Software Reliability Measurement: A Survey

Xiaobin Li

A Major Report

in

The Department

Of

Computer Science

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

July 2002

ABSTRACT

Software Reliability Measurement: A Survey

Xiaobin Li
Concordia University 2002

In complex software systems, reliability is the most important aspect of software quality, a multi-dimensional property including other factors like functionality, usability, performance, serviceability, capability, installability, maintainability, and documentation. Software reliability engineering is becoming a standard, widespread practice applicable to the different phases of the software development process.

The first chapter of the survey provides an introduction to software measurement. Traditional and object-oriented software metrics are analyzed in detail, comprehensive study of some empirical work is also provided in order to validate the usefulness of the selected software metrics. An overview of software reliability is then introduced from the basic terminologies to the reasons for the need of software reliability. Following is the classification of the existing software reliability measures and measurement tools discussed in several chapters. Firstly, the procedure of software reliability measurement procedure along with a framework is addressed. Secondly, software reliability modeling is introduced in detail together with model classification schemes. Thirdly, the relationship between software reliability engineering and Software development process is outlined. Fourthly, we show a classification of current development tools with some usage information. In the final part of this survey, the research directions of software reliability engineering are explored.

Acknowledgment

I would like to thank my supervisor Dr. Olga Ormandjieva, for her wisdom guidance through the survey and her tireless effort in reading several drafts of this report. I would also like to thank my wife, Xiaoyan Wu, for her love, patience, understanding, and support.

Table of Contents

List of Figures

List of Tables

Chapter1 Introduction

1.1 The Need of Reliable Software

Computers are essential part of our daily lives. They are used in diverse areas for various applications, for example, in air traffic control, nuclear power plants, aircraft, space shuttle, industrial process control, medical equipment, telecommunications and real-time military applications. The reliability of these computer systems is thus not only essential but also critical. A computer system is composed by two major components—hardware and software. Extensive research has been carried out on hardware reliability, however, the growing importance of software dictates that the focus shifts to software reliability. Since the demand for complex software systems has increased more rapidly than the ability to design, implement, test and maintain them, and the reliability of software system has become a major concern for our modern society.

In recent years, software has already become the major source of reported outages in many systems. Some software failures have impaired several high-visibility programs in the health industry, in some cases they even killed people. For example, in 1985 & 1986, the massive Therac-25 radiation therapy machine killed several patients after enjoying a perfect safety record for a long time. According to the later investigation, the reason caused the accident was that the sophisticated controlling system was malfunctioned by some software errors [1]. Another report was from the South West Thames Regional Health Authority (1993). On October 26, 1992, the Computer Aided Dispatch system of the London Ambulance Service broke down right after its installation, paralyzing the

capability of the world's largest ambulance service to handle 5000 daily requests in carrying patients in emergency situations. This led to serious consequences for many critical patients.

A recent inquiry revealed that a software design error and insufficient software testing caused an explosion that ended the maiden flight of the European Space Agency's (ESA) Ariane 5 rocket, less than 40 seconds after lift-off on 4 June 1996. The problems occurred in the flight control system and were caused by a few lines of Ada code containing three unprotected variables. The ESA report revealed that officials did not conduct a pre-flight test of the Ariane 5's inertial-reference system, which would have located the fault. The companies involved in this project had assumed that the same inertial-reference-system software would work in both Ariane4 and Ariane5. The ESA estimates that corrective measures will amount to US$362 million.

In the paper "The Role of Software in Recent Aerospace Accidents*" by Nancy G. Leveson [8], more aerospace accidents caused by unreliable software systems are introduced and analyzed, which includes the loss of the Mars Climate Orbiter in 1999; the destruction of the Mars Polar Lander sometime during the entry, deployment, and landing phase in the following year; the placing of a Milstar satellite in an incorrect and unusable orbit by Titan IV B-32/Centaur launch in 1999, etc.

Generally, software faults are more insidious and much more difficult to handle than physical defects. In theory, software can be made to be error-free, because unlike

hardware, software does not wear out. All design faults are presented from the time the software is installed into the computer. In principle, these faults could be removed completely. Yet the aim of perfect software remains elusive. The causes of software failures are enormous, and are introduced during each phase of software development life cycle. During the operation of software, wrong decisions are made because the particular inputs that triggered the problem had not been tested during the phase when faults could be corrected. Thus, we must determine the reliability of our systems before putting them into operation.

1.2 Reliability and Software Quality

Although software cannot be touched or seen, it is essential to the successful use of computers. In complex software systems, reliability is the most important aspect of software quality, a multi-dimensional property including other factors like functionality, usability, performance, serviceability, capability, installability, maintainability, and documentation. Software quality is one of the three most important software product characteristics, which are quality, cost, and schedule [2]. The purpose of software quality activity is to "identify, monitor, and control all activities, technical and managerial, which are necessary to ensure that the software achieves the specified SIL" and functional performance, safety, reliability and security requirements [3].

We will give a precise definition of software reliability in chapter 3. Informally, it is the probability that the software will work without failure for a specified period of time.

"Failure" means the program in its functioning has not met user requirements in some way.

Reliability represents a user-oriented view of software quality [4]. Saying so, we are comparing it with those developer-oriented measures. Initially, approaches to measure software quality were based on attempting to count the faults or defects found in a program. (There are a lot of related metrics, like faults per 1000 of codes) The problem is that even if faults found were correctly counted, they are still not good measures or status indicators. How much is enough? Is a large number good or bad? Maybe remaining faults are even better. Reliability is a much richer measure. It is customer- or user- oriented rather than developer-oriented. It is related to operation rather than the design of the system. It measures the frequency with which problems occur. It relates directly to operational experience and the influence of faults on the experience [4]. Thus, reliability is easy to be associated with cost. Moreover, it is more suitable for predicting and estimating trends and when objectives will be met.

Hence, understanding software reliability measurement and prediction has been an essential skill for both the software managers and software engineers. Fortunately, researches on software reliability and software reliability engineering (SRE) have been conducted during the past several decades, and more than 50 statistical models have been proposed for estimating of software reliability. Moreover, research on software reliability measurement has been conducted in different institutions and companies. SRE is currently becoming a standard [5]. For example, it is accepted as "best practice" by one of the major developers of telecommunications software (AT&T). Other organizations

that are using, experimenting with, or researching SRE are Alcatel, AT&T, Bellcore, BNR, Cray Research, Hewlett-Packard, Hitachi, IBM Corp., Jet Propulsion Laboratories, MITRE Corp., NASA, NCR Corp., Northern Telecom, and Toshiba [6].

1.3 An Overview Of This Survey

Software metrics are being used by the Software Assurance Technology Center (SATC) at NASA to help improve the reliability by identifying areas of the software requirements specification and code that can potentially cause errors [7]. In chapter2, we introduce basic definitions of software metrics and software measurement. We address the classifications of software metrics and how software metrics are related to software engineering life cycle. Some widely used traditional and object-oriented metrics are selected and addressed in detail. This chapter is rather independent than the rest chapters in this survey and worth its own.

In chapter 3, we provide basic definitions of software reliability and software reliability engineering, as well as some widely used basic terms in SRE.

In chapter 4, software reliability measurement, which is one of the three activities of software reliability engineering, are discussed. We address the procedure of software reliability measurement, as well as the framework. Classification of measures is also presented.

In chapter 5, software reliability modeling is discussed in detail. The reason is that software reliability modeling has received the most attention of SRE. This chapter presents some of the most important models that appeared in the recent literature, from both a historical and application perspective. A classification of models is provided to help understanding and using of models more easily.

In chapter 6, the relationship between SRE and Software Development Process is provided. SRE activities in different software development lifecycle are summarized.

In chapter 7, measurement tools are discussed. We will provide a classification of current developed tools as well as some usages information.

In chapter 8, research directions are discussed.

1.4 Reference

[1] Lee, I., and Iyer, R.K., "*Analysis of Software Halts in Tandem System*," Proceedings of the 3rd International Symposium on Software Reliability Engineering, October 1992, pp.227-236.

[2] PRESSMAN, R. S, "*Software Engineering: A Practitioner's approach*" 4th edition, McGraw-Hill Book Company, 1997.

[3] Gillies, A.C., Software Quality, Theory and management, Chapman Hall Computing Series, London, UK, 1992.

[4] Handbook of Software Reliability Engineering, McGraw Hill, editor M. Lyu, 1996

[5] John D. Musa, "*More Reliable Software Faster And Cheaper*: An overview of software reliability".

Http://members.aol.com/JohnDMusa/ARTweb.htm

[6] S.Rdalal, M.R.Lyu, and C.L.Mallows "*Software Reliabitity*", Bellcore, Lucent Technologies, AT&T Research.

[7] Dr. Linda Rosenberg, Dr. Ted Hammer and Dr. Jack Shaw "*Software Metrics And Reliability*" NASA GSFC

[8] The Role of Software in Recent Aerospace Accidents* Nancy G. Leveson, Ph.D.; Aeronautics and Astronautics Department, Massachusetts Institute of Technology; Cambridge MA leveson@mit.edu and http://sunnyday.mit.edu Keywords: software safety

Chapter2 A Survey of Software Metrics

Abstract

This chapter presents a survey of software metrics. Firstly, it reviews some basic concept of software metrics. Some traditional but still widely used software metrics are introduced next. In the third part, we present our focus – Object Oriented software metrics in detail incorporated with our comments. In order to validate the usefulness of the selected software metrics, in the fourth part, a comprehensive study of some empirical work is provided. At the last part, the conclusion is given and some future work is discussed as well.

2.1 Introduction

Definition of Software Metrics

Intuitively, people think the term metric is restricted to mathematic usage. It is hard to combine the word software and metric together.

The American Heritage Dictionary (Mifflin, 1991) defines a metric as:

1. designating, pertaining to the metric system, or

2. a standard of measurement.

However, in the field of software engineering, software metrics have developed for more than 30 years. There are many relatively similar definitions in different papers

for different purposes. Basically, software metrics is a simple quantitative measure derivable from any attribute of the software life cycle. Yet, in the realm of software engineering and mathematics, not everything is quantitative. Large part of mathematics, including most of logic is not quantitative [4]. Thus, we would like to present software metrics as follows:

Software metrics use mathematical presentation to quantify software development process, product and resources. It offers an efficient way for software engineers to control software development life cycle---to value the collections of software-related activities; measure artifacts, deliverables or documents and distribute software entities.

Ordinarily and traditionally, the measurement of the software process and product are studied and developed for use in predicting or evaluating the software development process. Productivity, product quality and product costs are examples that these metrics applied to. Information gained from these metrics and models can then be used in management and control of the development process, leading one hope to attain overall improvement. From another point of view, software design consumes resources and produces a product. Thus, the measurements of resources are important. Metrics on personnel, software and hardware including performance are used to help the effectiveness of organization.

Good metrics should not just stay on the description level. Instead, the actual prediction or estimation of process, product and resources should be presented. Hence,

a good software metrics should have the quality of simple, robust, easy of maintain, valid and objective (to the greatest extent possible). Moreover, metrics should have data values that belong to appropriate measurement scales for maximum utility in analytic studies and statistical analysis [2].

Classification of Software Metrics

Since software metrics is a relatively large concept, there are many ways to classify it. Broadly, with respecting to software design life cycle, software metrics can be classified as product metrics, process metrics and resources metrics. Same to the definition, the general subdivision of software metrics varies. Here, we present one popular classification based on the work of Fenton [3] and Meyer [4] in table2.1.

Product metrics are measures of the software products at any stage of the life cycle, from requirements to system delivered; process metrics are measures of the process used to obtained these products, such as overall development time, type of methodologies used and the customer satisfaction levels; resources metrics are measures of the behavior and the development environment.

<table>
<tr><td rowspan="5">Process Metrics</td><td colspan="3">Maturity Metrics – organization ~, resource ~, tech-management ~, document standards ~, data-management and analysis ~</td></tr>
<tr><td colspan="3">Management Metrics
1. Project management (milestone ~, risk ~ workflow ~, controlling management database ~)
2. Quality management (customer satisfaction ~, review ~, productivity ~, efficiency ~, quality assurance ~)
3. Configuration Management (change control ~, version control ~)</td></tr>
<tr><td colspan="3">Life Cycle Metrics – problem definition ~, requirement analysis and specification ~, design ~, implementation ~, maintenance ~</td></tr>
<tr><td rowspan="8">Product Metrics</td><td rowspan="3">Internal</td><td>Size metrics</td><td>Providing measures of how big a product is internally</td></tr>
<tr><td>Complexity metrics</td><td>Assessing how complex a product is</td></tr>
<tr><td>Style metrics</td><td>Assessing adherence to writing guidelines for product components</td></tr>
<tr><td rowspan="5">External</td><td>Product reliability metrics</td><td>Assessing the number of remaining defects</td></tr>
<tr><td>Functionality metrics</td><td>Assessing how much useful functionality the product provides</td></tr>
<tr><td>Performance metrics</td><td>Assessing product's use of available resources, computation speed</td></tr>
<tr><td>Usability metrics</td><td>Assessing a product's ease of learning and ease of use</td></tr>
<tr><td>Cost metrics</td><td>Assessing the cost of purchasing and using a product</td></tr>
<tr><td rowspan="3">Resources Metrics</td><td colspan="3">Personnel Metrics – Programming experience ~, Communication level ~, Productivity ~, Team Structure ~</td></tr>
<tr><td colspan="3">Software Metrics – Performance ~, Paradigm ~, Replacement ~</td></tr>
<tr><td colspan="3">Hardware Metrics – Performance ~, Reliability ~, Availability ~</td></tr>
</table>

Note: here, we use ~ to denote metrics

Table2.1 Classification of Software Metrics

Another way to categorize metrics is by the computational methodologies used. Grady pointed out this as primitive metrics and computed metrics [5]. Primitive metrics are those that can be directly observed, such as the program size (in LOC), number of defects observed in unit testing, or total development time for the project. Computed metrics are those that cannot be directly observed but are computed in some manner

from other metrics. Computed metrics are normally used for productivity. Such as LOC produced per person-month (LOC/person-month). Grady indicated that computed metrics are combinations of other metrics and thus are often more valuable in understanding or evaluating the software process than primitive metrics.

Since object-orient approach is becoming more and more popular in today's software development, there is a need to distinguish O-O design from traditional procedural programs. Object oriented system requires not only a different approach to design and implementation, but also a different approach to software metrics. Therefore, we classify software metrics as Object-Oriented metrics and traditional metrics. Traditional metrics are applied in procedure-oriented programs, which typically include Cyclomatic Complexity (CC), Lines of Code (LOC), Comment percentage (CP), Halstead's software metrics and etc. Whereas, object-oriented metrics mainly apply to evaluate object-oriented design in the following key features such as classes, objects, methods, coupling, cohesion, inheritance, polymorphism, encapsulation, and etc. From the point of effectiveness, researches [6, 7] have shown that a combination of selected traditional and object-oriented metrics provides the best results when we analyzing the overall quality of object-oriented system.

Measurement Scale of Software Metrics

While collecting software metrics data, software engineers have a specific purpose. The purpose can be for use in some process models, or product models, or help to build a resource model. Data can be used for calculations or some statistical analysis.

Thus, it is important to make sure of the types of information involved, before the data are actually collected and used. Metrics data are recognized by statisticians-nominal, ordinal, interval, ratio and absolute.

Scale	Possible Operations	Description	Examples
Nominal	=, ≠	Simplest possible measurement; empirical relation system consists only of different classes; no notion of ordering	1.Measure type of program 2. Classifying entities
Ordinal	<, >	Allow rank the various data values only, any mapping that preserves the ordering	1.Programmer productivity (low, medium, high)
Interval	+, -	Data can not only be ranked, but also can exhibit meaningful differences between values (powerful, but rare in practice)	1.Temperature measured on Fahrenheit or centigrade scale
Ratio	/	Measurement denotes a degree in relation to standard where a software entity manifest chosen property (most useful of measurement)	1.Program size, in LOC 2.Temperature (absolute)
Absolute	+, - , *, /	Absolute scale measurement is just counting; the attribute must always be of the form of 'number of occurrences of x in the entity'; only one possible measurement mapping (the actual count); all arithmetic is meaningful	1.Number of students in this class 2.Number of bugs found in one method

Table 2.2 Different Scales in Software Measurement Theory [8, 9]

It is important to be aware of what measurement scale is associated with a given metric. Many proposed metrics have values from an interval, ordinal, or even nominal scale. If the metric values are to be used in mathematical equations designed to represent a model of the software process, metrics associated with a ratio scale may be preferred, since ratio scale data allow most mathematical operations to be meaningfully applied. However, it seems clear that the values of many parameters

essential to the software development process cannot be associated with a ratio scale.

Motivation of software metrics

Software metrics are measures that are used to quantify software product, development resource and process. Thus, they are desired to assess or predict effort and cost of development process as well as the quality of software products [20]. In software development, the need for decreasing the probability of failing programs and decreasing available resources are explicit. Metrics are helpful to identify the causes of defects that have major effect on the software development. Also, metrics make it possible for software engineers to estimate and predict necessary resources for a project, software process, and work products.

Tom DeMarco says: " You cannot control what you can not measure." That is, without metrics there is not easy to determine whether the process or product is improving. Thus, metrics may help to evaluate the software in different phases to provide the indication of evolution effort. From this sense, metrics are helpful to establish the meaningful goals for software evolution. A baseline from which improvements should be measured can also be established. Currently, metrics are widely applied in a product to identify what kind of user requirements are likely to change, what kind of modules or classes are most fault prone, how much testing should be planned and so on.

2.2 Traditional Software Metrics

The early Software Metrics techniques were introduced in software area 30 years ago to measure the complexity of traditional program written by FORTURN, COBOL, C, etc. Traditional software complexity metrics are measures of the ease or difficulty of a programmer performing common programming tasks such as testing, understanding, or maintaining a program. Complexity metrics do not measure the complexity itself, but instead measure the degree to which those characteristics lead to complexity exist within the code and the degree to which those code characteristics occur in the code impact the ease or difficulty of a programmer working with the code. Some traditional complexity metrics still used widely and can be applied to object-oriented programs, such as Line Of Code (LOC), McCabe's Cyclomatic Complexity.

