.67**
Cohesion	**0.67**	**0.67**	**0.67**
Error distribution	0.65	0.65	0.65
Reviews, inspections and walkthroughs	0.61	0.61	0.61
Software development maturity	0.60	0.60	0.60
Requirements traceability	0.56	0.56	0.56
Functional size	0.50	0.53	0.55
Requirements compliance	0.50	0.50	0.50
Estimate of faults remaining in code	0.47	0.47	0.47
Cause & effect graphing	0.40	0.40	0.40
Completeness	0.36	0.36	0.36

Table 5-18 Rates of OO Families during the Implementation Phase

5.4.2.4 Testing Phase

Table 5-19 lists the rates of OO measures during the testing phase. The three top-ranked measures are "Failure rate", "Code defect density", and "Coverage factor". It should be noted that the importance of the OO missing measures in software reliability prediction decreases significantly in this phase. The top-ranked measures are failure and fault measures. This fact indicates that the OO design measures do not contribute much to software reliability prediction, even for OO systems.

Measure	Rate	Ranking
Failure rate	0.83	1
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
Mutation score	0.71	9
Requirements specification change requests	0.69	10
Test coverage	0.68	11
Class coupling	0.66	12
Class hierarchy nesting level	0.66	13
Error distribution	0.66	14

Number of children (NOC)	0.66	15
Number of class methods	0.66	16
Lack of cohesion in methods (LCOM)	0.65	17
Weighted method per class (WMC)	0.65	18
Man hours per major defect detected	0.63	19
Functional test coverage	0.62	20
Reviews, inspections and walkthroughs	0.61	21
Software capability maturity model	0.60	22
Requirements traceability	0.55	23
Number of faults remaining (error seeding)	0.51	24
Number of key classes	0.51	25
Function point analysis	0.50	26
Mutation testing (error seeding)	0.50	27
Requirements compliance	0.50	28
Full function point	0.48	29
Feature point analysis	0.45	30
Cause & effect graphing	0.44	31
Completeness	0.36	32

Table 5-19 Rates of OO Measures during the Testing Phase

Table 5-20 provides the OO family rates during the testing phase. The three top-ranked families are "Failure Rate", "Fault detected per unit of size", and "Fault-tolerant coverage factor". The measure "Coverage factor" introduces the family "Fault-tolerant coverage factor" during this phase. The rates and rankings in Table 5-20 indicate the fact that families related to failure ("Failure rate" and "Fault-tolerant coverage factor") play a more important role than families related to fault ("Fault detected per unit of size" and "Test adequacy"). These characteristics are also apparent from the extended structural representation in which failure measures are positioned closer to the indicator than the fault measures.

Family	Rate		
	Min	Median	Max
Failure rate	0.76	0.79	0.83
Fault detected per unit of size	0.75	0.75	0.83
Fault-tolerant coverage factor	**0.81**	**0.81**	**0.81**
Time taken to detect and remove faults	0.63	0.67	0.72
Test adequacy	0.62	**0.69**	**0.71**
Requirements specification change requests	0.69	0.69	0.69
Error distribution	0.66	0.66	0.66
Class behavioral complexity	**0.66**	**0.66**	**0.66**
Class inheritance breadth	**0.66**	**0.66**	**0.66**
Class inheritance width	**0.66**	**0.66**	**0.66**
Coupling	**0.66**	**0.66**	**0.66**
Class structural complexity	**0.65**	**0.65**	**0.65**
Cohesion	**0.65**	**0.65**	**0.65**
Reviews, inspections and walkthroughs	0.61	0.61	0.61
Software development maturity	0.60	0.60	0.60
Requirements traceability	0.55	0.55	0.55
Estimate of faults remaining in code	0.50	0.50	0.51
Functional size	0.45	**0.49**	**0.51**
Requirements compliance	0.50	0.50	0.50

Cause & effect graphing	0.44	0.44	0.44
Estimate of faults remaining per unit of size	0.40	0.40	0.40
Completeness	0.36	0.36	0.36

Table 5-20 Rates of OO Families during the Testing Phase

5.5 Summary

The discussion provided in this chapter is designed to incorporate in this report new measures generated by advances of software engineering which have occurred since the LLNL study was performed.

The missing measures discussed in this chapter were identified by experts. The measures covered the fault-tolerant computing environment, the mutation testing technique, the object-oriented development method, and one adaptation of "Function point". Eleven missing measures were initially identified. UMD eliminated the "Reliability trend indicator" because it is a reliability analysis approach rather than a missing measure.

The ranking criteria levels were assessed by UMD research team members according to the rational comparison to the experts' inputs, the software engineering literature, and field experts' inputs. The aggregation rates were calculated by applying the aggregation theory discussed in Chapter3.

The composition of families was revised to reflect the emergence of the missing measures. Two family groups corresponding to the non-OO technique and OO technique were identified. The rates and rankings of measures and families were reported separately for the two groups.

The impact analysis in this chapter shows that the introduction of the missing measures does not nullify everything described in Chapter 4. The "Coverage factor" is almost mandatory for the construction of any RPS for the real-time embedded system[8]. The "Mutation score" is highly recommended because it can reveal the percentage of failures that have not yet become manifest. It is a valuable support measure for the failure measures in the construction of RPSs.

FFP is an extension of function point in the field of real-time control systems for the purpose of functional size counting. The rate and ranking of this measure are lower than their predecessor, function point, because of the lack of experience with FFP despite the higher credibility level of FFP.

The OO measures cannot substitute for the traditional fault and failure measures in software reliability prediction. They provide means to clearly demonstrate the OO-specific design characteristics, such as the level of data abstraction, the depth of inheritance, and the degree of data encapsulation, etc. The rankings of these measures show that constructing the RPS directly from these measures is an arduous task.

As software engineering advances and new software engineering measures emerge, iterations of the study presented in this chapter should be performed to avoid obsolescence of the results presented in this chapter.

[SEL] SELAM, *Software Engineering in Applied Metrics.* Web: http://www.lmagl.qc.ca

[Arno73] Arnold, T. F., The Concept of Coverage and Its Effect on the Reliability Model of a Repairable System, *IEEE Transaction on Computers,* vol. c-22, no. 3, March 1973.

[Voas98] Voas, J. M., McGraw, G., *Software Fault Injection: Inoculating Programs Against Errors*, John Wiley & Sons, Inc., New York, 1998

[8] Most real-time embedded control systems are fault-tolerant systems.

[Lore94] Lorenz, M., Kidd, J., *Object-Oriented Software Metrics: A Practical Guide*, Prentice Hall, Inc. New Jersey, 1994

[Myer78] Myers, G. J., *Composite / Structural Design,* Van Nostrand Reinhold, New York, 1978.

[Khid94] Chidamber, S. R., Kemerer, F., A Metrics Suite for Object Oriented Design, *IEEE Transaction on Software Engineering,* Vol. 20, No. 6, June 1994

[Neal96] Neal, R. D., The Validation by Measurement Theory of Proposed Object-Oriented Metrics, *Dissertation,* Virginia Commonwealth University, Richmond, Va., 1996.

[Offu95] Offut, J., A Practical System for Mutation Testing: Help for the Common Programmer, *The Proceedings of the Twelfth International Conference on Testing Computer Software,* pp. 99-109, Washington, DC, June 1995.

CHAPTER 6 SUMMARY AND FUTURE RESEARCH

6.1 Summary

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

This study identified top-ranked software engineering measures in terms of their ability to predict software reliability. The top-ranked families were also identified in this study. A very strong indication of reliability might be obtained if the prediction is initiated from several compatible top-ranked families.

Chapter 2 investigated the relationships between measures and reliability prediction. It introduced three axes of classification important to the analysis of the measures. These axes are structural, life-cycle based and semantic. Semantic classification lead to the introduction of the concept of Family. The axes and axes' definitions are provided from Table 6-1 to Table 6-4. A graphical method was described for the purpose of structural representation. The creation of this graphical representation was shown to be another valuable tool in the analysis of a software engineering measure.

Axis	Definition
Indicator	Estimate or evaluation that provides a basis for decision-making. In this particular study, reliability is deemed an appropriate indicator.
Derived Measure	Any intermediate value which is neither an indicator nor a primitive measure.
Primitive Measure	Value resulting from the application of rules to a software attribute[1].

Table 6-1 Structural Classification Axis

Axis	Definition
Requirements Phase	Contains [IEEE610] phases: Concept and Requirements
Design Phase	Contains [IEEE610] phase: Design
Implementation Phase	Contains [IEEE610] phase: Implementation[2]
Testing Phase	Contains [IEEE610] phases: Test, Installation and Checkout Installation
Operation	Contains [IEEE610] phases: Operation, Maintenance, and Retirement

Table 6-2 Life-Cycle Classification Axis

[1] The user is referred to Chapter 2 (Section 2.1.1) for the explanation of the terms rule and software attribute.

[2] This phase contains unit testing.

Axis (Family)	Definition
Cause & effect graphing	*Cause & effect graphing*
Cohesion	*Cohesion*
Completeness	*Completeness*
Error distribution	*Error distribution*
Estimate of faults remaining in code	*Mutation testing (error seeding)*
	Number of faults remaining (error seeding)
Estimate of faults remaining per unit of size	*Bugs per line of code (Gaffney estimate)*
Failure rate	*Cumulative failure profile*
	Failure rate
	Mean time to failure
Fault detected per unit of size	*Code defect density*
	Design defect density
	Fault density
Fault-tolerant coverage factor	*Coverage factor*
Functional size	*Feature point analysis*
	Function point analysis
	Full function point[3]
Module structural complexity	*Cyclomatic complexity*
	Minimal unit test case determination
Requirements compliance	*Requirements compliance*
Requirements specification change requests	*Requirements specification change requests*
Requirements traceability	*Requirements traceability*
Reviews, inspections and walkthroughs	*Reviews, inspections and walkthroughs*
Software development maturity	*Software capability maturity model*
System architectural complexity	*Data flow complexity*
	Graph-theoretic static architecture complexity
	System design complexity
Test adequacy	*Mutation Score*
Test coverage	*Functional test coverage*
	Modular test coverage
	Test coverage
Time taken to detect and remove faults	*Fault-days number*
	Man hours per major defect detected

Table 6-3 Semantic Classification Axis for non-OO Systems

[3] Gray shadowing of a row is used to pinpoint missing measures, their rates, and the corresponding ranking.

Axis (Family)	Definition
Cause & effect graphing	*Cause & effect graphing*
Class behavioral complexity	*Number of class methods*
Class inheritance breadth	*Number of children (NOC)*
Class inheritance depth	*Class hierarchy nesting level*
Class structural complexity	*Weighted method per class (WMC)*
Cohesion	*Lack of cohesion in methods (LCOM)*
Coupling	*Class coupling*
Completeness	*Completeness*
Error distribution	*Error distribution*
Estimate of faults remaining in code	*Mutation testing (error seeding)*
	Number of faults remaining (error seeding)
Failure rate	*Cumulative failure profile*
	Failure rate
	Mean time to failure
Fault detected per unit of size	*Code defect density*
	Design defect density
	Fault density
Fault-tolerant coverage factor	*Coverage factor*
Functional size	*Feature point analysis*
	Function point analysis
	Full function point
	Number of key classes
Requirements compliance	*Requirements compliance*
Requirements specification change requests	*Requirements specification change requests*
Requirements traceability	*Requirements traceability*
Reviews, inspections and walkthroughs	*Reviews, inspections and walkthroughs*
Software development maturity	*Software capability maturity model*
Test adequacy	*Mutation score*
Test coverage	*Functional test coverage*
	Test coverage
Time taken to detect and remove faults	*Fault-days number*
	Man hours per major defect detected

Table 6-4 Semantic Classification Axis for OO Systems

This chapter also defined the concept of a Software Reliability Prediction System (RPS), which is a complete set of measures by which software reliability can be predicted. The RPS is composed of a root measure and several support measures as shown in Figure 6-1. The issue of selecting a software reliability prediction system was examined and a possible selection process suggested through Equation 2-8 to Equation 2-11.

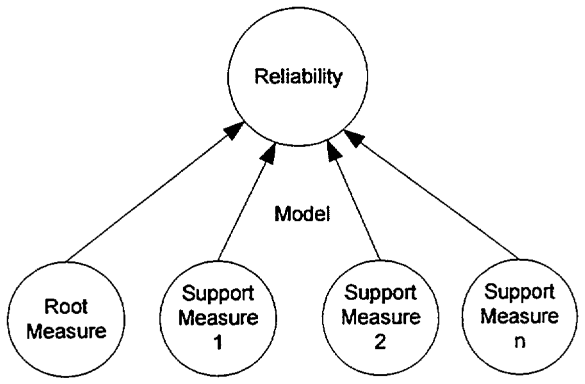


Figure 6-1 RPS

The point was made that the selection of a software reliability prediction system is a difficult task and that a simpler but related problem should be examined first: the problem of selecting single software engineering measures of high degree of validity which would be most relevant to reliability. The criteria for selection of the measures contain relevance, cost, benefit, validity, experience, credibility, and repeatability.

This chapter also showed that measures and interrelationships between measures need to be well understood before they are used. One should avoid the use of redundant measures and one should make sure that the set of measures at hand is complete from a software reliability prediction stand-point. Finally, once measures have been ranked separately, they need to be reinterpreted in the context of other measures.

Chapter 3 presented the methodology used to rank a pre-selected set of 30 software engineering measures (Table 6-5). The 30 measures were selected from the pool of measures identified in [LLNL98]. LLNL identified 78 software engineering measures related either directly or indirectly to software reliability and that might be appropriate to the study of digital I&C systems.

Bugs per line of code (Gaffney estimate)	Functional test coverage
Cause & effect graphing	Graph-theoretic static architecture complexity
Code defect density	Man hours per major defect detected
Cohesion	Mean time to failure
Completeness	Minimal unit test case determination
Cumulative failure profile	Modular test coverage
Cyclomatic complexity	Mutation testing (error seeding)
Data flow complexity	Number of faults remaining (error seeding)
Design defect density	Requirements compliance
Error distribution	Requirements specification change requests
Failure rate	Requirements traceability
Fault density	Reviews, inspections and walkthroughs
Fault-days number	Software capability maturity model
Feature point analysis	System design complexity
Function point analysis	Test coverage

Table 6-5 Pre-selected Software Engineering Measures

The methodology was based on the use of expert opinion elicitation to solicit the scores of software engineering measures. The scoring was performed with respect to seven ranking criteria: *Credibility, Repeatability, Cost, Benefit, Experience, Validation, and Relevance to Reliability*. The criteria are given in Table 6-6. The scoring was performed in terms of letter grades. A letter-conversion scheme translated the letter values to real numbers between 0 and 1. These numbers were then aggregated using an aggregation equation and a weighting scheme for the seven ranking criteria. The aggregated number served as the indicator of the "goodness" of the measure. A sensitivity analysis was further performed on all components of the analysis: letter-real conversion scheme, aggregation function form, weighting scheme. Since a priori one can not assess which aggregation scheme is correct, the purpose of such sensitivity analysis was to prove that the results obtained remain valid for a wide spectrum of aggregation schemes. All results of aggregation rates, rankings, and sensitivity analysis for the thirty pre-selected measures were presented in Chapter 4.

Set	Criterion	Definition
Quality Set	Credibility	Rates the measure in terms of its documented goals. A measure is considered to be *credible* if we judge it likely to support the specified goals.
	Experience	Rates the commercial *experience* in using the measure.
	Repeatability	A measure is considered *repeatable* if the repeated application of the measure by the same or different people results in similar results.
	Validation	Determines how extensively the measure has been validated.
Cost effectiveness	Benefit	Estimates the avoidance of costs that would be incurred if the measure was not used.
	Cost	Estimates the effort required to implement and use the measure.
Relevance	Relevance to Reliability	Scores the level at which the measure is relevant to software reliability prediction.

Table 6-6 Ranking Criteria and Their Definitions

Chapter 4 discussed the rates and rankings (with respect to software reliability prediction) obtained for the thirty measures selected at the beginning of the study. Some potential inconsistencies were examined and explained. The minimum, mean, and maximum rates for each family were calculated base on the rates of measures, composing the family.

The top-ranked measures were aggregated by an additive function with equal weights. These measures constitute the possible roots of software reliability prediction systems. Table 6-7 provides the top 3 measures, their semantic and structural classifications for each software development phase. Table 6-8 displays the top 3 families and their rankings.

Phase	Top-3 Measures	Semantic Classification	Structural Classification
Requirements Phase	Fault density	Fault detected per unit of size	Derived Measure
	Requirements specification change requests	Requirements specification change requests	Derived Measure
	Error distribution	Error distribution	Derived Measure
Design Phase	Design defect density	Fault detected per unit of size	Derived Measure
	Fault density	Fault detected per unit of size	Derived Measure
	Cyclomatic complexity	Module structural complexity	Derived Measure
Implementation Phase	Code defect density	Fault detected per unit of size	Derived Measure
	Design defect density	Fault detected per unit of size	Derived Measure
	Cyclomatic complexity	Module structural complexity	Derived Measure
Testing Phase	Failure rate	Failure rate	Derived Measure
	Code defect density	Fault detected per unit of size	Derived Measure
	Mean time to failure	Failure rate	Derived Measure

Table 6-7 Top-3 Measures Phase by Phase

It was proven however that rates and ranks remain relatively stable for different aggregation schemes. The sole exception was the change in equation form that signals noticeable changes in the results. This indicates that a more detailed study of the form of the aggregation equation should be performed. Stability, however, is reestablished when one analyzes the results by families rather than by single measure. The ranges of the correlation coefficients of the aggregation schemes are provided in Table 6-9.

Phase	Top-3 Families	Measures
Requirements Phase	Fault detected per unit of size	Fault density[4]
	Requirements specification change requests	Requirements specification change requests
	Error distribution	Error distribution
Design Phase	Fault detected per unit of size	*Design defect density Fault density
	Module structural complexity	*Cyclomatic complexity Minimal unit test case determination
	Time taken to detect and remove faults	*Fault-days number Man hours per major defect detected
Implementation Phase	Fault detected per unit of size	*Code defect density Design defect density Fault density
	Module structural complexity	*Cyclomatic complexity Minimal unit test case determination
	Time taken to detect and remove faults	*Fault-days number Man hours per major defect detected
Testing Phase	Failure rate	*Failure rate Mean time to failure Cumulative failure profile
	Fault detected per unit of size	*Code defect density Design defect density Fault density
	Module structural complexity	*Cyclomatic complexity Minimal unit test case determination

Table 6-8 Top-3 Families Phase by Phase

[4] It is important to note that other two elements of the family "Fault Detected Per Unit of Size", "Code defect density" and "Design defect density" are not applicable during the requirements phase.

* Highest ranked measure in family.

Correlation Coefficient Range	Rate Count
0 – 0.8	0
0.8 – 0.9	4
0.9 - 0.99	90
0.99 – 1.0	11

Table 6-9 Ranges of Correlation Coefficients

A final result of interest lies in the study of the impact of the ranking criteria. The study shows that optimal combinations of four, five and six criteria exist which generate a ranking that closely approximates the ranking obtained using seven criteria. Table 6-10 provides the top 5 criteria combinations through which a satisfactory aggregation can be obtained.

Rankings	Criteria Combination[5]
1	{Co., Be., Cr., Re., Exp., Va., Rel.}
2	{Co., Cr., Re., Exp., Va., Rel.}
3	{Cr., Re., Exp., Va., Rel.}
4	{Co., Be., Re., Exp., Va., Rel.}
5	{Co., Be., Re., Va., Rel.}

Table 6-10 Top-Ranked Criteria Combinations

The discussion provided in Chapter 5 was designed to incorporate current measures generated by advances in software engineering.

The missing measures discussed in this chapter were identified by experts. The measures covered the fault-tolerant computing environment, the mutation testing technique, the object-oriented development method, and one adaptation of "Function point". Eleven missing measures were initially identified. UMD eliminated the "Reliability trend indicator" because it is a reliability analysis approach rather than a missing measure. Table 6-11 lists 10 missing measures analyzed in this chapter.

FFP	Lack of cohesion in methods
Mutation score	Number of children
Coverage factor	Number of class methods
Class coupling	Number of key classes
Class hierarchy nesting level	Weighted method per class

Table 6-11 List of Missing Measures

[5] The abbreviations in this column "Co.", "Be.", "Cr.", "Re.", "Exp.", "Va.", and "Rel." stand for "Cost", "Benefit", "Credibility", "Repeatability", "Experience", "Validation", and "Relevance to Reliability", respectively.

The ranking criteria levels were assessed by UMD research team members using rational comparison with experts' input for analog measures, the software engineering literature, and field experts' input. The aggregation rates were calculated using the aggregation theory discussed in Chapter 3.

The composition of families was revised to reflect the emergence of the missing measures. Two family groups corresponding to the non-OO technique and OO technique were identified. The rates and rankings of measures and families were reported separately for the two groups. The top-3 measures and top-3 families for non-OO systems and OO systems are provided in Table 6-12 to Table 6-15.

Phase	Top-3 Measures	Semantic Classification	Structural Classification
Requirements Phase	Fault density	Fault detected per unit of size	Derived Measure
	Requirements specification change requests	Requirements specification change requests	Derived Measure
	Error distribution	Error distribution	Derived Measure
Design Phase	Design defect density	Fault detected per unit of size	Derived Measure
	Fault density	Fault detected per unit of size	Derived Measure
	Cyclomatic complexity	Module structural complexity	Derived Measure
Implementation Phase	Code defect density	Fault detected per unit of size	Derived Measure
	Design defect density	Fault detected per unit of size	Derived Measure
	Cyclomatic complexity	Module structural complexity	Derived Measure
Testing Phase	Failure rate	Failure rate	Derived Measure
	Code defect density	Fault detected per unit of size	Derived Measure
	Coverage factor	Fault-tolerant coverage factor	Derived Measure

Table 6-12 Top-3 Measures Phase by Phase for non-OO Systems

Phase	Top-3 Measures	Semantic Classification	Structural Classification
Requirements Phase	Fault density	Fault detected per unit of size	Derived Measure
	Requirements specification change requests	Requirements specification change requests	Derived Measure
	Error distribution	Error distribution	Derived Measure

Design Phase	Design defect density	Fault detected per unit of size	Derived Measure
	Fault density	Fault detected per unit of size	Derived Measure
	Fault-days number	Time taken to detect and remove faults	Derived Measure
Implementation Phase	Code defect density	Fault detected per unit of size	Derived Measure
	Design defect density	Fault detected per unit of size	Derived Measure
	Fault density	Fault detected per unit of size	Derived Measure
Testing Phase	Failure rate	Failure rate	Derived Measure
	Code defect density	Fault detected per unit of size	Derived Measure
	Coverage factor	Fault-tolerant coverage factor	Derived Measure

Table 6-13 Top-3 Measures Phase by Phase for OO Systems

Phase	Top-3 Families
Requirements Phase	Fault Detected per Unit of Size
	Requirements specification change requests
	Error Distribution
Design Phase	Fault Detected per Unit of Size
	Module Structural Complexity
	Time Taken to Detect and Remove Faults
Implementation Phase	Fault Detected per Unit of Size
	Module Structural Complexity
	Test Adequacy
Testing Phase	Failure Rate
	Fault Detected per Unit of Size
	Fault-tolerant Coverage Factor

Table 6-14 Top-3 Families Phase by Phase for non-OO Systems

Phase	Top-3 Families
Requirements Phase	Fault Detected per Unit of Size
	Requirements specification change requests
	Error Distribution
Design Phase	Fault Detected per Unit of Size
	Time Taken to Detect and Remove Faults
	Requirements specification change requests
Implementation Phase	Fault Detected per Unit of Size
	Test Adequacy
	Time Taken to Detect and Remove Faults
Testing Phase	Failure Rate
	Fault Detected per Unit of Size
	Fault-tolerant Coverage Factor

Table 6-15 Top-3 Families Phase by Phase for OO Systems

The analysis in this chapter shows that the introduction of the missing measures impacts the results obtained in Chapter 4 although not dramatically. It was established that the "Coverage factor" is almost mandatory for the construction of any RPS for the real-time embedded system[6]. The "Mutation score" is highly recommended because it can reveal the percentage of failures that have not manifested yet. It is a valuable support measure for the failure measures in the construction of RPSs.

FFP is an extension of function point in the field of real-time control systems for the purpose of functional size counting. The rate and ranking of this measure are lower than their predecessor, function point, because of the lack of experience with FFP despite its higher credibility level.

The OO measures provide means to clearly capture the OO-specific design characteristics, such as the level of data abstraction, the depth of inheritance, and the degree of data encapsulation, etc. The rankings of these measures show that constructing an RPS directly from these measures is an arduous task. They can not substitute for the traditional fault and failure measures.

It is clear that applying the top-ranked measures and families to establishing an RPS will lead to a significant improvement in predicting software reliability. However, current knowledge prevents the quantitative estimation of such improvement. Further experiments are required to investigate the quantitative reliability as a function of the RPS measures.

As software engineering advances and new software engineering measures emerge, iterations of the study presented in Chapter 5 should be performed to avert obsolescence of the results.

6.2 Future Research

The research presented in this report initiated a long-term study of the reliability prediction of software-based real-time digital systems. The following activities are recommended for future research.

