BIOINFO Computer Engineering
ISSN: 2249-3980 & E-ISSN: 2249-3999, Volume 3, Issue 1, 2014, pp.-043-056.
Available online at http://www.bioinfopublication.org/jouarchive.php?opt=&jouid=BPJ0000196

SOFTWARE RELIABILITY: A REVIEW

DESAI C.G.*
Department of MCA, MIT, Aurangabad- 431 009, MS, India.
*Corresponding Author: Email- chitra.desai@mit.asia

Received: November 16, 2013; Accepted: February 06, 2014

Abstract- To build high reliability software it is essential to concentrate on quality aspects of software from the very beginning of the software development life cycle. Therefore software reliability attribute is one of the important dimensions of software quality. However, several aspects contribute towards software reliability estimation. This paper provides an insight to all such aspects of software reliability. Initially the paper briefs introduction to Software reliability with a focus on factors affecting software reliability. Software reliability models are classified based on their characteristics and the various software reliability models are discussed here with classification. Different models are applied to different data set based on their growth pattern. The growth pattern is well identified using trend analysis. Various techniques for trend analysis are also discussed in this paper. Several parameters define how much more likely it is that one model will produce accurate estimates than another model. This can be best achieved by parameter estimation for model ranking. Here techniques for parameter estimation are also discussed for model ranking which will provide basis for accurate estimation of software reliability by choosing appropriate model. While many of the reliability growth models are purely empirical, some of the models are based on some specific assumptions about the fault detection/ removal process. The parameters of these models thus have some interpretations and thus possibly may be estimated using empirical relationships using static attributes. Two such parameters for software reliability - Exponential and Logarithmic are further discussed in this paper. Defect density plays an important role in software reliability estimation. A brief insight to defect density also contributes to this paper. Finally the paper concludes with the issues related to software reliability growth problem.

Keywords- Software reliability models, trend analysis, Exponential parameter, logarithmic parameter, model ranking, defect density

Introduction

Developing software is a challenging task. The challenge comes basically from the characteristics of the software itself, and because of these characteristics, the software can be said to prone towards containing faults. Even if know that the software contains faults, we generally do not know their exact identity. Viz., program proving and program testing can give the indication of the existence of the faults. Program proving is formal and mathematical while program testing is more practical and heuristic. The approach taken in program proving is to construct a finite sequence of logical statements, usually the output specification statement, to be proved [1]. Program testing is symbolic or physical execution of a set of test cases with the intent of exposing embedded faults in the program [1].

It is observed that these approaches are not that faultless and therefore a need for metrics for assessing software quality attributes has become significant. One such quantifiable metric of quality that is commonly used is software engineering practice is software reliability.

A number of conflicting views exist as to what software reliability is and how it should be quantified. One of the approaches parallels that of program proving whereby the program is either correct or incorrect. Software reliability in this case is binary in nature; an imperfect program has zero reliability and a perfect program has a reliability value of one. Thus software reliability can be defined as relative frequency of the times that the program performs as intended. This leads us to define software reliability as the probability of fault free operation, provided by software product under consideration, over a specified period of time in specified operational environment [2].

The next section focuses on significance of Software reliability proceeded by definition to software reliability

Uses of Software Reliability

Potential use of software reliability includes the following [2]:

Making intelligent system tradeoffs between reliability, performance, cost, schedules, and other factors for and between the programs themselves and other system elements as well. These tradeoffs will be made both at the start of the project and as it proceeds. They may involve combination of software reliability parameters with those of other system components to obtain system reliability estimates or allocation of system reliability goals among subsystem, one or more subsystems being computer programs.

Scheduling and monitoring progress of a testing effort b using continually updated estimates of current reliability. Included in the foregoing is determining when to terminate a testing effort.

Comparatively evaluating the effect on reliability of different design techniques, coding techniques, testing techniques and documentation approaches.

Looking at the potential benefits of software reliability forty years back efforts started in this area. Although modeling software reliability was very much influenced from that of hardware reliability, the concept of software differs much from that of hardware reliability.

Hardware versus Software Reliability

A fundamental difference in dealing with software versus hardware reliability is that hardware reliability tends to decrease in time, due to aging, wear-out, and so forth, where as software reliability tends to increase through time, due to the removal of bugs, or will remain the same if no action is taken. There are exceptions to both, of course. Hardware reliability may increase in time as a result of a reliability improvement program, and software reliability may decrease as a result of the introduction of new bugs. [Table-1] summaries the difference between hardware and software reliability [3].