Lines of Code (LOC)[7]

LOC is a widely used metric for program size. Size of a program is used to evaluate the ease of understanding by developers and maintainers. Size can be measured in a variety of ways, which include counting all physical lines of code, the number of statements, the number of blank lines, and the number of comment lines. Lines Of Code (LOC) count all lines. Non-comment Non-blank (NCNB) is sometimes referred to as Source Lines Of Code and counts all lines that are not comments and not blanks. Executable Statements (EXEC) is a count of executable statements regardless of number of physical lines of code. For example, in C, IF statement may be written as:

```
If (i < 20 && i >0){
```

X = 10;

Here, there are 3 LOC, 3NCNB, and 1 EXEC.

Cyclomatic Complexity – v(G)[7,9]

McCabe Cyclomatic complexity is to compute a number v(G) where G stands for the associated graph of the flowchart. v(G) refers to the number of edges minus the number of the nodes in the flowchart. Here, in flowchart graph, nodes mean the statements and decision boxes; edges mean the links between them. The cyclomatic complexity of such a graph can be computed by a simple formula as following as:

$$v(G)=e-n+2,$$

e: the number of edges; *n: the number of nodes in the graph;*

Take, for instance, the flowchart in figure 3. Here, e=9, n=8, then v(G)=e-n+2=9-8+2=3.

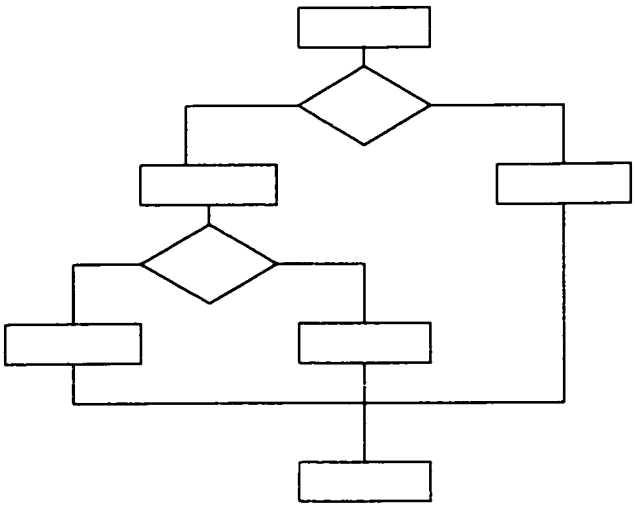

Figure2.1 Cyclomatic complexity flowchart with edges and nodes

Cyclomatic complexity can be used as a measure of program complexity and as a

guide to program development and testing [11]. An algorithm with a low cyclomatic complexity is generally better, but this may imply decreased testing and increased understandability. Thus low cyclomatic complexity is not good for program comprehension and maintenance.

Halstead's software metric

Halstead's software metric is used to estimate the number of errors in the program. It uses the number of distinct operators and the number of distinct operands in a program to develop expressions for the overall program length, volume and the number of remaining defects in a program. The length of the program is estimated:

$$N = N_1 + N_2$$

Where N_1= *total number of operators occurring in a program*

N_2= *total number of operands occurring in a program*

N= *length of the program*

And the Volume of the program is estimated:

$$V = N\log_2(n_1 + n_2)$$

Where V= *volume of the program*

n_1=*number of unique or distinct operators in a program*

n_2=*number of unique or distinct operands in a program*

Thus, we have:

$$N_1 = n_1\log_2 n_1$$

$$N_2 = n_2\log_2 n_2$$

There are tow empirical formulae proposed by Halstead to estimate the number of remaining errors in a program, respectively, are:

$$\hat{E} = \frac{v}{3{,}000} \qquad \text{and} \qquad \hat{E} = \frac{A}{3{,}000}$$

where $$A = \left(\frac{V}{\frac{2n_2}{n_1 N_2}} \right)^{\frac{2}{3}}$$

E= number of errors in the program

Halstead's theory of software metric is probably the most well known technique to measure the complexity in a software program and the amount of difficulty involved in testing a debugging the software.

2.3 Object-Oriented Software Metrics

Object-oriented technology is very popular in today's software industry. Compared to traditional procedure-oriented technology, they are often heralded as the silver bullet for enhancing software quality and reducing integrating and maintenance cost. However, there are substantially many challenges for object-oriented technology to deal with. For instance, there are no unified solutions to validate and evaluate the effectiveness of the existed software metrics, and there is no simplified industry-standard metrics suite to predict and assess the quality of object-oriented system.

Traditional procedure-oriented technology concerns with specific algorithm in procedures or functions, while object-oriented technology uses object as the key building blocks, which concentrates on the abstract object models of the real world. Even though some traditional metrics such as LOC and Cyclomatic Complexity still adapt to the new paradigm, object oriented technology which presents the different perspectives of the real world, should have some unique metrics to reflect its characteristics.

So far, there are various software metrics proposed for measuring object-oriented programs, which concern the different aspects of the object-oriented paradigm. In the following section, we narrow down our focus on the metrics that reflect object oriented key features such as inheritance, polymorphism, encapsulation, coupling and cohesion. Having surveyed plenty of papers, we introduce some popular or meaningful metrics suite such as Chidamber and Kemerer metrics, Briand et al coupling metrics, MOOD metrics, Benlarbi and Melo polymorphism metrics, and Bieman and Kang metrics by identifying what properties each metric going to reflect within the context of object orientation. Thus, their relationship can be seen in the following table.

Properties	Object-oriented metrics	Metrics suite and reference
Coupling	CBO, RFC	CK [15, 16]
	COF	MOOD [19, 21, 22]
	IFCAIC, ACAIC, OCAIC, FCAEC, DCAEC, OCAEC, IFCMIC, ACMIC, OCMIC, FCMEC, DCMEC, OCMEC, IFMMIC, AMMIC, OMMIC, FMMEC, DMMEC, OMMEC	Briand et al [14,17]
Cohesion	LCOM	CK [15, 16]
	TCC, LCC	Bieman and Kang [18, 19]
Inheritance	DIT, NOC	CK [15, 16]
	MIF, AIF	MOOD [19, 21, 22]
Polymorphism	POF	MOOD [19, 21, 22]
	OVO, SPA (D), DPA(D)	Benlarbi and Melo[23]
Encapsulation	MHF, AHF	MOOD [19, 21, 22]
Other	WMC	CK [15, 16]

Table2.3 Selected object oriented metrics

Coupling metrics

In the context of object-oriented paradigm, coupling describes the interdependency between methods and between object classes, respectively [12].

CBO (Coupling between object classes) [15,16]

CBO for a class is a count of the number of other classes to which it is coupled. An object is coupled to another object if one of them acts on the other, i.e., methods of one class use methods or attributes of another, or vice versa.

- 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.

- 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 maintenance is more difficult.
- 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.

RFC (Response For a Class) [15,16]

RFC is the response set for the class.

$$RFC = M \cup \bigcup_{i} R_i$$

Where

R_i = *Set of methods called by method "i"*

M = *Set of all methods in the class*

The response set of a class is a set of methods that can potentially be executed in response to a message received by an object of that class.

- If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding required on the part of the tester.
- The larger the number of methods that can be invoked from a class, the greater

the complexity of the class.

- A worst-case value for possible responses will assist in appropriate allocation of testing time.

Coupling Factor (COF) [19,21,22]

Coupling factor (COF) is defined as the ratio of the actual number of couplings to the maximum possible number of couplings in the system, excluding coupling due to inheritance. COF checks whether two classes related either by message passing or by semantic association links (reference by one class to an attribute or method of another class).

$$COF = \sum_{i=1}^{TC}[\sum_{j=1}^{TC} is\_client\ (C_i, C_j)] / (TC^2\text{-}TC)$$

Where:

$$is\_client\ (C_c, C_s) = \begin{cases} 1 & \text{iff} \quad C_c => C_s \wedge C_c \neq C_s \\ 0 & \text{otherwise} \end{cases}$$

$C_c => C_s$ represents the relationship between a client class C_c and a sever class C_s

TC is the number of classes

TC^2-TC refers to the maximum number of couplings in a system

Coupling metrics from Briand et al [14]

These measures focus on coupling as caused by interaction that occurs between classes [13]. The measures are built on three facets by relationship between classes (friendship, inheritance, and other), different types of interactions (attribute-class, class-method, and method-method), and the locus of impact of the interaction (import, export). Therefore, based on the combination of the above three facets, there are 18 different metrics such as IFCAIC, ACAIC, OCAIC, FCAEC, DCAEC, OCAEC, IFCMIC, ACMIC, OCMIC, FCMEC, DCMEC, OCMEC, IFMMIC, AMMIC, OMMIC, FMMEC, DMMEC, OMMEC. Definitions of these metrics are presented in the following table.

Briand et al coupling metrics **[14]**	
Name	Definition
IFCAIC: Inverse friend CA import coupling	These metrics count for the interactions between classes. • The first or first two letters represent the type of relationship considered (i.e., IF for Inverse Friend, F for Friend, D for descendant, A for ancestor, O for others). • The 2 letters afterwards capture the type of interaction (i.e., CA for class-attribute interaction, CM for class-method interaction, MM for method-method interaction). • The last 2 letters say the locus of impact. IC counts for import coupling, and EC counts for export coupling. (i.e., Import: class c is the client class in the interaction; Export: class c is the server class in the interaction.)
ACAIC: Ancestors CA import coupling	
OCAIC: Others CA Import coupling	
FCAEC: Friends CA export coupling	
DCAEC: Descendant CA export coupling	
OCAEC: Others CA export coupling	
IFCMIC: Inverse friend CM import coupling	
ACMIC: Ancestors CM import coupling	
OCMIC: Others CM import coupling	
FCMEC: Friends CM Export coupling	
DCMEC: descendant CM export coupling	
OCMEC: Others CM export coupling	
OMMIC: Others MM import coupling	
IFMMIC: Inverse Friend MM Import coupling	
AMMIC: Ancestors MM import coupling	
OMMEC: Others MM import coupling	
FMMEC: Friends MM export coupling	
DMMEC: Descendant MM export coupling	
OMMEC: Others MM export coupling	

Figure2.2 Briand et al coupling metrics

- The higher the export coupling of a class, the greater the impact of a change to the class on other classes.
- The higher the import coupling of a class C, the greater the impact of a change in other classes on C itself.
- Coupling based on friendship between classes is in general likely to increase the likelihood of a fault even more than other types of coupling, since friendship violates modularity in OO design.

Cohesion metrics

Cohesion describes the binding of the elements within one method and within one object class, respectively [12]. Classes with strong cohesion are easier to maintain, and furthermore, they greatly improve the possibility for reuse [12].

LCOM (lack of cohesion in methods) [15]

The LCOM is a count of the number of method pairs whose similarity is zero minus the count of method pairs whose similarity is not zero. Here, the degree of similarity based on the attributes in common by the methods. The larger the number of similar methods, the more cohesive the class is.

- Cohesiveness of methods within a class is desirable, since it promotes encapsulation.
- Lack of cohesion implies classes should probably be split into two or more sub-classes.
- Any measure of disparateness of methods helps identify flaws in the design of classes.
- Low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.

Bieman and Kang's class cohesion measures [18,19]

Bieman and Kang concern the cohesion among class components including attributes

and classes. Their class cohesion measures are based on the direct or indirect connectivity between a pair of methods. Two methods are directly connected if they use one or more common attributes. In contrast, two methods that are connected through other directly connected methods are indirectly connected.

Here, NDC(C) is the number of directly connected methods in a class C. NIC(C) is the number of indirectly connected methods in a class C. NP(C)=N* (N-1)/2 is the maximum possible number of connections in a class.

Tight Class Cohesion (TCC) is defined to be a ratio of the number of directly connected methods in a class, NDC(C), to the maximum possible number of connections in a class NP(C).

$$TCC(C) = NDC(C)/NP(C)$$

Loose Class Cohesion (LCC) is defined to be a ratio of all directly connected methods, NDC(C), and indirectly connected methods, NIC(C), in a class to the maximum possible number of connections in a class, NP(C).

$$LCC(C)=(NDC(C)+NIC(C))/NP(C)$$

Inheritance metric

Inheritance is a reuse mechanism that allows programmers to define objects incrementally by reusing previously defined objects as the basis for new objects [24].

Depth of inheritance tree (DIT) [15]

The depth of a class within the inheritance hierarchy is the maximum number of steps from the class node to the root of the tree and is measured by the number of ancestor class [7].

- The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior.
- Deeper trees constitute greater design complexity, since more methods and classes are involved.
- The deeper a particular class is in the hierarchy, the greater the potential reuse of inherited methods.

Number of children (NOC) [15]

The number of children is the number of immediate subclasses subordinated to a class in the class hierarchy. It is an indicator of the potential influence a class can have on the design and system [7].

- Greater the number of children, greater the reuse, since inheritance is a form of reuse.
- Greater the number of children, the greater the likelihood of improper abstraction of the parent class is. If a class has a large number of children, it may be a case of misuse of sub-classing.
- 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.

Method Inheritance Factor (MIF) [19,21,22]

The Method Inheritance Factor is defined as the ratio of the number of inherited methods to the total number of the methods that can be invoked in association with the classes in a system. Here, for each class $C_1, C_2 \ldots C_n$, a method counts as 0 if it has not been inherited and 1 if it has been inherited.

$$MIF = \sum_{i=1}^{TC} M_i(C_i) / \sum_{i=1}^{TC} M_a(C_i)$$

Where

$M_i(C_i)$ is the number of methods inherited(and not overridden) in C_i

$M_a(C_i)$ is the number of methods that can be invoked in association with C_i

TC is the total number of classes

Attribute Inheritance Factor (AIF) [19,21,22]

AIF is defined as the ratio of the number of inherited attributes to the total number of available attributes (locally defined plus inherited) for all classes in a system.

$$AIF = \sum_{i=1}^{TC} A_i(C_i) / \sum_{i=1}^{TC} A_a(C_i)$$

Where

$A_i(C_i)$ is the number of inherited attributes in C_i

$A_a(C_i)$ is the number of attributes that can be invoked associated with Ci

TC is the number of classes

The larger of the metrics values, the greater the number of methods or attributes it is likely to inherit, making it more complex to predict its behavior.

Polymorphism metrics

Polymorphism means having the ability to take several forms. For object-oriented systems, polymorphism allows the implementation of a given operation to be dependent on the object that contains the operation [24].

Polymorphism Factor (POF) [19,21,22]

POF is the number of methods that redefine inherited methods, divided by the maximum number of possible distinct polymorphic situations (the latter represents the case in which all new methods in a class are overridden in all its derived classes). Thus, POF is an indirect measure of the relative amount of dynamic binding in a system.

$$POF = \sum_{i=1}^{TC} M_o(C_i) / \sum_{i=1}^{TC} [M_n(C_i) \times DC(C_i)]$$

Where

$M_n(C_i)$ is the number of new methods

$M_o(C_i)$ is the number of overriding methods

$DC(C_i)$ is the descendants count(the number of classes descending from

C_i)

TC is the number of classes

Polymorphism metrics by Benlarbi and Melo [23]

- **Overloading**

Overloading means methods have same names with different signatures within a class scope. The OVO metrics is to gauge the degree of methods generality in a class by counting the number of member functions that implemented the same operation such as add "+ ", minus "-" operation. Here, overl(f_i, C) counts the number of items that the function member f_i is overloaded in the class C.

$$OVO=\sum f_{i \in C} overl(f_i, C)$$

- **Static polymorphism**

Static polymorphism refers to method overriding. That is, methods have the same name and different signatures among inheritance related classes, and the corresponding binding occurs in compile-time. Spoly(C_i, C) counts the number of static polymorphism functions that appear in C_i and C.

Static polymorphism in ancestors:

$$SPA(C) = \sum C_{i \in Ancestors(C)} Spoly(C_i, C)$$

Static polymorphism in descendents:

$$SPD(C) = \sum C_{i \in Descendents(C)} Spoly(C_i, C)$$

- **Dynamic polymorphism**

  Dynamic polymorphism occurs in the same name and same signature in an overridden method, and the corresponding binding occurs in run-time. $Dploy(C_i, C)$ counts the number of dynamically polymorphic functions that appear in C_i and C.

  Dynamic polymorphism in ancestors:

$$DPA(C) = \sum C_{i \in Ancestors(C)} Dpoly(C_i, C)$$

  Dynamaic polymorphism in descendents:

$$DPD(C) = \sum C_{i \in Descendents(C)} Dpoly(C_i, C)$$

Encapsulation metrics

Encapsulation means separating the external aspects of an object that are accessible to other objects, from the internal implementation details of the object that are hidden for other objects. Encapsulation prevents a program from becoming interdependent that a small change has massive effects. [24]

Method Hiding Factor (MHF) [21,22]

MHF is defined as the ratio of the number of the invisibilities of all methods to the

total number of methods declared in all classes. The invisibility of a method is the percentage of the total classes from which this method is not visible.

$$MHF = \sum_{i=1}^{TC} \sum_{m=1}^{Md(Ci)} (1\text{-}V(M_{mi})) / \sum_{i=1}^{TC} M_d(C_i)$$

Where:

$$V(M_{mi}) = \sum_{j=1}^{TC} is\_visible(M_{mi}, C_j) / (TC\text{-}1)$$

$$is\_visible(M_{mi}, C_j) = \begin{cases} 1 & \text{iff } j \neq i \quad \wedge C_j \text{ may call } M_{mi} \\ 0 & \text{otherwise} \end{cases}$$

$M_d(C_i)$ is the number of methods declared in a class (not inherited)

TC is the number of classes

Attribute Hiding Factor (AHF) [21,22]

AHF is defined as the ratio of the number of the invisibilities of all attributes to the total number of attributes defined in all classes. AHF was defined in an analogous fashion, but using attributes rather than methods.

$$AHF = \sum_{i=1}^{TC} \sum_{m=1}^{Ad(Ci)} (1\text{-}V(A_{mi})) / \sum_{i=1}^{TC} A_d(C_i)$$

Where:

$$V(A_{mi}) = \sum_{j=1}^{TC} is\_visible(A_{mi}, C_j) / (TC\text{-}1)$$

$$\text{is\_visible}(A_{mu}, C_j) = \begin{cases} 1 & \text{iff} \quad j \neq i \quad \wedge C_j \text{ can reference } A_{mu} \\ 0 & \text{otherwise} \end{cases}$$

A_d (C_i) is the number of attributes declared in a class (not inherited)

TC is the number of methods declared in a class

Other metrics

Weighted methods per class (WMC) [15,16]

WMC is a count of the complexities of the methods implemented within a class (Complexities here refers to Cyclomatic Complexity). WMC belongs to the popular CK metrics suite and reflect the complexity of classes.