1. The report, its methodology and the results presented should be peer-reviewed.

[6] Most real-time embedded control systems are fault-tolerant systems.

2. A larger set of expert opinion aggregation functions is recommended. UMD examined two forms of aggregation functions: additive and multiplicative. However, these only constitute a limited, though reasonable, set of aggregation functions typically in use for multiple objectives' aggregation [Keen76]. Further research on this topic is recommended.

3. Top-ranked software reliability prediction systems (RPSs) need to be fully identified for each life-cycle phase. The roots of top RPS's have been identified. It is now necessary to identify their support measures.

4. A reliability threshold under which an application is not acceptable needs to be determined for each phase. These thresholds are designed for the V&V process.

5. The research presented in this report needs to be validated through experiments. Validation includes 1) identifying applications for which the top-ranked RPSs are either available or recoverable, 2) predicting their operational reliability based on the RPSs for each phase of the life-cycle, 3) assessing the actual operational reliability, 4) comparing estimation and prediction. Preliminary research needs to be carried out to determine how many applications are required to perform this validation.

[IEEE610] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE, 1990.

[LLNL98] J. D. Lawrence, et al., *Assessment of Software Reliability Measurement Methods for Use in Probabilistic Risk Assessment*, Technical report UCRL-ID-136035, FESSP, Lawrence Livermore National Laboratory. 1998

[Keen76] R. L. Keeney, H. Raiffa, *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*, John Wiley & Sons, New York, 1976.

APPENDIX A SOFTWARE ENGINEERING MEASURES

This appendix contains brief descriptions of the measures used in this study (see Table A-1). The measures that were identified by LLNL but not considered in this study are listed in Table A-2. The measures are listed in alphabetic order. Table entries in bold-face type denote measures note described in the IEEE standard. Plain-text entries denote those measures described in the IEEE standard.

Bugs per line of code (Gaffney estimate)	Graph-theoretic static architecture complexity
Cause & effect graphing	Lack of cohesion in methods
Class coupling	Man hours per major defect detected
Class hierarchy nesting level	Mean time to failure
Code defect density	Minimal unit test case determination
Cohesion	Modular test coverage
Completeness	**Mutation score**
Coverage factor	**Mutation testing (error seeding)**
Cumulative failure profile	**Number of children**
Cyclomatic complexity	**Number of class method in a class**
Data flow complexity	Number of faults remaining (error seeding)
Design defect density	**Number of key classes**
Error distribution	Requirements compliance
Failure rate	**Requirements specification change requests**
Fault density	Requirements traceability
Fault-days number	**Reviews, inspections and walkthroughs**
Feature point analysis	**Software capability maturity model**
Full function point	**System design complexity**
Function point analysis	Test coverage
Functional test coverage	**Weighted method per class**

Table A-1 Measures

Bugs per line of code (Lipow estimate)
Bugs per line of code (Stetter estimate)
BVA model
Cost
Coupling
Data structure metrics
Defect indices
Design structure
Failure analysis using elapsed time
Fault density
Functional complexity
Graph-theoretic dynamic architecture complexity
Graph-theoretic generalized static architecture complexity
Independent process reliability
Input domain models (Brown-Lipow model)
Input domain models (Miller model)
Interface complexity
K-out-of-n model
Markov reliability model
Mean time to discover next K faults
Micro complexity
Multiversion software
Number of conflicting requirements
Number of entries & exits per module
Operation (functional) complexity
Operator complexity
Project initiation reliability prediction
Reliability block diagrams
Reliability growth function
Reliability prediction as a function of development environment
Reliability prediction as a function of software characteristics
Reliability prediction during software testing
Reliability prediction for the operational environment
RELY—Required software reliability
Residual fault count
Run reliability
Schedule
Software documentation
Software maturity index
Software process capability determination (SPICE)
Software purity level
Software release readiness
Software science reliability measure
Source listings
System operational availability
System performance reliability
Test accuracy (error seeding)
Testability analysis
Testing sufficiency

Table A-2 Measures not Considered in This Study

A.1 Bugs per Line of Code

Categories

Structural Level	derived
Life-Cycle Coverage	implementation[1] (artifact)

Application

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

Primitive

S	number of executable source statements

Definition

F_i	estimated number of faults in the *ith* module
F	total number of faults in the complete program

Implementation

Gaffney Model [1]

Empirical formulas are derived, yielding slightly different estimates for assembly code and high-level language (Jovial, in this case) code. Since the differences in the coefficients are smaller than the estimated errors in the basic theory, only the latter is given here. It is

$$F = 4.2 + 0.0015S^{4/3}$$

The power (4/3) implies that this estimate should be used for modules, not a complete program. If there are N modules in the program, and F_i is the estimated number of faults in the i^{th} module, then the number of faults in the complete program can be estimated to be

$$F = \sum_{i=1}^{N} F_i$$

[1] It is important to note that the measure is also applicable in the later stages. For instance, the measure Bugs per Line of Code is applicable from the implementation phase and remains applicable afterwards.

Remarks

These models are based on the Halstead measures, which are not considered particularly credible in this study, so should be used with considerable caution. In particular, the measures do not account for differences in the type of code (business data processing, scientific calculations, real time control, compilers, and so forth) or the degree of sophistication of the development organization (as measured, for example, by its CMM level).

There are rough industry estimates that, ignoring all else, one should expect 5–30 faults per thousand executable lines of code. The measures given here may be an improvement on this, and may therefore be suitable as a starting point for deriving a more accurate estimate. It is difficult to see any other value to them.

The measures are easy to calculate.

The measures are simplistic and ignore many aspects of the software and its development, so are not likely to be very accurate.

References

1. John E. Gaffney, "Estimating the number of faults in code," *IEEE Trans. Soft. Eng.* 10, 4 (July 1984), 459-464.

A.2 Cause & Effect Graphing

Categories

• ***Structural Level***	derived
• ***Life-Cycle Coverage***	requirement (artifact)

Application

Cause and effect graphing aids in identifying requirements that are incomplete and ambiguous. This measure explores the inputs and expected outputs of a program and identifies the ambiguities. Once these ambiguities are eliminated, the specifications are considered complete and consistent.

Cause and effect graphing can also be applied 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.

Primitives

List of causes

List of effects

Definitions

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

Implementation

A cause and effect graph is a formal translation of natural language specification into its input conditions and expected outputs. The graph depicts a combinatorial logic network.

To begin, identify all requirements of the system and divide them into separate identifiable entities. Carefully analyze the requirements to identify all the causes and effects in the specification. After the analysis is completed, assign each cause and effect a unique identifier. For example, E1 for effect one or I1 for input one.

To create the cause and effect graph, perform the following steps:

1. Represent each cause and each effect by a node identified by its unique number.

2. 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.

3. Annotate the graph with constraints describing combinations of causes and effects that are impossible because of semantic or environmental constraints.

4. Identify as an ambiguity 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.

The measure is computed as follows:

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

To derive test cases for the program, convert the graph into a limited entry decision table with "effects" as columns and "causes" as rows. For each effect, trace back through the graph to find all combinations of causes that will set the effect to be TRUE. Each such combination is represented as a column in the decision table. The state of all other effects should also be determined for each such combination. Each column in the table represents a test case.

References

1. *IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software*, IEEE, 1988.

2. Elmendorf, W. R. *Cause-Effect Graphs on Functional Testing.* Poughkeepsie: IBM Systems Development Division, TR-00.2487, 1973.

3. Myers, Glenford J. *The Art of Software Testing.* New York, Wiley-Interscience, 1979

4. Powell, B. P., ed. *Validation, Verification, and Testing Technique and Tool Reference Guide.* National Bureau of Standards Special Publication 500-93, 1982.

A.3 Class Coupling

Application

Class Coupling [3], which is also called coupling between object classes (CBO) in [1], is designed to examine how one class relates to other classes.

In practice, one wants to build systems that get their work done by requesting services from other objects. This means that one class can leverage the other classes' services. However, the level of this service availability should be limited to the level of complexity that one can handle. In other words, the amount of coupling should remain below a certain threshold.

Definitions

Class Coupling is defined as the sum total of other classes to which a class is coupled [1]. Intuitively, coupling refers to the degree of interdependence between parts of design. In ontological terms, "two objects are coupled if and only if at least one of them acts upon the other. X is said to act upon Y if the history of Y is affected by X, where history is defined as the chronologically ordered states that a thing traverses in time." [2, p 547]

Class Coupling relates to the notion that an object is coupled to another object if one of them acts on the other, i.e., methods of one use methods or instance variables[2] of another.

Some empirical observations with regard to *Class Coupling* are listed as follows:

1. Excessive coupling between object classes is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse it in another application.

[2] A name that allows one object (instance) to refer to another one. The instance variables make up an object's state data. In some literature instance variable is also called attribute.

2. In order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design, and therefore the maintenance is more difficult.

3. A measure of coupling is useful to determine how complex the testing of various parts of a design is likely to be. The higher the inter-object class coupling, the more rigorous the testing needs to be.

Implementation

Chidamber and Kemerer [1] advocated a formal representation of an object X as

$X = <x, p(x)>$

where

x	the substantial individual, namely, the entity that object X represents.
$p(x)$	the finite collection of *x's* properties

Thereafter, let $X=<x, p(x)>$ and $Y=<y, p(y)>$ be two objects,

$p(x) = \{Mx\} \cup \{Ix\}$

$p(y) = \{My\} \cup \{Iy\}$

where

$\{Mi\}$ is the set of methods and $\{Ii\}$ is the set of instance variables of object i.

Using the above definition of coupling, any action by {Mx} on {My} or {Iy} constitutes coupling, as does any action by {My} on {Mx} or {Ix}. Therefore, any evidence of a method of one object using methods or instance variables of another object constitutes coupling. Since objects of the same class have the same properties, two classes are coupled when methods declared in one class use methods or instance variables of the other class.

Reflecting the definitions and principles described above, the potential steps for evaluating the "Class Coupling" of a class X are

1. Extract the set of methods {Mx} and instance variables {Ix} of the class X.

2. For each class Y in the system other than the class X, extract the set of methods {My} and instance variables {Iy}.

3. Examine the behavior of {Mx}, if there is any action on {My} or {Iy}, skip step 4.

4. Examine the behavior of {My}, if there is any action on {Mx} or {Ix}, go to step 5, otherwise, go to step 6.

5. Class X and Y are coupled, increase the sum of the measure by 1. Then go to step 1.

6. Class X and Y are not coupled. Go to step 1.

Reference:

1. S. R. Chidamber, F. Kemerer, A Metrics Suite for Object Oriented Design, *IEEE Transactions on Software Engineering,* Vol. 20, No. 6, June 1994

2. I. Vessey and R. Weber, Research on Structured Programming: An Empiricist's Evaluation, *IEEE Transactions on Software Engineering,* vol. SE-10, 1984

3. M. Lorenz, J. Kidd, *Object-Oriented Software Metrics: A Practical Guide*, Prentice Hall, Inc. New Jersey, 1994

A.4 Class Hierarchy Nesting Level

Application

This measure assesses how many classes can potentially affect this class [1].

Definition

Classes are organized for inheritance purposes hierarchically in a tree structure, with the base or the topmost class called the root. The nesting level is the distance[3] in this hierarchy between the root and the class [2] [3]. For instance, in Figure A-1 the hierarchy nesting level of class F is 2. In case of multiple inheritance, this measure is the maximum length from the node to the root of the tree. This measure was also defined by Chidamber and Kemerer as Depth of Inheritance Tree (DIT) [1].

Large nesting numbers indicate a design problem, where developers are overly zealous in finding and creating objects. This will usually result in subclasses that are not specialization of all the superclasses. A subclass should ideally extend the functionality of the superclasses.

[3] The distance is measured as the number of classes (circles in Figure A-1) between the root and the class being measured, which includes root but excludes current class.

Implementation

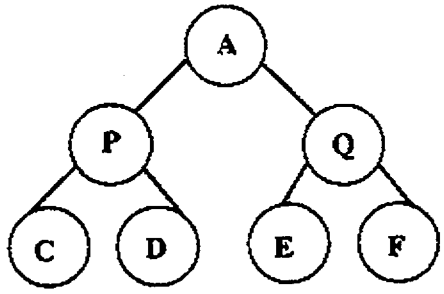


Figure A-1 An Example of Class Hierarchy Diagram

1. Construct the class hierarchy diagram of the class being counted (for example, Figure A-1 is the class hierarchy diagram that is applicable to class C, D, E, and F).

2. Count the number of ancestors of the class being counted. For example, if one tries to count the DIT value of class C, then the set of ancestors of class C is {P, A}. Therefore the DIT of class C is 2.

After the above steps, we can count the nesting level from the top of the class hierarchy or the bottom of the framework. Lorenz and Kidd suggested the threshold of this measure is 6. That is, if the value of this measure of a class is over 6, this class then needs to be reexamined or redesigned for over-design [2] [4].

Reference:

1. S. R. Chidamber, F. Kemerer, A Metrics Suite for Object Oriented Design, *IEEE Transactions on Software Engineering,* Vol. 20, No. 6, June 1994

2. B. Henderson-Sellers, Object-Oriented Metrics: Measures of Complexity, Prentice Hall Inc. 1996.

3. M. Lorenz, J. Kidd, *Object-Oriented Software Metrics: A Practical Guide*, Prentice Hall, Inc. New Jersey, 1994

4. R. D. Neal, The Applicability of Proposed Object-Oriented metrics to Developer Feedback in Time to Impact Development, *NASA/WVU Software IV&V Facility, Software Research Laboratory, Technical Report Series,* NASA-IVV-96-004, NASA, 1996.

A.5 Cohesion

Categories

- ***Structural Level*** derived
- ***Life-Cycle Coverage*** design (artifact)

Application

The goal of this measure is to indicate the "goodness" of a design.

Cohesion was introduced by Myers [1] in 1978 to serve as an indication of "goodness" of a design. There is a widespread belief that high cohesion yields better software designs, which in turn should lead to fewer faults, and thus higher reliability.

Cohesion falls in the category of "good design principles" and is widely believed to promote good software design. Determination of its values is frequently somewhat subjective. Additional discussion can be found in [3].

Definitions

Cohesion is defined to be "the degree of functional relatedness of processing elements within a single module."

Implementation

Decision tables that permit cohesion and coupling to be determined for a module are presented in Table A-3. This table is ordered from poorer forms of cohesion to preferred forms.

Table A-3 Determination of Cohesion [1]

Difficult to describe the module function	Y	N	N	N	N	N	N	N
Module performs more than one function	-	-	Y	Y	Y	Y	Y	N
Only one function performed per invocation	-	-	Y	N	N	N	Y	-
Each function has an entry point	-	-	N	-	-	-	Y	-
Module performs related class of functions	-	N	Y	Y	-	-	-	-
Functions are related to problem procedure	-	-	-	N	Y	Y	-	-
All of the functions use the same data	-	-	-	-	N	Y	Y	-
Coincidental cohesion	X	X						
Logical cohesion			X					

Classical cohesion				X				
Procedural cohesion					X			
Communicational cohesion						X		
Informational cohesion							X	
Functional cohesion								X

Remarks

Ref. 4 provides a theoretically more precise measure of functional cohesion. This was not included here because of doubts as to its practicality.

References

1. J. Myers, *Reliable Software through Composite Design*, Petrocelli (1975).

2. Han S. Son and Poong H. Seong, "Quantitative evaluation of safety-critical software at the early development stage: an interposing logic system software example."

3. Edward Yourdon and Larry L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice-Hall (1979).Rel. Eng. And System Safety 50, 3 (1995), 261-269.

4. James M. Bieman and Linda M. Ott, "Measuring functional cohesion," IEEE Trans. Soft. Eng. 20, 8 (August 1994), 644-657.

5. James M. Bieman and Byung-Kyoo Kang, "Measuring design-level cohesion," IEEE Trans. Soft. Eng. **24**, 2 (February 1998), 111-124.

A.6 Completeness

Categories

• ***Structural Level***	derived
• ***Life-Cycle Coverage***	requirement (artifact)

Application

This measure determines the completeness of the software specification during the requirements phase. Also, the values determined for the primitives associated with the completeness measure can be used to identify problem areas within the software specification.

Primitives

The completeness measure consists of the following primitives:

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 not using all conditions, or options or both.

B_{10} = number of decision points

B_{11} = number of condition options without processing

B_{12} = number of condition options

B_{13} = number of calling routines with parameters not agreeing with 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

Implementation

The completeness measure (CM) is the weighted sum of ten derivatives expressed as

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

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.

To calculate the completeness measure

1. The definitions of the primitives for the particular application must be determined.

2. The priority associated with the derived measure must also be determined. This prioritization would affect the weights used to calculate the completeness measure.

Each primitive value would then be determined by the number of occurrences related to the definition of the primitive.

Each derived measure is determined as follows:

$D_1 = (B_2 - B_1)/B_2$ = functions satisfactorily defined

$D_2 = (B_4 - B_3)/B_4$ = data references having an origin

$D_3 = (B_6 - B_5)/B_6$ = defined functions used

$D_4 = (B_8 - B_7)/B_8$ = referenced functions used

$D_5 = (B_{10} - B_9)/B_{10}$ = all condition options at decision points

$D_6 = (B_{12} - B_{11})/B_{12}$ = all condition options with processing at decision points are used

$D_7 = (B_{14} - B_{13})/B_{14}$ = calling routine parameters agree with the called routine's defined parameters

$D_8 = (B_{12} - B_{15})/B_{12}$ = all condition options that are set

$D_9 = (B_{17} - B_{16})/B_{17}$ = processing follows set condition options

$D_{10} = (B_4 - B_{18})/B_4$ = data references have a destination

References

1. IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE, 1988.

2. Murine, g. e. On Validating Software Quality Metrics. 4th *Annual IEEE Conference on Software Quality,* Phoenix, Arizona, Mar. 1985.

3. San Antonio, R, and Jackson, K. Application of Software Metrics During Early Program Phase. *Proceedings of the National Conference on Software Test and Evaluation,* Feb. 1982.

A.7 Coverage Factor

Application

This measure reflects the ability of the system to automatically recover from the occurrence of a failure during normal system operation.

Definitions

Coverage = Probability [system recovers | failure occurs], which means the probability that the system can recover from a failure. [1] [2]

Implementation

Several different models for predicting coverage in a fault-tolerant system are listed here. They include models for permanent, intermittent, and transient errors[4]. These models are described below:

A.7.1 Discrete time Markov chain

A high-level abstraction of a typical error-handling model may divide the process into sequential phases, forming a discrete time Markov chain (DTMC) [2]. According to [3], the matrix of eventual exit probabilities given an entry state is given by ($[I-P]^{-1}R$), where $P = [p_{ij}]$ denotes the transition probability matrix, and p_{ij} is the probability that the next state will be an error handling state j given that the current state is the error handling state i. $R = [r_{ij}]$ where r_{ij} is the probability of reaching an exit state j from an error handling state i.

[4] The terminology in this description is borrowed from [6]:

A *failure* occurs when the delivered service deviates from the specified service.

An *error* is that part of the system state which is liable to lead to failure.

The cause of an error is a *fault*.

Upon occurrence, a fault creates a *latent error*, which becomes effective when it is activated.

If an error, once activated, remains effective for a long time (relative to the time needed to detect and handle it), it may be considered *permanent*. If the error cycles relatively quickly between the active and latent states, it is considered *intermittent*. If the error, once activated, becomes latent, and remains latent for a long time, it is considered a *transient error*.

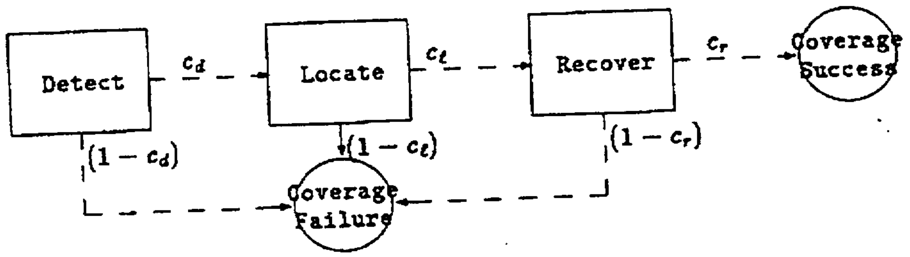


Figure A-2. Three phases of error handling from a permanent effective error

A three-phase error-handling model of detection, location, and recovery is presented in Figure A-2 (cited from [2]) with the assumption that errors are permanently effective, which means that once errors has been activated, they remain effective. Thus system coverage c is given by:

$$c = c_d \times c_l \times c_r$$

where c_d, c_l, c_r are the probabilities that the system reaches the "Locate" state from the "Detect" state, the "Recover" state from the "Locate" state, the "Coverage Success" state from the "Recover" state, respectively.

Some models, like the one proposed by the designer of CAST [4], combined the concept of transient restoration with a permanent recovery model (like the one in Figure A-2) into a single model shown in Figure A-3 (cited from [2]).

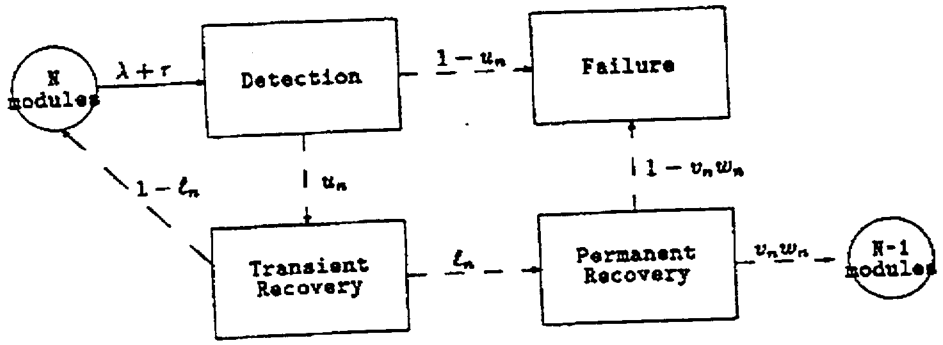


Figure A-3 CAST recovery model

In this CAST model, errors are activated at a total rate of λ (permanent rate) + τ(transient rate); they are detected with a probability ofu_n. Failure to detect the error is conservatively assumed to "pollute" the system with more errors resulting in a system failure. After detection, transient recovery is attempted; it is successful (if the error is transient) with probability 1- l_n, where l_n is the transient leakage. Unsuccessful transient recovery leads to permanent recovery where the cause of the error (the fault) is located with a probability of v_n and the system recovers with a probability of w_m[5].

A.7.2 Continuous time Markov chain (CTMC)

The only difference between CTMC and DTMC stems from the fact that the labels on the arcs of a CTMC model represent the rate at which the state changes occur, rather than simply probabilities as in the DTMC models.

An example of CTMC model is shown in Figure A-4 (cited from [2]). In this model, state A is entered on activation of the error. In this state A, the effective error begins to pollute the system with more errors (at rate ρ to state P). The effective error may be detected (state D) at the rate δ before affecting delivered service. The error may become latent (state B) before producing more errors or being detected. Global error detection mechanisms may detect an error (at rate ε with probability q) before the delivered service is affected [2].

The model shown in Figure A-4 can be used to represent either permanent or intermittent errors. Parameters α and β represent the rates at which an effective error becomes latent and vice versa. Thus if α and β are set to 0, this model represents permanent errors.

The coverage factor, that is, the probability of going from state A to state D is given by:

$$c = \frac{\delta}{\delta + \rho} + \frac{q\rho}{\delta + \rho}$$

[5] The parameters with the subscript n represent the parameters of the *nth* module.

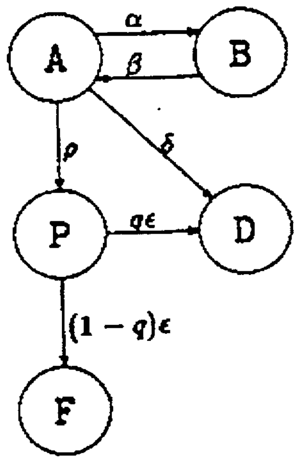


Figure A-4 A CTMC model

A.7.3 ESPN Models

ESPN (Extended stochastic Petri net) models combine both local and global timing in the same model. Please refer to [5] for a more detailed explanation. Figure A-5 (cited from [2]) represents a coverage model for a system that combines hardware and software error detection techniques. Errors that are not detected by hardware checking may be detected by a diagnostic program that is run periodically. The diagnostic unit is periodically executed even if there is no indication of an error in the unit, so as to detect latent errors in the system.

When a latent error is activated, a token is deposited in the place labeled *effective error*, enabling transition *T1*. Transition *T1* fires immediately, and deposits a token in the place labeled *perm*, with probability *p*, or in place *inter A* (active intermittent) with probability *i*, or place *trans* with probability *t*. If the effective error is permanent, its representative token remains in the corresponding place, just as a permanent error remains in the system. If the error is intermittent, the token will circulate between the *inter A* and *inter B* places. If the error is transient, its representative token will eventually move to the *trans gone* place. While the error is not benign, transitions *T2* or *T3* may be enabled. (An arc with a small circle signifies an *inhibitor*[6] *arc*.)