WMC Considers a Class C_1, with methods $M_1, \ldots M_n$ *those are defined in the class.* Let $c_1, \ldots, c_n$ be the Cyclomatic Complexity of the methods. Thus, WMC = $\sum_{i=1}^{n} C_i$

- The number of methods and the complexity of methods involved is a predictor of how much time and effort is required to develop and maintain the class.
- The larger the number of methods in a class the greater the potential impact on children, since children will inherit all the methods defined in the class.
- Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse.

Comments on Object-Oriented metrics

In addition to these selected metrics, there are many other metrics proposed in the literature. We find most measurement models are expressed as a formula. To understand the formula is not difficult. However, it's really difficult to obtain a clear picture of each metric and understand their practical meanings in object-oriented systems, since "there is little understanding of the motivation and empirical hypotheses behind many of these new measures. It is often difficult to determine how such measures relate to one another and for which application they can be used. [13] ". Moreover, we think if there is no automatic tool to support the computation of the metrics suite, it has less practical meaning to discuss the concept of metrics, for software metrics is somewhat a statistic computation.

However, one encouraging phenomenon is that many researchers have done or are undertaking theoretical and empirical studies to test the current popular software metrics. So far, the metrics developed by Chidamber and Kemerer [15, 16] are the most popular ones, because they are widely referenced, and most commercial metrics collection tools available include these metrics. Briand et al coupling metrics are also comprehensive, since they also receive a considerable amount of empirical studies. In section 4, we will discuss the empirical validation work of these popular software metrics to discover their effect in software quality.

2.4 Empirical validation studies

Recent evidence indicates that most faults in software applications are found in only a few of a system's components [25]. Therefore, if software metrics can identify the fault-prone classes, we can improve the whole software qualities by refining those classes. Consequently, we are going to investigate the relationship between the metrics and software quality such as fault-proneness by studying the empirical work that have been published in the literature on software metrics [21,27,26,28,29,25]. The following paragraphs abstract out the result of each selected empirical studies.

In [28], Victor R. Basili et al empirically investigated the suite of Chidamber and Kemerer (briefly, CK) metrics as predictors of fault-prone classes to determine whether they can be used as early quality indicators. The study showed that CBO, RFC, LCOM, DIT, NOC and WMC (exept LCOM) are useful to predict class fault-proneness during the high and low level design phases of the life cycle.

In [21], R. Harrison et al describe the results of an investigation into MOOD metrics. Their empirical results indicated that the MOOD metrics operate at the system level. Comparing with the CK metrics, they found the two sets are complementary, offering different assessments of a system. For example, the coupling metrics (CBO and COF) are closely related, in that CBO offers a class-level view of coupling, whereas COF offers a systems level view. Thus, the MOOD metrics could be of use to project mangers, as the metrics operate at a system level, providing an overall assessment of a system. However, R.Harrison also declared that these metrics need further empirical

validations to establish the causal correlation between the metrics and external quality attributes of system such as reliability, maintainability, testability, etc.

In [29], Daniela Glasberg et al performed an empirical study to test CK metrics [15,16] and Briand coupling metrics [14]. Their result indicated that the Depth of Inheritance (DIT) has a quadratic relationship with fault-proneness. However, the Number of Children (NOC) metrics was not associated with fault-proneness after controlling for the confounding effect of the descendent-based export coupling metrics. Export coupling metrics were found positively associated with fault-proneness. In [26], El Emam K. indicated that cohesion metrics tend not to be good predictors of fault-proneness. Further work should be performed to provide the precise definition of cohesion and develop corresponding effective metrics. Additionally, in [25], El Emam K. et al found that an export coupling (EC) metrics had the strongest association with fault-proneness, indicating a structural feature that may be symptomatic of a class with a high probability of latent faults.

In [27], Briand et al performed a comprehensive empirical validation of many object-oriented software metrics. They used univariate analysis and multivariate analysis to understand their interrelationships, their individual impact on class fault-proneness, and, when used together, their capability to predict where faults are located. The univariate analysis showed "that many coupling and inheritance measures are strongly related to the probability of fault detection in a class. In particular, coupling induced by method invocations, the rate of change in a class due

to specialization, and the depth of a class in its inheritance hierarchy appear to be important quality factors. On the other hand, cohesion, as currently captured by existing measures, does not seem to have a significant impact on fault proneness. This is likely to reflect two facts: (1) the weak under-standing we currently have of what this attribute is supposed to capture, (2) the difficulty to measure such a concept through syntactical analysis only. " As to multivariate analysis, the results showed "that by using some of the coupling and inheritance measures, very accurate models can be derived to predict in which classes most of the faults actually lie. When predicting fault-prone classes, the best model shows a percentage of correct classifications about 80% and finds more than 90% of faulty classes. "

In [26], El Emam K. provided some concrete guidelines for quality management in object-oriented application while reviewing CK metrics and Briand coupling metrics. First, since export and import coupling are the most important metrics, and can be collected at the early design states, allowing for the early quality management, we should assign best people to work on classes with high values on the coupling metrics. Second, if the historical data is available, we should rank classes by their predicted fault-proneness. Similarly, we should assign best people to work on the classes with the largest predicted fault-proneness. Third, more experienced inspection teams should work for classes with high fault-proneness, and develop more test cases for those classes.

2.5 Future work

Even though software metrics has been a subject area over 30 years, it has barely penetrated into mainstream software engineering. Having exploring research of software metrics, we observe two intuitional reasons. **One reason** is that most metrics have been defined by an individual or a team and then tested and used only in a very limited environment, thus it is hard to achieve the adoption in software industry. **Another reason** is that most software metrics have not addressed their most important requirement: to provide information to support quantitative managerial decision-making during the software lifecycle [10].

Based our survey, we think further work about the current software metrics may necessarily focus on the following directions:

Unification

To increase the adoption of the software metrics application in real software life cycle, it should have a unified notation to address their definition and underlying principles. Moreover, there should have one unified empirical and systematic approach to evaluate and validate the usefulness of these metrics.

Standardization

It is recognized that a single software metric cannot predict and evaluate software quality effectively; however, too much software metrics also introduce redundancies and confusions. Therefore, future research may identify one industry-standard

software metrics suite to reflect the quality issues of software products.

Tools

Since software metrics use to quantify software development process, product and resources. That is, they present in statistic formulas, so it's necessary to develop some useful tools or some components easily embedded in specific software to automate the computation of the software metrics. Some design decision suggestions based on the computation of the metrics may also be incorporated into these tools.

Applications

Software metrics are validated widely in predicting the fault-proneness of the system. Further empirical studies may test their capability in software maintainability, testability reliability or other aspects of software quality.

2.6 Conclusion

In this chapter, we survey software metrics by reviewing the key concept related to software metrics, introducing the selected popular traditional and object-oriented metrics, and studying the comprehensive empirical validation experiments. Based on the analysis of the results from empirical studies, we discover that most empirical validation work focuses on testing fault-proneness of software system and declares inheritance and coupling measures being the most effective metrics, which are strongly related to the probability of fault detection in object-oriented system. Especially, coupling metrics play a vital role in software quality assurance and should

have extra attentions by the software developers and managers.

Reliability measurement is a subset of software measurement technology. Software metrics and software reliability measurement are interrelated technologies. Some metrics can be directly used as reliability models. In chapter5, the relationship between metrics and models will be presented in detail.

2.7 Reference

[1] PRESSMAN, R. S. "*Software Engineering: A Practitioner's approach*" 4th edition. McGraw-Hill Book Company. 1997.

[2] Conte, S. D., H. E. Dunsmore, and V. Y. Shen, "*Software Engineering Metrics and Models*", Menlo Park, Calif Benjamin/Cummings, 1986

[3] Fenton NE, "*Software Metrics - A Rigorous Approach*" Chapmann & Hall, London, 1991

[4] Bertrand Meyer, "*The role of object-oriented metrics*", in *Computer* (IEEE), vol. 31, no. 11, November 1998, pages 123-125.

[5] Grady, R. B. and D. R. Caswell, "*Software Metrics: Establishing a Company-Wide Program*", Engle- wood Cliffs, N. J. Prentice-Hall, 1987

[6] Tegarden, D., Sheetz, S., Monarchi, D, "*Effectiveness of Traditional Software Metrics for Object-Oriented Systems*", Proceedings: 25th Hawaii International

Confernce on System Sciences, January 1992, pp. 359-368.

[7] Linda H. Rosenberg, "*Applying and Interpreting Object Oriented Metrics*", presented at the Software Technology Conference, Utah, April 1998.

http://satc.gsfc.nasa.gov/support/STC_APR98/apply_oo/apply_oo.html

[8] Norman Fenton, "*Software Metrics For Control And Quality Assurance Course Overview*", 2000

http://www.dcs.qmul.ac.uk/~norman/Courses/mod_903/slides/slides_2000/all_slides_2000_blue/

[9] Al Ehrbar , "*Software measures*", John Wiley & Sons, 1998

[10] Norman E Fenton and Martin Neil, "*Software Metrics: Roadmap*", 2000

http://www.agena.co.uk/postcript_papers/metrics_roadmap.pdf

[11]McCabe, "*Object Oriented Tool User's Instructions*", McCabe & Associates, 1994

[12] Johann Eder, Certi Kappel, and Michael Klagenfurt, "*Coupling and cohesion in object-oriented systems*", 1992

http://citeseer.nj.nec.com/cache/papers/cs/2479/ftp:zSzzSzftp.ifs.uni-linz.ac.atzSzpubzSzpublicationszSz1993zSz0293.pdf/eder92coupling.pdf

[13] Lionel C. Briand, John W. Daly, and Jurgen wust, "*A unified framework for coupling measurement in object-oriented systems*", 1996

http://www.iese.fraunhofer.de/network/ISERN/pub/technical_reports/isern-96-14.pdf

[14] Lionel Briand, Prem Devanbu, Walcelio Melo, "*An investigation into coupling measures for C++*", 1997

http://citeseer.nj.nec.com/cache/papers/cs/1336/http:zSzzSzwww.research.att.comzSz~premzSzicse97.pdf/briand97investigation.pdf

[15] Shyam R. Chidamber and Chris F. Kemerer, "*A METRICS SUITE FOR OBJECT ORIENTED DESIGN*", IEEE Transitions on Software Engineering, 1994

http://www.pitt.edu/~ckemerer/clnieee.pdf

[16] Shyam R. Chidamber and Chris F. Kemerer, "*Towards a Metrics Suite for Object Oriented Design*", In Proc. of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91), pages 197--211. ACM, 1991

[17] Bindu S. Gupta, "*A critique of cohesion measures in the object-oriented paradigm*", 1997

http://citeseer.nj.nec.com/cache/papers/cs/2183/ftp:zSzzSzcs.mtu.eduzSzpubzSzottzSzreportszSzmehra-thesis.pdf/gupta97critique.pdf

[18] Linda M.Ott and James M. Bieman, "*Program slices as an abstraction for cohesion measurement*", 1998

ftp://ftp.cs.colostate.edu/pub/bieman/istPreprint98.pdf

[19] Fernando Brito e Abreu and Rita Esteves, Miguel Goulão, "*The Design of Eiffel*

programs: Quantitative Evaluation Using the MOOD Metrics", Proceedings of TOOLS'96 USA, Santa Barbara, California, July 1996.

[20] I.Brooks, "*Object-oriented metrics collection and evaluation with a software process*", Proc. OOPSLA '93 Workshop Processes and Metrics for Object-Oriented Software Development, Washington, D.C., 1993

[21] R.Harrison, S.J. Counsell, R.V. Nithi, "*An evaluation of the MOOD set of object-oriented software metrics*", 1998

http://www.computer.org/tse/ts1998/e0491abs.htm

[22] Fernando Brito e Abreu and Walcélio Melo, "*Evaluating the Impact of Object-Oriented Design on Software Quality*", 1996

http://www2.umassd.edu/SWPI/ESEG/3IntSoftMetSymp.pdf

[23] Sdida Benlarbi and Walcelio L. Melo, "*Polymorphism Measures for Early Risk Prediction*", 1999

http://delivery.acm.org/10.1145/310000/302652/p334-benlarbi.pdf?key1=302652&key2=8583876101&coll=portal&dl=ACM&CFID=1931617&CFTOKEN=83627676

[24] Victor Laing and Charles Coleman, "*Principal Components of Orthogonal Object-Oriented Metrics*", (323-08-14), 2001

http://satc.gsfc.nasa.gov/support/OSMASAS_SEP01/Principal_Components_of_Orthogonal_Object_Oriented_Metrics.pdf

[25] Khaled El Emam, Walcelio Melo, Javam C. Machado, "*The prediction of faulty classes using object-oriented design metrics*", The journal of systems and software 56(2001) 63-75, 2000

http://www.mestradoinfo.ucb.br/Prof/wmelo/jss-oo.pdf

[26] El-Emam, K., "*Object-Oriented Metrics: A Review of Theory and Practice*", 2001

ftp://ai.iit.nrc.ca/pub/iit-papers/NRC-44190.pdf

[27] Lionel C. Briand, Jürgen Wüst, John W. Daly 1 , and D. Victor Porter, "*Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems*", 2000

http://www.sce.carleton.ca/faculty/briand/jss.pdf

[28] Victor R.Basili, Lion C. Briand, and Walcelio L. Melo, "*A validation of object oriented metrics as quality indicators*", IEEE transactions on software engineering, Vol 22,No. 10, 1996

http://www.mestradoinfo.ucb.br/Prof/wmelo/ieeetse1996.pdf

[29] Daniela Glasberg, Khaled El Emam, Walcelio Melo, Nazim Madhavji, "*Validating object-oriented design metrics on a commercial java application*", 2000

http://www.mestradoinfo.ucb.br/Prof/wmelo/NCR_1080.pdf

Chapter 3 Overview of Software Reliability Engineering (SRE)

The analysis of the reliability of a system must be based on precisely defined concepts. And the fundamental definitions of reliability must depend on concepts from probability theory [1]. This chapter, we provide the basis for quantifying the reliability of a system, from the definition to expression of software reliability as well as terms related to software reliability engineering.

3.1 Definition of Software Reliability

According to the 1991 IEEE standard of software engineering, **Software Reliability** is defined as the probability of failure-free software operation for a specified period of time in a specified environment [2]. Further, John D. Musa extended the definition as the probability that a system or a capability of a system functions without failure for a specified period in a specified environment [3]. The period may be specified in natural units or time units.

From the definition, we can see that software reliability depends on how software is used. Software usage information is thus an important part of reliability evaluation. This includes the environment information where software is used, as well as the information on the actual frequency of usage of different operations, functions, or features that the system offers. The usage information is quantified through operational profiles, which we will address later in this chapter.

Furthermore, from the definition, we can see that "**Time**" is an important issue in measuring. "Time" is execution exposure that software receives through usage. It is usually measured in central processing unit (CPU) execution-time, calendar-time or clock time. Experience indicates that the best metric is the actual central processing unit (CPU) execution time. However, in many practical test environments, CPU execution time is not always available. Thus, it is necessary to reformulate measurements, and reliability models, in terms of other exposure metrics, such as calendar-time, clock-time, number of executed test cases (or runs), fraction of planned test cases executed, inservice-time, or structural coverage. In considering which time to use, it is necessary to weigh factor, such as availability of data for computation of a particular metric, error-sensitivity of the metric, availability of appropriate reliability models, etc.

In Musa's definition, the concept of natural units is relatively new to software reliability. A natural unit is a unit other than time that is related to the amount of processing performed by a software-based product, such as pages of output, transactions, telephone calls, jobs, semiconductor wafers, queries or application program interface calls [3]. Availability is the average probability that a system or a capability of a system is currently functional in a specified environment. If you are given an average down time per failure, availability implies a certain reliability. Failure intensity, used particularly in the field of software reliability engineering, is simply the number of failures per natural or time unit. It is an alternative way of expressing reliability.

3.1.1 Software Reliability Versus Hardware Reliability

Computers are composed by hardware and software. Increasing use of firmware and embedded software is blurring the boundary lines between software and hardware. To avoid failures, hardware and software must all be stable and reliable. Thus, we can define software reliability as a part of system reliability, which is composed by hardware and software reliability.

The development of hardware reliability theory has a long history and a successful record. Hardware reliability encompasses a wide spectrum of analyses that strive systematically to reduce or eliminate system failures which adversely affect product performance [4]. Reliability also provides the basic approach for assessing safety and risk analysis. Software reliability strives systematically to reduce or eliminate system failures which adversely affect performance of a software program.