There is also a set of two places and transitions that represent the state of the running process: *norm* (normal operation), or *diag* (diagnostics). Initially, a token is present in the *norm* place, and cycles around through these two places. When the token is in the *norm* place, the effective error propagates within the system. The error may be detected by local (hardware) error detection mechanisms with probability *s* (transition *T2*); if not detected, the system is *polluted.* These additional errors may be detected by some global error detection mechanisms (transition *T4*) with probability *q*. When the system is undergoing diagnostics, transition *T3* is enabled and the error may be detected with

[6] An inhibitor arc is analogous with the negative gate in logic theory.

probability *c*. Each of the probabilities *c*, *q*, and *s* are conditional, and are conditioned in the event that an error exists.

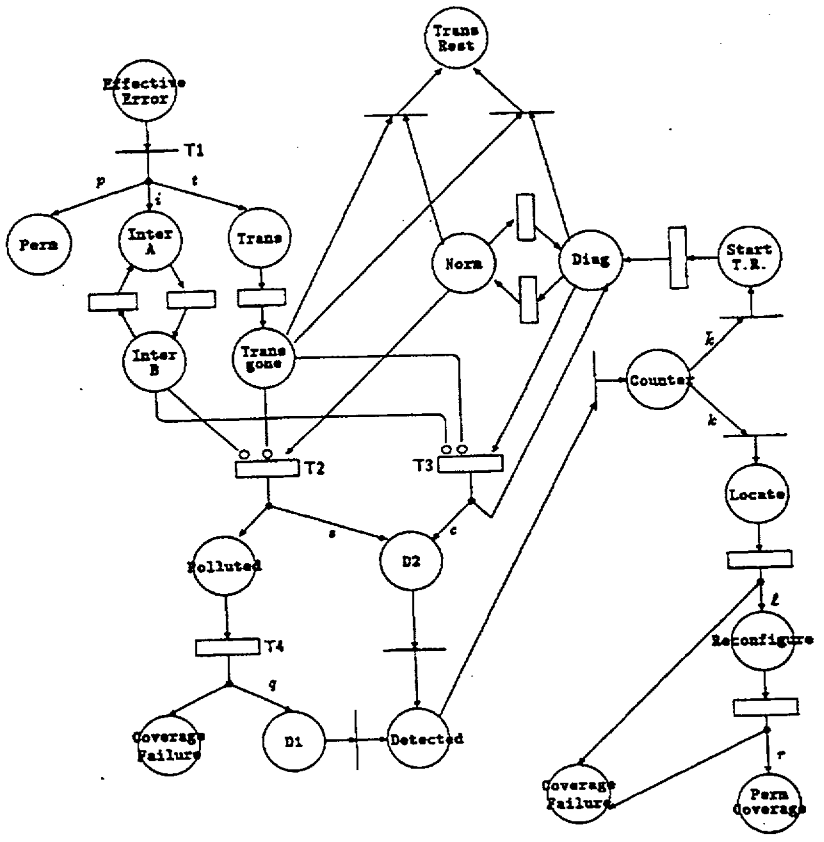


Figure A-5 An ESPN coverage model

Once an error is detected, the recovery processes may begin. The *counter* place counts the number of times an error is detected. The first *k* times it is detected, the system attempts some recovery, denoted by the place labeled *start T.R.*, after which the system undergoes diagnosis. If the transient error disappears in the meantime, the transient restoration (*trans rest*) exit is taken. If an error is detected more than *k* times, permanent recovery is commenced.

The methodology used solving an ESPN model depends upon the distributions chosen for the transition firing times. If all the firing times are assumed to be exponentially distributed, then the ESPN can be converted to a Markov chain for solution [5]. Under certain conditions, the net may be solved as a semi-Markov process or it may be simulated for solution.

Reference

1. 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, no. 3, March 1973

2. J. B. Dugan, K. s. Trivedi, "Coverage Modeling for Dependability Analysis of Fault-Tolerant Systems", *IEEE Transactions on Computers,* vol. 38, no. 6, June 1989

3. U. N. Bhat, *Elements of Applied Stochastic Processes,* 2nd ed. New York, Wiley, 1984

4. R. B. Conn, P. M. Merryman, and K. L. Whitelaw, "CAST – A complementary analytic-simulative technique for modeling fault-tolerant computing systems", *Proceedings AIAA Comput. Aerosp. Conf.*, Los Angeles, CA, Nov. 1977, pp. 6.1-6.27

5. M. K. Molloy, "Performance analysis using stochastic Petri nets," *IEEE Trans. Comput.*, vol. C-31, pp. 913-917, Setp. 1982.

6. J. Laprie, Dependable computing and fault-tolerance: Concepts and terminology," *Proc. Fifteenth Int. Symp. Fault-Tolerant Comput.,* July 1985, pp. 2-7.

A.8 Cumulative Failure Profile

Categories

- ***Structural Level*** derived
- ***Life-Cycle Coverage*** implementation(process)

Applications

The goal of this measure is to

1. Predicate reliability through the use of failure profiles;
2. Estimate additional testing time to reach an acceptability reliable system;
3. Identify modules and subsystems that require additional testing.

Primitives

f_i	total number of failures of a given severity level in a given time interval, i =1, ...

Implementation

Plot cumulative failures versus a suitable time base. The curve can be derived for the system as a whole, subsystems, or modules.

References

1. IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE, 1988.

2. Mendis, K. S., Quantifying Software Quality, *Quality Progress,* May 1982, pp 18-22.

3. Musa, J. D., Iannino, A., and Okumoto, K., *Software Reliability: Measurements, Prediction, Application,* New York, McGraw-Hill, 1987.

4. Shooman, M. L., *Software Engineering Design/Reliability/Management,* New York, McGraw Hill, 1983, pp 329-335.

5. Trachtenberg, M., Discovering How to Ensure Software Reliability, *RCA Engineer,* vol 27, no 1, Jan/Feb 1982, pp 53-57.

A.9 Cyclomatic Complexity

Categories

• ***Structural Level***	derived
• ***Life-Cycle Coverage***	design(artifact)

Application

This measure determines the structure complexity of a coded module. The use of this measure is designed to limit the complexity of a module, thereby promoting understandability of the module and the number of minimum logical testing path.

Definitions & primitives

N = number of nodes (sequential groups of program statements)

E = number of edges (program flows between nodes)

SN = number of splitting nodes (nodes with more than one edge emanating from it)

RG = number of regions (areas bounded by edges with no edges crossing)

Implementation

Using regions, or nodes and edges, a strongly connected graph of the module is required. A strongly connected graph is one in which a node is reachable from any other node: this is accomplished by adding an edge between the exit node and the entry node. Once the graph is constructed, the measure is computes as follows:

$$C = E - N + 1$$

The cyclomatic complexity is also equivalent to the umber of regions (RG) or the number of splitting nodes plus one (SN + 1). If a program contains an N-way predicate, such as a CASE statement with N cases, the N-way predicate contributes N-1 to the count of SN.

References

1. IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE, 1988.

2. Basili, V. R., Perricone, B. T., Software Errors and Complexity: An Empirical Investigation, *Communications of the ACM,* Jan. 1984.

3. McCabe, T. J., *Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric,* National Bureau of Standards Special Publication 500-99, Dec. 1982.

A.10 Data or Information Flow Complexity

Categories

• ***Structural Level***	derived
• ***Life-Cycle Coverage***	design(artifact)

Application

The measures Data or Information Flow Complexity are designed to measure the structural complexity or procedural complexity of a system.

This measure can be used to evaluate:

1. The information flow structure of large scale systems

2. The procedure and module information flow structure

3. The complexity of the interconnections between modules

Moreover, this measure can also be used to indicate the degree of simplicity of relationships between subsystems and to correlate total observed failures and software reliability with data complexity.

Primitives

lfi = local flows into a procedure

datain = number of data structures from which the procedure retrieves data

lfo = local flows from a procedure

dataout = number of data structures that the procedure updates

length = number of source statements in a procedure (excludes comments in a procedure)

Implementation

Determine the flow of information between modules or subsystems or both by automated data flow techniques, HIPO charts, etc.

A local flow from module A to B exists if one of the following holds true:

(1) A calls B,

(2) B calls A and A returns a value to B that is used by B, or

(3) Both A and B are called by another module that passes a value from A to B.

Values of primitives are obtained by counting the data flow paths directly into and out of the modules. The two intermediate derived measures fanin and fanout are defined as:

fanin = lfi + datain

faout = lfo + dataout

The information flow complexity (IFC) is IFC = $(\text{fanin} * \text{fanout})^2$

Weighted IFC = length * $(\text{fanin} * \text{fanout})^2$

References

1. IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE, 1988.
2. Bieman, J. W., Edwards, W. R., *Measuring Software Complexity,* Technical Report 83-5-1, Computer Science Department, University of Southwestern Louisiana, Feb. 1983.
3. Bieman, J. W., Edwards, W. R., *Measuring Data Dependency Complexity.* Technical Report 83-5-3, Computer Science Department, University of Southwestern Louisiana, Feb. 1983.
4. Bieman, J. W., Baker, A., Clites, P., et al, A Standard Representation of Imperative Language Program for Data collection and Software Measures Specification. *Journal of Systems and Software,* Dec. 1987.
5. Henry, S., Kafura, D., Software Structure Metrics Base on Information Flow, *IEEE Transactions on Software Engineering,* vol SE-7 (5), Sept. 1981.

A.11 Defect Density

Categories

- ***Structural Level*** derived

- *Life-Cycle Coverage* design(process)

Application

This measure indicates whether the inspection process is effective.

The defect density measure can be used after design and code inspections of new development or large block modifications. If the defect density is outside the norm after several inspections, it is an indication that the inspection process requires further scrutiny.

Primitives

Establish severity levels for defect designation.

D_i = total number of unique defects detected during the *ith* design or code inspection process.

I = total number of inspections.

KLSOD = in the design phase, the number of source lines of design statements in thousands.

KSLOC = in the implementation phase, the number of source lines of executable code and non-executable data declarations in thousands.

Implementation

Establish a classification scheme for severity and class of defect. For each inspection, record the product size, and the total number of unique defects.

For example, in the design phase, calculate the ratio.

$$DD = \frac{\sum_{i=1}^{I} D_i}{KSLOD}$$

This measure assumes that a structured design language is used. However, if some other design methodology is used, then some other unit of defect density has to be developed to conform to the methodology in which the design is expressed.

References

1. IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE, 1988.

2. Dobbins, J., Buck, R., Software Quality in the 80's. *Trends and Applications Proceedings,* IEEE Computer Society, 1981.

3. Dobbins, J., Buck, R., Software Quality Assurance, *Concepts, Journal of the Defense Systems Management College,* Autumn, 1982.

4. Fagan, Michael, E., Design and Code Inspection to Reduce Errors in Program Development. *IBM Systems Journal,* vol 15, no 3, Jul. 1976, pp 102-211.

A.12 Error Distribution

Categories

• ***Structural Level***	derived
• ***Life-Cycle Coverage***	requirements (artifact)

Application

This measure is designed to rank the failure modes. The search for the causes of software faults and failures involves the analysis of the defect data collected during each phase of the software development. Distribution of the errors allows ranking of the predominant failure modes.

Primitives

Error description notes the following points:

1. Associated faults
2. Types
3. Severity
4. Phase introduced
5. Preventive measure
6. Discovery mechanism, including reasons for earlier non-detection of associated faults.

Implementation

The primitives for each error are recorded and the errors are counted according to the criteria adopted for each classification. The number of errors is then plotted for each class. Examples of such distribution plots are shown in Figure A-6. In the three examples of Figure A-6, the errors are classified and counted by phase, by the cause, and by the cause for deferred fault detection. Other similar classification could be used such as the type of steps suggested to prevent the reoccurrence of similar errors or the type of steps suggested for earlier detection of the corresponding faults.

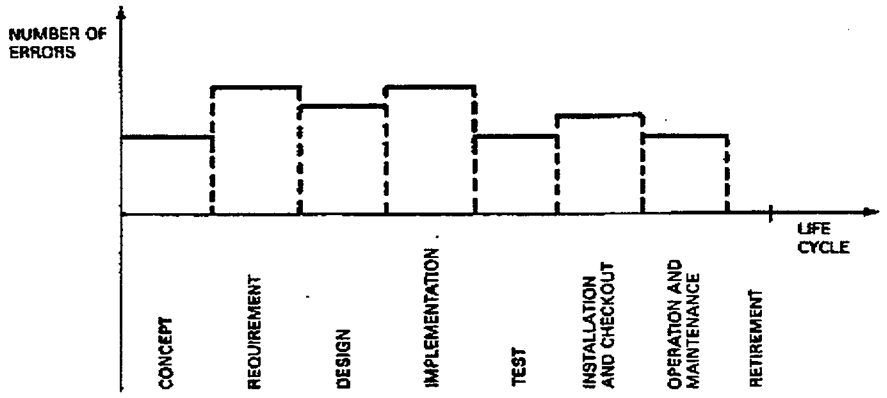
References

1. IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE, 1988.

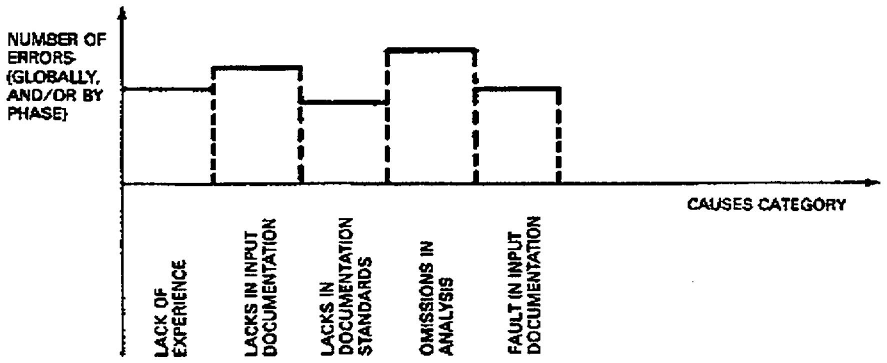
Figure A-6 Error Analysis

IEEE
Std 982.2-1988

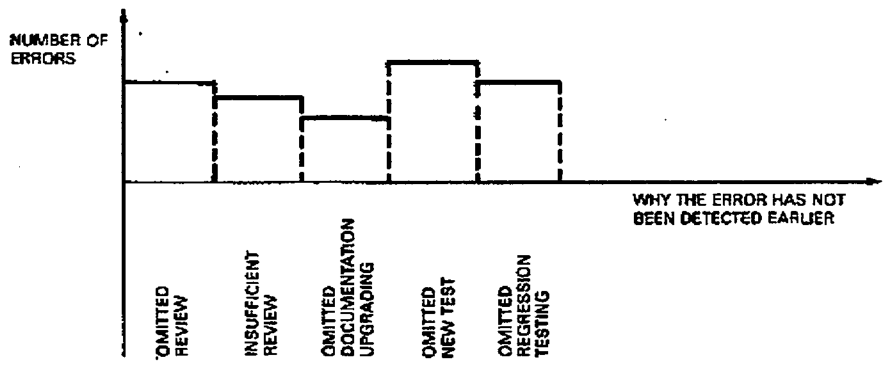
IEEE GUIDE FOR THE USE OF IEEE STANDARD DICTIONARY



(a) Error Distribution by Phase



(b) Error by Cause Category



(c) Suggested Causes for Error Detection Deferral

A.13 Failure Rate

Categories

- ***Structural Level*** derived
- ***Life-Cycle Coverage*** testing (artifact)

Application

This measure is used to indicate the growth in the software reliability as a function of test time.

Primitives

t_i = observed times between failure (for example, execution time) for a given severity level, i = 1, ...

f_i = number of failures of a given severity level in the ith time interval

Implementation

The failure rate $\lambda(t)$ at any point in time can be estimated from the reliability function, R(t), which in turn can be obtained from the cumulative probability distribution, F(t), of the time until the next failure using any of the software reliability growth models such as the non-homogeneous Poisson process

$$\lambda(t) = -\frac{1}{R(t)}\left[\frac{dR(t)}{dt}\right]$$

(NHPP) or a Bayesian type model. The failure rate is

where

$$R(t) = 1 - F(t)$$

References

1. IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE, 1988.

2. Miller, D. R., Exponential Order Statistic Models of Software Reliability Growth, *IEEE Transactions on Software Engineering,* vol SE-12, no 1, Jan. 1986.

3. Okumoto, K., A Statistical Method for Quality Control, *IEEE Transaction on Software Engineering,* vol SE-11, no 12, Dec. 1985, pp 1424-1430.

4. Gingpurwalla, N., Soyer, R., Assessing (Software) Reliability Growth Using a Random Coefficient Autoregressive Process and Its Ramifications, *IEEE Transactions on Software Engineering,* vol SE-11, no 12, Dec. 1985, pp 1456-1463.

5. Yamada, S., Osaki, S., Software Reliability Growth Modeling: Models and Applications, *IEEE Transactions on Software Engineering,* vol SE-11, no 12, Dec. 1985, pp 1431-1437.

A.14 Fault Density

Categories

• ***Structural Level***	derived
• ***Life-Cycle Coverage***	requirements (artifact)

Application

This measure indicates the fault density of a specific program for the given severity levels. In particular this measure can be used to perform the following functions:

1. Predicate remaining faults by comparison with expected fault density.
2. Determine if sufficient testing has been completed based on predetermined goals for severity class.
3. Establish standard fault densities for comparison and prediction.

Primitives

Establish the severity levels for failure designation.

F = total number of unique faults found in a given time interval resulting in failures of a specified severity level

KSLOC = number of source lines of executable code and non-executable data declarations in thousands.

Implementation

Establish severity, failure types and fault types.

1. Failure types might include I/O (input, output, or both) and user. Fault types might result from design, coding, documentation, and initialization.
2. Observe and log each failure.
3. Determine the program fault(s) that caused the failure. Classify the faults by type. Additional failures may be found resulting in total faults being greater than the number of failures observed, or one fault may manifest itself by several failures. Thus, faults and failure density may both be measured.

4. Determine total lines of executable and non-executable data declaration source code (KLSOC).

5. Calculate the fault density for a given severity levels as F_d = F/KLSOC.

References

1. IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE, 1988.

2. Bowen, J. B., A Survey of Standards and Proposed Metrics for Software Quality Metrics, *Computer,* 1979, 12 (8), pp 37-41.

3. Shooman, M. L., *Software Engineering Design/Reliability/Management,* New York, McGraw Hill, 1983, pp 325-329.

A.15 Fault-Days Number

Categories

• ***Structural Level***	derived
• ***Life-Cycle Coverage***	testing (process)

Application

This measure represents the number of days that faults spend in the software system from their creation to their removal.

Primitives

Phase when the fault was introduced in the system.

Date when the fault was introduced in the system.

Phase, date and time when the fault is removed.

FDi = fault days for the ith fault.

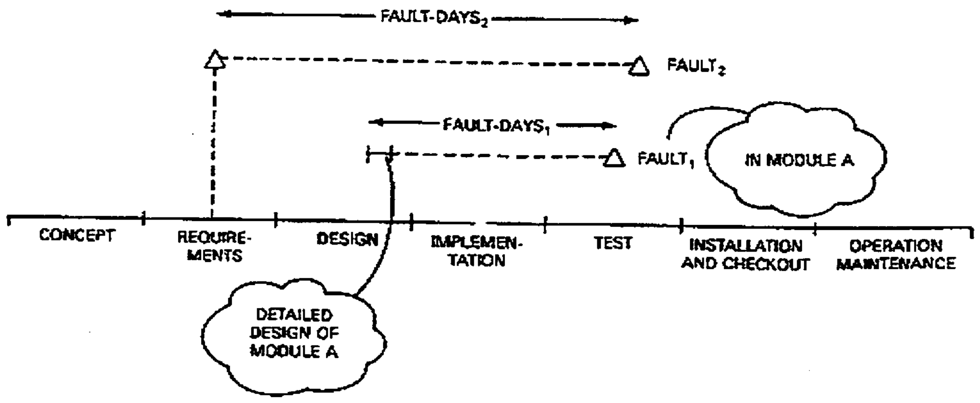
Note: For more meaningful measures, the time unit can be made relative to test time to operational time.

Implementation

For each fault detected and removed, during any phase, the number of days from its creation to its removal is determined (fault-days).

The fault-days are then summed for all faults detected and removed, to get the fault-days number at system level, including all faults detected/removed up to the delivery date. In cases when the creation date for the fault is not known, the fault is assumed to have been created at the middle of the phase in which it was introduced.

In Figure A-7 the fault-days for the design fault for module A can be accurately calculated because the design approval date for the detailed design of module A is known. The fault introduced during the requirements phase is assumed to have been created at the middle of the requirement phase because the exact knowledge of when the corresponding piece of requirement was specified, is not known.



The measure is calculated as shown is Figure A-7.

Figure A-7 Calculation of Fault-Days

References

1. IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE, 1988.

2. Mills, Harland, D., Software Development, *IEEE Transactions on Software Engineering*, vol SE-4, no 4, Dec. 1976.

A.16 Feature Point Analysis

Categories

• ***Structural Level***	derived
• ***Life-Cycle Coverage***	requirements (artifact)

Application

This measure is designed to determine the functional size of software, especially for real-time or embedded software applications.

This measure can be used starting in the requirements specification phase and throughout the remainder of the software life cycle as a basis to assess software quality, costs, documentation and productivity. Feature Points are gaining acceptance as a measure of software size, especially for real-time or embedded software applications. Feature Points measure the size of an entire application as well as that of software enhancements, regardless of the technology used for its development and / or maintenance.

Primitives

ILF, EIF, EI, EO, EQ, Algorithms;

The 14 software characteristics: Data Communications, distributed data processing, performance application, heavily used configuration, transaction rate, on-line data entry, end-user efficiency, complex processing, reusability, installation ease, operational ease, multiple sites, facilitate change;

The value adjustment factor

Implementation

Feature Point Analysis measures software size by counting six distinct software attributes. Two of these address the software program data requirements of an end user and are referred to as Data Functions (items 1 and 2 below). The remaining four address the user's need to access data and are referred to as Transactional Functions (items 3, 4, 5 and 6 below).

1. Internal Logical Files (ILF) (logical groups of data maintained in an application)
2. External Interface Files (EIF) (logical groups of data used by one application but maintained by another application)
3. External Inputs (EI) (which maintain internal logical files)
4. External Outputs (EO) (reports and data leaving the application)
5. External Inquiries (EQ) (combination of a data request and data retrieval)
6. Algorithms (bounded computational problems that are included within a software component)

These six attributes are rated using a single weight as shown in Table A-4.

Table A-4 Computing Feature Point Measure

Measurement Parameters	**Count**	**Weight**	**Weighted Value (Count x Weight)**
Number of Internal Logical		7	

Files			
Number of External Interface Files		7	
Number of External Inputs		4	
Number of External Outputs		5	
Number of External Inquires		4	
Number of Algorithms		3	
		Total Feature-Point Count:	

Thus, an Unadjusted Feature Point is defined as follows:

Unadjusted Feature Point = (External Inputs X Weight) +

(External Outputs X Weight) +

(Logical Internal Files X Weight) +

(Logical Interface Files X Weight) +

(Inquiries X Weight) +

(Algorithms X Weight)

The Unadjusted Feature Point Count is modified by a Value Adjustment Factor that assesses the design characteristics of the software under consideration. The Unadjusted Feature Point count is multiplied by the Value Adjustment Factor that considers the system's technical and operational characteristics and is calculated by answering questions about the following 14 software characteristics:

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. **On-Line Data Entry.** On-line data entry and control information functions are provided in the application.

7. **End-User Efficiency.** The on-line functions provided emphasize a design for end-user efficiency.

8. **On-Line Update.** The application provides on-line update for the internal logical files.

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 to be installed at multiple sites for multiple organizations.

14. **Facilitate Change.** The application has been specifically designed, developed and supported to facilitate change.

The Function Point Counting Practices Manual gives specific guidelines for determining the "Degree of Influence" from 0 to 5 for each of fourteen "general system characteristics." These are also suggested for use in computing the Value Adjustment Factor for Feature Points. This calculation provides us with the Adjusted Feature Point count.

The following formula converts the total of the Degrees of Influence assigned above to the Value Adjustment Factor:

Value Adjustment Factor = Total Degree of Influence X .01 + .65

The Value Adjustment Factor measures software design characteristics and changes significantly only when design changes are made to the software.

Since such design changes occur infrequently, the Value Adjustment Factor is the most stable part of the Feature Point count. The Value Adjustment Factor is then applied to the Unadjusted Feature Points (the total of the weighted counts) to establish the Adjusted Feature Point Count. This represents the size of the application and can be used to compute several measures as discussed in the Interpretation section of this document.

Adjusted Feature Points = (Unadjusted Feature Points) X

(Value Adjustment Factor)

This is referred to as the Feature Point Count, denoted by Feature Point, in the following.

The measure of interest is that of software quality, defined by

Software Quality = 1 - [(Number of defects found) / Feature Point]

Each of the functional components of a software system is analyzed in this way and each component's Feature Point score is added to the total to derive a total Feature Point Count for the application.

Once the Feature Point Count has been calculated, it can be used as a measure of software quality as defined above. The software quality measure can be computed during each phase of the software life cycle. The software quality measure is judged to be better as the value computed for the measure approaches the value 1.