Although hardware reliability models cannot be applied to software since software is not physical, the analysis and comparison of the two will help us understand software reliability better and benefit us in developing software reliability modeling. The table below summarizes the differences between hardware and software reliability:

Software Reliability	Hardware Reliability
Failures are primarily due to design faults. Repairs are made by modifying the design to make it robust against conditions that can trigger a failure.	Failures are caused by deficiencies in design, production, and maintenance.
There is no wear-out phenomena. Software errors occur without warning. "Old" code can exhibit an increasing failure rate as a function of errors induced while making upgrades.	Failures are due to wear or other energy-related phenomena. Sometimes a warning is available before a failure occurs.
There is no equivalent to preventive maintenance for software.	Repairs can be made which would make the equipment more reliable through maintenance.
Reliability is not time dependent. Failures occur when the logic path that contains an error is executed. Reliability growth is observed as errors are detected and corrected.	Reliability is time related. Failure rates can be decreasing, constant, or increasing with respect to operating time.
External environment conditions do not affect software reliability. Internal environmental conditions, such as insufficient memory or inappropriate clock speeds do affect software reliability.	Reliability is related to environmental conditions.
Reliability cannot be predicted form a knowledge of design, usage, and environmental stress factors.	Reliability can be predicted in theory from physical bases.
Reliability cannot be improved by redundancy, since this will simply replicate the same error. Reliability can be improved by diversity.	Reliability can usually be improved by redundancy
Failure rates of software components are not predictable.	Failure rates of components are somewhat predictable according to known patterns.
Software interfaces are conceptual.	Hardware interfaces are visual
Software design does not use standard components.	Hardware design uses standard components.

Table 3.1. Difference between hardware and software reliability [14]

3.1.2 Measures of Reliability

The quality of software, and in particular its reliability, can be measured in a number of ways. **Availability** and **Failure Intensity** are the most widely used two ways of measuring. Before we continue the discussing, it is necessary for us to distinguish two situations. In one situation, there are no fault removals during testing and maintenance phases of software development life cycle. No detected problems will be reported before the releases of a product. In the other situation, there are detected problems. We are experiencing fault identifications and corrections. The quality of the product improves over time, and we talk about **reliability growth**. In this verse, we address availability and failure intensity separately:

Failure Intensity

Failure intensity is a metric that is commonly used to describe software reliability. It is defined as the number of failures experienced per unit "time" period [3]. Sometimes, the term **Fault rate** is used instead. **Mean time to failure** (MTTF) is an interesting associated measure. Failure intensity can be computed for all kinds of experienced failures. It is a good measure which can reflect the user's perspective of software quality.

Suppose that the reliability function for a system is given by R(t). In situation 1, when there is no fault found, it is possible to describe the R (t) by using constant failure intensity, λ, and a very simple exponential relationship:

$$R(t) \approx e^{-\lambda t}$$

't' is the duration of the mission. For instance, let's suppose we are using a system which is under representative and unchanging conditions, and the faults causing any reported failures are not being removed. Let the number of failures observed over 1,000 hours of operation be 6. Then, failure intensity is about $\hat{\lambda} = 6/1000 = 0.006$ failures per hour. From the above equation, and given that the system operates correctly at time t = 10 hours, the probability that the system will **not** fail during a 10 hour mission is about $R(10) = e^{-0.006*10} = 0.941$.

While in situation 2, when reliability growth is presented, failure intensity, $\lambda(\tau)$ becomes a decreasing function of time τ during which software is exposed to testing and usage under representative (operational) conditions [4]. Thus, the situation is much more complex that the previous one. There is a large number of software reliability models that address this situation in [5]. All models have some advantages and disadvantages. It is extremely important that an appropriate model be chosen on a case-by-case basis [4, 5]. Two typical models are the "basic execution time" (BET) model [6, 7] and the Logarithmic-Poisson execution time (LPET) model [4, 8]. Both models assume that every detected failure is immediately and perfectly repaired, and the testing uses operational profiles.

The BET model represents a class of "finite-failure" models for which the mean value functions tends towards a level asymptote as exposure time grows.

The **BET** failure intensity $\lambda(\tau)$ with exposure time τ is:

$$\lambda(\tau) = \lambda_0 e^{-\frac{\lambda_0}{V_0}\tau}$$

λ_0 is the initial intensity.

V_0 is the total expected number of failures.

On the other hand, the LPET model is a representative of a class of models called "infinite-failure" models since it allows an unlimited number of failures.

The **LPET** failure intensity $\lambda(\tau)$, with exposure time τ is:

$$\lambda(\tau) = \frac{\lambda_0}{\lambda_0 \theta \tau + 1}$$

$\lambda(0)$ is the initial intensity

θ is called the failure intensity decay parameter since:

$$\lambda(\tau) = \lambda(\mu) = \lambda_0 e^{-\theta \mu}$$

and $\mu(\tau)$ is the mean number of failures experienced by time τ, i.e.,

$$\mu(\tau) = \frac{1}{\theta} \ln(\lambda_0 \theta \tau + 1))$$

Availability

Availability is another important practical measure for software reliability. In telecommunication industry, availability and unavailability targets are used to measure the stability of system. For example, the Bellclore unavailability for telecommunication

network elements is about 3 minutes of downtime per year [5]. Here, availability implies certain reliability.

The availability of a system is defined as the probability that the system is successful at time [0 t]. We can compute availability for software as we do for hardware. It is the ratio of up time to the sum of up times plus down time. Mathematically, it is called **average availability**, and shows as below:

$$\text{Availability} = \frac{\text{System up time (MTTF)}}{\text{System up time (MTTF)} + \text{System down time (MTTR)}}$$

where

$$\text{System down time} = \text{Failure intensity} * \text{Mean time to repair (MTTR)} \quad [9]$$

Here, the failure intensity is a figure computed for serious failures and not those that involves only minor degradation of the system [9].

From the definition, we call the availability as **instantaneous availability**, when the system will be available at any random time t during its life. A Markov model was developed by Trivedi (1975) [10] that depicts the concept of software availability. A more recent paper by Laprie (1992) [11] improved the model and dealt with the evaluation of availability during the operational phase.

In the definition, if the time [0,t] is long enough, then the availability approaches **steady state availability,** which, given some simplifying assumptions, can be described by the following relationships [12, 13]:

$$\text{Steady state Availability} = \frac{\lambda}{\lambda + \rho}.$$

$$\lambda = \frac{\text{Total number of Failures in T}}{\text{Total time system was operational during period T}}$$

$$\rho = \frac{\text{Total number of failures in period T}}{\text{Total time system was under repair or recovery during Period T}}$$

*λ is **fault intensity**.*

*ρ is the **repair rate**, which is the number of repaired failures per unit time.*

From the previous defined equations, we can see that two measures that directly influence the availability of a system are its failure rate (failure intensity) and its field repair rate. Failure rate is usually connected to both the operational profile and the process of problem resolution and correction. Recovery rate depends on the operational usage profile, the type of problem encountered, and the field response to that problem.

In practice, reliability and availability models are used to predict future unavailability of a system. Of course, only the data up to the point from which the prediction is being made would be available. The prediction would differ from the true value depending on how well the model describes the system. The figure below shows the empirical unavailability

data and fits for two simple models. Both models appear to predict future system behavior well.

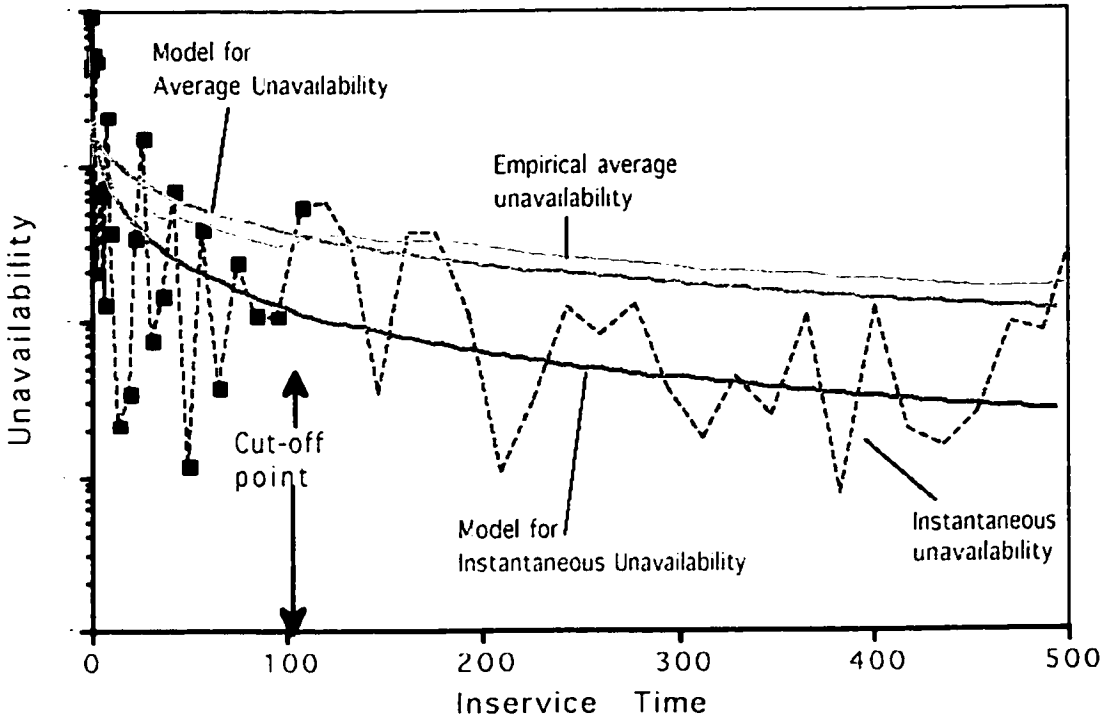

Figure 3.1. Unavailability fitting using LPET and constant repair rate with data up to "cut-off point" only [5]

3.2 Software Reliability Engineering

Software reliability engineering is a currently a daily practiced technique in many engineering disciplines. By definition, software reliability engineering (SRE) is the "applied science of predicting, measuring, and managing the reliability of software-based systems to maximize customer satisfaction" [9]. It is the quantitative study of the operational behavior of software based systems with respect to user requirements concerning reliability, it therefore includes [5]:

1. Software reliability measurement, which includes estimation and prediction, with the help of software reliability models established in the literature;

2. The attributes and metrics of product design, development process, system architecture, software operational environment, and their implications on reliability;

3. The application of this knowledge in specifying and guiding system software architecture, development, testing, acquisition, use, and maintenance.

In later chapters, we will address reliability measurement, modeling and some other issues in detail. In figure2, we present an overview of a general SRE process.

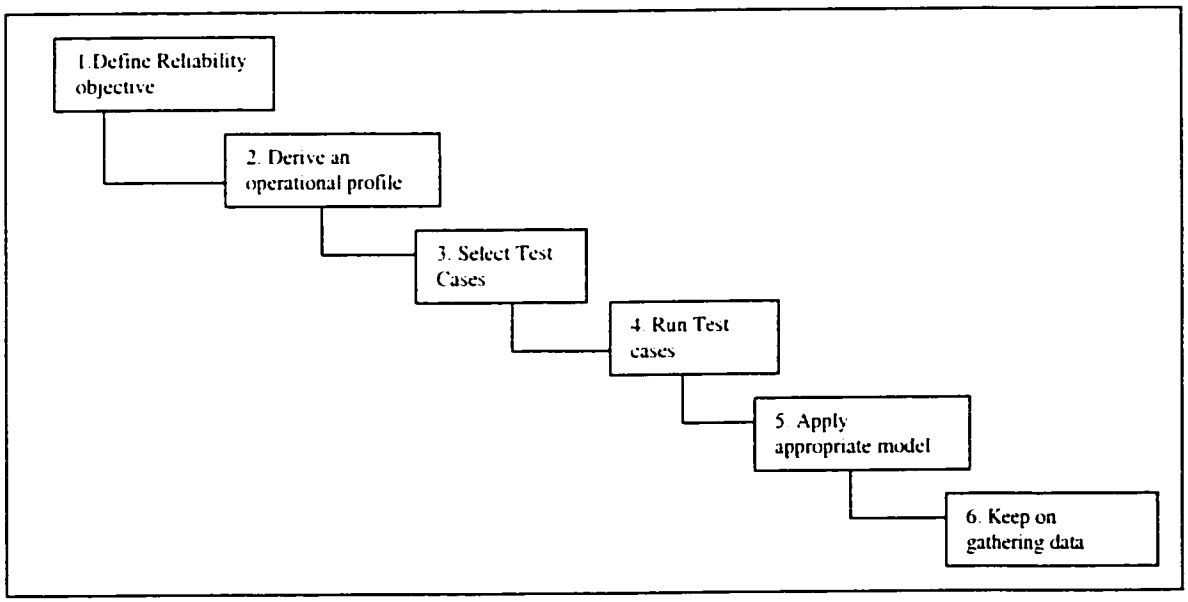

Figure3.2 SRE General Process

In step1, the reliability objective is determined by specifying balance among key quality objectives (e.g., reliability, cost, delivery date).

In step2, an **operational profile** is derived. This is a usage distribution, i.e., the extent to which each operation of the software system is expected to be exercised. By definition,

Operational profile is a set of relative frequencies (or probabilities) of occurrence of disjoint software operations during its operational use [5]. A software-based system may have one or more operational profiles. Operational profiles are used to select **test cases** and direct development, testing and maintenance efforts towards the most frequently used or most risky components. Construction of an operational profile is preceded by definition of a **customer** profile, a **user** profile, a **system mode** profile, and a **functional** profile. The usual participants in this iterative process are system engineers, high-level designers, test planners, product planners, and marketing. The process starts during the requirements phase and continues until the system testing starts. **Profiles** are constructed by creating detailed hierarchical lists of customers, users, modes, functions and operations that the software needs to provide under each set of conditions. For each item it is necessary to estimate the probability of its occurrence (and possibly **risk** information) and thus provide a quantitative description of the profile. If usage is available as a rate (e.g., transactions per hour) it needs to be converted into probability. In discussing profiles, it is often helpful to use tables and graphs and annotate them with usage and criticality information.

In step3, test cases are selected in accordance with the operational profile to ensure that the system testing usage is similar to the usage environment expected after release of the system.

In step4, test cases are executed. For each test that results in failure the time of failure is noted, in terms of execution time, and the execution time clock is stopped. After repair the clock is started again, and the testing resumes.

In step5, an appropriate software reliability model is applied to the distribution of the failure points over time, and an estimate of the reliability is determined. If the reliability objective has been reached, the system is released. Thus, reliability modeling is an essential element of the reliability estimation process. It determines if a product meets its reliability objective and is ready for release.

In step6, failure data continue to be gathered after system release.

3.2.1 Some Basic Terms

Here it is necessary to provide some widely used terms in Software Reliability Engineering to avoid confusing.

Error. Human action that results in software containing a fault. Examples include omission or misinterpretation of user requirements in a software specification, and incorrect translation or omission of a requirement in the design specification. [15]

Failure. (1) The termination of the ability of a functional unit to perform its required function. (2) An event in which a system or system component does not perform a required function within specified limits. A failure may be produced when a fault is

encountered [15]. Failure can be caused by hardware or software faults (defects), or by how-to-use errors.

Fault. (1) An accidental condition that causes a functional unit to fail to perform its required function unit to fail to perform its required function. (2) A manifestation of an error in software. A fault, if encountered, may cause a failure. Synonymous with bug [15]. Inherent faults are the faults that are associated with a software product as originally written, or modified. Faults that are introduced through fault correction, or design changes, from a separate class modification faults. An associated measure is fault density. Faults are results of errors, or mistakes.

Fault density. The number of faults per thousand lines of executable source code.

Software reliability management. The process of optimizing the reliability of software through a program that emphasizes software error prevention, fault detection and removal, and the use of measurements to maximize reliability in light of project, constraints such as resources (cost), schedule, and performance.

3.3 Reference

[1] Alfs T.Berztiss "*Software Reliability Engeneering*" university of Pittsburgh

[2] Institute of Electrical and Electronics Engineers, ANSI/IEEE Standard Glossary of Software Terminology, IEEE Std. 729-1992, 1991

[3] John D. Musa, "*More Reliable Software Faster And Cheaper: An Overview Of Software Reliability*".

Http://members.aol.com/JohnDMusa/ARTweb.htm

[4] J.D.Musa, "*Operational Profiles In Software Reliability Engineering*," IEEE software, vol. 10(2), pp. 14-32, March 1993

[5] Handbook of Software Reliability Engineering, McGraw Hill, editor M. Lyu, January 1996.

[9] J.D.Musa, and W.W.Everet, "*Software-Reliability Engineering: Technology For The 1990s.*," IEEE software , VOL 7. pp. 36-43, November 1990

[10]Trvedi, A.K., "*Computer software Reliability: Many-State Markov Modeling Techniques*," Ph.D dissertation, Polytechnic Institute of Brooklyn, June, 1975

[11] Aprie, J.-C., Dependability Basic Concepts and terminology, Dependable computing and Fault-tolerant systems, Vol.5, J-C. Laprie (ed.), Springer Verlag, Wien, New York, 1992

[12] K.S Trivedi, Probability & Statistics with Reliability, Queuing, and computer science applications, Prentice-Hall, Englewood Cliffs, N.J., 1982

[13] M.L.shooman, software engineering, McGraw-Hill, New York, 1983

[14]Debra S.Herrmann, software safety and reliability, ISBN 0-7695-0299-7

[15] *IEEE Standard Dictionary of Measures to Produce Reliable Software* (IEEE Std 982.1-1988). New York, N.Y.: Institute of Electrical and Electronic Engineers, Inc., 1989.

Chapter 4 Software Reliability Measurement Procedure

As we have introduced in chapter 2, 3, software measurement is a quantitative assessment of the degree to which a software product or process possesses a given attribute; reliability measurement is a main activity of software reliability engineering. Thus, in this chapter we will discuss the issues in software reliability measurement from the definition to the framework of measurement, as well as a classification of measures.

4.1 Definition of SR Measurement

Software reliability measurement is the application of statistical inference procedures to failure data taken from software testing and operation to determine software reliability [1]. It mainly includes two activities: reliability estimation and reliability prediction.

Estimation evaluates current software reliability by applying statistical inference techniques to failure data obtained during system test or system operation [2]. This is a measure regarding the achieved reliability from the past until the current point. We use reliability estimation to assess the current reliability and determine whether a reliability model is a good fit in retrospect.