Remarks

The software quality measure based on Feature Point Analysis allows the software production process to be quantified in terms of the quality of the software produced and it is easily measured. Organizations that have adopted Feature Point Analysis as a software measure claim to realize many benefits, including improved quality, improved project estimating, better understanding of project and maintenance productivity, more disciplined management of changing project requirements, and user requirements.

Feature Points were originally an extension to the Function Point measure and have similar disadvantages to Function Points. To use Feature Points effectively, training in the calculation of Feature Points is required.

Feature Point Analysis has proven to be an accurate technique for sizing, documenting, and communicating a system's capabilities. It has been successfully used to evaluate the functions of real-time and embedded code systems, such as robot-based warehouses and avionics, as well as traditional data processing. As computing environments become increasingly complex, it can potentially prove to be a valuable tool that helps to measure the software quality.

The measure is easy to understand throughout the software life cycle phases, even where direct measurement of reliability is not possible. Even when more direct measures of software quality and reliability are available, the software quality measure provides an additional perspective as to the effectiveness of the design and implementation processes and thus, has the potential of adding credibility to the product.

References

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

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

3. Heller, R., "An Introduction to Function Point Analysis," Newsletter from Process Strategies, Inc., No. 4, Fall 1996, (http://www.processtrat.com/)

4. Pressman, R., *Software Engineering—A Practitioner's Approach*, McGraw-Hill, 1992.

A.17 Full Function Point

Application

Full Function Point (FFP) was proposed with the aim of offering a functional size measure specifically adapted to real-time software. [1] [2]

Definition

FFP is a functional measure based on the standard function point analysis (FPA) technique. It was designed for both management information system (MIS) and real-time software. Since FFP is an extension of the standard FPA, all rules of FPA are included in the FFP counting process. However, a small number of subsets of FPA rules dealing with control concepts have been expanded considerably. The control aspect of real-time control software is addressed by new function types.

FFP introduces two new Control Data Function Types named Updated Control Group (UCG) and Read-only Control Group (RCG). A UCG is a group of control data updated by the application. The control data live for more than one transaction. A RCG is a group of control data used, but not updated, by the application being counted. The control data live for more than one transaction, too.

The following four new Control Process Function Types address the sub-processes of real-time software.

1. External Control Entry ECE: processes control data coming from outside the application's boundary[7].

2. External Control Exit ECX: An ECX is a unique sub-process. It is identified from a functional perspective. The ECX process control data goes outside the application's boundary.

3. Internal Control Read ICR. An ICR is a unique sub-process. It is identified from a functional perspective. The ICR reads control data.

4. Internal Control Write ICW: is a unique sub-process. The ICW writes control data.

The Control Data Function Types, which contribute to the overall size, fall into two categories.

[7] The boundary of a piece of software is the conceptual frontier between this piece and the environment in which it operates, as it is perceived externally from the perspective of its users. The boundary allows the measurer to distinguish, without ambiguity, what is included inside the measured software from what is part of the measured software's operating environment.

1. Multiple occurrences Control Data Function Types, which can be either updated or read only by the process. These are similar to the Internal Logical files and External Interface files counted for FPA.

2. Single occurrence Control Data Function Types. These data groups may be maintained by the processes (Updated Control Group-UCG) or only read by the processes (Read only Control Group-RCG). The single occurrence data groups contain all instances of single control values used by the processes. There may be only one instance of a UCG or RCG per application.

Implementation

The FFP involves applying a set of rules and procedures to a given piece of software, as it is perceived from the perspective of its inherent functional user requirements [1] [2]. An overview of the counting by FFP is summarized in Figure A-8 [3]. After identifying the counting boundary, FFP analysis includes the following steps:

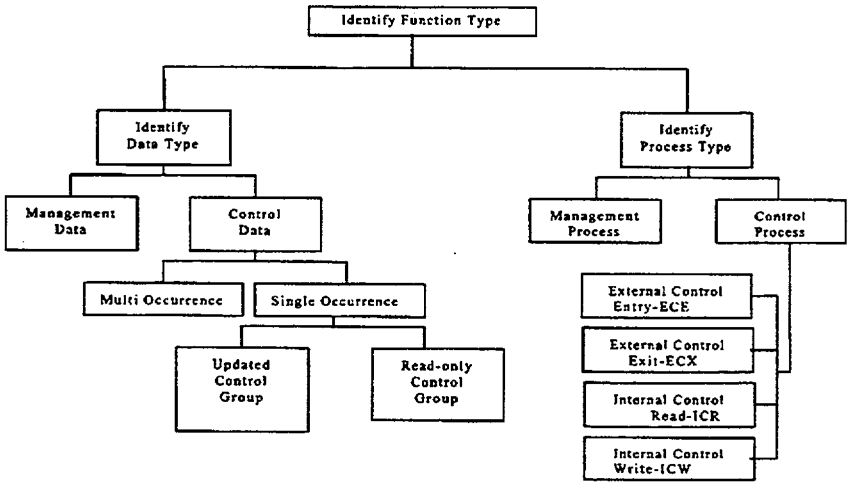


Figure A-8 Overview of Full Function Point Counting

1. Counting Management Function Types. For the Management Function Types, namely, ILF, EIF, EI, EO, and EQ, the counting procedure and point assignment rules are unchanged in FFP. This step covers the counting of Management Data and Management Process shown in Figure A-8.

2. Counting Control Data Function Types. Control Data Function Types, include UCG and RCG, are classified into single occurrence groups of data and multiple occurrence groups of data. The point of these kinds of data groups is determined by the number of Data Element Types (DETs) and Record Elements Types (RETs) and the corresponding complexity matrix (please refer to the description of FPA in this appendix and [4] for more detailed explanation). The point of single occurrence Control Data Function Types depends only on the number of DETs.

3. Counting Control Process Function Types. The number of points assigned to Control Process Functions Types, namely, ECE, ECX, ICW, and ICR, depends on the number of DETs. Once the number of DETs is determined, the number of points is determined by the matrix in [5], pp. 14.

4. Determine the unadjusted FFP and the adjust factor. The determination of the unadjusted FFP count and the technical complexity factor is the same as that of FPA.

References

1. SELAM, Software Engineering in Applied Metrics. web: http://www.lmagl.qc.ca

2. UQAM, Software Engineering Management Research Laboratory, Universite' du Quebec a Montreal. web: http://www.lmagl.qc.ca

3. 1999, N. Kececi, M. Li, C. Smidts, "Function Point Analysis: An application to a nuclear reactor protection system," *Probabilistic Safety Assessment - PSA' 99,* August 22-25, 1999, Washington, DC.

4. IFPUG (1994). *Function Point Counting Practices Manual, Release 4.0*, International Function Point Users Group – IFPUG, Westerville, Ohio, 1994

5. D. St-Pierre, etc. *Full Function Points: Counting Practices Manual, Technical Report 1997-04*, Software Engineering Management Research laboratory and Software Engineering Laboratory in Applied Metrics (SELAM), September, 1997.

A.18 Function Point Analysis

Categories

• ***Structural Level***	derived
• ***Life-Cycle Coverage***	requirements (artifact)

Application

This measure is designed to determine the functional size of software.

This measure can be used starting in the requirements specification phase and throughout the remainder of the software life cycle as a basis to assess software quality, costs, documentation and productivity. Function points have gained acceptance as a primary measure of software size. Function points accurately measure the size of an entire application as well as that of software enhancements, regardless of the technology used for its development and/or maintenance.

Primitives

ILF, EIF, EI, EO, EQ;

The 14 software characteristics: Data Communications, distributed data processing, performance application, heavily used configuration, transaction rate, on-line data entry, end-user efficiency, complex processing, reusability, installation ease, operational ease, multiple sites, facilitate change;

The value adjustment factor

Implementation

Function Point Analysis measures software size by counting five distinct software attributes. Two of these address the software program data requirements of an end user and are referred to as Data Functions (items 1 and 2 below). The remaining three address the user's need to access data and are referred to as Transactional Functions (items 3, 4, and 5 below).

1. Internal Logical Files (logical groups of data maintained in an application)
2. External Interface Files (logical groups of data used by one application but maintained by another application)
3. External Inputs (which maintain internal logical files)
4. External Outputs (reports and data leaving the application)
5. External Inquiries (combination of a data request and data retrieval)

These five attributes are rated as having low, average, or high importance in the analysis. The rating matrix for inputs is shown in the table and illustrates the rating process. The importance of each component is then weighted according to Table A-5.

Table A-5 Computing Function Point Measure

Measurement Parameters	**Count**	**Low**	**Average**	**High**	**Weighted Value (Count x Weight)**
Number of Internal Logical Files		7	10	15	
Number of External Interface Files		5	7	10	
Number of External Inputs		3	4	6	
Number of External Outputs		4	5	7	

Number of External Inquires		3	4	6	
				Total Function Point Count:	

The total Function Point count is based upon an Unadjusted Function Point Count that is defined as follows:

Unadjusted Function Points = (Internal Logical Files X Weight) +

(External Interface Files X Weight) +

(External Inputs X Weight) +

(External Outputs X Weight) +

(External Inquiries X Weight)

The Unadjusted Function Point Count is modified by a Value Adjustment Factor that assesses the design characteristics of the software. The Unadjusted Function Point count is multiplied by the Value Adjustment Factor. This factor considers the system's technical and operational characteristics and is calculated by answering questions about the following 14 software characteristics:

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. **On-Line Data Entry.** On-line data entry and control information functions are provided in the application.

7. **End-User Efficiency.** The on-line functions provided emphasize a design for end-user efficiency.

8. **On-Line Update.** The application provides on-line update for the internal logical files.

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 to be installed at multiple sites for multiple organizations.

14. **Facilitate Change.** The application has been specifically designed, developed and supported to facilitate change.

The Function Point Counting Practices Manual gives specific guidelines for determining the "Degree of Influence" from 0 to 5 for each of fourteen "general system characteristics." Each of these factors is scored based on their influence on the system being counted. The resulting score will increase or decrease the Unadjusted Function Point count by 35%. This calculation provides us with the Adjusted Function Point count.

The following formula converts the total of the Degrees of Influence assigned above to the Value Adjustment Factor:

Value Adjustment Factor = Total Degree of Influence X .01 + .65

The Value Adjustment Factor measures software design characteristics and changes significantly only when design changes are made to the software.

Since such design changes occur infrequently, the Value Adjustment Factor is the most stable part of the Function Point count. The Value Adjustment Factor is then applied to the Unadjusted Function Points (the total of the weighted counts) to establish the Adjusted Function Point Count. This represents the size of the application and can be used to compute several measures as discussed in the Interpretation section of this document.

Adjusted Function Points = (Unadjusted Function Points) X

(Value Adjustment Factor)

This is referred to as the Function Point Count, denoted by *FP*, in the following. The measure of interest is that of software quality, defined by

Software Quality = 1 - [(Number of defects found) / *FP*]

Each of the functional components of a software system is analyzed in this way and each component's FP score is added to the total to derive a total Function Point count for the application.

Once the Function Point Count, *FP*, has been calculated, it can be used as a measure of software quality as defined above. The software quality measure can be computed during each phase of the software life cycle. The software quality measure is judged to be better as the value computed for the measure approaches the value 1.

Remarks

The software quality measure based on Function Point Analysis allows the software production process to be quantified in terms of the quality of the software produced and it is easily measured. Organizations that have adopted Function Point Analysis as a software measure claim to realize many benefits including improved: project estimating; understanding of project and maintenance activity productivity, management of changing project requirements and user requirements. Function Points can be converted to an equivalent size in terms of lines of source code (LOC).

The following table, adapted from Ref. 4, provides a rough estimate of the average number of lines of code required to build one Function Point (FP) using various types of computer programming languages:

Programming Language	**LOC / FP (Average)**
Assembly Language	300
COBOL	100
FORTRAN	100
Pascal	90
Ada	70
Object-Oriented Languages	30
Fourth Generation Languages	20
Code Generators	15

Additional information on the conversion of Function Points to lines of code can be found in Ref. 2.

Function Points were originally designed to be applied to business information processing type applications. Capers Jones proposed extensions to the Function Point measure that may enable the concept to be applied to scientific and real-time application software. To use Function Points effectively, training in the calculation of Function Points is required.

Function Point Analysis has proven to be an accurate technique for sizing, documenting and communicating a system's capabilities. It has been successfully used to evaluate the functions of real-time and embedded code systems, such as robot based warehouses and avionics, as well as traditional data processing. As computing environments become increasingly complex, it is proving to be a valuable tool that accurately reflects the systems we deliver and maintain.

The measure is easy to understand throughout the software life cycle phases, even where direct measurement of reliability is not possible. Even when more direct measures of software quality and reliability are available, the software quality measure provides an additional perspective as to the effectiveness of the design and implementation processes and thus, has the potential of adding credibility to the product.

References

1. Jones, C., *Programming Productivity*, McGraw-Hill, Inc., 1986.
2. Jones, C., *Applied Software Measurement*, McGraw-Hill, Inc., 1991.
3. Heller, R., An Introduction to Function Point Analysis, Newsletter from Process Strategies, Inc., No. 4, Fall 1996, (http://www.processtrat.com/)
4. Pressman, R., *Software Engineering—A Practitioner's Approach*, McGraw-Hill, 1992.

A.19 Function Test Coverage

Readers are referred to the Section A.26 for the description of this measure.

A.20 Graph-Theoretic Complexity for Architecture

Categories

• ***Structural Level***	derived
• ***Life-Cycle Coverage***	design (artifact)

Application

Complexity measures can be applied early in the product cycle for development trade-offs as well as to assure system and module comprehensibility adequate for correct and efficient maintenance. Many system faults are introduced in the operational phase by modifications to systems that are reliable but difficult to understand. In time, a system's entropy increase making a fault insertion more likely with each new change. Through complexity measures the developer plans ahead for correct change by establishing initial order and thereby improves the continuing reliability of the system throughout its operational life.

There are three graph-theoretic complexity measures for software architecture:

(1) Static complexity - A measure of software architecture, as represented by a network of modules, useful for design tradeoff analyses. Network complexity is a function based on the countable properties of the modules (nodes) and network.

(2) Generalized static complexity --- a measure of software architecture, as represented by a network of modules and the resources used. Since resources are acquired or released when program are invoked in other modules, it is desirable to measure the complexity associated with allocation of those resources in addition to the basic (static) network complexity.

(3) Dynamic complexity --- A measure of software architecture, as represented by a network of modules during execution, rather than at rest, as is the case for the static measures. For example, modules may execute at different frequencies.

Primitives

K = number of resources, index by k = 1, ... , K

E = number of edges, indexed by i = 1, ..., E

N = number of nodes, indexed by j = 1, ..., N

c_j = complexity for program invocation and return along each edge e_j as determined by the user (such as operating system complexity)

Resource status array R(k,E)

$$r_{ki} = \begin{cases} 1 & \text{if kth resource is required for the ith edge } (e_i) \\ 0 & \text{otherwise} \end{cases}$$

d_k = complexity for allocation of resource k as determined by the user (for example, complexity associated with a procedure used to gain exclusive access to common data)

Implementation

Using nodes and edges, a strongly connected graph of the network is required. a strongly connected graph is one in which a node is reachable from any other node. This is accomplished by adding an edge between the exit node and the entry node. Each node represents a module that may or may not be executed concurrently with another module. Each edge represents program invocation and return between modules. In this case the edges are called single paths.

1. Static Complexity - Once a strongly connected graph is constructed, with modules as nodes, and transfer of control as edges, the static complexity is calculated as $C = E - N + 1$

2. Generalized Static complexity - Resources (storage, time, logic complexity, or other measurable factors) are allocated when programs are invoked in other modules. Given a

network and resources to be controlled in the network, the generalized static complexity associated with allocation of these resources is $C = \sum_{i=1}^{E}(c_i + \sum_{k=1}^{K}(d_k \cdot r_{ki}))$

3. Dynamic Complexity - A change in the number of edges may result from module interruption due to invocations and returns. An average dynamic network complexity can be derived over a given time period to account for the execution of modules at different frequencies and also for module interruption during execution. Dynamic complexity is calculated using the formula for static complexity at various points in time. The behavior of the measure is then used to indicate the evolution of the complexity of the software.

References

1. IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE, 1988.

2. Hall, N. R., Complexity Measures for Systems Design, Ph.D. Dissertation, Polytechnic Institute of New York, Brooklyn, Jun. 1983.

3. Hall, N. R., Preiser, S., Dynamic Complexity Measures for Software Design, *Proceedings of Total Systems Reliability Symposium*, IEEE Computer Society, Dec. 1983.

4. Hall, N. R., Preiser, S., Combined Network Complexity Measures. *IBM Journal of Research and Development*, Jan. 1984.

5. McCabe, T. J., A Complexity Measure, *IEEE Transactions on Software Engineering*, vol SE-2, no 4, Dec. 1976, pp 308-320.

A.21 Information Flow Complexity

Readers can be referred to the Section A.10 for the description of the measure Information Flow Complexity.

A.22 Lack of Cohesion in Methods (LCOM)

Application

This measure is a relative indicator of cohesion of a class. The "relative" originates from the fact that this measure is the subtraction of the number of related method pairs from the number of unrelated method pairs within the class under measurement [1]. Therefore the value of LCOM is a comparison between the number of correlated methods and the number of irrelevant methods from a design perspective (because whether two methods are correlated is determined by whether there is any instance variable shared by both of them. This criterion is based on the design perspective).

Definition

Consider a class C_I with n methods $M_1, M_2, \ldots, M_n$. Let $\{I_j\}$ be the set of instance variables used by method M_i. Then there are n such sets $\{I_1\}, \{I_2\}, \ldots, \{I_n\}$. Let $P = \{(\{I_i, I_j) \mid I_i \cap I_j = \varnothing\}$ and $Q = \{(\{I_i, I_j) \mid I_i \cap I_j \neq \varnothing\}$. If all n sets $\{I_1\}, \{I_2\}, \ldots, \{I_n\}$ are $\varnothing$ then let $P = \varnothing$.

The LCOM is defined as [1]:

$$\text{LCOM} = |P| - |Q|, \text{ if } |P| > |Q|$$ [8]

or

$$\text{LCOM} = 0, \text{ otherwise}$$

LCOM is the sum total of the number of method pairs whose similarity[9] is 0 minus the number of method pairs whose similarity is not zero. The larger the number of similar methods[10], the more cohesive the class, which is consistent with traditional notions of cohesion that measure the inter-relatedness between portions of a program [2].

The LCOM value provides a measure of the relative disparate nature of methods in the class. A smaller number of disjoint pairs (elements of set P) implies greater similarity of methods. LCOM is intimately tied to the instance variables and methods of a class, and therefore is a measure of the attributes of an object class.

The following are observations that relate to the value of this measure:

1. Cohesiveness of methods within a class is desirable, since it promotes encapsulation.
2. Lack of cohesion signals classes should probably be split into two or more subclasses.
3. Any measure of disparateness of methods helps identify flaws in the design of classes.
4. Low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.

Implementation

1. Identify the n methods $M_1, M_2, \ldots, M_n$ in the class under measurement.
2. Identify the n sets of $\{I_1\}, \{I_2\}, \ldots, \{I_n\}$.
3. Identify P and Q.

---

[8] |P| is defined as the number of elements contained in the set P.

[9] The degree of similarity between two methods M_1 and M_2 in class C_I is given by: $\sigma() = \{I_1\} \cap \{I_2\}$ where $\{I_1\}$ and $\{I_2\}$ are the sets of instance variables used by M_1 and M_2.

[10] Similar methods are also called correlated methods in this description. They are methods that share at least one instance variables.

4. Calculate the LCOM.

References

1. S. R. Chidamber, F. Kemerer, A Metrics Suite for Object Oriented Design, *IEEE Transactions on Software Engineering,* Vol. 20, No. 6, June 1994

2. S. R. Schach, *Software Engineering 2nd Edition,* Richard d. Irwin, Inc., Aksen Associates, Inc. Boston, 1993.

A.23 Man Hours per Major Defect Detected

Categories

• ***Structural Level***	derived
• ***Life-Cycle Coverage***	design (process)

Application

This measure is created to supply a quantitative figure that can be used to evaluate the efficiency of the design and code inspection process.

The design and code inspection processes are two of the most effective defect removal processes available. The early removal of defects at the design and implementation phases, if done effectively and efficiently, significantly improves the reliability of the developed product and allows a more controlled test environment.

Primitives

T_1 = time expended by the inspection team in preparation for design or code inspection meeting.

T_2 = time expended by the inspection team in conduct of a design or code inspection meeting.

S_i = number of major (nontrivial) defects detected during the *ith* inspection.

I = total number of inspections to date.

Implementation

At each inspection meeting, record the total preparation time expended by the inspection team. Also, record the total time expended in conducting the inspection meeting. All defects are recorded and grouped into major/minor categories. (A major defect is one which must be corrected for the product to function within specified requirements.)

The inspection time is summarized and the defects are cumulatively added. The computation should be performed during design and code. If the design is not written in a structural design language, then this measure can be only applied during the implementation phase.

The man hours per major defect detected is

$$M = \frac{\sum_{i=1}^{I} (T_1 + T_2)_i}{\sum_{i=1}^{I} S_i}$$

This computation should be initiated after approximately 8000 lines of detailed design or code have been inspected.

References

1. IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE, 1988.

2. Fagan, Michael, E., Design and Code Inspection to Reduce Errors in Program Development, *IBM Systems Journal,* vol 15, no 3, Jul. 1976, pp 182-211.

A.24 Mean Time to Failure

Categories

- ***Structural Level*** derived
- ***Life-Cycle Coverage*** testing (process)

Application

The goal of this measure is for hypothesis testing a specified MTTF requirement.

Primitive

Mean time to failure is the basic parameter required by most software reliability models. Computation is dependent on accurate recording of failure time (t_i), where t_i is the elapsed time between the ith and the (i-1)st failure. Time units used should be as precise as feasible. CPU execution time provides more resolution than wall-clock time. Thus CPU cycles would be more appropriate for a software development environment. For an operational environment that might require less resolution, an estimate based on wall-clock time could be used.

Implementation

Detailed record keeping of failure occurrences that accurately track the time (calendar or execution) at which the faults manifest themselves is essential. If weighting or organizing the failures by complexity, severity, or the reinsertion rate is desired, detailed failure analysis must be performed to determine the severity and complexity. Prior failure experience or model fitting analysis (for example, goodness-of-fit-test) can be used to select a model representative of a failure process, and to determine a reinsertion rate of faults.

References

1. IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE, 1988.
2. Currit, P. A., Dyer, M., Mills, H. D., Certifying the Reliability of Software, *IEEE Transaction on Software Engineering*, vol SE-12, no 1, Jan. 1986, pp 3-11.
3. Musa, J. D., Okumoto, K. A., Logarithmic Poisson Execution Time Model for Software Reliability Measurement, *Proceedings of the 7th International Conference on Software Engineering*, 1984, pp 230-238.
4. Rushforth, C., Staffanson, F., Crawford, A., Software Reliability Estimation Under Conditions of Incomplete Information, University of Utah, RADC-TR-230, Oct. 1979.
5. Shooman, M. L., Trivedi, A. K, A Many-State Markov Model for Computer Software Performance Parameters, *IEEE Transactions on Reliability*, vol R-25, no 2, Jun. 1976, pp 66-68.
6. Sukert, A. N., *A Software Reliability Modeling Study*, RADC-TR-76-247, Aug. 1976.
7. Wagoner, W. L., *The Final Report on Software Reliability Measurement Study*, Aerospace Corporation, Report Number TOR-0074(4112), Aug. 1973.

A.25 Minimal Unit Test Case Determination

Categories

• ***Structural Level***	derived
• ***Life-Cycle Coverage***	design (artifact)

Application

This measure determines the number of independent paths through a module so that a minimal number of covering test cases can be generated for unit test.

Primitives

N = number of nodes; a sequential group of program statements

E = number of edges; program flow between nodes

SN = number of splitting nodes; a node with more than 1 edge emanating from it

RG = number of regions; in a graph with no edges crossing, an area bounded by edges

Implementation

The cyclomatic complexity is first computed using the cyclomatic complexity measure described in 6. The complexity of the module establishes the number of distinct paths. The user constructs test cases along each path so all edges of the graph are traversed. This set of test cases forms a minimal set of cases that covers all paths through the module.

References

1. IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE, 1988.

2. Conte, S. D., Dunsmore, H. E., Shen, V. Y., *Software Engineering Metrics and Models*, Menlo Park: Benjamin/Cummings Publishing Co, 1986.

3. McCabe, T. J., A Complexity Measure, *IEEE Transactions on Software Engineering*, Vol SE-2, no 4, Dec. 1976, pp 308-320.

4. McCabe, T. J., *Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric*, National Bureau of Standards Special Publication 500-99, Dec. 1982.

A.26 Modular Test Coverage

Categories

• ***Structural Level***	derived
• ***Life-Cycle Coverage***	testing (process)

Application

This measure quantifies a software test coverage index for a software delivery.

The primitives counted may be either functions or modules. The operational user is most familiar with the system and will report system problems in terms of functional requirements rather than module test requirements. It is the task of the evaluator to obtain or develop the functional requirements an associated module cross-reference table.

Primitives

FE = number of the software functional (modular) requirements for which all test cases have been satisfactorily completed.

FT = total number of software functional (modular) requirements.

Implementation

The test coverage index is expressed as a ratio of the number of software functions (modules) tested to the total number of software functions (modules) that make up the users' (developers') requirements. This ratio is expressed as

$$\text{Functional (modular) test coverage index} = \frac{FE}{FT}$$

References

1. IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE, 1988.

2. Ceriani, M., Cicu, A., Maiocchi, M., A Methodology for Accurate Software Test Specification and Auditing, *Computer Program Testing*, B. Chandrasekaran and S. Radicchi, eds, North Holland Publishing Company, 1981.

3. Henderson, J. B., Hierarchical Decomposition: Its Use in Testing, *Testing Techniques Newsletter,* vol 3, no 4, San Francisco: Software Research Associate, Nov. 1980.

4. Maiocchi, M., Mazzetti, A., Villa, M., TEFAX: An Automated Test Factory for Functional Quality Control of Software Projects, Colloque de Genie Logiciel #2, Nice, 1984.

5. Miller, E., Coverage Measure Definition Reviewed, *Testing Technique Newsletter*, vol 3, no 4, San Francisco: Software Research Associate, Nov. 1980.

A.27 Mutation Testing

Categories

• ***Structural Level***	derived
• ***Life-Cycle Coverage***	testing (process)

Application

The goal of this measure is to examine the ability of the test data to differentiate between a correct program and an incorrect one.

Several known error types are inserted into the program and the program is executed with the specified test cases and in the testing environment. This allows the estimation of the number of errors remaining in the program.

Primitives

Number of seeded errors found

Total number of seeded errors

Number of real errors found

Implementation

This measure, denoted by *MTR*, is based on the assumption that the ratio of the seeded errors found to the total number of seeded errors is approximately equal to the ratio of the number of real errors found to the number of real errors. Thus, *MTR* is defined as follows:

$$MTR \cong \frac{Number\ of\ seeded\ errors\ found}{Total\ number\ of\ seeded\ errors} = \frac{Number\ of\ real\ errors\ found}{Total\ number\ of\ real\ errors}$$

The equation that defined *MTR* allows one to solve for any of the variables given knowledge of the other three. In particular, one can estimate the total number of real errors remaining and the associated testing effort. If all seeded errors were found, this is an indication that either the test cases are adequate, the inserted mutations (seeded errors) do not represent the distribution of real errors or the seeded errors were too easy to find.

Remarks

The measure is easy to calculate and provides an indication that the test cases are adequate to locate software errors. This measure is applicable to algorithmic solution and generally results in good estimates of operational reliability.

The generation of mutations may be labor intensive. The error types and seeded errors must be a statistical distribution of real errors in the program for this method to be of value. Even for a small program the number of mutants can be quite large.

A mutation of a correct program is another program that exhibits differences from the correct one (these are referred to as "seeded errors" even though they are intentionally inserted into the program). These errors are inserted one at a time and reflect those errors that may be made by a "competent programmer." An example of such of such an error is replacing "<" by ">" in a conditional. See Refs. 1, 2, and 3 for additional detail on the mutation testing measure.

References

1. Peng, W. and Wallace, D., *Software Error Analysis*, NIST Special Publication 500-209, 1993.

2. Myers, G., *The Art of Software Testing,"* John Wiley & Sons, 1979.

3. Royer, T., *Software Testing Management: Life on the Critical Path,"* Prentice-Hall, Inc., 1993.

A.28 Number of Children (NOC)

Application

NOC is the count of the immediate subclasses of the class being measured. NOC was presented by Chidamber and Kemerer as a measure of complexity [1] [2].

Definition

NOC is defined as the number of immediate subclasses subordinated to a class in the class hierarchy. This measurement has the following viewpoints [1]:

1. The greater the number of children, the greater the reuse, since inheritance is a form of reuse.

2. The greater the number of children, the greater the likelihood of improper abstraction of the parent class. If a class has a large number of children, it may be a case of misuse of subclassing.

3. The number of children gives an idea of the potential influence a class has on the design. If a class has a large number of children, it may require more testing of the methods in that class.

Implementation

1. Construct the class hierarchy diagram (see Figure A-1).

2. Identify the immediate subclasses of the class under measurement. For instance, if the class P in Figure A-1 is under measurement, then the immediate subclasses of class P are class C and class D.

3. Sum up the number of such subclasses. For instance, that number is 2 in the previous example. The number is NOC.

References

1. S. R. Chidamber, F. Kemerer, A Metrics Suite for Object Oriented Design, *IEEE Transactions on Software Engineering,* Vol. 20, No. 6, June 1994

2. R. D. Neal, The Applicability of Proposed Object-Oriented Metrics to Developer Feedback in Time to Impact Development, *NASA/WVU Software IV & V Facility, Software Research Lab.,* NASA-IVV-96-004, 1996.

A.29 Number of Class Methods in a Class

Application

This measure assesses software size in terms of the number of methods in a class [1].

The number of class methods can indicate the amount of commonalty being handled for all instances. It can also indicate poor design if the services handled by individual instances[11] are handled by the class itself.

Definition

Methods are the behaviors that a class can exhibit. Generally they are activated by a message sent to the class. The number of methods available to the class affects the size of the class.

Implementation

1. Identify all the methods within the class under measurement. These methods include the overridden methods.

2. Count the number of the methods retrieved by step 1.

3. This number is the value of the measure number of class methods in a class.

References

1. M. Lorenz, J. Kidd, *Object-Oriented Software Metrics: A Practical Guide*, Prentice Hall, Inc. New Jersey, 1994

A.30 Number of Faults Remaining (Error Seeding)

Categories

- ***Structural Level*** derived
- ***Life-Cycle Coverage*** requirements (artifact)

Application

This measure estimates the faults remaining in a program. The estimated number of faults remaining in a program is related to the reliability of the program. There are many sampling techniques that estimate this number. This section describes a simple form of seeding that assumes a homogeneous distribution of a representative class of faults.

This measure can be applied to any phase of the software life cycle. The search for faults continues for a determined period of time that may be less than that required to find all seeded faults. The measure is not computed unless some non-seeded faults are found.

[11] The instance means the subclass in this context.

Primitives

N_s = number of seeded faults

n_s = number of seeded faults found

n_F = number of faults found that were not intentionally seeded

Implementation

A monitor is responsible for error seeding. The monitor inserts (seeds) N_s faults representative of the expected indigenous faults. The test team reports to the monitor the faults found during a test period of predetermined length.

Before seeding, a fault analysis is needed to determine the types of faults and their relative frequency of occurrences expected under a particular set of software development conditions. Although an estimate of the number of faults remaining can be made on the basis of very few inserted faults, the accuracy of the estimate (and hence the confidence in it) increases as the number of seeded faults increases.

Faults should be inserted randomly throughout the software. Personnel inserting the faults should be different and independent of those persons later searching for the faults. The process of searching for the faults should be carried out without knowledge of the inserted faults. The search should be performed for a previously determined period of time (or effort) and each fault reported to the central monitor.

Each report fault should be reviewed to determine if it is in the class of faults being studied and, if so, if it is a seeded or an indigenous fault. The maximum likelihood estimate of the number of indigenous (unseeded) faults in the specified class is

$$\overline{NF} = \frac{n_F N_s}{n_s}$$

where $\overline{NF}$ is truncated to the integer value. The estimate of the remaining number of faults is then

$$\overline{NF}_{rem} = \overline{NF} - n_F$$

The probability of finding n_F of NF indigenous faults and n_F of N_s seeded faults, given that there are $(n_F + n_s)$ faults found in the program is $C(N_s, n_s)\ C(NF, n_F)\ /\ C(NF + N_s, n_F + n_s)$, where the function $C(x, y) = x!/(x-y)!y!$ is the combination of "x" things taken at "y" at a time. Using this relation one can calculate confidence intervals.

References

1. IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE, 1988.

2. Thayer, T. A., Kipow, M., Nelson, E. C., *Software Reliability: A Study of Large Project Reality,* New York: North Holland Publishing Co, 1978.

3. Basin, S. L, *Estimation of Software Error Rates Via Capture-Recapture Sampling,* Palo Alto: Science Applications Inc, Sept 1973.

4. Basin, S. L., *Measuring the Error Content of Software,* Palo Alto: Science Applications Inc, Sept 1974.

5. Bowen, J. B., Saib, S. H., *Association of Software Errors, Classifications to AED language Elements,* Prepared under Contract Number NAS 2-10550 by Hughes Aircraft Company, Fullerton, and General Research Corporation, Santa Barbara, California, Nov 1980.

6. Bowen, J. B., Saib, S. H., *Error Seeding Technique Specification,* Prepared under Contract Number NAS 2-10550 by Hughes Air-craft Company, Fullerton, and General Research Corporation, Santa Barbara, California, Dec 1980.

7. DACS, *Rudner Model, Seeding/Tagging,* Quantitative Software Models, DACS, SSR-1, Mar 1979, pp 8-56 to 3-58.

8. Duran, J. W., Wjorkowski, J. J., Capture-Recapture Sampling for Estimating Software Error Content, *IEEE Transactions on Software Engineering,* vol SE-7, Jan. 1981.

9. Feller, W., *An Introduction to Probability Theory and Its Applications,* New York: John Wiley and Sons, Inc, 1957, p 43.

10. *System Specification for Flexible Inter-connect* Specification Number SS078779400, May 15,1980.

11. Lipow, M., *Estimation of Software Package Residual Errors,* TRW Software Series SS-72-09, Redondo Beach: TRW E&D, 1972.

12. McCabe, T. J., *Structural Testing,* Columbia, Maryland: McCabe & Associates, Inc, 1984.

13. Mills, H. D., *On the Statistical Validation of Computer Programs,* FSC-72-6015, Gaithersburg, Maryland: Federal Systems Division, Inter-national Business Machines Corporation, 1972.

14. Ohba, M., et al. *S-shape Reliability Control Curve: How Good Is It?* Proceedings COMPSACS2, IEEE Computer Society, 1982, pp 38-44.

15. Rudner, B., Seeding/Tagging Estimation of Software Errors: Models and Estimates. RADC-TR-15, 1977.

16. Seber, G. A. F., *Estimation of Animal Abundance and Related Parameters,* 2nd ed, New York: McMillan Publishing Co, 1982.

A.31 Number of Key Classes

Application

This measure estimates the number of key classes in a system. The value of this measure is an indicator of effort required to develop the system.

Definitions

Key classes are central to the business domain being developed. Key classes are also the central points of reuse on future projects, since they are highly likely to be needed in other domains in the business [1]. The number of key classes is an indication of the volume of work needed in order to develop an application. It is also one indication of the amount of long-term reusable objects that will be developed as a part of this effort for applications dealing with the same or similar problem domain.

Implementation

Usually, we can determine if a class is key by asking questions such as,

1. Could I easily develop applications in this domain without this class?
2. Would a customer consider this object important?
3. Do many scenarios involve this class?

Answers to these questions will segregate classes into categories of key and support. In general, project experiences have shown that you can expect 20-40 percent of your classes being categorized as key domain classes, with the rest being support classes. Low numbers of key classes may indicate that you need to explore more of your business domain to discover important abstractions to simulate your business.

Reference

1. M. Lorenz, J. Kidd, *Object-Oriented Software Metrics: A Practical Guide*, Prentice Hall, Inc. New Jersey, 1994

A.32 Requirement Compliance

Categories

- ***Structural Level*** derived

- ***Life-Cycle Coverage*** requirements (artifact)

Application

The goal of this measure is to verify requirements' compliance by using system verification diagram (SVDs).

SVD is a logical interconnection of stimulus response elements (for example, stimulus and response) which detect inconsistencies, incompleteness, and misinterpretations.

Primitives

Des = decomposition elements:

Stimulus - external input

Function - defined input/output process

Response - result of the function

Label - numerical DE identifier

Reference - specification paragraph number

Requirement errors detected using SVDs:

N_1 = number due to inconsistencies

N_2 = number due to incompleteness

N_3 = number due to misinterpretation

Implementation

The implementation of an SVD is composed of the following phases:

(1) The decomposition phase is initiated by mapping the system requirement specifications into stimulus/response elements (Des). That is, all keywords, phases, functional and/or performance requirements and expected outputs are documented on decomposition forms.

(2) The graph phase uses the Des from the decomposition phase and logically connects them to form the SVD graph.

(3) The analysis phase examines the SVD from the graph by using connectivity and reachability metrics. The various requirements error types are determined by examining the system verification diagram and identifying errors as follows:

a) Inconsistencies --- Decomposition elements that do not accurately reflect the system requirement specification.

b) Incompleteness --- Decomposition elements that do not completely reflect the system requirement specification.

c) Misinterpretation --- Decomposition element that do not correctly reflect the system requirement specification. This error may occur during translation of the requirements into decomposition elements, constructing the connectivity and reachability matrices.

An analysis is also made of the percentages for the various requirement error types for the respective categories' inconsistencies, incompleteness, and misinterpretation.

Inconsistencies (%) = $(N_1/(N_1+N_2+N_3)) * 100$

Incompleteness (%) = $(N_2/(N_1+N_2+N_3)) * 100$

Misinterpretation (%) = $(N_3/(N_1+N_2+N_3)) * 100$

This analysis can aid also in future software development efforts.

References

1. IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE, 1988.

2. Fischer, K. F., Walker, M. G., Improved Software Reliability Through Requirement Verification, *IEEE Transactions on Reliability*, vol R-28, no 3, Aug. 1979, pp 233-239.

A.33 Requirements Specification Change Requests

Categories

• ***Structural Level***	derived
• ***Life-Cycle Coverage***	requirement (artifact)

Application

This measure indicates the stability of the functional requirements.

The requirements phase of the software life cycle has the greatest potential for improving the quality of the resulting system and helping to control the software development cost. It has been observed that a significant cause of project failure and poor quality in software systems is frequent changes to the requirements.

Primitives

Requested changes to the requirements specification

Implementation

The requirements specification change request measure, denoted by RSCR, is defined as the number of change requests that are made to the requirements specification. The requested changes are counted from the time of the first release of the requirements specification document to the time when the product begins it operational life. Thus, RSCR is defined as:

RSCR = Σ(requested changes to the requirements specification),

where the summation is taken over all requirement change requests initiated during the software development life cycle.

RSCR is an indication of the quality of the resulting software system. Evidence suggests that the system quality decreases as the size of RSCR increases.

Remarks

The RSCR is easy to compute and clearly shows the stability and/or growth of functional requirements throughout the software life cycle, by life cycle phase.

The use of this measure in conjunction with Function Points or Feature Point counts can be used to show status and trends in requirements growth.

The measure is easy to understand throughout the software life cycle phases, even where direct measurement of reliability is possible. Even when more direct measures of reliability are available, RSCR measure provides an additional view of the effectiveness of the functional specification process used and has the potential of adding credibility to the product.

References

1. Moller, K. and Paulish, D., *Software Metrics, A Practitioner's Guide to Improved Product Development*, Chapman and Hall Computing, 1993.

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

A.34 Requirements Traceability

Categories

• ***Structural Level***	derived
• ***Life-Cycle Coverage***	design (artifact)

Application

This measure aids in identifying requirements that are either missing from, or in addition to, the original requirements.

Primitives

R1 = number of requirements met by the architecture.

R2 = number of original requirements.

Implementation

A set of mappings from the requirements in the software architecture to the original requirements is created. Count each requirement met by the architecture (R1) and count each of the original requirements (R2). Compute the traceability measure (TM):

$$TM = \frac{R1}{R2} \times 100\%$$

References

1. IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE, 1988.

2. Henninger, K., Specifying Software Requirements for Complex Systems, New Techniques and Their Application, *IEEE Transaction on Software Engineering,* vol SE-6, no 1, Jan. 1980, pp 1 – 14.

3. Perriens, M. P., Software Requirements Definition and Analysis with PSL and QBE, *IBM Technical Report FSD 80-0003*, Bethesda, IBM Federal Systems Division, Oct. 1980.

4. Yeh, R., Zave, P., et al, Software Requirements: A Report on the State of the Art, *Computer Science Technical Report Series TR-949*, College Park, Maryland, University of Maryland, Oct. 1980.

A.35 Reviews, Inspections and Walkthroughs

Categories

• ***Structural Level***	derived
• ***Life-Cycle Coverage***	requirements (process)

Application

This measure identifies the number of satisfied checklist items for each product.

Performing technical reviews, walkthroughs and/or inspections may affect the overall quality of the software development work products (e.g., software requirements specification, test plan, design description, code, etc.) This measure uses various checklists and captures the number of checklist items satisfied for each specific work product of interest.

Primitives

NC_i = the number of checklist items satisfied for that work product

N = the total number of checklist items applicable to that work product.

Implementation

Several checklists exist that can be used to evaluate a particular work product to the software development process; e.g., a checklist for system testing (see Ref. 2, pages 371-374). Other checklists are found in references (TBD).

This type of measure is defined as follows:

For a specific work product and selected checklist, the figure of merit, denoted by *FOM*, be defined as:

$$FOM = [(\Sigma\,(NC_i)\,)\,/\,N] * 100,$$

where

NC_i = the number of checklist items satisfied for that work product

N = the total number of checklist items applicable to that work product.

FOM is only an indication of the quality of the resulting software system. Confidence in the specific work product increases non-linearly as the *FOM* approaches 100%.

Remarks

The measure is easy to calculate and provides a quality check for each work product at each life cycle phase of the software development effort.

The measure is easy to understand throughout the software life cycle phases, even where direct measurement of reliability is possible. Even when more direct measures of reliability are available, this measure provides an additional view of the effectiveness of development process used and has the potential of adding credibility to the product.

References

1. Moller, K. and Paulish, D., *Software Metrics, A Practitioner's Guide to Improved Product Development*, Chapman and Hall Computing, 1993.

2. Freedman, D. and Weinberg, G., *Handbook of Walkthroughs, Inspections and Technical Reviews: Evaluating Programs, Projects, and Products*, Dorset House Publishing Company, 1990.

3. Redmill, F., *Dependability of Critical Computer Systems 1*, Elsevier Applied Science, 1988.

A.36 Software Capability Maturity Model (CMM)

Categories

- ***Structural Level*** primitive
- ***Life-Cycle Coverage*** requirements (process)

Application

The goal of this measure is to describe the principles and practices underlying software process maturity and is intended to help software organizations improve the maturity of their software processes.

The SW-CMM is a framework that describes the 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.

Primitives

This measure, denoted by L_i, where $i = 1, 2, 3, 4, 5$, is based on the assumption that the predictability, effectiveness, and control of a project's or an organization's software processes—and hence the production of higher-quality software—are believed to improve as the organization moves up these five levels. While not rigorous, empirical evidence to date supports this belief.

L_i is defined as follows: L_i represents the project's or organization's software process maturity as measured using one of the Software Engineering Institute's CMM-based appraisal instruments.

The CMM-based appraisal methods rate an organization's software process maturity and classifies it as one of the following levels:

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 heroics.

2. **Repeatable.** Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.

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.

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

5. **Optimizing.** Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies.

Remarks

The measure is easy to calculate and provides an of software process maturity. This can be used to infer the quality of the resulting software products that are developed by each maturity level process.

This is not a direct measure of the reliability of the software developed by an L_i process.

Except for Level 1, each maturity level is decomposed into several key process areas that indicate the areas an organization should focus on to improve its software process.

The key process areas at Level 2 focus on the software project's concerns related to establishing basic project management controls. They are requirements management, software project planning, software project tracking and oversight, software subcontract management, software quality assurance, and software configuration management.

The key process areas 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. They are organization process focus, organization process definition, training program, integrated software management, software product engineering, intergroup coordination, and peer reviews.

The key process areas at Level 4 focus on establishing a quantitative understanding of both the software process and the software work products being built. They are quantitative process management and software quality management.

The key process areas at Level 5 cover the issues that both the organization and the projects must address to implement continual, measurable software process improvement. They are defect prevention, technology change management, and process change management.

Each key process area is described in terms of the key practices that contribute to satisfying its goals. The key practices describe the infrastructure and activities that contribute most to the effective implementation and institutionalization of the key process area. The intention in setting down the key practices is not to require or espouse a specific model of the software life cycle, a specific organizational structure, a specific separation of responsibilities, or a specific management and technical approach to development. The intention, rather, is to provide a description of the essential elements of an effective software process.

The key practices are intended to communicate principles that apply to a wide variety of projects and organizations, that are valid across a range of typical software applications, and that will remain valid over time. Therefore, the approach is to describe the principles and leave their implementation up to each organization, according to its culture and the experiences of its managers and technical staff.

References

1. Mark C. Paulk, Bill Curtis, Mary Beth Chrissis, and Charles V. Weber, "Capability Maturity Model for Software, Version 1.1," Software Engineering Institute, CMU/SEI-93-TR-24, DTIC Number ADA263403, February 1993.

2. Mark C. Paulk, Charles V. Weber, Suzanne M. Garcia, Mary Beth Chrissis, and Marilyn W. Bush, "Key Practices of the Capability Maturity Model, Version 1.1," Software Engineering Institute, CMU/SEI-93-TR-25, DTIC Number ADA263432, February 1993.

3. Mark C. Paulk, Bill Curtis, Mary Beth Chrissis, and Charles V. Weber, "Capability Maturity Model, Version 1.1," *IEEE Software*, **10**, 4 (July 1993), pp. 18-27.

A.37 System Design Complexity

Categories

• ***Structural Level***	derived
• ***Life-Cycle Coverage***	design (artifact)

Application

The goal of this measure is to identify the system complexity.

Systems with high complexity are more likely to contain faults than systems with lower complexity. One measure of system complexity that has some experimental validation is system design complexity. An expected fault rate can be derived from the design complexity.

Primitives

Let

n = number of modules in the system

$f(i)$ = fanout of the ith module

$v(i)$ = number of I/O variables in the ith module

Derived quantities are:

$S(i)$ = structural complexity of the ith module

$D(i)$ = data complexity of the ith module

St = program structural complexity

Dt = program data complexity

C = program design complexity

Fr = fault rate, in terms of expected faults per KLOC

Here, "fanout" is defined, for a module, to be the number of other modules called (found by counting the number of "call" statements in the module). I/O variables are distinct arguments exchanged between the module and the rest of the program, and includes distinct arguments in a calling sequence and referenced global variables.

Module design complexity is a combination of structural complexity and data complexity, and system design complexity is a function of the average complexity of the various module. The following equations are used:

$$S(i) = f^2(i)$$
$$D(i) = \frac{v(i)}{f(i)+1}$$
$$St = \sum_{i=1}^{n} f^2(i)$$
$$Dt = \frac{1}{n}\sum_{i=1}^{n}\frac{v(i)}{f(i)+1}$$
$$C = St + Dt$$
$$Fr = 0.4 \times C - 5.2$$

Implementation

The fault rate is given in terms of delivered lines of code, and is derived from experimental data. Only a limited number of cases were used to fit the curve, and the parameters may need to adjusted for other organizations.

Remarks

This measure does consider more than one aspect of complexity, which is better than most complexity measures.

This appears a useful technique for predicting faults in code, once the detailed design or code is available.

References

1. David N. Card and Robert L. Glass, *Measuring Software Design Quality*, Prentice-Hall (1990).

A.38 Test coverage

Categories

- ***Structural Level*** derived
- ***Life-Cycle Coverage*** testing (process)

Application

The goal of this measure is to identify 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, in particular, unit, functional, system and acceptance tests. Developers, using the program class of primitives, can apply the measure in unit test to obtain a measure of thoroughness of structural tests. System tester can apply the measure in two ways: First, by focusing on requirements primitive, the system tester can gain o user-view of the thoroughness of functional tests. Second, by focusing on the program class of primitives, the system tester can determine the amount of implementation in the operational environment.

Primitives

The primitives for test coverage are in two classes, program and requirements. For program, there are two type: functional and data. The program functional primitives are either modules, segments, statements, branches (nodes), or paths. Program data primitives are equivalence classes of data. Requirements primitives are either test cases or functional capabilities.

Implementation

Test coverage (TC) is the percentage of requirement primitives implemented times the percentage of primitives executed during a set of tests. A simple interpretation of test coverage can be expressed by the following formula:

$$TC(\%) = \frac{(implemented\ capabilities)}{(required\ capabilities)} \times \frac{(program\ primitives\ tested)}{(total\ program\ primitives)} \times 100$$

References

1. IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE, 1988.
2. Demillo, R. A., Martin, r. J., *Software Test and Evaluation: Current Defense Practices Overview,* Georgia Institute of Technology, Jun. 1983.
3. McCabe, T., *Tutorial Text: Structured Testing*, IEEE Computer Society Press, 1981.

4. Miller, E., *Tutorial: Program Testing Techniques,* IEEE Computer Society Press, 1977.

A.39 Test Mutation Score

Application

This measure is designed initially for the purpose of providing a measure of the efficiency of a testing data set T. A high score indicates that T is very efficient for the program P with respect to mutation fault exposure.

Definitions

A mutation is a single-point, syntactically correct change, introduced in the program P to be tested. The mutation score, denoted *ms,* is the ratio of the non-equivalent mutants of P (i.e. those which are distinguishable from P for at least one data point of the input domain) which are killed by a specific test data set T [1].