Prediction predicts future software reliability based upon available software metric and measures [2]. During different software development stage, prediction involves different techniques:

1. During system design and coding phase, when failure data is not available, some metrics obtained from the software development process and the characteristics of the resulting product are used to predict reliability of the software upon testing or delivery.
2. During system test and operation phase, when failure data are available, the estimation techniques are used to parameterize and verify software reliability models, which can perform future reliability prediction.

Most current software reliability models fall in the estimation category to do reliability prediction. Still, some prediction models are proposed and described in the literature. In next chapter, we will discus software reliability modeling in detail.

4.1.1 Classification of Measures

In the IEEE Guide of dictionary of measures to produce reliable software [3], measurements are classified into two kinds. One is "functional classification"; another is "lifecycle classification". The lifecycle classification addresses the problem of "when are the measures applied" to assist the user in more practical considerations. The "lifecycle" which is the software development lifecycle is divided into early, middle, and late segments and covers the following: Concepts, Requirement, Design, Implementation, Test, Installation and Checkout, operation and Maintenance, Retirement.

According to the functional classification, measures are classified based on two main objectives of reliability measurement: product maturity assessment and process maturity assessment. The first objective is a certification of product readiness for operation

(including an estimation of support necessary for corrective maintenance). The second objective is an ongoing evaluation of a software factory's capability to produce products of sufficient quality for user needs. The process maturity assessment includes the product maturity assessment with the objective of process repair when necessary [3].

Product measures address cause and effect of the static and dynamic aspects of both projected reliability prior to operation, and operational reliability. As an example, reliability may change radically during the maintenance effort, due to the complexity of the system design. These product measures cover more than the correctness aspect of reliability; they also address the system utility aspect of reliability. Table1 including six product measure subcategories addresses these dimensions of reliability:

Category	Sub-category	Function
Product Maturity Assessment	Errors Faults Failures	Count of defects with respect to human cause, program bugs, and observed system malfunctions
	Mean-Time-to-Failure Failure Rate	Derivative measures of detect occurrence and time
	Reliability Growth Projection	The assessment of change in failure-freeness of the product under testing and in operation
	Remaining Product Faults	The assessment of fault-freeness of the product in development, test, or maintenance
	Completeness Consistency	The assessment of the presence and agreement of the presence and agreement of all necessary software system parts
	Complexity	The assessment of complicating factors in a system
Process Maturity Assessment	Management Control	The assessment of guidance of the development and maintenance processes
	Coverage	The assessment of the presence of all necessary activities to develop or maintain the software product
	Risk Benefit Cost Evaluation	The assessment of the process tradeoffs of cost, schedule, and performance

Table 4.1 Functional Classification

(Table4.1 is based on the classification of functional measure classification [3])

4.2 Measurement Framework

Software reliability measurement takes place in an environment that includes user needs and requirements, a process for developing products meeting those requirements, and user environment within which the delivered software satisfies those needs. This measurement environment establishes a framework for determining and interpreting indicators of software reliability. [4]

Measurement framework has been proposed in a lot of papers, books and IEEE standards, like in [2,4,5,6]. These frameworks are normally developed focusing on the practice of SRE. The purpose is to enhance an organization's ability to deliver timely, high-quality products through an application of SRE practices, as well as to help ensure that software vendors deliver high-quality component products. Here we present the framework developed by M.R. Lyu, shown as in figure4.1:

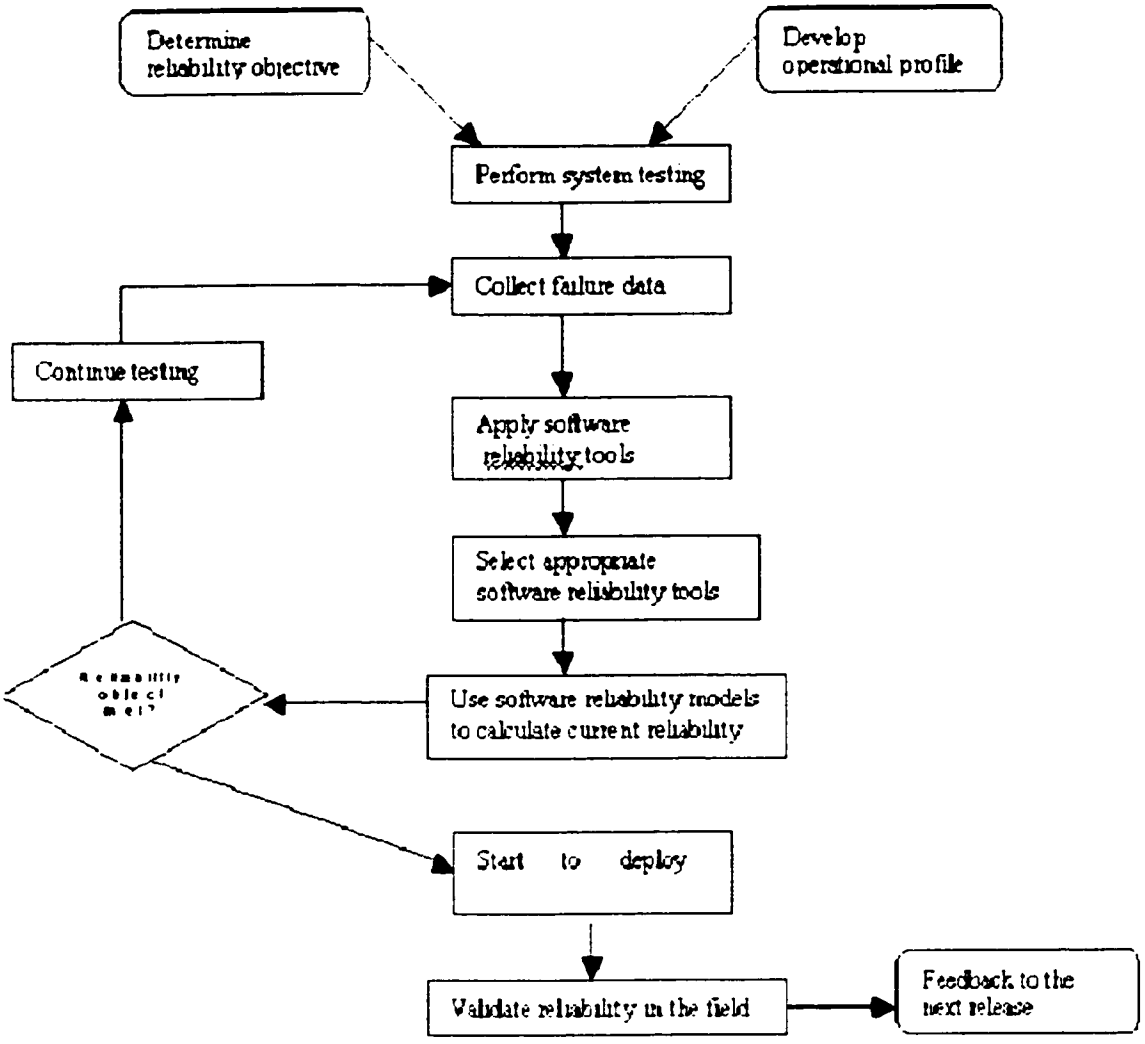

Figure4.1 Software Reliability Measurement Procedure Overview

It can be seen that there are four major components in this software reliability measurement process, namely,

(1) reliability objective,

(2) operational profile

(3) reliability modeling and measurement, and

(4) reliability validation

We have the list observations from this framework:

(1) According to this framework, quality is first defined quantitatively from the customer's viewpoint by defining failures and failure severity, by determining a reliability objective, and by specifying balance among key quality objective (e.g., reliability, delivery date, cost).

(2) Customer usage is quantified by developing an operational profile. As we discussed in chapter3, operational profile is a set of disjoint alternatives of system operation and their associated probabilities of occurrence. The construction of an operational profile encourages testers to select test cases according to the system's operational usage, which contributes to more accurate estimation of software reliability in the field. We will discuss the implementation of operational profile in the later verse of this chapter.

(3) Resources are then managed by advocating the employment of operational profile and quality objectives to guide through software development phases (e.g., design, implementation, testing).

(4) Reliability during testing is tracked to determine product release, using appropriate software reliability measurement models and tools. This activity may be repeated until a certain reliability level has been achieved.

(5) Reliabilities are analyzed in the field to validate the reliability engineering effort and to provide feedback for product and process improvements.

Reliability modeling is an essential element of the reliability estimation process. It determines if a product meets its reliability objective and is ready for release. It is required to use a reliability model to calculate, from failure data collected during system testing (such as failure report data and test time), various estimates of a product's

reliability as a function of test time. These reliability estimates can provide the following information useful for product quality management [2]:

(1) The reliability of the product at the end of system testing.

(2) The amount of (additional) test time required to reach the product' reliability objective.

(3) The reliability growth as a result of testing.

(4) The predicted reliability beyond the system testing already performed.

4.2.1 Implement Operational Profile

From chapter 3 and the previous verse of this chapter, we can see operational profile is an important aspect in SRE. A reliability estimate established by the SRE process is meaningful only if the conditions under which the software is tested closely correspond to the conditions under which it will be used in the field [7]. This ensures the test cases to have the same distribution according to type as the inputs after the system has been released. This distribution is just the operational profile. Lyu and Musa [2, 5] discussed the implementation of operational profile in detail. Normally, the development of an operational profile may take five steps, as follows:

(1) Definition of client (customer) types. A client (customer) is a person, a group, or an institution that acquires the system. Table4.2 illustrates a hypothetical customer profile for telephone switch software. This relates to product software, i.e., to software that is

developed for a number of different clients. Consider a generic automated enquiry system that responds to telephone calls by customers. The system consists of prompts to the caller, call routing, canned responses, the possibility of speaking to a customer service representative, and wait lines. This structure is the same for all systems, only the content of the prompts and the canned responses differs from system to system. The systems for airlines will be more similar to each other than to a system that handles health insurance enquiries. Airlines and health insurance companies thus form two client types.

Custermer Group	Probability
Local Carrier	0.7
Long distance Carrier	0.3

Table 4.2 Customer profile for telephone switch software

(2) Definition of user types. A user is a person, a group, or an institution that employs the system, it can even be other software (as in automatic business-to-business e-commerce) who will use the system in the same way. In the case of the enquiry system, some users will update the canned responses, and this will happen more often for airlines than for health insurance companies. Other users will listen to responses, and still others will want to talk to customer representatives.

(3) Definition of system modes. A system mode is a set of functions or operations that are grouped for convenience in analyzing execution behavior [5]. System can switch among modes; two or more modes can be active at any present time. These are the major operational modes of a system, such as system initialization, reboot after a failure,

overload due to excessive wait line build-up, gathering of system usage statistics, monitoring of customer service representatives, selection of the language in which the customer will interact with the system.

(4) Definition of functional profile. A function may represent one or more tasks, operations, parameters, or environmental variables. In this step, each system mode is broken down into the procedures or transactions that it will need, and the probability of use of each such component is estimated. The functional profile is developed during the design phase of the software process.

(5) Definition of operational profile. The functional profile relates to the abstractions produced as part of software design. The abstractions of the functional profile are now converted into actual operations, and the probabilities of the functional profile are converted into probabilities of the operations. The operations are the ones that are tested. Their profile will determine verification and validation resources, test cases and the order of their execution. The operations need to be associated with actual software commands and input states. These commands and input states are then sampled in accordance with the associated probabilities to generate test cases. Particular attention should be paid to generation of test cases that address critical and special issues.

The five steps are actually part of any software development process. The operations have to be defined sooner or later in any case, and a process based on the five steps can be an effective way of defining operations. For example, user types can be identified with

actors and system modes with use cases in use case diagrams of UML [8]. Identification of user types and system modes by means of use case diagrams can take place very early in the software process.

What is specific to SRE and what is difficult is the association of probabilities with the operations. They include the uncertainty of how a new software system will be used. Many systems are used in ways totally different from how the original developers envisaged their use. The Internet is the most famous example. Another problem is that requirements keep changing well after the initial design stage.

Fortunately, it has been shown that reliability models are rather insensitive to errors in the operational profile [9]. Note that reliability is usually presented as a value together with a confidence interval associated with the value [5]. Adams [10] shows how to calculate confidence intervals that take into account uncertainties associated with the operational profile. Note that Adams defines reliability as the probability that the software will not fail for a test case randomly chosen in accordance with the operational profile.

4.3 Reference

[1] M.R Lyu and A.Nikora, "Using Software Reliability Models More Effectively," IEEE Software, July 1992

[2] M.R Lyu (ed), handbook of software reliability engineering, McGraw-Hill and IEEE Computer Society Press, New York, 1996,

[3] *IEEE Standard Dictionary of Measures to Produce Reliable Software* (IEEE Std 982.1-1988). New York, N.Y.: Institute of Electrical and Electronic Engineers, Inc., 1989.

[4] *IEEE Standard Dictionary of Measures to Produce Reliable Software* (IEEE Std 982.2-1988). New York, N.Y.: Institute of Electrical and Electronic Engineers, Inc., 1989.

[5] Musa, John D.; Iannino, Anthony; & Okumuto, Kazihra. *Software Reliability Measurement, Prediction, Application.* New York, N.Y.: McGraw-Hill, 1987.

[6] Baumert, John H. *Software Measures and the Capability Maturity Model* (CMU/SEI-92-TR-25). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1992.

[7] Denise M. Woit, "*Operational Profile Specification, Test Case Generation, and Reliability Estimation for Modules*", 1994 http://citeseer.nj.nec.com/cache/papers/cs/5949/http:zSzzSzwww.sarg.ryerson.cazSzsargzSzpaperszSz199402-CRL281zSzindex.phtmlzSzsarg.pdf/woit94operational.pdf

[8] G. Booch, J. Rumbaugh, and I. Jacobson, The Unified Modeling Language User Guide, Addison-Wesley, 1999.

[9] A. Pasquini, A.N. Crespo, and P. Matrella, Sensitivity of reliability-growth models to operational profile errors vs testing accuracy, IEEE Trans. Reliability 45, 1996, 531{540 (errata ibid 46, 1997, 68).

[10] T. Adams, Total variance approach to software reliability estimation, IEEE Trans. Software Eng. 22, 1996, 687-688.

Chapter 5 Software Reliability Modeling

5.1 Introduction

In the field of software reliability engineering, one particular aspect that has received the most attention is software reliability modeling. It is rational, since all activities in SRE are based on models established in literature. Models are the basis of SRE. During the past 25 years, research activities in software reliability engineering have been studied and more than 50 statistical models have been developed.

A **software reliability model** specifies the general form of the dependence of the failure process on the principal factors that affect it: fault introduction, fault removal, and the operational environment [1]. Figure5.1 shows the basic idea of reliability modeling:

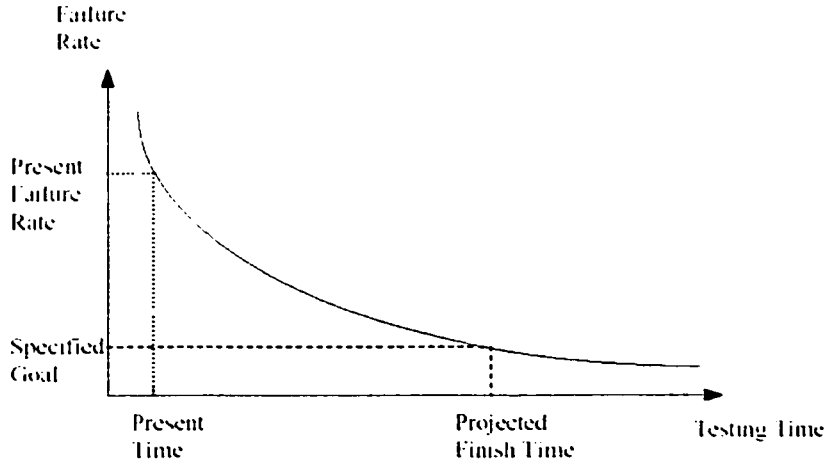

Figure5.1 Basic idea of software reliability modeling [2]

From figure5.1, we can see that the failure rate is decreasing due to the discovery and removal of software failures. At any "present time", it is possible to observe a history of the failure rate of the software. Software reliability modeling forcasts the curve of the

failure rate by statistical evidence [2]. Thus the testing time and the reliability of the software could be predicted.

5.2 Relationship of Metrics and Models

Models, metrics are two interrelated technologies in the measurement technology.

We have talked about metrics in chapter2. Here, it is necessary for us to identify the relationship between metrics and models to avoid confusing.

Generally speaking, a model is an equation (or set of equations) that takes software metrics data as input parameters and calculates some other parameter. The output from a model is therefore a software metric, too, and may be used as input to another model. For example, a key input parameter for many cost and some reliability models is the software size. When these models are used to predict the cost or reliability of yet-to-be-developed software, the software size is not known but can be estimated by feeding project estimates into a sizing model [3]. Figure5.2 shows this relationship:

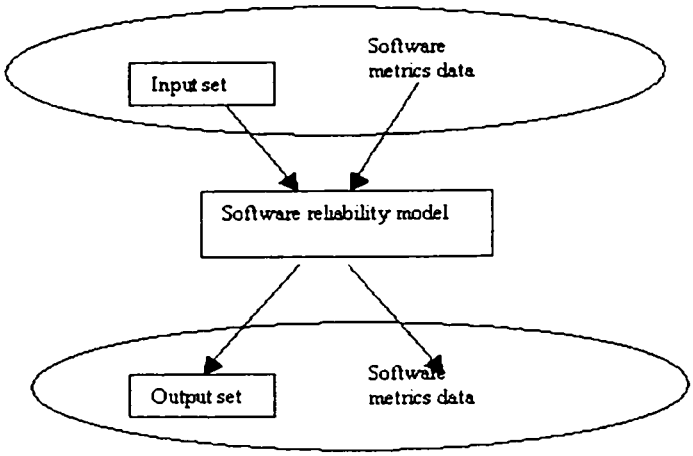

Figure5.2 Metrics and models

5.3 Classification Of Software Reliability Models

Even through software is pervasive in today's world, and the need of reliability engineering in software development is obvious, not many software practitioners, developers, or users utilize software reliability models to evaluate computer software reliability. A survey conducted in the late 1990s by the American Society for Quality (ASQ) reported that only 4% of the participants responded positively when asked if they could use a software reliability model.

There are many reasons for this phenomenon, one of them is because these participants do not know how to select and apply these models. Numerous analytical models for predicting reliability and fault content in a software system are available. These models attempt to capture and quantify the inherent uncertainties in a software production process. However, none of the single model is sufficient for measuring the software reliability. Besides, the accuracy of these models varies. Thus, it is necessary to have an overview of these proposed models, and learn how and when to apply them. A good way to perform this is to classify these models and study them.

5.3.1 Classification Schemes

During the past researches, many classification schemes have been developed and introduced. Here we pick up three most widely used classification scheme.

5.3.1.1 Musa And Okumoto Classification Scheme

In his early research, Musa and Okumoto [4] developed the first classification scheme. It allows relationships to be established for models within the same classification groups and shows where model development has occurred. For this scheme, Musa and Okumoto classified models in terms of five different attributes. They are:

1. Time domain: Wall clock versus execution time.
2. Category: The number of failures that can be experienced in infinite time. This is either finite or infinite, the two subgroups.
3. Type: The distribution of the number of failures experienced by time t.
4. Class: (Finite failure category only). Functional form of the failure intensity expressed in terms of time.
5. Family: (Infinite failure category only). Functional form of the failure intensity function expressed in terms of the expected number of failures experienced.

This classification scheme has a strong inference to the field of software reliability engineering. Many later researches follow the scheme, the handbook of reliability engineering by Lyu [2] utilize this scheme to introduce different models.

5.3.1.2 Hoang Pham Classification Scheme

In his book "software reliability and testing" [5], Hoang Pham classified the software reliability models into two types: the deterministic and the probabilistic.

The deterministic model is used to study the number of distinct operators and operands in a program as well as the number of errors and the number of machine instructions in the program. Performance measures of the deterministic type are obtained by analyzing the program texture and do not involve any random event [5].

The probabilistic model represents the failure occurrences and the fault removals as probabilistic events. The probabilistic software reliability models are classified into five different groups:

- Error seeding
- Failure rate
- Curve fitting
- Reliability growth
- Non-homogeneous Poisson process

Error Seeding: There are two types of errors: indigenous and induced. This group of models estimates the number of errors in a program by using the multistage sampling technique. The unknown number of indigenous is estimated from the number of induced errors and the ratio of the two types of errors obtained from the debugging data.

Fault Rate: This group of models is used to study the functional forms of the per-fault failure rate and program failure rate at the failure intervals.

Curve fitting: This group of models uses regression analysis to study the relationship between software complexity and the number of errors in a program, the number of changed, failure rate, or time between failures.

Reliability Growth: This group of models measures and predicts the improvement of reliability through the debugging process.

Non-homogeneous Poisson process (NHPP): This group of models provides an analytical framework for describing the software failure phenomenon during testing. The main issue in the NHPP model is to estimate the mean value function of the cumulative number of failures experienced up to a certain time point.

Markov: A Markov process has the property that the future behavior of the process depends only on the current state and is independent of its past history [2]. This group of models is a general way of representing the software failure process. The number of remaining faults is modeled as a stochastic counting process [4]. When a continuous-time discrete-state Markov chain is adapted, the state of the process is the number of remaining faults, and time between failures is the sojourning time from one state to another. If we assume that the failure rate of the program is proportional to the number of remaining faults, linear death process and linear birth-and-death process are two models readily available. The former assumes that the remaining errors are monotonically non-increasing, whereas the latter allows faults to be introduced during debugging.

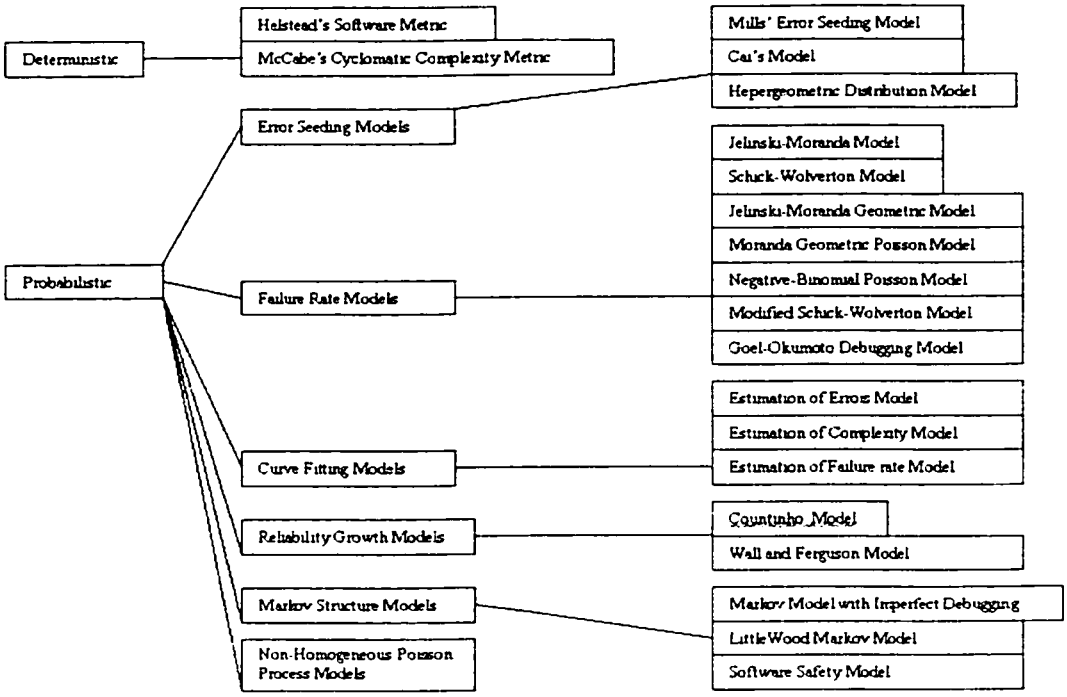

Figure5.3 Overview of Hoang Pham's Classification with some representive models

Model Introduction

In this verse, we will have a look at two deterministic models. Because there are too many models, it is hard to cover them all in this survey. In the next verse, we will have look at another classification scheme developed by Kishor S.Trivedi. Thus, we will introduce more models there. For the overview of Hoang Pham's classification, please reference figure3.

Halstead's Software Metric

Halstead's software metric is used to estimate the number of errors in the program, it falls in the deterministic category. For detail reference, please check chapter2.

McCabe's Cyclomatic Complexity Metric

In chapter 2, we have talked about this metric. The reason to mention it here is to notify that this metric can be used directly to model the reliability of software.

5.3.1.3 Kishor S. Trivedi Classification

Trivedi [6] classifies the popular software reliability models as data-domain models and time-domain models. An overview of Trivedi's classification is provided in figure4:

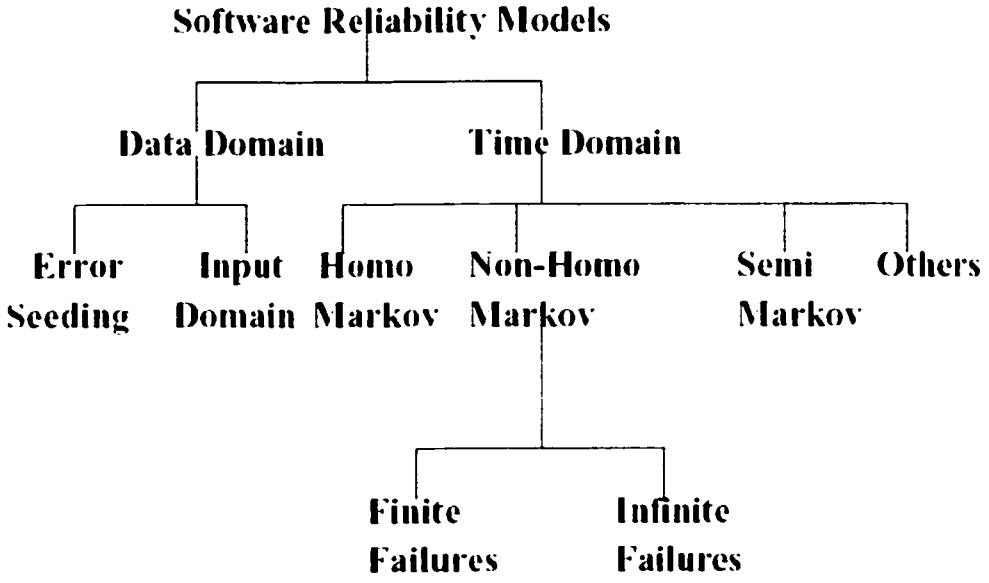

Figure5.4 Overview of Trivedi's classification

Data-domain models

This group of models focus on the fault content of the software product under consideration. They select sample data sets representative of the expected operational usage for the purpose of estimating the residual number of faults in the software under consideration [6]. It is further classified as (1) **error seeding models** and (2) **input-**

domain models. We have introduced error seeding models in the previous verse. Here, let us have a look at input-domain models.

(2) Input-domain models: This group of models use the execution of an input state as the index of function reliability. The ratio of the number of the successfully execution inputs to the total number of inputs gives an estimate of the reliability of the software product [6]. The basic method involves the generation of test cases from an input distribution, which represents the operational usage of the program. Since it is difficult to obtain this distribution, the input domain is partitioned into a set of equivalence classes.

Time-domain models

This group of models model the underlying failure process of the software under consideration, and use the observed failure history as a guideline, in order to estimate the residual number of faults in the software, and the test time required to detect them. Let us have a look at the subcategories.

(1) **Homogeneous Markov Models**: This group of models assume that the initial number of faults in a program is unknown but fixed. The number of faults forms the state space of homogeneous Markov chain.

(2) **Non-Homogeneous Markov Models (NHPP):** See **5.3.1.2**

(3) **Semi-Markov models:** This group of models assumes that the initial number of faults in the software product is unknown but fixed, and that the failure intensity of the

software, or the transition rate from a given state, depends not only on the number of residual faults in the software, also on the time elapsed in the state.

Category	Representative models
Error-seeding Models	Mills' Hypergeometric Model
Input-domain models	Nelson Input domain Model ; Brown and Lippw Input Domain Model ;
Homogeneous Markov Model	Jelinski-Moranda de-eutrophication Model; Goe-Okumoto Imperfect Debugging Model LittleWood Markov Model
Finite Failures NHPP Models	Goel-Okumoto NHPP Model; Delayed S-Shaped NHPP Model; Inflection S-Shaped Model; C1 NHPP Model; Pareto NHPP Model; LittleWood NHPP Model; Hyperexpeonetial NHPP Model
Infinite Failures Models	Musa-Okumoto Logarithmic Poisson Execution Time Model ; Duane Model ; Log-Power NHPP Model;
Other Models	Littilewood-Verall Bayesian Model; Littlewood and Keiller Bayesian Model; Random Coefficient Autoregressive Model

Table 5.1 Some Representative Models According To Trivedi's Classification

Model Introduction

In this section, we will have a close look at some representative models according to Trivdi's classification scheme, as shown in Table5.1.

Mills' Hypergeometric Model [2, 4, 7]

The model was originally developed in 1970 by Mill, and with later improvements by Baisin(1973) and Huang(1984).

This model estimates the number of errors in a program by introducing seeded errors into the program. From the debugging, the unknown number of inherent errors could be estimated. In this model, a known number faults, n_1, are induced errors, in a software product. n_1 originally has N indigenous faults, N is estimated from testing by a

hypergeometric distribution. The probability that exactly k out of r detected faults are seeded faults is given by:

$$P_k(N) = \frac{\binom{n_1}{r}\binom{N - n_1}{r - k}}{\binom{N}{r}}$$

where

N = total number of inherent errors

r = total number of errors removed during debugging

n_I = total number of induced errors

k= total number of induced errors in r removed errors

Since n_I, r, k are known, the maximum likelihood estimates of N, can be shown as:

$$\hat{N} = \frac{n_1 r}{k}$$

In Mill's equation, the probability of selecting a sample of size r is given by:

$$p = \binom{N}{r}$$

Since the total number of faults is $N + n_I$, the above probability is incorrect, Basin [8] gives the actual probability by:

$$p = \binom{N + n_1}{r}$$

$$\hat{N} = \frac{n_1(r - k)}{k}$$

Nelson Input Domain Model [6]

To begin the discussion, let us first identify a word '*run*'. A *run* is the execution of an input state. The reliability is estimated by running the *run*. The *run* is randomly chosen

from a set $\{E_i := 1,2,\ldots,N\}$. Each E_i is a set of data values needed to make a run. The random sampling of n is done according to a probability vector P_i, where P_i is the probability that E_i is sampled. The probability vector $\{P_i : I= 1,2,\ldots, N\}$ defines the operational profile or the user input distribution. If the number of failures is k, then the estimate of reliability is given by

$$\hat{R} = 1 - \frac{k}{n} = \frac{n-f}{n}$$

Jelinski-Moranda De-eutrophication Model [2, 9]

This model is one of the earliest models proposed; moreover it is still used today. It has the following assumptions:

1. The software has N faults at the beginning of the test.
2. Each of the faults is independent and all faults will cause a failure during testing.
3. The repair process is instantaneous and perfect, i.e., the time to remove the fault is negligible, new faults will not be introduced during fault removal.
4. The software failure rate is proportional to the current content of the system.

The program failure rate at the *i*th failure interval is given by

$$\lambda(t_i) = \phi\,[N-(i-1)],\quad i=1, 2,\ldots,N$$

ϕ = a proportional constant, the contribution any one fault makes to the overall program;

N = the number of initial faults in the program

t_i = the time between the $(i-1)$ and the ith failures

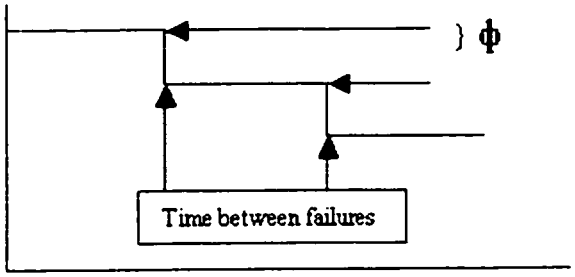

Figure5.5 De-eutrophication process [2]

The software reliability function is:

$$R(t_i) = e^{-\phi(N-i+1)t}$$

When the failure data set t_i is given and ϕ is known. The maximum likelihood estimation of N can be obtained by solving the following equation:

$$\sum_{i=1}^{n} \frac{1}{N-(i+1)} = \Phi \sum_{i=1}^{n} t_i$$

When ϕ is unknown, we can obtain ϕ and N by solving the following equations:

$$\Phi = \frac{\sum_{i=1}^{n} \frac{1}{N-(i-1)}}{\sum_{i=1}^{n} t_i}$$

and

$$n\sum_{i=1}^{n} t_i = \left[\sum_{i=1}^{n} [N-(i-1)]t_i\right]\left[\sum_{i=1}^{n} \frac{1}{N-(i-1)}\right]$$

• Based on the basic JM model, much has been written in the literature and many variation of it have been proposed. Goel-Okumoto Debugging Model and LittleWood Markov Model are two of them, as we discuss below.

Goel-Okumoto Debugging Model [2, 4, 10]

This model extends the basic JM model by adding an assumption:

1. A fault is removed with probability p whenever a failure occurs.

The failure rate function of the base JM model with imperfect debugging at the ith failure interval becomes

$$\lambda(t_i) = \phi\,[N - p(i-1)],\quad i = 1, 2, \ldots, N$$

The reliability function is

$$R(t_i) = e^{-\phi(N-p(i-1))t_i}$$

• The basic JM model assumes that the fault removal process is perfect. However, it is not practical. The Geol-Okumoto debugging model overcomes this shortcoming.

Hyperexponential Growth Model (NHPP) [4, 11]

This model is based on the list assumptions

1. A program has a number of clusters of modules.
2. Each cluster has a different initial number of errors and a different failure rate.

Since the sum of exponential distributions becomes a hypereponential distribution, the mean value function (m(t)) of the hyperexponential class NHPP model is

$$m(t) = \sum_{i=1}^{n} a_i[1 - e^{-b_i t}]$$

where

n = *number of clusters of modules*

a_i = *number of initial faults in cluster i*

b_i = *failure detection rate of each fault in cluster i*

The failure intensity λ (t) is given by

$$\lambda(t) = a\sum_{i=1}^{n} a_i b_i e^{-b_i t}$$

NHPP S-shaped Model [2, 12]

S-shaped means the reliability growth curve is an S-shaped curve which means that the curve crosses the exponential curve from below and the crossing occurs once and only once. The detection rate of faults, where the error detection rate changes with time, become the greatest at a certain time after testing begins, after which it decreases exponentially.

It is based on the following assumptions:

1. The error detection rate differs among faults
2. Each time a software failure occurs, the software error which caused it is immediately removed, and no new errors are introduced.

This can be shown in the following differential equation:

$$\frac{\partial m(t)}{\partial t} = b(t)[a - m(t)]$$

where

a = expected total number of faults that exist in the software before testing

b(t) = failure detection rate, also called the failure intensity of a fault

m(t) = expected number of failures detected at time t

Solve the above equation, we have

$$m(t) = a\left(1 - e^{\int_0^t b(u)du}\right)$$

• Faults are covered by other faults at the beginning of the testing phase, and before these faults are actually removed, the covered faults remain undetected.

• Software testing process usually involves a learning process where testers become familiar with the software products, environments, and software specifications.

• There are several S-shaped models, e.g., delayed S-shaped models, and inflection S-shaped models. (We will introduce in the following sections.)