A mutant is the code after a syntactical modification. A mutant is killed by a test case that causes the mutant program to produce "altered" output. Equivalent mutants are mutant programs that are functionally equivalent to the original program and therefore cannot be killed by any test case. The goal of mutation is to find test cases that kill all nonequivalent mutants. [2]

Implementation

A set of mutants of P consists of a set of programs which differ from P in containing one mutation from a given list of faults representative of the most likely faults introduced by programmers using the language of P. A mutant is killed (i.e. distinguished from P) by a test set T if its output history differs from that of the original program P.

For instance, one of the possible mutants of the following FORTRAN procedure

```
  SUBROUTINE M1(X, MAG)
  MAG = 1
  DO 1 I = 1, N
1 MAG = MAG + X(I)**2
  MAG = SQRT(MAG)
  RETURN
  END
```

is to change the target of the loop, which is the labeled "1" statement, to another labeled target, which is the labeled "1" statement below.

```
  SUBROUTINE M1(X, MAG)
  MAG = 1
  DO 1 I = 1, N
```

```
    MAG = MAG + X(I)**2
  1 MAG = SQRT(MAG)
    RETURN
    END
```

Mutation analysis has already been used in experiments conducted on FORTRAN, COBOL and C programs. A specific mutant generator is required for each programming language, in order to produce syntactically correct changes that should be representative of the programmers' faults.

It is debatable whether a mutation score is a convincing measure of the actual fault revealing power of a test set. Nevertheless, let us note that (1) mutations are faults related to the program structure, and (2) by essence, structural testing should aim at tracking down faults related to implementation (while functional testing should focus on other fault types). Thus mutations representative of the likely faults committed by the programmer form a fault set consistent with structural testing; and the mutation score is a meaningful measure at least for assessing the relative efficiency of different structural testing methods. In any case, a high mutation score indicates that the test set strongly probes the program structure (thus, has a high fault revealing power), while a low score reveals inadequacies. Moreover, despite the fact that mutations are simple changes, they can produce errors that are representative of the subtle errors caused by real faults.

References

1. P. Thevenod-Fosse, C. Mazuet, Y. Crouzet, "On Statistical Structural Testing of Synchronous Data Flow Programs," *Proceedings of 1st European Dependable Computing Conference (EDCC-1)*, Berlin, Germany, LNCS, Springer Verlag, 1994, pp. 250-267.

2. J. M. Voas, G. McGraw, *Software Fault Injection: Inoculating Programs Against Errors*, John Wiley & Sons, Inc., New York, 1998

A.40 Weighted Method per Class (WMC)

Application

WMC is the sum of weighted methods in a class. Each method within the class is weighted by some sort of complexity metric and this weight is summed up so as to arrive at WMC [2][3]. WMC was presented by Chidamber and Kemerer as a measure of complexity [3].

Definitions

Consider a class C, with methods $M_1, M_2, \dots M_n$ defined in C. Let $c_1, c_2, \dots c_n$ be the complexity of the methods respectively. Then WMC is defined [3]:

Equation A-1

$$WMC = \sum_{i=1}^{n} c_i$$

The following assigned weights are used to compute method complexity [1]:

API calls	5.0
Assignments	0.5
Binary expressions or arithmetic operator	2.0
Keyword messages or messages with parameters	3.0
Nested expressions	0.5
Parameters	0.3
Primitive calls	7.0
Temporary variables	0.5
Unary expression or messages without parameters	1.0

Implementation

1. Inspect the method *i* of the class *C* and identify the complexity of each statement in this method according to the mapping of weights described in the previous table.

2. The complexity of method *i* (c_i) is defined as the sum of the results obtained in step 1.

3. Calculate *WMC* according to the Equation A-1.

References

1. M. Lorenz, J. Kidd, *Object-Oriented Software Metrics: A Practical Guide*, Prentice Hall, Inc. New Jersey, 1994.

2. R. D. Neal, The Applicability of Proposed Object-Oriented metrics to Developer Feedback in Time to Impact Development, *NASA/WVU Software IV&V Facility, Software Research Laboratory, Technical Report Series,* NASA-IVV-96-004, NASA, 1996.

3. S. R. Chidamber, F. Kemerer, A Metrics Suite for Object Oriented Design, *IEEE Transactions on Software Engineering,* Vol. 20, No. 6, June 1994.

APPENDIX B SOFTWARE ENGINEERING MEASURES QUESTIONNAIRE

Please read the report, then answer questions (1) to (10). Use tables from Table B-1E to Table B-10E as examples for answering the questions.

THIS QUESTIONNAIRE IS TO BE ANSWERED BY EXPERTS ONLY

QUESTIONS:

1. Please print your name: ______________________________

2. Please print your title: ______________________________

3. For each software engineering measure listed in Table B-1 check if:

   (i) you were an inventor of the measure or closely associated with the invention of the measure.

   (u) you were a user of this measure on projects or experiments. Then

   Specify the number of projects

   Specify the size of these projects (KLOCs or FPs). Give minimum, maximum and average size.

   Specify the context of applications, e.g. safety critical, aerospace, nuclear, etc. This column in Table B-1 contains only an index list of applications related to this measure. The detailed context is described in Table B-2.

   (l) your knowledge of the measure is obtained from the reading of published materials (books, papers, reports)

   (w) your knowledge of the measure is the result of your attendance to workshops and/or conferences (specify the duration of these events in number of 8-hours days).

   (o) No experience.

   Please fill Table B-1 and Table B-2.

4. Rate the measures given according to the ranking criteria specified in section 3 by filling the "rate" columns in Table B-3 and B-4. In Table B-3 rate all criteria except the two relevance criteria. The two relevance criteria are software life cycle phase dependent. In Tables B-4 rate the relevance criteria for the requirements, design, implementation and testing phases.

   (Note: Please do not rate measures for which you are not qualified).

5. For each measure in the list, associate a degree of confidence (a number between 0 and 1) to your rating of the measure. This degree of confidence reflects your knowledge and experience in the measure. A degree of confidence of "0" means that you are absolutely

unsure of your rating (and consequently you should not have rated the measure)! A degree of confidence "1" means that you are absolutely sure of your rating.

If necessary, you can specify a degree of confidence for individual ranking criteria. For instance, you could be absolutely certain of your evaluation of the rate of ranking criteria X for measure A, but absolutely unsure of your evaluation of the rate of ranking criteria Y (for this same measure.) This would translate into the following degrees of confidence:

Deg. of confidence (X|A) = 1
Deg. of confidence (Y|A) = 0

Please fill the confidence columns in Table B-3 and B-4.

6. Identify measures highly correlated with the measure currently being rated. Highly correlated signifies that the current measure can replace the correlated measure, i.e. the information provided by each measure is almost identical. For example, consider "failure rate" and "mean time to failure", these two measures are highly correlated since one can be derived from the other and vice versa.

   Please fill Table B-5 by specifying names of highly correlated measures.

7. Identify possibly missing measures that will help us in realizing the aims of the projects.

8. Define the "missing measures" to a level understandable to any of your colleagues. Append these descriptions to this questionnaire. The description should include facts that would help others make an assessment of the ranking criteria's rate for the particular measure identified. Please fill Table B-8. If you have more than one "missing measure", there should exist more than one Table B-8s. Please caption them as Table B-8-1, Table B-8-2, ..., etc.

9. Perform steps (3), (4), (5) and (6) for all "missing measures" identified and complete Table B-6, Table B-7, Table B-9 and Table B-10 respectively.

10. Add to your report any comments you might have on the ranking criteria.

Table B-1 Sources of Knowledge

Measure	**(i)**	**(u)**					**(l)**	**(w)**	**(o)**
		number of projects	size of projects			context of application[1]			
			minimum	maximum	average				
Bugs per line of code (Gaffney estimate)									
Cause & effect graphing									
Code defect density									
Cohesion									
Completeness									
Cumulative failure profile									
Cyclomatic complexity									
Data flow complexity									
Design defect density									
Error distribution									
Failure rate									
Fault density									
Fault-days number									
Feature point analysis									
Function point analysis									
Functional test coverage									
Graph-theoretic static architecture complexity									
Man hours per major defect detected									
Mean time to failure									
Minimal unit test case determination									
Modular test coverage									
Mutation testing (error seeding)									
Number of faults remaining (error seeding)									
Requirements compliance									

[1] This contains an index list of applications' description. The detailed description of applications is in Table B-2.

Measure	**(i)**	**(u)**					**(l)**	**(w)**	**(o)**
		number of projects	size of projects			context of application			
			minimum	maximum	average				
Requirements specification change requests									
Requirements traceability									
Reviews, inspections and walkthroughs									
Software capability maturity model									
System design complexity									
Test coverage									

Table B-2 Software Reliability Engineering Measures Related Applications' Descriptions

Application's Index	Description
1	
2	
3	
4	
5	
6	
. . .	

Table B-1E Example of Table B-1 Sources of Knowledge

Measure	**(i)**	**(u)**					**(l)**	**(w)**	**(o)**
		number of projects	size of projects			context of application[1]			
			minimum	maximum	average				
Failure rate							x		
Function Point Analysis		2	150 FP	450 FP	300 FP	1,2			

Table B-2E Example of Table B-2 Software Reliability Engineering Measures Related Applications' Descriptions

Application's Index	Description
1	PACS system reliability evaluation. PACS is a real-time gate security control system.
2	GWRPS system safety evaluation. GWRPS is a safety critical, real-time nuclear plant protection system.

[1] This contains an index list of applications' description. The detailed description of applications is in Table B-2E.

Table B-3 Evaluation of Measures by Criteria (except relevance criteria)

Measure	Time-liness		Cost		Benefits		Credibility		Repeat-ability		Experience		Validation	
	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence
Bugs per line of code (Gaffney estimate)														
Cause & effect graphing														
Code defect density														
Cohesion														
Completeness														
Cumulative failure profile														
Cyclomatic complexity														
Data flow complexity														
Design defect density														
Error distribution														
Failure rate														
Fault density														
Fault-days number														
Feature point analysis														
Function point analysis														
Functional test coverage														
Graph-theoretic static architecture complexity														
Man hours per major defect detected														

Measure	**Time-liness**		**Cost**		**Benefits**		**Credibility**		**Repeat-ability**		**Experience**		**Validation**	
	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence
Mean time to failure														
Minimal unit test case determination														
Modular test coverage														
Mutation testing (error seeding)														
Number of faults remaining (error seeding)														
Requirements compliance														
Requirements specification change requests														
Requirements traceability														
Reviews, inspections and walkthroughs														
Software capability maturity model														
System design complexity														
Test coverage														

Table B-4 Phase-Based Relevance to Reliability/Review

Measure	Requirement				Design				Implementation				Testing			
	Relevance to Reliability		Relevance to Review		Relevance to Reliability		Relevance to Review		Relevance to Reliability		Relevance to Review		Relevance to Reliability		Relevance to Review	
	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence
Bugs per line of code (Gaffney estimate)																
Cause & effect graphing																
Code defect density																
Cohesion																
Completeness																
Cumulative failure profile																
Cyclomatic complexity																
Data flow complexity																
Design defect density																
Error distribution																
Failure rate																
Fault density																

Measure	**Requirement**				**Design**				**Implementation**				**Testing**			
	Relevance to Reliability		**Relevance to Review**		**Relevance to Reliability**		**Relevance to Review**		**Relevance to Reliability**		**Relevance to Review**		**Relevance to Reliability**		**Relevance to Review**	
	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence
Fault-days number																
Feature point analysis																
Function point analysis																
Functional test coverage																
Graph-theoretic static architecture complexity																
Man hours per major defect detected																
Mean time to failure																
Minimal unit test case determination																
Modular test coverage																
Mutation testing (error seeding)																
Number of faults remaining (error seeding)																
Requirements compliance																
Requirements specification change requests																
Requirements traceability																
Reviews, inspections and walkthroughs																

Measure	Requirement				Design				Implementation				Testing			
	Relevance to Reliability		Relevance to Review		Relevance to Reliability		Relevance to Review		Relevance to Reliability		Relevance to Review		Relevance to Reliability		Relevance to Review	
	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence
Software capability maturity model																
System design complexity																
Test coverage																

Table B-3E Example of Table B-3 Evaluation of Measures by Criteria (except relevance criteria)

Measure	Time-liness		Cost		Benefits		Credibility		Repeat-ability		Experience		Validation	
	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence
Bugs per line of code (Gaffney estimate)	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

Table B-4E Example of Table B-4 Phase-Based Relevance to Reliability/Review

Measure	Requirement				Design				Implementation				Testing			
	Relevance to Reliability		Relevance to Review		Relevance to Reliability		Relevance to Review		Relevance to Reliability		Relevance to Review		Relevance to Reliability		Relevance to Review	
	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence
Bugs per line of code (Gaffney estimate)	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

Table B-5 Correlation Between Measures[*]

Measure	Correlated Measure 1	Correlated Measure 2	Correlated Measure 3	Correlated Measure 4	Correlated Measure 5	Correlated Measure 6	Correlated Measure 7	Correlated Measure 8	Correlated Measure 9	Correlated Measure 10
Bugs per line of code (Gaffney estimate)										
Cause & effect graphing										
Code defect density										
Cohesion										
Completeness										
Cumulative failure profile										
Cyclomatic complexity										
Data flow complexity										
Design defect density										
Error distribution										
Failure rate										
Fault density										
Fault-days number										
Feature point analysis										
Function point analysis										
Functional test coverage										

[*] If you have more than 10 highly correlated measures in one row please attach a copy of this form.

Measure	Correlated Measure 1	Correlated Measure 2	Correlated Measure 3	Correlated Measure 4	Correlated Measure 5	Correlated Measure 6	Correlated Measure 7	Correlated Measure 8	Correlated Measure 9	Correlated Measure 10
Graph-theoretic static architecture complexity										
Man hours per major defect detected										
Mean time to failure										
Minimal unit test case determination										
Modular test coverage										
Mutation testing (error seeding)										
Number of faults remaining (error seeding)										
Requirements compliance										
Requirements specification change requests										
Requirements traceability										
Reviews, inspections and walkthroughs										
Software capability maturity model										
System design complexity										
Test coverage										

Table B-5E Example of Table B-5 Correlation Between Measures

Measure	Correlated Measure 1	Correlated Measure 2	Correlated Measure 3	Correlated Measure 4	Correlated Measure 5	Correlated Measure 6	Correlated Measure 7	Correlated Measure 8	Correlated Measure 9	Correlated Measure 10
Measure A	Measure B	Measure C	Measure D							
Measure B	Measure A	Measure E	Measure F	Measure G						

Table B-6 Sources of Knowledge (Missing Measures)

Measure	(i)	(u)					(l)	(w)	(o)
		number of projects	size of projects			context of application[1]			
			minimum	maximum	average				

[1] This contains an index list of applications' description. The detailed description of applications is in Table B-7.

Table B-7 Software Reliability Engineering Measures Related Applications' Descriptions

Application's Index	Description
1	
2	
3	
4	
. . .	

Table B-6E Example of Table B-6 Sources of Knowledge (Missing Measures)

Measure	(i)	(u)					(l)	(w)	(o)
		number of projects	size of projects			context of application [1]			
			minimum	maximum	average				
Measure A								2	

Table B-7E Example of Table B-7 Software Reliability Engineering Measures Related Applications' Descriptions

Application's Index	Description
1	
2	
. . .	

TableB-8 Description of Missing Measures

Name:
Author(s):
Reference:
Description:

Table B-8E Example of Table B-8 Description of Missing Measures

Name:	Coupling
Author(s):	Myers
Reference:	Xxx
Description:	Coupling is an indication of "goodness" of a design. There is a widespread belief that low coupling yield better software designs. …

TABLE B-9 Evaluation of Measures by Criteria (missing Measures) (except relevance criteria)

Measure	Time-liness		Cost		Benefits		Credibility		Repeat-ability		Experience		Validation	
	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence

Table B-9E Example of Table B-9 Evaluation of Measures by Criteria (except relevance criteria) (Missing Measures)

Measure	Time-liness		Cost		Benefits		Credibility		Repeat-ability		Experience		Validation	
	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence
Metric A	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

Table B-10 Phase-Based Relevance to Reliability/Review (Missing Measures)

Measure	Requirement				Design				Implementation				Testing			
	Relevance to Reliability		Relevance to Review		Relevance to Reliability		Relevance to Review		Relevance to Reliability		Relevance to Review		Relevance to Reliability		Relevance to Review	
	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence

Table B-10E Example of Table B-10 Phase-Based Relevance to Reliability/Review (Missing Measures)

Measure	Requirement				Design				Implementation				Testing			
	Relevance to Reliability		Relevance to Review		Relevance to Reliability		Relevance to Review		Relevance to Reliability		Relevance to Review		Relevance to Reliability		Relevance to Review	
	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence	Rate	Confidence
Metric A	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

APPENDIX C SENSITIVITY ANALYSIS: SCHEMES AND RESULTS

C.1 Sensitivity Analysis on Criteria Levels

C.1.1 Variations on Criteria Levels

Table C-1 Criteria Levels for Scheme 1

Cost		Experience			Benefits	Credibility	Repeatability	Validation	Relevance to Reliability
W	1.00	W	1.00	A	1.00	1.00	1.00	1.00	1.00
M	0.90	M	0.55	B	0.90	0.90	0.85	0.85	0.90
Q	0.75	L	0.20	C	0.60	0.70	0.45	0.40	0.80
Y	0.30	E	0.15	D	0.30	0.60	0.25	0.25	0.75
T	0.00	N	0.00	E	0.10	0.35	0.00	0.00	0.20
				F	0.00	0.00			0.00

Table C-2 Criteria Levels for Scheme 2

Cost		Experience			Benefits	Credibility	Repeatability	Validation	Relevance to Reliability
W	1.00	W	1.00	A	1.00	1.00	1.00	1.00	1.00
M	0.75	M	0.75	B	0.80	0.80	0.75	0.75	0.80
Q	0.50	L	0.50	C	0.60	0.60	0.50	0.50	0.60
Y	0.25	E	0.25	D	0.40	0.40	0.25	0.25	0.40
T	0.00	N	0.00	E	0.20	0.20	0.00	0.00	0.20
				F	0.00	0.00			0.00

Table C-3 Criteria Levels for Scheme 3

Cost		Experience			Benefits	Credibility	Repeatability	Validation	Relevance to Reliability
W	1.00	W	1.00	A	1.00	1.00	1.00	1.00	1.00
M	0.90	M	0.90	B	0.90	0.90	0.90	0.90	0.90
Q	0.80	L	0.80	C	0.80	0.80	0.80	0.80	0.80
Y	0.70	E	0.70	D	0.70	0.70	0.70	0.70	0.70
T	0.00	N	0.00	E	0.60	0.60	0.00	0.00	0.60
				F	0.00	0.00			0.00

Table C-4 Criteria Levels for Scheme 4

Cost		Experience			Benefits	Credibility	Repeatability	Validation	Relevance to Reliability
W	1.00	W	1.00	A	1.00	1.00	1.00	1.00	1.00
M	0.30	M	0.30	B	0.40	0.40	0.30	0.30	0.40
Q	0.20	L	0.20	C	0.30	0.30	0.20	0.20	0.30
Y	0.10	E	0.10	D	0.20	0.20	0.10	0.10	0.20
T	0.00	N	0.00	E	0.10	0.10	0.00	0.00	0.10
				F	0.00	0.00			0.00

Table C-5 Criteria Levels for Scheme 5

Cost		Experience			Benefits	Credibility	Repeatability	Validation	Relevance to Reliability
W	1.00	W	1.00	A	1.00	1.00	1.00	1.00	1.00
M	0.90	M	0.30	B	0.90	0.40	0.90	0.30	0.90
Q	0.80	L	0.20	C	0.80	0.30	0.80	0.20	0.80
Y	0.70	E	0.10	D	0.70	0.20	0.70	0.10	0.70
T	0.00	N	0.00	E	0.60	0.10	0.00	0.00	0.60
				F	0.00	0.00			0.00

C.1.2 Rates Corresponding to Criteria Level Variations

Table C-6 Rates for the Requirements Phase

Measure	Scheme 1	Scheme 2	Scheme 3	Scheme 4	Scheme 5
Cause & effect graphing	.45	.41	.66	.19	.44
Error distribution	.70	.65	.78	.51	.70
Fault density	.73	.69	.81	.59	.78
Fault number days	.63	.62	.70	.44	.53
Feature point analysis	.44	.44	.65	.21	.44
Function point analysis	.51	.49	.67	.30	.54
Number of faults remaining (error seeding)	.45	.43	.62	.18	.38
Requirements compliance	.52	.50	.69	.28	.49
Requirements specification change requests	.71	.68	.79	.55	.72
Reviews, inspections and walkthroughs	.62	.61	.71	.48	.60
Software capability maturity model	.62	.57	.70	.43	.61

Table C-7 Rates for the Design Phase

Measure	Scheme 1	Scheme 2	Scheme 3	Scheme 4	Scheme 5
Cause & effect graphing	.43	.40	.64	.19	.42
Cohesion	.45	.45	.61	.26	.41
Completeness	.33	.36	.56	.20	.35
Cyclomatic complexity	.74	.69	.75	.60	.71
Data flow complexity	.63	.62	.73	.42	.57
Design defect density	.77	.73	.83	.60	.76
Error distribution	.70	.65	.78	.51	.70
Fault density	.76	.72	.82	.60	.79
Fault number days	.73	.67	.80	.47	.63
Feature point analysis	.47	.46	.69	.22	.48
Function point analysis	.54	.51	.71	.31	.57
Graph-theoretic static architecture complexity	.52	.48	.70	.30	.49
Man hours per major defect detected	.65	.63	.71	.51	.65
Mean time to failure					
Minimal unit test case determination	.60	.58	.70	.44	.56
Number of faults remaining (error seeding)	.45	.43	.62	.18	.38
Requirements compliance	.50	.50	.68	.28	.48
Requirements specification change requests	.71	.68	.78	.55	.70
Requirements traceability	.58	.57	.71	.40	.56
Reviews, inspections and walkthroughs	.61	.60	.71	.48	.59
Software capability maturity model	.62	.58	.70	.44	.61
System design complexity	.56	.55	.72	.34	.51

Table C-8 Rates for the Implementation Phase

Measure	Scheme 1	Scheme 2	Scheme 3	Scheme 4	Scheme 5
Bugs per line of code (Gaffney estimate)	.44	.45	.58	.28	.40
Cause & effect graphing	.40	.38	.61	.18	.40
Code defect density	.83	.77	.85	.65	.82
Cohesion	.37	.38	.53	.22	.33
Completeness	.33	.36	.56	.20	.35
Cyclomatic complexity	.77	.72	.80	.61	.75
Data flow complexity	.60	.57	.69	.40	.53
Design defect density	.77	.73	.83	.60	.76
Error distribution	.69	.66	.77	.51	.69
Fault density	.77	.72	.82	.60	.79
Fault number days	.73	.67	.80	.47	.63
Feature point analysis	.47	.46	.67	.21	.45
Function point analysis	.55	.51	.69	.31	.55
Graph-theoretic static architecture complexity	.45	.44	.64	.28	.43
Man hours per major defect detected	.63	.62	.69	.51	.63
Minimal unit test case determination	.65	.63	.75	.46	.61
Modular test coverage					
Mutation testing (error seeding)					
Number of faults remaining (error seeding)	.47	.44	.67	.19	.43
Requirements compliance	.51	.49	.68	.28	.49
Requirements specification change requests	.71	.68	.78	.55	.70
Requirements traceability	.57	.57	.71	.40	.56
Reviews, inspections and walkthroughs	.61	.60	.71	.48	.60
Software capability maturity model	.61	.58	.70	.44	.61
System design complexity	.55	.53	.71	.33	.50

Table C-9 Rates for the Testing Phase

Measure	Scheme 1	Scheme 2	Scheme 3	Scheme 4	Scheme 5
Bugs per line of code (Gaffney estimate)	.37	.38	.50	.25	.32
Cause & effect graphing	.45	.41	.66	.19	.44
Code defect density	.83	.78	.85	.65	.82
Cohesion	.37	.38	.53	.22	.33
Completeness	.33	.36	.56	.20	.35
Cumulative failure profile	.80	.76	.83	.64	.78
Cyclomatic complexity	.74	.69	.78	.60	.73
Data flow complexity	.6	.57	.69	.40	.53
Design defect density	.76	.71	.82	.59	.75
Error distribution	.69	.67	.77	.51	.69
Failure rate	.87	.85	.88	.76	.86
Fault density	.77	.72	.82	.60	.79
Fault number days	.75	.71	.82	.49	.64
Feature point analysis	.43	.43	.62	.20	.41
Function point analysis	.50	.48	.65	.30	.51
Functional test coverage	.61	.57	.76	.29	.56
Graph-theoretic static architecture complexity	.45	.44	.64	.28	.43
Man hours per major defect detected	.65	.63	.71	.53	.65
Mean time to failure	.81	.81	.87	.64	.78
Minimal unit test case determination	.71	.64	.80	.47	.66
Modular test coverage	.70	.70	.83	.53	.67
Mutation testing (error seeding)	.47	.44	.67	.19	.42
Number of faults remaining (error seeding)	.50	.46	.71	.20	.47
Requirements compliance	.50	.48	.68	.27	.48
Requirements specification change requests	.70	.67	.78	.55	.70
Requirements traceability	.57	.56	.71	.40	.56
Reviews, inspections and walkthroughs	.62	.61	.70	.49	.59
Software capability maturity model	.61	.58	.70	.44	.61
System design complexity	.55	.53	.71	.33	.50
Test coverage	.70	.67	.82	.55	.73