Delayed S-Shaped NHPP Model [2, 12]

The delayed S-Shaped software reliability growth model was proposed to model the software fault removal phenomenon in which there is a time delay between the actual detection of the fault and its reporting. The test process in this case can be seen as consisting of two phase: fault detection and fault isolation. The mean value function, m(t), and the failure intensity λ (t), is given by:

Assume: $$b(t) = \frac{b^2 t}{bt + 1}$$

Where

b = the error detection rate per error in the steady-state

From the m(t) of NHPP S-Shaped model, we have

$$m(t) = a[1-(1+bt)e^{-bt}]$$

$$\lambda(t) = ab^2te^{-bt}$$

The reliability of the software system is

$$R(s \mid t) = e^{-[m(t+s)-m(t)]}$$

$$= e^{-a[(1+bt)e^{-bt}-(1+b(t+s))e^{-b(t+s)}}$$

The expected number of errors remaining in the system at time t is given by

$$N(t) = m(\infty) - m(t)$$

$$= a(1+bt)e^{-bt}$$

NHPP Inflection S-Shaped Model [11]

This model was proposed to analyze the software failure detection process where the faults in a program are mutually dependent. Mutually dependent means that some faults are not detectable before other faults are removed. Moreover, the probability of failure detection at any time is proportional to the current number of detectable faults in the software. The isolated faults can be entirely removed. The mean value function, m(t), and the failure intensity λ (t), is given by:

Assume: $$b(t) = \frac{b}{1+\beta e^{-bt}}$$

b = *failure-detection rate*

β = *the inflection factor*

$$m(t) = \frac{a}{1+\beta e^{-bt}}(1-e^{-bt})$$

$$\lambda(t) = \frac{ab(1+\beta)e^{-bt}}{(1+\beta e^{-bt})^2}$$

The expected number of remaining errors at time t is thus given by

$$m(\infty) - m(t) = \frac{a(1+\beta)e^{-bt}}{(1+\beta e^{-bt})}$$

Musa-Okumoto Logarithmic Poisson Execution Time Model [2, 4, 13]

This model assumes the number of failures experienced by time τ (τ is the execution time) is Non-homogeneous Poisson process (NHPP), with the mean value function, **m(τ)** given by

$$m(\tau) = \frac{1}{\theta}\ln(\lambda_0\theta\tau + 1)$$

where λ_0 = *initial failure intensity*

θ = *the failure decay parameter* ($\theta > 0$)

The failure intensity is given by

$$\lambda(\tau) = \frac{\lambda_0}{(\lambda_0\theta\tau + 1)}$$

The program reliability is given by

$$R(\tau_i \mid \tau_{i-1}) = \left[\frac{\lambda_0\theta\tau_{i-1} + 1}{\lambda_0\theta(\tau_{i-1} + \tau_i) + 1}\right]^{1/\theta}$$

• The total number of failures for this model is infinite. It is very likely that the number of inherent faults in a program is finite.

• It would be very useful if we could relate the execution time component parameters of the model to characteristics of the software product, the development process, and the execution environment. [4]

• the relationship of failure intensity and execution time is show as figure 6 below:

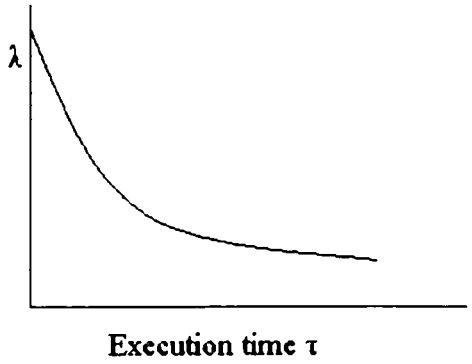

Figure5.6 Failure Intensity For The Logarithmic Passion Model

5.4 Summary

So far, three software reliability model classification schemes have been introduced as well as some representative models. It will be very hard to introduce all the previously reported models in this survey due to the time and space limitation, and it will be meaningless.

One problem with models is very similar to metrics, which there is an overwhelming number of models has been proposed to address the issue of software reliability assessment. We must be aware that no single model can be recommended universally to users under any circumstances. In the mean while, the best models may vary from time to time and differ form application to application. Classification is one efficient way of studying and applying models.

5.5 Reference

[1] S.R.Dalal. ; M.R.Lyu. ; C.L.Mallows. *Software Reliability*, Bellvore, Lucent Technologies, At&T research.

Http://www.cse.cuhk.edu.hk/~lyu/paper_ps/ENCBSTT4.ps

[2] M.R Lyu (ed), handbook of software reliability engineering, McGraw-Hill and IEEE Computer Society Press, New York, 1996,

[3] "A History of Software Measurement at Rome laboratory",

http://www.dacs.dtic.mil/techs/history/toc.html

[4] Musa, John D.; Iannino, Anthony; & Okumuto, Kazihra. *Software Reliability Measurement, Prediction, Application.* New York, N.Y.: McGraw-Hill, 1987

[5] Huang Pham, "software reliability and testing", IEEE computer society press, the institute of electrical and electronics engineers., Inc. 1995.

[6] http://www.ee.duke.edu/~kst/

[7] H.D.Mills, "On the Statistical Validation of Computer Programs", IBM Federal Syst. Div., Gaithersburg, MD, Rep. 72-6015, 1972

[8]S.L.Basin, "Estimation of software Error Rates via Capture-Recapture Sampling," Science Application, Inc., Palo Alto, CA, Sept. 1973

[9] A.A. Abdel-Ghally, P.Y.Chan, and B.LittleWood, "Evaluation of competing software reliability Predictions," IEEE Trans. On Software Engineering, VOL. SE-12, No.,9 September 1986.

[10] A.L.Goel, "software reliability models: Assumptions, Limitations, and Aplicability," IEEE Trans, on software engineering, VOL SE-11, NO12, December 1985.

[11] M.Ohba, "software reliability analysis models," IBM J.Res Develop, VOL. 28, No.t, July 1884

[13] J.D.Musa, "A theory of software reliability and it's application," IEEE Trans. On software Engineering, vol.SE-1, No.3, September 1975

Chapter 6 SRE & Software Development Lifecycle

6.1 Introduction

Software Reliability Engineering (SRE) is a practical augment of software engineering. Recently, it was accepted as "best practice" by one of the major developers of telecommunications software (AT&T). Many other organizations are using, experimenting with, or researching SRE. In this chapter, we will have a look at the benefits and cost of practicing software reliability engineering. Then, we will analyze the relationship between software development life cycle and software reliability engineering activities to solve the problems like, "when are the measures applied?" and "what should be done to augment the software engineering by software reliability engineering?"

6.2 Benefits and Costs

Due to proprietary reasons, the direct economic report of practicing SRE is hard to access. Previous studies show that the cost-benefit ratio of applying SRE can be six or more [1]. As one example of the proven benefit of SRE, AT&T applied SRE to two different releases of a switching system, International Definity PBX. Customer-reported problem decreased by a factor of 10, the system test interval decreased by a factor of 2, and total development time decreased 30%. No serious service outages occurred in 2 years of deployment of thousands of systems in the field [2]. Another example was from Tierney (1997), the late 1997 survey showed that Microsoft had applied software reliability engineering in 50 percent of its software development groups, including projects such as Windows and Word. The benefits they observed were increased test

coverage, improved estimates of amount of test required, useful metrics that helped them establish ship criteria, and improved specification reviews [3].

Generally, SRE will support the software development in the following ways:

• *Satisfy customer requirements more precisely.*

Precisely defined reliability requirements help to testers to verify that the finished product meets customers' needs before it is released.

• *Better schedule control.*

SRE avoids wasting time for unnecessary testing, and help delivering the exact reliability needed by the customer.

• *Increase productivity.*

The productivity is improved by using the Software Reliability Measurement technologies to focus on developing and testing for exactly the reliability needed.

• *Better Resource allocation*

SRE supports to predict the amount of system test resources needed, avoiding waste.

Although software reliability engineering brings a lot of benefits, the cost of practicing is affordable. It is estimated that routine application of SRE does not add more than several percent to the overall cost of a project. A medium to large project involving 40-100 persons may require pre-project activities totaling about one to two person-weeks, definition of operational profiles may require on to three person months, and routine collection and analysis of project failure and effort data may cost between one half to one person-day per week. [4]

6.3 Software Lifecycle versus SRE activities

A software lifecycle provides a systematic approach to developing, using, operating, and maintaining a software system. The standard IEEE computer dictionary has defined the software lifecycle as:

"That period of time in which the software is conceived, developed and used."

There are many different definition of software lifecycle, and the classification of software lifecycle varies. The IEEE standard dictionary of measures to produce reliable software classifies the software lifecycle into early, middle, and late segments, as shown in table6.1:

Lifecycle segments	Phases	Reliability Objective
Early	Concepts Requirements Design	The early segment relates to the potential causes of system reliability
Middle	Implementation Test Installation and checkout	The middle segment relates to the reduction of process errors that can improve the efficiency of software development.
Late	Operations and Maintenance Retirement	The late segment relates to actual system performance reliability.

Table6.1 Lifecycle Classification (according to [5])

In this classification scheme, the software reliability engineering activities are not clearly defined, and the boundary between each segment is vague. Thus, it is not widely accepted.

A more popular software development lifecycle phase is given in the handbook of software reliability engineering in [2], which shows in figure6.1:

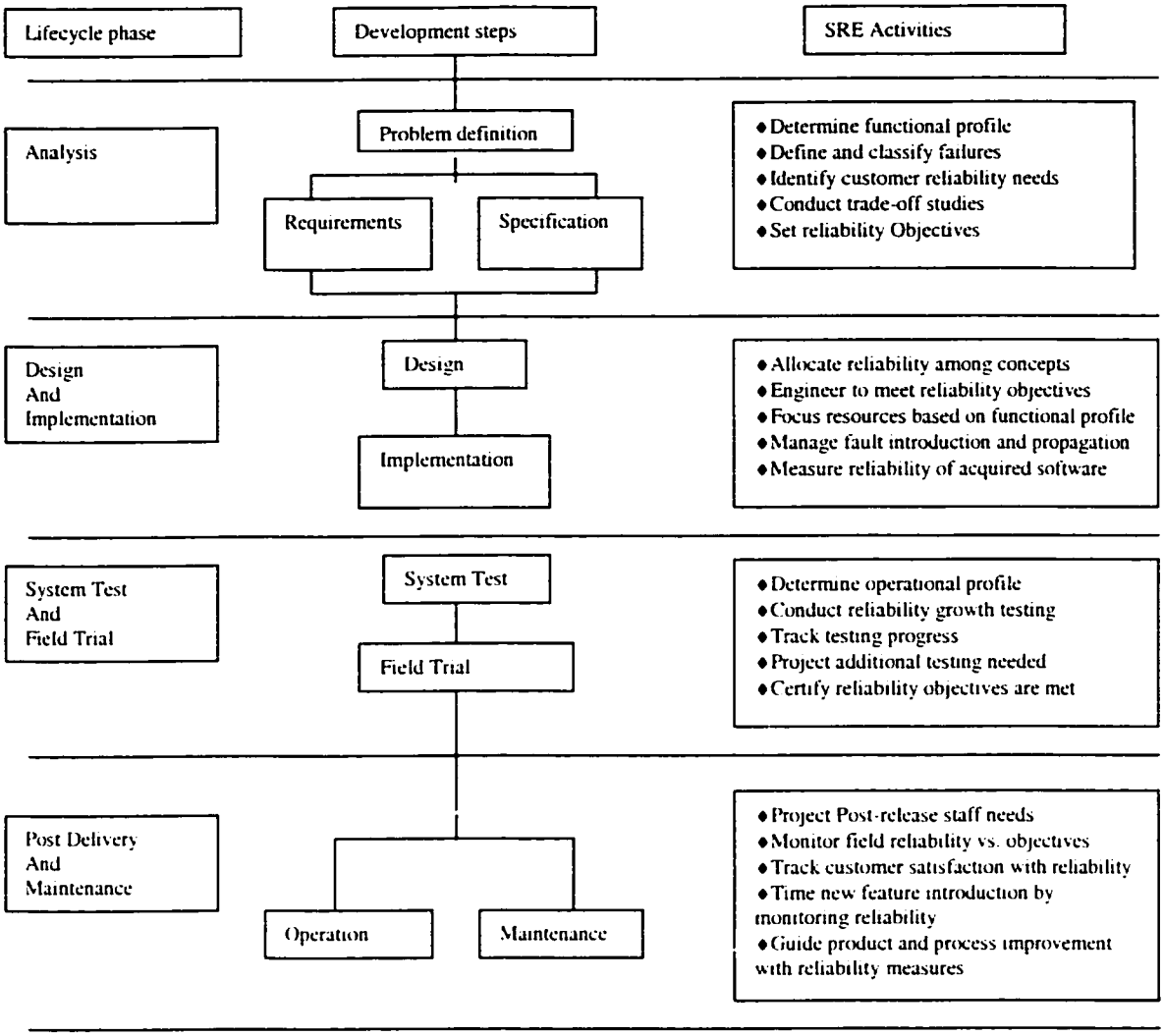

Figure6.1. SRE activities in the software product lifecycle (based on [2])

Figure1 also illustrates SRE activities across the phases of the software product lifecycle. The phase as well as SRE activities need not follow a neat sequence. There is considerable and overlap and iteration. Each SRE activity is therefore simply placed in the phase in which most of the effort occurs. The rest sections discuss the SRE activities of each phase in detail.

6.3.1 SRE During Analysis Phase

The analysis phase is the first step in the software development process. It is also the most important phase in the whole process and the foundation of building a successful software product. The purpose of the analysis phase is to define the requirements and provide specifications for the subsequent phases and activities. It is composed of three major activities: Problem definition, requirements and specifications.

Problem definition develops the problem statement and the scope of the project. Requirement activity consists of requirement collection and requirement analysis. Which further includes a feasibility study and documentation. The specification is transforming the user-oriented requirements into a precise form oriented to the needs of software engineers. According to the ANSI/IEEE standards: "The requirements specification shall clearly and precisely describe the essential functions, performances, design constraints, attributes, and external interfaces. Each requirements shall be defined such that its achievement is capable of being objectively verified...". Research indicates that increased effort and care during specification will generate significant rewards in terms of dependability, maintainability, productivity, and general software quality. [6,7,8]

Software reliability engineering augments analysis phase through a set of activities. Table2 shows the summary of these activities and their benefits:

Activity Name/No.	Content	Benefit
Activity 1: Define and classify failures	•Define failure from customers' perspective •Group identified failures into a group of severity classes from customer's perspective •Usually 3-4 classes are sufficient	Release software at a time that meets customer reliability needs but is as early and inexpensive as possible
Activity 2: Identify customer reliability needs	•What is the level of reliability that the customer needs? •Who are the rival companies and what are rival products and what is their reliability?	
Activity 3: Determine operational profile	•Based on the tasks performed and the environmental factors	Speed up time to market by saving test time, reduce test cost, have a quantitative measure
Activity 4: Conduct trade-off studies	•Reliability and functionality •Reliability, cost and delivery date	Increase market share by providing a software product that matches better to customer needs
Activity 5: Set reliability objectives	•Explicit requirement statements from a request for proposal or standard documents •Customer satisfaction with a previous release or similar product •Capabilities of competition •Trade-offs with performance, delivery date and cost •Warranty, technology capabilities	Release software at a time that meets customer reliability needs but is as early and inexpensive as possible.

Table 6.2 Summary of software reliability activities and benefits (base on [2])

6.3.2 SRE During Design and Implementation Phase

The design stage is concerned with building the system to perform as required. It involves translating a requirements specification into a design of the software product. There are two stages of design: architecture design and detailed design. The architecture design expresses the system concept in terms of hardware and software components and the interfaces among them and with the external environment [9]. Detailed design is about designing the program and algorithmic details. The activities within detailed design are program structure, program language and tools, validation and verification, test planning, and design documentation.

Implementation stage involves translating the design into the code of a programming language, beginning when the design document is baselines. Coding consists of the following activities: identifying reusable modules, code editing, code inspection, and final test planning. The output of this stage is an operational system. [6]

Software reliability engineering augments the design and implementation phase through a set of activities. Table3 shows the summary of these activities and their benefits:

Activity Name/No.	Content	Benefit
Activity 6: Allocate reliability among components	•Determine which systems and components are involved and how they affect the overall system reliability	Reduce development time and cost by striking better balance among components
Activity 7: Engineer to meet reliability objectives	•Plan by using fault tolerance, fault removal and fault avoidance technologies	Reduce development time and cost with better design
Activity 8: Focus resources based on operational profiles	•Operational profile guides the designer to focus on features that are supposed to be more critical	Speed up time to market by guiding development priorities, reduce development cost
Activity 9: Manage fault introd-uction and propagation	•Reliability and functionality •Reliability, cost and delivery date	Maximize cost-effectiveness of reliability improvement
Activity 10: Measure reliability of acquired software	•Certification test using reliability demonstration chart	Reduce risks to reliability, schedule, cost from unknown software and systems

Table 6.3 Summary of software reliability activities and benefits (base on [2])

Notice: Activity 7,8,9 are normally involved in the design stage; activity 8,9,10 are normally involved in the implantation stage. In the design stage, activity 8 can help a developer focus on what is really important from the customer's standpoint, also it can help allocate effort during the implementation stage, based on the relative usage and criticality of different functions.

6.3.3 SRE During The System Test And Field Test Phase

Testing is the verification and validation activity for the software product. The goals of the testing phase are (1) to affirm the quality of the product by finding and eliminating faults in the program, (2) to demonstrate the presence of all specified functionality in the product, and (3) to estimate the operational reliability of the software. [10]

The system test tests all the subsystems as a whole to determine whether specified functionality is performed correctly as the results of the software. The field test, also called field trial, beta test, is to install the product in a user environment and allows the user to test the product where customers often lead the test and define and develop the test cases. Normal activities include: (1) execution of system and field acceptance tests, (2) checkout of the installation configurations and (3) validation of software functionality and quality. SRE augments the testing process by the following activities shown in table4:

Activity Name/No.	Content	Benefit
Activity 11: Determine operational profile used for testing	• Decide upon need of multiplicity of operational profile • Decide upon ultrareliable operations	Reduce the chance of critical operations going unattended, speed up time to market by saving test time, reduce test cost
Activity 12: Track testing progress and certify that reliability objectives are met	• Conduct feature test, regression test and performance and load test • Conduct reliability growth test	Know exactly what reliability the customer would experience at different points in time if the software is released at those points
Activity 13: Certify that reliability objectives and release criteria are met	• Check accuracy of data collection • Check whether test operational profile reflects field operational profile • Check customer's definition of failure matches with what was defined for testing the product	Release software at a time that meets customer reliability needs but is as early and inexpensive as possible; verity that the customer reliability needs are actually met

Table6.4 Summary of software reliability activities and benefits during the system test and field test phase (base on [2])

Normally, activity 11, 12 are used in the system test stage; activity 13 is used in the field test stage.

6.3.4 SRE During The Post-delivery And Maintenance Phase

This phase is the final phase in the software lifecycle. Operation phase usually contains activities like installation, training. Maintenance is defined as "modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment." [11]. Maintenance activities include system improvement and replacement strategies. There are four types maintenance: correction maintenance, adaptive maintenance, perfective maintenance, and emergency maintenance.

SRE augments this phase by the activities described in the table5:

Activity Name/No.	Content	Benefit
Activity 14: Project post-release staff needs	•Customer's staff for system recovery; supplier's staff to handle customer-reported failures and to remove faults	Reduce post-release costs with better planning
Activity 15: Monitor field reliability vs. objectives	•Collect post release failure data systematically	Maximize likelihood of pleasing customer with reliability
Activity 16: Track customer satisfaction with reliability	•Survey product features with sample customer set	
Activity 17: Time new feature introduction by monitoring reliability	•New features bring new defects. Add new features desired by the customers if they can be managed without sacrificing reliability of the whole system	Ensure that software continues to meet customer reliability needs in the field
Activity 18: Guide product and process improvement with reliability measures	•Root-cause analysis for the faults •Why the fault was not detected earlier in the development phase and what should be done to reduce the probability of introducing similar faults	Maximize cost-effectiveness of product and process improvements selected

Table6.5 Summary of Software Reliability activities and Benefits

6.4 Summary

SRE presents a lifecycle approach to manage the software reliability. Through practicing SRE activities, software engineers and managers can estimate and predict the rate of failure occurrence in software.

6.5 Reference

[1]. W. Ehrlich, B. Prasanna. J. Sampfel, J. Wu, "*Determining the Cost of Stop-Test Decisions*," IEEE Software, Vol 10(2), pp 33-42., 1993

[2]. Handbook of Software Reliability Engineering, McGraw Hill, editor M. Lyu, 1996.

[3]. http://www.geocities.com/itopsmat/SoftwareEngineeringReliabilityModel.pdf

[4] http://www.computer.org/proceedings/issre/0443/04430062abs.htm

[5] IEEE Standard Dictionary of Measures to Produce Reliable Software (IEEE Std 982.2-1988). New York, N.Y.: Institute of Electrical and Electronic Engineers, Inc., 1989.

[6] Software Development Life Cycle Models

http://www.cs.ucd.ie/staff/dwilson/home/comp3013/sdlc.ppt

[7] Roger S. Pressman, "*Software Engineering: A Beginner's Guide*" Published 1988

[8] Donald J. Reifer (Editor), "*Software Management*", IEEE Computer Society; ISBN: 0818680016; 5th edition (September 1997)

[9] Ian Sommerville, "*Software Engineering (6th Edition)*" Addison-Wesley Pub Co; ISBN: 020139815X; 6th edition (August 7, 2000)

[10] S. Norman: Software Testing Tools, Ovum Ltd, London, 1993.

[11] IEEE Standard Glossary of Software Engineering Terminology, [ANSI/IEEE Std. 729-1983]

Chapter 7 Software Reliability Tools

7.1 The need of tools

There are around 50 software reliability models proposed in the literature. To practice SRE, engineers must apply these models during different phases of software development lifecycle. Since the engagement and application of software reliability models and the evaluation and interpretation of model results involve tiring computation-intensive tasks, it is widely believed that the only practical usage of reliability models is through software reliability tools. In the *handbook of software reliability engineering* by M.R.Lyu [1], the advantages of using tools are summarized as follows:

♦Tools provide most of the features needed in executing a software reliability analysis, resulting in a decrease of programming time.

♦ Comparing multiple models on the same failure data and changing the analysis to use a different model is easier to accomplish.

♦ Tools provide better error detection because many potential types of errors have been identified and are checked for automatically. The chance of a bug in the tool itself is very small.

♦ They provide a general framework for reliability estimation and prediction. Their basic structure is from the theories developed by researchers and uses the terminology of those models.

7.2 Criteria of selecting tools

Current available tools in the field of software reliability are enormous. Appendix A of *The Handbook of Software Reliability Engineering*, edited by M.R. Lyu, lists the characteristics of seven quantitative software reliability estimation and prediction tools. Annex B of *Software Safety and Reliability* [2], written by Debra S.Herrmann, lists sixteen commercial products except for the previous seven tools. Many other reliability estimation and prediction tools' lists can be found in the Internet resources [3,4]. How to select a proper tool is thus an issued placed in front of an analyst. Generally, the following criteria should be considered in selecting a tool for an organization [1]:

1. Availability of the tool for the company's computer system.
2. Cost of installing and maintaining the program.
3. Number of Studies likely to be done.
4. Types of system to be studied.
5. Quality of the tool documentation and support.
6. Ease of learning the tool
7. Flexibility and power of the tool.
8. Availability of tools, either in-house or on a network.
9. Goals and questions to be answered by the study.
10. Models and statistical techniques understood by the analyst
11. Schedule for the project and type of data collected.
12. Tool's ability to communicate the nature of the model and the results to a person other than the analyst.

7.3 Classification of Tools

There can be many ways to classify the tools, from the tools' characteristics to the development time, or from the models' point of view to see how models are applied. In this survey, we make the classification based on the time when the tools can be applied to predict or estimate the reliability during the software development lifecycle. Current tools can be broadly classified into three categories:

1. Tools which use static complexity metrics at the end of the development phase as inputs and either classify the modules into fault-prone or non fault-prone categories, or predict the number faults in a software module. An example of such a tool is the *Emerald system* [5].

The Emerald system uses the Datrix software analyzer to collect about 38 basic software metrics from the source code. Based on the experience in assessing previous software products, these metrics are used to identify patch-prone modules. The Emerald system can thus be used to determine the quality of a software product after the development phase, or before the test phase. It does not offer the capability of obtaining predictions based on the failure data collected during the testing phase.

2. Tools which accept failure data during the functional testing of the software product to calibrate a software reliability growth model based on the data, and use the calibrate model to make predictions about the future. AT&T SRE Toolkit, SMERFS, SoRel and CASRE are examples of such tools [1].

The very good description and comparison of such tools can be found in [1]. These tools are applied very late in the software development lifecycle. They can be used to estimate software reliability using the failure data to drive one or more of the software reliability growth model.

Here, we highlight the CARSE (Computer-aided software reliability estimation system). Major corporations including AT&T, Lucent, Microsoft, NASA, IBM, Motorola, and Nortel have used CASRE [6]. It is implemented as a software reliability modeling tools that address the ease-of-use issue as well as other issues. As a result, CASRE has a much better user interface. It is designed for the Windows environment. A Web-based version is also available. The high-level architecture for CASRE is shown in figure 1:

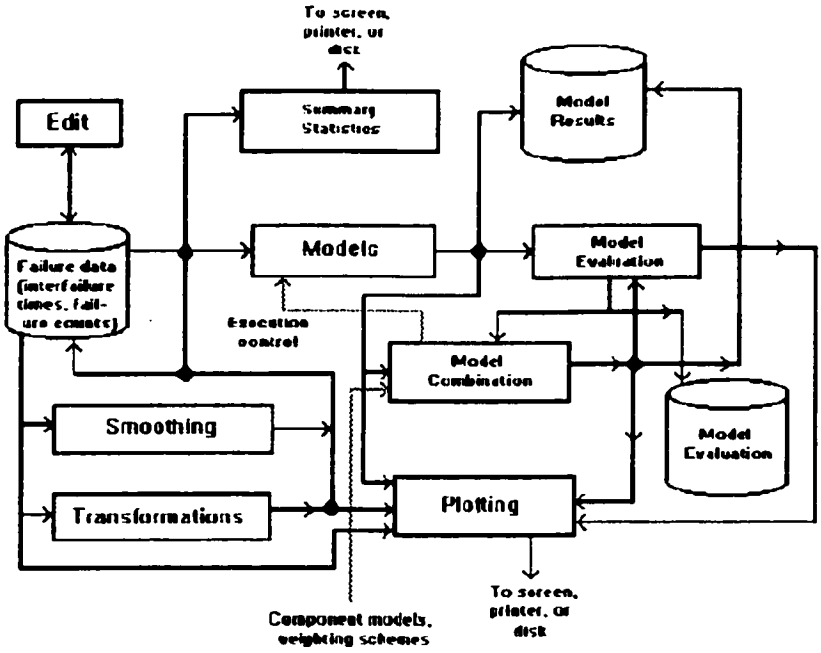

Figure7.1 High-Level Architecture For CASRE [7]

3. Tools which track the quality of a software product and provide insights throughout the lifecycle of the software. An example of such a tool is Software Reliability Estimation and Prediction Tool (SREPT) [8].

SREPT is design to suit the increasing need for a tool that can be used to track the quality of a software product during the software development process, right from the architectural phase all the way up to the operational phase. The high level architecture for SREPT is shown in figure2:

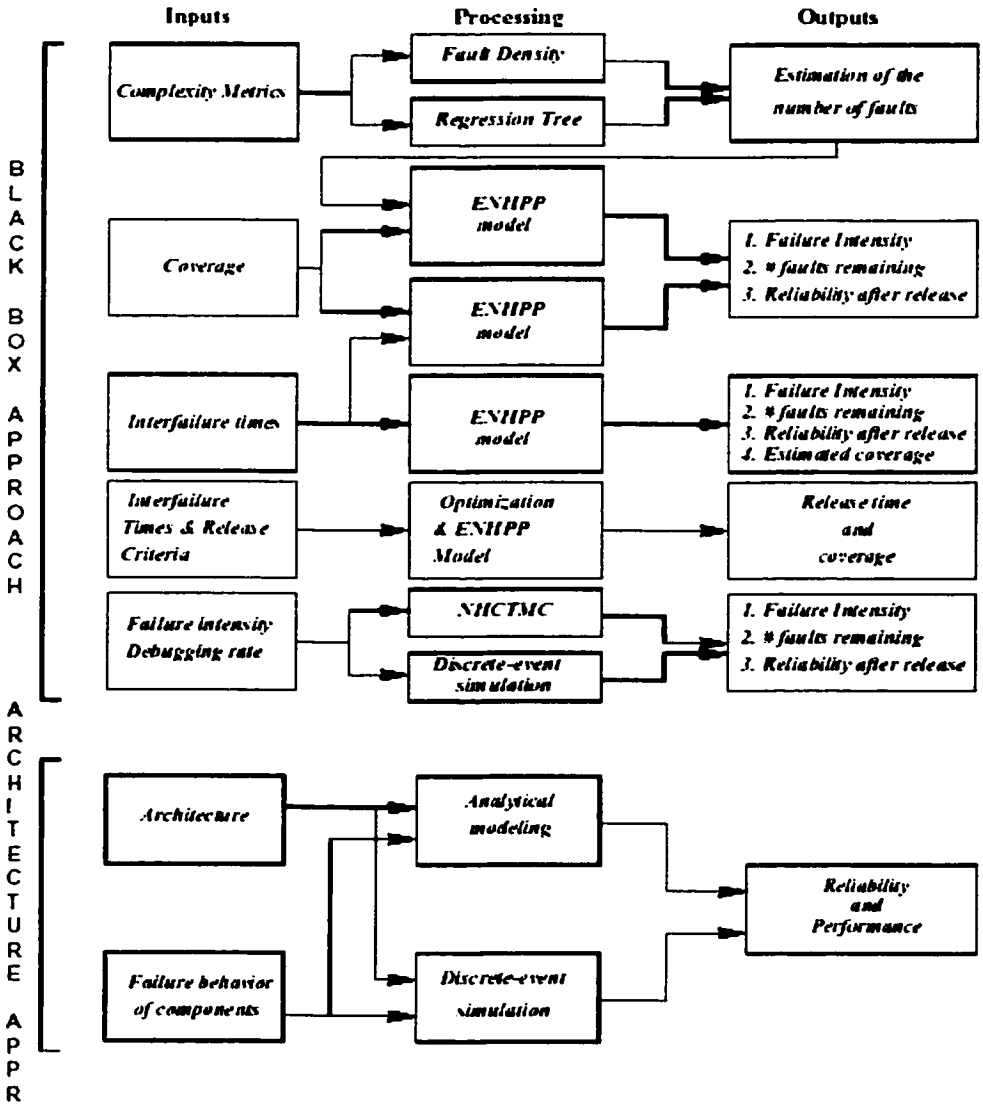

Figure7.2 High Level Architecture Of SREPT

SREPT uses black-box-based approach and architecture-based approach to software reliability prediction. Black-box-based approaches treat the software as a whole without considering its internal structure, basic techniques used are complexity metrics, test coverage, interfailure times-data, for the pre-test and testing phases. Architecture-based approaches use the internal control structure of the software to predict the reliability of software, it is applied at all stages in the software' lifecycle.

SREPT is a more recently developed tool, which has a user-friendly form in a GUI-based environment. It is expected to have a powerful impact in the area of software reliability estimation and prediction.

7.4 Summary

In this chapter, we provided a general description of selecting criteria for SRE tools as well as a broad classification of tools. Two tools (CASRE and SREPT) are highlighted to gain a global understanding of SRE tools.

7.5 Reference:

[1] Handbook of Software Reliability Engineering, McGraw Hill, editor M. Lyu, 1996.

[2] Debra S.Herrmann, *Software Safety and Reliability*, published by the IEEE Computer Society 10662 Los Vaqueros Circle, 1999.

[3] http://rac.iitri.org/InfoResources/Rac_Tools.html

[4] http://www.enre.umd.edu/

[5] J.P. Hudepohl, S.J. Aud, T.M. Khoshgoftaar, E.B. Allen, and J. Mayrand, \Eme-rald: Software Metrics and Models on the Desktop", *IEEE Software*, September 1996, pp. 56-60.

[6] http://www.cse.cuhk.edu.hk/~lyu/book/reliability/casre.html

[7] Michael R. Lyu, "*Design, Testing, and Evaluation Techniques for Software Reliability Engineering*". http://citeseer.nj.nec.com/32587.html

[8] http://www.ee.duke.edu/~kst/srept.html

Chapter 8 Research Directions

Based on this survey software reliability engineering, the further work about the current software metrics may necessarily focus on the following directions:

Model development

Models are the base of software reliability engineering. Even though more than 50 models have been proposed during the past 25 years, the development of new models is always essential. Most existing models for predicting software reliability are based purely on the observation of software product failures. These models also require a considerable amount of failure data to obtain an accurate reliability prediction. However, information concerning the development of software product, the method of failure detection, environmental factors, etc., are ignored. Many researchers are currently developing new models concerning these factors.

Standardization

It is recognized that no single software reliability model is appropriate to predict and estimate system reliability; however, multiple models may also introduce redundancies and confusions. Therefore, future research may identify one industry-standard software reliability models suite to reflect the reliability issues of software products.

Tools

Before 1996, tools were mostly developed as research projects by industrial, government, or academic labs. Many of them are not commercial products, and only very limited

models are applied. Since it is widely believed that the only practical usage of reliability models is through software reliability tools, it is necessary to develop more useful tools or some components easily embedded in specific software to automate the reliability application.

Adoption

To increase the adoption of SRE in real software development lifecycle, the SRE activities should be defined more clearly. This is especially true for software reliability prediction in the earlier phases of the lifecycle (e.g., the analysis and the design phase). There are a lot of works to do in specification reliability, requirement reliability to enhance the robustness of the earlier phases.

Reliability and Reuse over the Internet

As computers become more interconnected, it is being apparent that there are many benefits to being able to communicate with other computers and combine computing power. However, communication over long distances, and especially over the Internet, is fairly unreliable. There are many different points where problems could arise - from transmission delays to locating the proper computer to query. Through various techniques such as redundancy, fault tolerance, and reliable protocols can be applied to improve communications between computers.

Database reliablity

SRE approach can also be applied to estimate the level of completeness of data base schemas [1]. The study has just started and needs further work. Current approach is as

follows. A set of queries likely to be put to a database under construction is collected from prospective users, together with usage ratings. A usage rating is an integer between 1 and 5 that indicates the relative frequency with which a particular query will be put. A test suite is constructed in which each query has as many copies as its usage rating, and this collection of queries is randomized. The queries are put to the database one after another, and if the database schema has to be extended to deal with a query, this is registered as a failure. The failure data are analyzed using SRE techniques, thus providing a quantitative estimate of the completeness of the schema.

Component Reliability

Software components are the most promising idea extant for the efficient design of quality software system. Most of the research in components is devoted to specification, design, reuse, and cataloging of the components. The reliability of component is also very important, but has received less attention. Currently, the theory of component reliability has just been establishing [2]. A lot of future works (e.g., experimental validation, tool support for component developers and system designers) are required to be done.

Reference:

[1]. A.T. Berztiss and K.J. Matjasko, Queries and the incremental construction of conceptual models, in Information Modeling and Knowledge Bases VI, pp. 175{ 185, IOS Press, 1995.

[2]. "Theory of software reliability based on components", Dick Hamlet Portland State University Portland, OR, USA hamlet@cs.pdx.edu Dave Mason Denise Woit Ryerson.