C.1.3 Rankings Corresponding to Criteria Level Variations

Table C-10 Rankings for the Requirements Phase

Measure	Scheme 1	Scheme 2	Scheme 3	Scheme 4	Scheme 5
Cause & effect graphing	10	11	9	10	9
Error distribution	3	3	3	3	3
Fault density	1	1	1	1	1
Fault number days	4	4	5	5	7
Feature point analysis	11	9	10	9	10
Function point analysis	8	8	8	7	6
Number of faults remaining (error seeding)	9	10	11	11	11

Requirements compliance	7	7	7	8	8
Requirements specification change requests	2	2	2	2	2
Requirements traceability					
Reviews, inspections and walkthroughs	6	5	4	4	5
Software capability maturity model	5	6	6	6	4

Table C-11 Rankings for the Design Phase

Measure	Scheme 1	Scheme 2	Scheme 3	Scheme 4	Scheme 5
Cause & effect graphing	20	20	18	20	18
Cohesion	18	18	20	17	19
Completeness	21	21	21	19	21
Cyclomatic complexity	3	3	6	3	3
Data flow complexity	8	8	7	11	11
Design defect density	1	1	1	2	2
Error distribution	6	6	4	6	5
Fault density	2	2	2	1	1
Fault number days	4	5	3	8	7
Feature point analysis	17	17	16	18	17
Function point analysis	14	14	10	14	10
Graph-theoretic static architecture complexity	15	16	14	15	15
Man hours per major defect detected	7	7	11	5	6
Minimal unit test case determination	11	11	13	10	13
Number of faults remaining (error seeding)	19	19	19	21	20
Requirements compliance	16	15	17	16	16
Requirements specification change requests	5	4	5	4	4
Requirements traceability	12	12	9	12	12
Reviews, inspections and walkthroughs	10	9	12	7	9
Software capability maturity model	9	10	15	9	8
System design complexity	13	13	8	13	14

Table C-12 Rankings for the Implementation Phase

Measure	Scheme 1	Scheme 2	Scheme 3	Scheme 4	Scheme 5
Bugs per line of code (Gaffney estimate)	20	18	21	16	21
Cause & effect graphing	21	21	20	23	20
Code defect density	1	1	1	1	1
Cohesion	22	22	23	19	23
Completeness	23	23	22	21	22
Cyclomatic complexity	4	4	5	2	4
Data flow complexity	12	12	13	13	14
Design defect density	3	2	2	4	3
Error distribution	7	7	7	6	6
Fault density	2	3	3	3	2
Fault number days	5	6	4	9	8
Feature point analysis	17	17	18	20	17
Function point analysis	15	15	14	15	13

Graph-theoretic static architecture complexity	19	20	19	17	18
Man hours per major defect detected	9	9	15	7	7
Minimal unit test case determination	8	8	8	10	9
Number of faults remaining (error seeding)	18	19	17	22	19
Requirements compliance	16	16	16	18	16
Requirements specification change requests	6	5	6	5	5
Requirements traceability	13	13	9	12	12
Reviews, inspections and walkthroughs	11	10	11	8	11
Software capability maturity model	10	11	12	11	10
System design complexity	14	14	10	14	15

Table C-13 Rankings for the Testing Phase

Measure	Scheme 1	Scheme 2	Scheme 3	Scheme 4	Scheme 5
Bugs per line of code (Gaffney estimate)	29	29	30	24	30
Cause & effect graphing	26	27	24	29	24
Code defect density	2	3	3	2	2
Cohesion	28	28	29	25	29
Completeness	30	30	28	26	28
Cumulative failure profile	4	4	4	4	4
Cyclomatic complexity	8	9	11	6	7
Data flow complexity	18	17	21	18	19
Design defect density	6	6	7	7	6
Error distribution	13	12	13	12	10
Failure rate	1	1	1	1	1
Fault density	5	5	6	5	3
Fault number days	7	7	8	14	14
Feature point analysis	27	26	27	27	27
Function point analysis	22	21	25	20	20
Functional test coverage	17	18	14	21	17
Graph-theoretic static architecture complexity	25	24	26	22	25
Man hours per major defect detected	14	14	18	10	13
Mean time to failure	3	2	2	3	5
Minimal unit test case determination	9	13	10	15	12
Modular test coverage	12	8	5	11	11
Mutation testing (error seeding)	24	25	23	30	26
Number of faults remaining (error seeding)	23	23	15	28	23
Requirements compliance	21	22	22	23	22
Requirements specification change requests	10	11	12	9	9
Requirements traceability	19	19	17	17	18
Reviews, inspections and walkthroughs	15	15	19	13	16
Software capability maturity model	16	16	20	16	15
System design complexity	20	20	16	19	21
Test coverage	11	10	9	8	8

C.2 Sensitivity Analysis on Weight Sets

C.2.1 Variations on Weight Sets

Table C-14 Weights Used in Weight Sensitivity Analysis

	Cost	Benefits	Credibility	Repeatability	Experience	Validation	Relevance to Reliability
Weight 1	0.14	0.14	0.14	0.14	0.14	0.14	0.14
Weight 2	0.13	0.13	0.13	0.13	0.13	0.13	0.25
Weight 3	0.08	0.08	0.17	0.17	0.08	0.08	0.33
Weight 4	0.245	0.045	0.088	0.036	0.130	0.239	0.216
Weight 5	0.20	0.03	0.10	0.17	0.16	0.14	0.20
Weight 6	0	0	0.25	0.25	0.25	0	0.25

C.2.2 Rates Corresponding to Variations on Weight Sets

Table C-15 Rates for the Requirements Phase

Measure	Weight 1	Weight 2	Weight 3	Weight 4	Weight 5	Weight 6
Cause & effect graphing	0.45	0.44	0.46	0.47	0.50	0.48
Completeness	0.41	0.44	0.46	0.52	0.49	0.39
Error distribution	0.70	0.71	0.73	0.79	0.75	0.73
Fault density	0.73	0.69	0.66	0.79	0.78	0.75
Fault number days	0.63	0.55	0.52	0.63	0.66	0.63
Feature point analysis	0.44	0.39	0.36	0.47	0.46	0.38
Function point analysis	0.51	0.46	0.40	0.54	0.54	0.49
Number of faults remaining (error seeding)	0.45	0.40	0.39	0.42	0.46	0.43
Requirements compliance	0.52	0.53	0.53	0.59	0.57	0.53
Requirements specification change requests	0.71	0.71	0.70	0.81	0.77	0.70
Reviews, inspections and walkthroughs	0.62	0.60	0.59	0.64	0.62	0.64
Software capability maturity model	0.62	0.61	0.60	0.67	0.65	0.67

Table C-16 Rates for the Design Phase

Measure	Weight 1	Weight 2	Weight 3	Weight 4	Weight 5	Weight 6
Cause & effect graphing	0.43	0.40	0.41	0.44	0.47	0.45
Cohesion	0.45	0.47	0.47	0.60	0.53	0.38
Completeness	0.33	0.29	0.26	0.39	0.37	0.24
Cyclomatic complexity	0.74	0.69	0.66	0.78	0.80	0.77
Data flow complexity	0.63	0.62	0.63	0.72	0.70	0.61
Design defect density	0.77	0.78	0.79	0.87	0.84	0.80
Error distribution	0.70	0.71	0.73	0.79	0.75	0.73
Fault density	0.76	0.74	0.73	0.83	0.82	0.80
Fault number days	0.73	0.73	0.77	0.79	0.81	0.81
Feature point analysis	0.47	0.44	0.43	0.51	0.51	0.43
Function point analysis	0.54	0.51	0.47	0.59	0.58	0.54
Graph-theoretic static architecture complexity	0.52	0.51	0.53	0.53	0.57	0.58
Man hours per major defect detected	0.65	0.64	0.63	0.76	0.73	0.65

Minimal unit test case determination	0.60	0.52	0.47	0.63	0.63	0.56
Number of faults remaining (error seeding)	0.45	0.40	0.39	0.42	0.46	0.43
Requirements compliance	0.50	0.51	0.51	0.57	0.55	0.51
Requirements specification change requests	0.71	0.71	0.71	0.81	0.77	0.71
Requirements traceability	0.58	0.57	0.57	0.67	0.63	0.55
Reviews, inspections and walkthroughs	0.61	0.59	0.58	0.64	0.62	0.63
Software capability maturity model	0.62	0.61	0.60	0.67	0.65	0.67
System design complexity	0.56	0.60	0.64	0.67	0.65	0.56

Table C-17 Rates for the Implementation Phase

Measure	Weight 1	Weight 2	Weight 3	Weight 4	Weight 5	Weight 6
Bugs per line of code (Gaffney estimate)	0.44	0.45	0.49	0.50	0.52	0.47
Cause & effect graphing	0.40	0.35	0.34	0.39	0.43	0.39
Code defect density	0.83	0.83	0.84	0.89	0.90	0.91
Cohesion	0.37	0.33	0.28	0.48	0.41	0.24
Completeness	0.33	0.29	0.26	0.39	0.37	0.24
Cyclomatic complexity	0.77	0.74	0.72	0.82	0.84	0.82
Data flow complexity	0.60	0.56	0.55	0.67	0.65	0.55
Design defect density	0.77	0.78	0.78	0.87	0.84	0.79
Error distribution	0.69	0.68	0.68	0.76	0.73	0.70
Fault density	0.77	0.75	0.74	0.84	0.83	0.81
Fault number days	0.73	0.73	0.77	0.79	0.81	0.81
Feature point analysis	0.47	0.45	0.44	0.52	0.51	0.44
Function point analysis	0.55	0.52	0.49	0.60	0.59	0.55
Graph-theoretic static architecture complexity	0.45	0.40	0.39	0.43	0.49	0.47
Man hours per major defect detected	0.63	0.60	0.58	0.73	0.70	0.62
Minimal unit test case determination	0.65	0.61	0.59	0.71	0.70	0.65
Number of faults remaining (error seeding)	0.47	0.43	0.43	0.45	0.48	0.45
Requirements compliance	0.51	0.52	0.52	0.58	0.56	0.52
Requirements specification change requests	0.71	0.71	0.71	0.81	0.77	0.71
Requirements traceability	0.57	0.57	0.56	0.67	0.62	0.54
Reviews, inspections and walkthroughs	0.61	0.59	0.59	0.64	0.62	0.64
Software capability maturity model	0.61	0.61	0.59	0.67	0.65	0.67
System design complexity	0.55	0.58	0.62	0.65	0.63	0.54

Table C-18 Rates for the Testing Phase

Measure	Weight 1	Weight 2	Weight 3	Weight 4	Weight 5	Weight 6
Bugs per line of code (Gaffney estimate)	0.37	0.32	0.31	0.39	0.41	0.34
Cause & effect graphing	0.45	0.44	0.46	0.47	0.50	0.48
Code defect density	0.83	0.83	0.84	0.89	0.90	0.91
Cohesion	0.37	0.33	0.28	0.48	0.41	0.24
Completeness	0.33	0.29	0.26	0.39	0.37	0.24
Cumulative failure profile	0.80	0.81	0.84	0.89	0.87	0.86
Cyclomatic complexity	0.74	0.69	0.66	0.78	0.80	0.77
Data flow complexity	0.60	0.56	0.55	0.67	0.65	0.55
Design defect density	0.76	0.76	0.76	0.86	0.83	0.78
Error distribution	0.69	0.68	0.68	0.76	0.73	0.70
Failure rate	0.87	0.88	0.91	0.92	0.93	0.95

Fault density	0.77	0.75	0.74	0.84	0.83	0.82
Fault number days	0.75	0.76	0.80	0.81	0.83	0.84
Feature point analysis	0.43	0.37	0.34	0.46	0.45	0.36
Function point analysis	0.50	0.44	0.38	0.53	0.53	0.47
Functional test coverage	0.61	0.63	0.68	0.66	0.67	0.66
Graph-theoretic static architecture complexity	0.45	0.40	0.39	0.43	0.49	0.47
Man hours per major defect detected	0.65	0.63	0.62	0.75	0.73	0.65
Mean time to failure	0.81	0.83	0.87	0.87	0.87	0.86
Minimal unit test case determination	0.71	0.71	0.72	0.79	0.77	0.74
Modular test coverage	0.70	0.72	0.78	0.76	0.77	0.77
Mutation testing (error seeding)	0.47	0.46	0.48	0.51	0.49	0.45
Number of faults remaining (error seeding)	0.50	0.48	0.51	0.50	0.53	0.51
Requirements compliance	0.50	0.51	0.51	0.57	0.55	0.51
Requirements specification change requests	0.70	0.70	0.70	0.81	0.77	0.70
Requirements traceability	0.57	0.56	0.55	0.66	0.62	0.54
Reviews, inspections and walkthroughs	0.62	0.60	0.60	0.65	0.63	0.65
Software capability maturity model	0.61	0.61	0.59	0.67	0.65	0.67
System design complexity	0.55	0.58	0.62	0.65	0.63	0.54
Test coverage	0.70	0.72	0.75	0.78	0.78	0.79

C.2.3 Rankings Corresponding to Variations on Weight Sets

Table C-19 Rankings for the Requirements Phase

Measure	Rank 1	Rank 2	Rank 3	Rank 4	Rank 5	Rank 6
Cause & effect graphing	10	9	8	11	9	9
Completeness	12	10	9	9	10	11
Error distribution	3	1	1	2	3	2
Fault density	1	3	3	3	1	1
Fault number days	4	6	7	6	4	6
Feature point analysis	11	12	12	10	11	12
Function point analysis	8	8	10	8	8	8
Number of faults remaining (error seeding)	9	11	11	12	12	10
Requirements compliance	7	7	6	7	7	7
Requirements specification change requests	2	2	2	1	2	3
Reviews, inspections and walkthroughs	6	5	5	5	6	5
Software capability maturity model	5	4	4	4	5	4

Table C-20 Rankings for the Design Phase

Measure	Rank 1	Rank 2	Rank 3	Rank 4	Rank 5	Rank 6
Cause & effect graphing	20	19	19	19	19	17
Cohesion	18	17	17	14	17	20
Completeness	21	21	21	21	21	21
Cyclomatic complexity	3	6	6	6	4	4
Data flow complexity	8	8	8	8	8	10
Design defect density	1	1	1	1	1	3
Error distribution	6	5	4	5	6	5
Fault density	2	2	3	2	2	2

Fault number days	4	3	2	4	3	1
Feature point analysis	17	18	18	18	18	18
Function point analysis	14	15	15	15	14	15
Graph-theoretic static architecture complexity	15	16	13	17	15	11
Man hours per major defect detected	7	7	9	7	7	8
Minimal unit test case determination	11	13	16	13	12	13
Number of faults remaining (error seeding)	19	20	20	20	20	19
Requirements compliance	16	14	14	16	16	16
Requirements specification change requests	5	4	5	3	5	6
Requirements traceability	12	12	12	10	11	14
Reviews, inspections and walkthroughs	10	11	11	12	13	9
Software capability maturity model	9	9	10	9	9	7
System design complexity	13	10	7	11	10	12

Table C-21 Rankings for the Implementation Phase

Measure	Rank 1	Rank 2	Rank 3	Rank 4	Rank 5	Rank 6
Bugs per line of code (Gaffney estimate)	20	18	16	18	17	17
Cause & effect graphing	21	21	21	23	21	21
Code defect density	1	1	1	1	1	1
Cohesion	22	22	22	19	22	23
Completeness	23	23	23	22	23	22
Cyclomatic complexity	4	4	5	4	2	2
Data flow complexity	12	14	14	12	10	13
Design defect density	3	2	2	2	3	5
Error distribution	7	7	7	7	7	7
Fault density	2	3	4	3	4	4
Fault number days	5	5	3	6	5	3
Feature point analysis	17	17	18	17	18	20
Function point analysis	15	16	17	15	15	12
Graph-theoretic static architecture complexity	19	20	20	21	19	18
Man hours per major defect detected	9	10	12	8	8	11
Minimal unit test case determination	8	8	9	9	9	9
Number of faults remaining (error seeding)	18	19	19	20	20	19
Requirements compliance	16	15	15	16	16	16
Requirements specification change requests	6	6	6	5	6	6
Requirements traceability	13	13	13	11	13	15
Reviews, inspections and walkthroughs	11	11	11	14	14	10
Software capability maturity model	10	9	10	10	11	8
System design complexity	14	12	8	13	12	14

Table C-22 Rankings for the Testing Phase

Measure	Rank 1	Rank 2	Rank 3	Rank 4	Rank 5	Rank 6
Bugs per line of code (Gaffney estimate)	29	29	28	30	29	28
Cause & effect graphing	26	25	24	26	24	23
Code defect density	2	2	3	2	2	2
Cohesion	28	28	29	25	28	30
Completeness	30	30	30	29	30	29
Cumulative failure profile	4	4	4	3	3	3

Cyclomatic complexity	8	12	14	11	8	10
Data flow complexity	18	19	20	16	16	18
Design defect density	6	5	7	5	7	8
Error distribution	13	13	12	12	13	13
Failure rate	1	1	1	1	1	1
Fault density	5	7	9	6	5	6
Fault number days	7	6	5	7	6	5
Feature point analysis	27	27	27	27	27	27
Function point analysis	22	24	26	22	23	25
Functional test coverage	17	15	13	17	15	15
Graph-theoretic static architecture complexity	25	26	25	28	26	24
Man hours per major defect detected	14	14	15	14	14	16
Mean time to failure	3	3	2	4	4	4
Minimal unit test case determination	9	10	10	9	10	11
Modular test coverage	12	8	6	13	11	9
Mutation testing (error seeding)	24	23	23	23	25	26
Number of faults remaining (error seeding)	23	22	21	24	22	21
Requirements compliance	21	21	22	21	21	22
Requirements specification change requests	10	11	11	8	12	12
Requirements traceability	19	20	19	18	20	20
Reviews, inspections and walkthroughs	15	17	17	20	19	17
Software capability maturity model	16	16	18	15	17	14
System design complexity	20	18	16	19	18	19
Test coverage	11	9	8	10	9	7

C.3 Sensitivity Analysis on Equations

C.3.1 Variations on Aggregation Functions

Users can refer to Chapter 3 Section 3.8.3 for a detailed discussion on aggregation functions.

C.3.2 Rates Corresponding to Variations on Aggregation Functions

Table C-23 Rates for the Requirements Phase

Measure	Equation 1	Equation 2
Cause & effect graphing	0.45	0.57
Completeness	0.41	0.59
Error distribution	0.70	0.70
Fault density	0.73	0.70
Fault number days	0.63	0.62
Feature point analysis	0.44	0.51
Function point analysis	0.51	0.54
Number of faults remaining (error seeding)	0.45	0.53
Requirements compliance	0.52	0.65
Requirements specification change requests	0.71	0.72
Reviews, inspections and walkthroughs	0.62	0.60
Software capability maturity model	0.62	0.59

Table C-24 Rates for the Design Phase

Measure	Equation 1	Equation 2
Cause & effect graphing	0.43	0.54
Cohesion	0.45	0.58
Completeness	0.33	0.43
Cyclomatic complexity	0.74	0.69
Data flow complexity	0.63	0.68
Design defect density	0.77	0.76
Error distribution	0.70	0.70
Fault density	0.76	0.72
Fault number days	0.73	0.72
Feature point analysis	0.47	0.56
Function point analysis	0.54	0.58
Graph-theoretic static architecture complexity	0.52	0.61
Man hours per major defect detected	0.65	0.64
Minimal unit test case determination	0.60	0.59
Number of faults remaining (error seeding)	0.45	0.53
Requirements compliance	0.50	0.63
Requirements specification change requests	0.71	0.72
Requirements traceability	0.58	0.64
Reviews, inspections and walkthroughs	0.61	0.60
Software capability maturity model	0.62	0.59
System design complexity	0.56	0.69

Table C-25 Rates for the Implementation Phase

Measure	Equation 1	Equation 2
Bugs per line of code (Gaffney estimate)	0.44	0.59
Cause & effect graphing	0.40	0.49
Code defect density	0.83	0.76
Cohesion	0.37	0.44
Completeness	0.33	0.43
Cyclomatic complexity	0.77	0.71
Data flow complexity	0.60	0.64
Design defect density	0.77	0.76
Error distribution	0.69	0.68
Fault density	0.77	0.73
Fault number days	0.73	0.72
Feature point analysis	0.47	0.57
Function point analysis	0.55	0.59
Graph-theoretic static architecture complexity	0.45	0.54
Man hours per major defect detected	0.63	0.62
Minimal unit test case determination	0.65	0.65
Number of faults remaining (error seeding)	0.47	0.55
Requirements compliance	0.51	0.64
Requirements specification change requests	0.71	0.72
Requirements traceability	0.57	0.64
Reviews, inspections and walkthroughs	0.61	0.60
Software capability maturity model	0.61	0.58
System design complexity	0.55	0.68

Table C-26 Rates for the Testing Phase

Measure	Equation 1	Equation 2
Bugs per line of code (Gaffney estimate)	0.37	0.47
Cause & effect graphing	0.45	0.57
Code defect density	0.83	0.76
Cohesion	0.37	0.44
Completeness	0.33	0.43
Cumulative failure profile	0.80	0.76
Cyclomatic complexity	0.74	0.69
Data flow complexity	0.60	0.64
Design defect density	0.76	0.75
Error distribution	0.69	0.68
Failure rate	0.87	0.81
Fault density	0.77	0.73
Fault number days	0.75	0.73
Feature point analysis	0.43	0.50
Function point analysis	0.50	0.53
Functional test coverage	0.61	0.71
Graph-theoretic static architecture complexity	0.45	0.54
Man hours per major defect detected	0.65	0.64
Mean time to failure	0.81	0.80
Minimal unit test case determination	0.71	0.71
Modular test coverage	0.70	0.74
Mutation testing (error seeding)	0.47	0.57
Number of faults remaining (error seeding)	0.50	0.59
Requirements compliance	0.50	0.63
Requirements specification change requests	0.70	0.72
Requirements traceability	0.57	0.63
Reviews, inspections and walkthroughs	0.62	0.61
Software capability maturity model	0.61	0.58
System design complexity	0.55	0.68
Test coverage	0.70	0.66

C.3.3 Rates Corresponding to Variations on Aggregation Functions

Table C-27 Rankings for the Requirements Phase

Measure	Equation 1	Equation 2
Cause & effect graphing	10	9
Completeness	12	7
Error distribution	3	2
Fault density	1	3
Fault number days	4	5
Feature point analysis	11	12
Function point analysis	8	10
Number of faults remaining (error seeding)	9	11
Requirements compliance	7	4
Requirements specification change requests	2	1
Reviews, inspections and walkthroughs	6	6
Software capability maturity model	5	8

Table C-28 Rankings for the Design Phase

Measure	Equation 1	Equation 2
Cause & effect graphing	20	19
Cohesion	18	17
Completeness	21	21
Cyclomatic complexity	3	7
Data flow complexity	8	8
Design defect density	1	1
Error distribution	6	5
Fault density	2	2
Fault number days	4	4
Feature point analysis	17	18
Function point analysis	14	16
Graph-theoretic static architecture complexity	15	12
Man hours per major defect detected	7	9
Minimal unit test case determination	11	14
Number of faults remaining (error seeding)	19	20
Requirements compliance	16	11
Requirements specification change requests	5	3
Requirements traceability	12	10
Reviews, inspections and walkthroughs	10	13
Software capability maturity model	9	15
System design complexity	13	6

Table C-29 Rankings for the Implementation Phase

Measure	Equation 1	Equation 2
Bugs per line of code (Gaffney estimate)	20	16
Cause & effect graphing	21	21
Code defect density	1	2
Cohesion	22	22
Completeness	23	23
Cyclomatic complexity	4	6
Data flow complexity	12	10
Design defect density	3	1
Error distribution	7	7
Fault density	2	3
Fault number days	5	5
Feature point analysis	17	18
Function point analysis	15	15
Graph-theoretic static architecture complexity	19	20
Man hours per major defect detected	9	13
Minimal unit test case determination	8	9
Number of faults remaining (error seeding)	18	19
Requirements compliance	16	11
Requirements specification change requests	6	4
Requirements traceability	13	12
Reviews, inspections and walkthroughs	11	14
Software capability maturity model	10	17
System design complexity	14	8

Table C-30 Rankings for the Testing Phase

Measure	Equation 1	Equation 2
Bugs per line of code (Gaffney estimate)	29	28
Cause & effect graphing	26	24
Code defect density	2	4
Cohesion	28	29
Completeness	30	30
Cumulative failure profile	4	3
Cyclomatic complexity	8	12
Data flow complexity	18	17
Design defect density	6	5
Error distribution	13	13
Failure rate	1	1
Fault density	5	7
Fault number days	7	8
Feature point analysis	27	27
Function point analysis	22	26
Functional test coverage	17	10
Graph-theoretic static architecture complexity	25	25
Man hours per major defect detected	14	16
Mean time to failure	3	2
Minimal unit test case determination	9	11
Modular test coverage	12	6
Mutation testing (error seeding)	24	23
Number of faults remaining (error seeding)	23	21
Requirements compliance	21	19
Requirements specification change requests	10	9
Requirements traceability	19	18
Reviews, inspections and walkthroughs	15	20
Software capability maturity model	16	22
System design complexity	20	14
Test coverage	11	15

C.4 Ranking Criteria Validation Experiment

This experiment attempts to validate the ranking criteria chosen in the study. This is done by varying the weights of the different ranking criteria. Table C-31 provides the static weight sets (explained below) used in this experiment. Table C-32 provides the variations on the weight sets, the correlation coefficients, and the virtual distances, which indicate how close a variation is to the pre-selected 5 static weight sets.

The rates of thirty pre-selected measures are computed for each candidate weight set and we adopt the following notations:

$Rate_1$	The rate set corresponding to weight set 1 in Table C-31
$Rate_2$	The rate set corresponding to weight set 2 in Table C-31
$Rate_3$	The rate set corresponding to weight set 3 in Table C-31
$Rate_4$	The rate set corresponding to weight set 4 in Table C-31
$Rate_5$	The rate set corresponding to weight set 5 in Table C-31
Rate(i)	The rate set corresponding to the *ith* variation on weight set in Table C-32
ρ_1	The correlation coefficient of $Rate_1$ and Rate(i)
ρ_2	The correlation coefficient of $Rate_2$ and Rate(i)
ρ_3	The correlation coefficient of $Rate_3$ and Rate(i)
ρ_4	The correlation coefficient of $Rate_4$ and Rate(i)
ρ_5	The correlation coefficient of $Rate_5$ and Rate(i)

The virtual distance VD for each variation in Table C-32 is defined as

$$VD = \sum_{j=1}^{5} (1 - \rho_j)^2$$

VD is sorted in ascending order in Table C-32.

Table C-31 Static Weight Sets

Weight Set	Cost	Benefits	Credibility	Repeatability	Experience	Validation	Relevance to Reliability
1	0.14	0.14	0.14	0.14	0.14	0.14	0.14
2	0.13	0.13	0.13	0.13	0.13	0.13	0.25
3	0.08	0.08	0.17	0.17	0.08	0.08	0.33
4	0.245	0.045	0.088	0.036	0.130	0.239	0.216
5	0.20	0.03	0.10	0.17	0.16	0.14	0.20

Table C-32 Variations on Weight Set, Correlation Coefficients, and Virtual Distances

Cost	Benefits	Credibility	Repeatability	Experience	Validation	Relevance to Reliability	ρ_1	ρ_2	ρ_3	ρ_4	ρ_5	VD
0.17		0.17	0.17	0.17	0.17	0.17	0.9970	0.9883	0.9680	0.9865	0.9983	0.001352
		0.20	0.20	0.20	0.20	0.20	0.9941	0.9876	0.9683	0.9758	0.9874	0.001941
0.17	0.17		0.17	0.17	0.17	0.17	0.9954	0.9866	0.9598	0.9883	0.9936	0.001991
0.20	0.20		0.20		0.20	0.20	0.9749	0.9845	0.9823	0.9730	0.9804	0.002294
0.17	0.17	0.17		0.17	0.17	0.17	0.9902	0.9851	0.9571	0.9899	0.9831	0.002542
	0.17	0.17	0.17	0.17	0.17	0.17	0.9930	0.9871	0.9680	0.9691	0.9806	0.00257
0.25		0.25		0.25		0.25	0.9800	0.9823	0.9663	0.9752	0.9849	0.002688
0.20		0.20		0.20	0.20	0.20	0.9880	0.9822	0.9537	0.9954	0.9877	0.002777
0.20			0.20	0.20	0.20	0.20	0.9895	0.9798	0.9527	0.9898	0.9947	0.00289
0.17	0.17	0.17	0.17		0.17	0.17	0.9762	0.9839	0.9861	0.9620	0.9769	0.002993
0.17	0.17	0.17	0.17	0.17		0.17	0.9867	0.9846	0.9764	0.9549	0.9849	0.003232
0.20	0.20	0.20		0.20		0.20	0.9812	0.9839	0.9678	0.9672	0.9769	0.003255
0.20	0.20	0.20			0.20	0.20	0.9688	0.9832	0.9795	0.9756	0.9675	0.003327
0.20		0.20	0.20	0.20		0.20	0.9828	0.9801	0.9725	0.9576	0.9887	0.003375
0.20	0.20		0.20	0.20		0.20	0.9837	0.9816	0.9671	0.9611	0.9862	0.00339

Cost	Benefits	Credibility	Repeatability	Experience	Validation	Relevance to Reliability	ρ_1	ρ_2	ρ_3	ρ_4	ρ_5	VD
			0.25	0.25	0.25	0.25	0.9867	0.9796	0.9535	0.9782	0.9828	0.003526
			0.33		0.33	0.33	0.9683	0.9816	0.9824	0.9649	0.9717	0.00368
	0.20		0.20	0.20	0.20	0.20	0.9873	0.9810	0.9554	0.9719	0.9768	0.003841
0.20		0.20	0.20		0.20	0.20	0.9684	0.9760	0.9793	0.9617	0.9768	0.004014
	0.25		0.25		0.25	0.25	0.9719	0.9849	0.9844	0.9592	0.9663	0.004057
0.25		0.25			0.25	0.25	0.9617	0.9764	0.9737	0.9789	0.9696	0.004086
		0.25	0.25	0.25		0.25	0.9836	0.9837	0.9776	0.9481	0.9795	0.004146
		0.25	0.25		0.25	0.25	0.9715	0.9821	0.9871	0.9551	0.9701	0.004211
0.25			0.25		0.25	0.25	0.9616	0.9712	0.9701	0.9696	0.9767	0.004664
			0.33	0.33		0.33	0.9787	0.9794	0.9668	0.9526	0.9789	0.004676
0.25			0.25	0.25		0.25	0.9755	0.9726	0.9584	0.9615	0.9876	0.004722
		0.25		0.25	0.25	0.25	0.9803	0.9771	0.9497	0.9788	0.9707	0.004744
		0.33		0.33		0.33	0.9748	0.9807	0.9664	0.9580	0.9674	0.004965
	0.20	0.20	0.20		0.20	0.20	0.9727	0.9832	0.9868	0.9493	0.9640	0.005072
		0.33			0.33	0.33	0.9602	0.9787	0.9779	0.9661	0.9562	0.005601
	0.20	0.20	0.20	0.20		0.20	0.9807	0.9813	0.9746	0.9394	0.9694	0.005979
	0.20	0.20		0.20	0.20	0.20	0.9777	0.9751	0.9484	0.9690	0.9619	0.006203
	0.25		0.25	0.25		0.25	0.9772	0.9783	0.9654	0.9433	0.9684	0.006396
0.25	0.25				0.25	0.25	0.9529	0.9709	0.9607	0.9796	0.9570	0.006871
0.20	0.20			0.20	0.20	0.20	0.9734	0.9679	0.9314	0.9866	0.9694	0.00756
	0.25	0.25			0.25	0.25	0.9585	0.9762	0.9742	0.9545	0.9460	0.007939
0.25	0.25			0.25		0.25	0.9620	0.9658	0.9403	0.9636	0.9624	0.00891
	0.25	0.25		0.25		0.25	0.9671	0.9731	0.9587	0.9421	0.9514	0.009218
0.25				0.25	0.25	0.25	0.9665	0.9602	0.9225	0.9895	0.9706	0.00969
0.33				0.33		0.33	0.9534	0.9568	0.9307	0.9678	0.9656	0.011067
0.33					0.33	0.33	0.9368	0.9555	0.9460	0.9779	0.9527	0.011615
					0.50	0.50	0.9371	0.9606	0.9532	0.9648	0.9384	0.012728
				0.33	0.33	0.33	0.9573	0.9540	0.9177	0.9702	0.9507	0.014034
	0.33				0.33	0.33	0.9401	0.9622	0.9540	0.9539	0.9306	0.014068
	0.25			0.25	0.25	0.25	0.9580	0.9554	0.9203	0.9616	0.9440	0.014706
				0.50		0.50	0.9479	0.9558	0.9316	0.9480	0.9452	0.015046

Cost	Benefits	Credibility	Repeatability	Experience	Validation	Relevance to Reliability	ρ_1	ρ_2	ρ_3	ρ_4	ρ_5	VD
0.17	0.17	0.17	0.17	0.17	0.17		0.9769	0.9422	0.9021	0.9484	0.9652	0.017337
0.20	0.20	0.20	0.20		0.20		0.9585	0.9370	0.9222	0.9286	0.9523	0.019122
	0.33			0.33		0.33	0.9437	0.9515	0.9280	0.9320	0.9303	0.020191
	0.25	0.25	0.25		0.25		0.9592	0.9412	0.9282	0.9151	0.9388	0.021238
0.25	0.25	0.25			0.25		0.9525	0.9324	0.9025	0.9517	0.9408	0.022182
0.20		0.20	0.20	0.20	0.20		0.9668	0.9296	0.8883	0.9457	0.9628	0.022861
	0.20	0.20	0.20	0.20	0.20		0.9686	0.9365	0.8976	0.9301	0.9467	0.02323
		0.33	0.33		0.33		0.9528	0.9324	0.9199	0.9176	0.9426	0.023307
		0.25	0.25	0.25	0.25		0.9657	0.9310	0.8907	0.9339	0.9509	0.024663
	0.33		0.33		0.33		0.9528	0.9297	0.9047	0.9197	0.9337	0.027095
0.20	0.20	0.20	0.20			0.20	0.9231	0.9427	0.9670	0.8906	0.9292	0.027268
0.25	0.25		0.25		0.25		0.9465	0.9191	0.8922	0.9318	0.9459	0.028609
0.25	0.25	0.25				0.25	0.9109	0.9431	0.9656	0.9018	0.9158	0.029098
0.25	0.25		0.25			0.25	0.9120	0.9375	0.9617	0.8914	0.9261	0.030363
		0.33	0.33			0.33	0.9188	0.9437	0.9740	0.8813	0.9227	0.030506
		0.50			0.50		0.9409	0.9227	0.8935	0.9375	0.9228	0.030684
	0.25	0.25	0.25			0.25	0.9236	0.9472	0.9739	0.8787	0.9177	0.030779
	0.33		0.33			0.33	0.9168	0.9476	0.9751	0.8806	0.9158	0.031621
0.25		0.25	0.25		0.25		0.9365	0.9128	0.8985	0.9169	0.9414	0.032288
0.20	0.20	0.20		0.20	0.20		0.9554	0.9194	0.8647	0.9470	0.9391	0.033298
		0.50				0.50	0.9038	0.9448	0.9751	0.8915	0.9058	0.033555
			0.50			0.50	0.9069	0.9401	0.9723	0.8814	0.9193	0.033577
	0.33	0.33			0.33		0.9400	0.9248	0.8972	0.9213	0.9095	0.034208
0.20	0.20		0.20	0.20	0.20		0.9558	0.9151	0.8624	0.9383	0.9470	0.034715
0.25		0.25	0.25			0.25	0.9068	0.9272	0.9544	0.8829	0.9233	0.03564
0.33		0.33				0.33	0.8940	0.9285	0.9549	0.8983	0.9125	0.036393
			0.50		0.50		0.9356	0.9089	0.8831	0.9162	0.9305	0.037956
0.33		0.33			0.33		0.9269	0.9038	0.8729	0.9430	0.9297	0.038948
0.20	0.20	0.20	0.20	0.20			0.9482	0.9125	0.8828	0.8933	0.9384	0.039258
	0.33	0.33				0.33	0.9027	0.9400	0.9654	0.8786	0.8922	0.040621
0.25		0.25		0.25	0.25		0.9436	0.9045	0.8472	0.9456	0.9363	0.042688

Cost	Benefits	Credibility	Repeatability	Experience	Validation	Relevance to Reliability	ρ_1	ρ_2	ρ_3	ρ_4	ρ_5	VD
	0.25		0.25	0.25	0.25		0.9459	0.9082	0.8569	0.9165	0.9249	0.044455
	0.25	0.25	0.25	0.25			0.9429	0.9105	0.8825	0.8736	0.9191	0.04759
		0.33	0.33	0.33			0.9401	0.9038	0.8747	0.8770	0.9259	0.049185
0.33			0.33			0.33	0.8832	0.9098	0.9377	0.8747	0.9110	0.049282
	0.25	0.25		0.25	0.25		0.9371	0.9046	0.8516	0.9172	0.9088	0.050252
0.25			0.25	0.25	0.25		0.9362	0.8924	0.8378	0.9284	0.9368	0.051068
0.33	0.33					0.33	0.8728	0.9166	0.9380	0.8884	0.8878	0.052021
		0.33		0.33	0.33		0.9344	0.8985	0.8425	0.9238	0.9139	0.052607
			0.33	0.33	0.33		0.9362	0.8951	0.8413	0.9157	0.9239	0.053158
0.25		0.25	0.25	0.25			0.9307	0.8915	0.8611	0.8840	0.9317	0.053987
0.25	0.25	0.25		0.25			0.9286	0.8900	0.8409	0.8947	0.9122	0.0613
						1.00	0.8552	0.9137	0.9458	0.8712	0.8682	0.065319
0.33			0.33		0.33		0.9023	0.8719	0.8449	0.9028	0.9168	0.06636
	0.50					0.50	0.8639	0.9151	0.9404	0.8598	0.8585	0.068943
0.25	0.25		0.25	0.25			0.9152	0.8702	0.8249	0.8688	0.9100	0.080004
0.33	0.33				0.33		0.8936	0.8666	0.8142	0.9268	0.8878	0.081589
0.33		0.33		0.33			0.9091	0.8651	0.8127	0.8882	0.9065	0.082756
0.50						0.50	0.8318	0.8796	0.9058	0.8684	0.8664	0.086819
		0.50		0.50			0.9094	0.8698	0.8186	0.8679	0.8862	0.088454
	0.33		0.33	0.33			0.9103	0.8692	0.8255	0.8454	0.8871	0.092251
	0.33	0.33		0.33			0.9064	0.8726	0.8260	0.8548	0.8718	0.092812
0.25	0.25			0.25	0.25		0.9012	0.8586	0.7867	0.9108	0.8877	0.095858
	0.50				0.50		0.8765	0.8562	0.8064	0.8857	0.8447	0.11058
			0.50	0.50			0.8947	0.8478	0.8010	0.8391	0.8844	0.11309
0.33			0.33	0.33			0.8765	0.8267	0.7796	0.8421	0.8856	0.131905
	0.33			0.33	0.33		0.8772	0.8388	0.7692	0.8730	0.8493	0.133197
0.33				0.33	0.33		0.8721	0.8256	0.7500	0.8956	0.8698	0.137166
	0.33	0.33	0.33				0.8604	0.8508	0.8694	0.7691	0.8391	0.137975
					1.00		0.8476	0.8221	0.7664	0.8846	0.8341	0.150322
0.25	0.25	0.25	0.25				0.8440	0.8293	0.8437	0.7770	0.8437	0.152062
		1.00					0.8376	0.8358	0.8569	0.7705	0.8170	0.159983

Cost	Benefits	Credibility	Repeatability	Experience	Validation	Relevance to Reliability	ρ_1	ρ_2	ρ_3	ρ_4	ρ_5	VD
				0.50	0.50		0.8600	0.8173	0.7430	0.8687	0.8420	0.16123
		0.50	0.50				0.8348	0.8227	0.8465	0.7508	0.8294	0.173497
0.50					0.50		0.8179	0.7866	0.7315	0.8785	0.8336	0.193284
0.33	0.33	0.33					0.8147	0.8056	0.8110	0.7738	0.8079	0.195882
	0.50	0.50					0.8241	0.8244	0.8359	0.7455	0.7812	0.201355
	0.50		0.50				0.8168	0.8025	0.8181	0.7149	0.7965	0.228317
0.33	0.33			0.33			0.8264	0.7761	0.7010	0.8137	0.8146	0.238778
0.33		0.33	0.33				0.7858	0.7687	0.7867	0.7311	0.8024	0.256267
0.33	0.33		0.33				0.7657	0.7445	0.7535	0.7052	0.7778	0.317251
	0.50			0.50			0.7929	0.7491	0.6772	0.7572	0.7572	0.327914
			1.00				0.7403	0.7209	0.7443	0.6512	0.7483	0.395721
0.50				0.50			0.7578	0.7005	0.6196	0.7661	0.7655	0.402775
0.50		0.50					0.7149	0.7025	0.7128	0.6997	0.7371	0.411589
				1.00			0.7567	0.7040	0.6227	0.7385	0.7369	0.426784
0.50			0.50				0.6251	0.6008	0.6138	0.5873	0.6634	0.732723
	1.00						0.5366	0.5406	0.5315	0.4629	0.4682	1.216517
0.50	0.50						0.4997	0.4843	0.4675	0.5070	0.5131	1.27997
1.00							0.1064	0.0880	0.0779	0.1679	0.1702	3.861543

APPENDIX D DATA COLLECTED FOR THE MISSING MEASURES

This appendix contains the levels assessed by the University of Maryland team for each of the missing measures.

Table D-1 Ranking Criteria Levels for the Missing Measures

Measure	Cost	Benefits	Credibility	Repeatability	Experience	Validation
Class coupling	W	F+	C	A	M	A
Class hierarchy nesting level	W	F+	C	A	M	A
Coverage	Q	D	A	B	M+	A
Full Function Point	Q	E+	D+	C	M	C+
Lack of Cohesion in Methods	W	F+	D	A	M	A
Mutation Score	Q	C	C	A	E	A
Number of Children	W	F+	C	A	M	A
Number of Class methods	W	F+	C	A	M	A
Number of Key Classes	W	F+	D	E	M	A
Weighted Method Complexity	W	F+	D	A	M	A

Table D-2 Level for the Relevance to Reliability Criterion (Per Phase)

Measure	Requirements	Design	Implementation	Testing
Class coupling		D -	D -	E +
Class hierarchy nesting level		D -	D -	E +
Coverage				B
Full Function Point	E -	D -	D -	F +
Lack of Cohesion in Methods		D -	D -	E +
Mutation Score			D	D
Number of Children		D -	D -	E +
Number of Class methods		D -	D -	E +
Number of Key Classes		D -	D -	E +
Weighted Method Complexity		D -	D -	E +

APPENDIX E GLOSSARY

Aggregation framework

An aggregation framework (also called aggregation scheme) is defined as the set {aggregation equation, weights, a letter-real conversion scheme}.

Aggregated rate

An aggregated rate (also called measure's rate, or rate) is a real value ranging from 0 to 1. The rate is an indicator of the measure's capability to predict software reliability. The higher the aggregated rate value, the more capable the measure is of predicting software reliability.

Aggregation scheme

See the entry for "Aggregation framework".

Algorithm

An algorithm is a straightforward procedure for combining two or more measures. The output of the algorithm represents one or more characteristics of the software product under study.

Architectural model

A model that puts emphasis on the architecture of the software and derives reliability estimates by combining estimates obtained for the different modules of the software.

Attribute (object-oriented)

Some data (state information) for which each object in a class has its own value.

Availability

Term used to define whether or not a measure is available in a particular software development phase.

Class (object-oriented)

A group or set of objects sharing common attributes.

Cohesion

The manner and degree to which the tasks performed by a single software module are related to one another.

Coupling

The manner and degree of interdependence between software modules.

Derived measure

An intermediate value which is neither an indicator nor a primitive measure.

Early prediction model

A model that uses the analyst's knowledge of the software development process to predict reliability early in the software development process, i.e., during the design or coding phase.

Extrinsic characteristics

The extrinsic validity of the measure is defined by its degree of relevance to reliability.

Family

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.

Fault-tolerant system

A system that is able to continue normal operation despite the presence of faults.

Indicator

Estimates or evaluations that provide a basis for decision-making. In this particular study, reliability is deemed an appropriate indicator for decision making.

Input domain model

A model that uses the properties of the input domain of the software to derive a correctness probability estimate from test cases that executed properly.

Intrinsic validity

The intrinsic validity of a measure depends on how well it performs with respect to quality ranking criteria (defined later) and cost effectiveness ranking criteria.

Method (object-oriented)

A function or behavior of an object.

Model

A procedure for combining measures to produce an estimate or evaluation based on a series of assumptions. Each assumption is an idealization of reality. The procedure is logically deduced from the assumptions.

Mutation testing

A testing methodology in which two or more program mutations are executed using the same test cases to evaluate the ability of the test cases to detect differences in the mutations.

Object-oriented development

A software development technique in which a system or component is expressed in terms of objects and connections between those objects.

Primitive measure

Primitive measures are values resulting from the application of rules to software attributes.

Ranking criterion

Used to assess the reliability prediction potential of a software engineering measure.

Ranking criteria level

Each ranking criterion is quantified into levels. These levels provide a qualitative estimate of the "goodness" of a measure with respect to the criterion.

Reliability growth model

A model that characterizes the improvement in reliability that results from correction of faults. This category of models uses failure data information and trends observed in the failure data to derive reliability predictions.

Root of RPS

A root of a software reliability prediction system is a measure that constitutes the starting point of the system and should be supplemented by additional measures which will complete the system.

Reliability Prediction System (RPS)

A complete set of software engineering measures from which software reliability can be predicted.

Rule

A rule is a mapping of the software attribute to a subset of the field of real or integer numbers.

Software Attribute

Software attributes are properties of the software, such as the functional size, structural complexity, etc.

Software development phase

The period of time that begins when a software product is conceived and ends when the software is no longer available for use. The software development phase typically includes a concept phase, requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase, and, retirement phase. In this study the above phases are grouped into four phases:

- the requirements phase, which includes the concept and requirements phases,
- the design phase, which includes the design phase,
- the implementation phase, which includes the implementation and unit testing phases,
- the test phase, which includes the test phases.

Software Engineering

1. The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.
2. The study of approaches as in 1.

Software Engineering Measure

A measure of the degree to which a software system, component, or process possesses a given software attribute.

Software error

A human action that produces an incorrect result.

Software failure

The inability of a system or component to perform its required functions within specified performance requirements.

Software fault

An incorrect step, process, or data definition in a computer program.

Software Reliability

Software reliability is defined here as the probability of successfully performing the safety function on demand with no unintended functions that might affect safety.

Software-based safety critical digital I&C system

A computer-based sub-system that controls and monitors the safe execution of a process plant or airborne system etc. This computer-based system performs the following functions:

(1) measurement of process variables such as temperature, flow rate, and pressure, (2) execution of a control strategy, (3) actuation of such devices as valves and switches that enable the process to implement the control strategy, and (4) generation of reports to engineers and management indicating equipment status and performance.

Structural level

A structural level is any of the following: software attribute, primitive measure, derived measure or indicator.

Support measure

Measure used to supplement the root of a RPS.

NRC FORM 335 (2-89) NRCM 1102, 3201, 3202 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/GR-0019 UMD-RE-2000-23
2. TITLE AND SUBTITLE Software Engineeing Measures for Predicting Software Reliability in Critical Digital Systems	3. DATE REPORT PUBLISHED MONTH: November YEAR: 2000
	4. FIN OR GRANT NUMBER K6007
5. AUTHOR(S) C .Smidts, M. Li	6. TYPE OF REPORT Technical
	7. PERIOD COVERED *(Inclusive Dates)* January 1999 - August 2000

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.)*

University of Maryland
Center for Reliability Engineering
College Park, MD 20742

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 Technology
Office of Nuclear Regulatory Research
U.S. Nuclear Regulatory Commission
Washington, DC 20555-0001

10. SUPPLEMENTARY NOTES

R. Brill, NRC Project Manager

11. ABSTRACT *(200 words or less)*

This report presents the University of Maryland (UMD) research to identify measures and families for the prediction and assessment of the reliability of software-based digital systems.

A set of software engineering measures from which the potential reliability of a digital I&C system can be predicted is developed from a set of 30 pre-selected software engineering measures. These measures are derived from a pool of 78 software engineering measures identified by Lawrence Livermore National Laboratory (LLNL). The concepts of structural classification, software development life-cycle classification, and family are presented. These 30 measures are categorized using these concepts. The concept of RPS and an extended structural representation are introduced to bridge the gap between software engineering measures and reliability. Expert opinion is elicited as the input in ranking the pre-selected 30 measures in terms of software reliability prediction. 10 missing measures are identified and ranked. The potential impact of these 10 missing measures on the ranking of the pre-selected 30 measures is analyzed. The top-ranked measures and families are presented in this report. Use of the families of measures in each software development phase can lead to a quantitative prediction of software reliability.

12. KEY WORDS/DESCRIPTORS *(List words or phrases that will assist researchers in locating the report.)* Software Reliability Measures; Software Reliability Measure classification; Software Measure families; Ranking of Software Reliability Measures; Measures of Software quality	13. AVAILABILITY STATEMENT unlimited
	14. SECURITY CLASSIFICATION *(This Page)* unclassified *(This Report)* unclassified
	15. NUMBER OF PAGES
	16. PRICE

UNITED STATES
NUCLEAR REGULATORY COMMISSION
WASHINGTON, D.C. 20555-0001

1975 25 2000
years