Because of the large number of differences, one might conclude that methods developed for hardware would not be appropriate for analysis of software reliability. Many of the techniques used in the analysis of hardware reliability are also useful in software reliability as well, but care must be taken in selecting models and interpreting results in software context.

Table 1- Comparison of Hardware and Software

Hardware	Software
A small anomaly may lead to a predictable failure or have little or no effect.	One incorrect bit may lead to disaster
Design and production predominate.	Nearly 100% design
Can test all events	The number of events is huge and events tend to be unique to software
Cause of failure is design, manufacture, maintenance, and misuse.	Failures are due to design defects.
Redundancy can be used to increase reliability.	Redundancy does not necessarily lead to improvement.
Maintenance improves reliability	Reprogramming may introduce new errors
Physical laws may describe failures.	A comparable law does not exit.
Interface are physical structures	Interfaces are conceptual
Standard parts are commonly used.	Standard parts are seldom used.

Software Reliability

Software Reliability has been defined as the probability that a software fault, which causes deviation from required output by more than specified tolerances in a specified environment, does not occur during a specified exposure period. Thus reliability can be formally defined as:

$$R(i) = P[\text{No failures in i runs}] \quad (1)$$

Or

$$R(t) = P[\text{No failures in interval } (0,t)] \quad (2)$$

Assuming that inputs are selected independently according to some probability distribution function, we have

$$R(i) = [R(1)]\, i = (R)\, I \quad (3)$$

Where R=R (1). We can define the reliability R as follows:

$$R = 1 - \lim (nf/n) \quad (4)$$

Where

n = number of runs,

nf = number of failures in n runs.

This is the operational definition of software reliability.

Software reliability is a function of many factors like software development methodology, validation methods and also the languages in which the program is written.

Failure intensity is an alternative way of expressing reliability. Let R be the reliability, λ the failure intensity and t the execution time. Then as shown in [Fig-1].

$$R(t) = \exp(-\lambda t) \quad (5)$$

The failure intensity statement is more economical since only one number is needed. Failure intensity like reliability is defined with respect to a specified operational profile. The relationship between failure intensity and reliability depends on the reliability model employed if these values are changing.

[Fig-2] shows that as faults are removed, failure intensity tends to drop and the reliability tends to increase.

Software Error, Fault and Failure

The following definitions are commonly used in the software engineering literature [4]:

- Error: Human action which results in software containing a fault.
- Fault: A manifestation of an error in software; a fault if encountered may cause a failure.
- Failure: An unacceptable result produced when fault is encountered.

Even though these three things have different meanings, they are often used interchangeably in the literature.

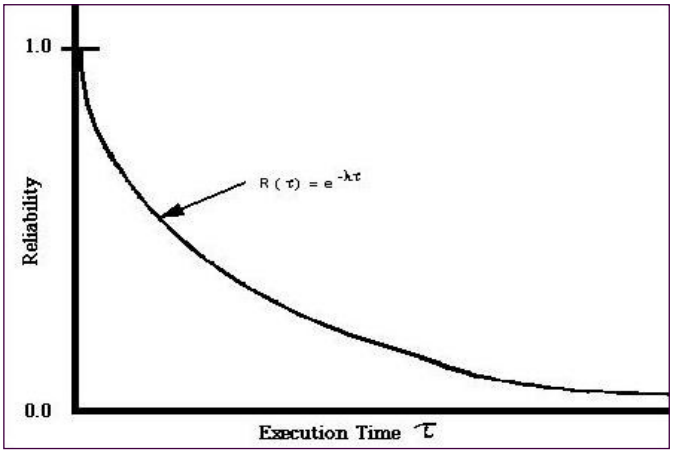

Fig. 1- Reliability Function in Relation to Failure Intensity

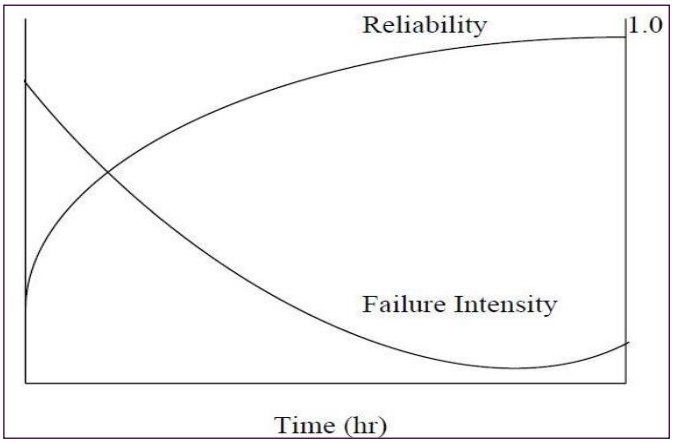

Fig. 2- Decreases in Failure Intensity Increases Reliability

Time Relevant to Software Reliability

Three kinds of time are relevant to software reliability:

- The execution time for the program is the time required by a processor to execute the instructions of the program.
- Calendar time is the regular time we are familiar with.
- Clock time, used occasionally, represents the elapsed time from start to end of the program execution on a running computer. It includes wait time and execution time of other programs.

Factors Influencing Software Reliability

The main factors that affect software reliability are fault introduction, fault removal, and operational profile. Fault introduction depends primarily on the characteristics of the developed code (code written or modified for the program) and the development process. The code characteristics with the greatest effect are size. Development characteristics include the software engineering technologies and tools employed and the average level of experience of programmers. Note that code is developed when adding features or removing faults. Time, the operational profile and the quality of the repair activity affect fault removal.

The reliability of a software based product depends on how the computer and other external elements will use it [5]. Making a good reliability estimate depends on testing the product as if it were in the field. The operational profile, a quantitative characterization of how the software will be used, is therefore essential in any software reliability engineering (SRE) application. It is the fundamental concept which must be understood in order to apply SRE effectively and with any degree of validity.

A profile is a set of independent possibilities called elements, and their associated probability of occurrence. If operation A occurs 60 percent of time, B occurs 30 percent, and C occurs 10 percent, for example the profile is [A, 0.6…B, 0.3…C, 0.1]. The OP is the set of independent operations that a software system performs and their associated probabilities. Developing an OP for a system involves one or more of the following five steps [Fig-3]:

- Find the customer profile
- Establish the user profile
- Define the system mode profile
- Determine the functional profile
- Determine the operational profile

Since most of the foregoing factors are probabilistic in nature and operate over time, software reliability models are generally developed as models of random processes.

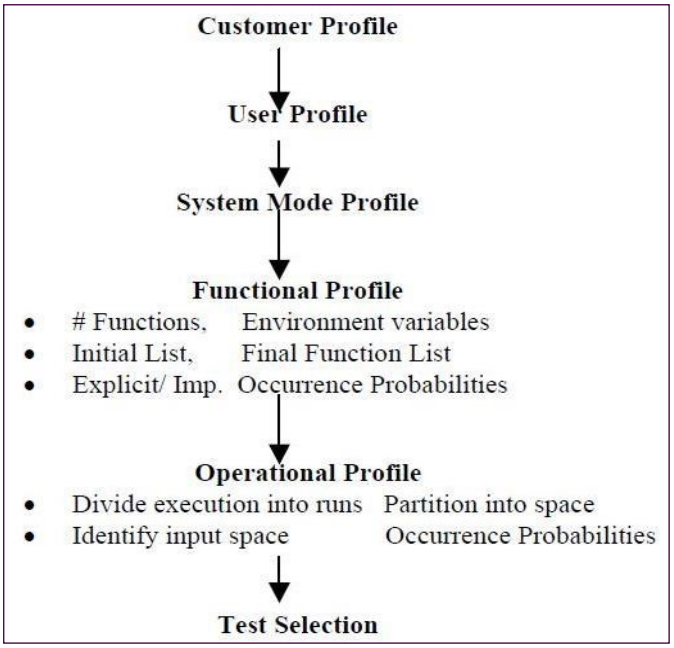

Fig. 3- Processes for Developing Operational Profile

The model are distinguished from each other by probability distribution of failure times or numbers of failure experienced and by the form of the variation of the random process with time. Since the failure rate of the software changes over time as the software is modified to correct faults, the prediction procedure like the one shown in [Fig-4] provide values for the parameters of a software reliability growth model. A reliability growth model can be used to forecast what the failure rate λ(τ) will be at any time τ into the system test.

Conversely, a growth model can be used to forecast when a particular failure rate objective will be reached. The amount of execution time to reach an objective can be translated into calendar time for schedule and resource estimate. Reliability progress is evaluated as a part of prediction process.

The next section focuses on software reliability models by first providing the overview of the software reliability models and then by classifying them under appropriate heads.

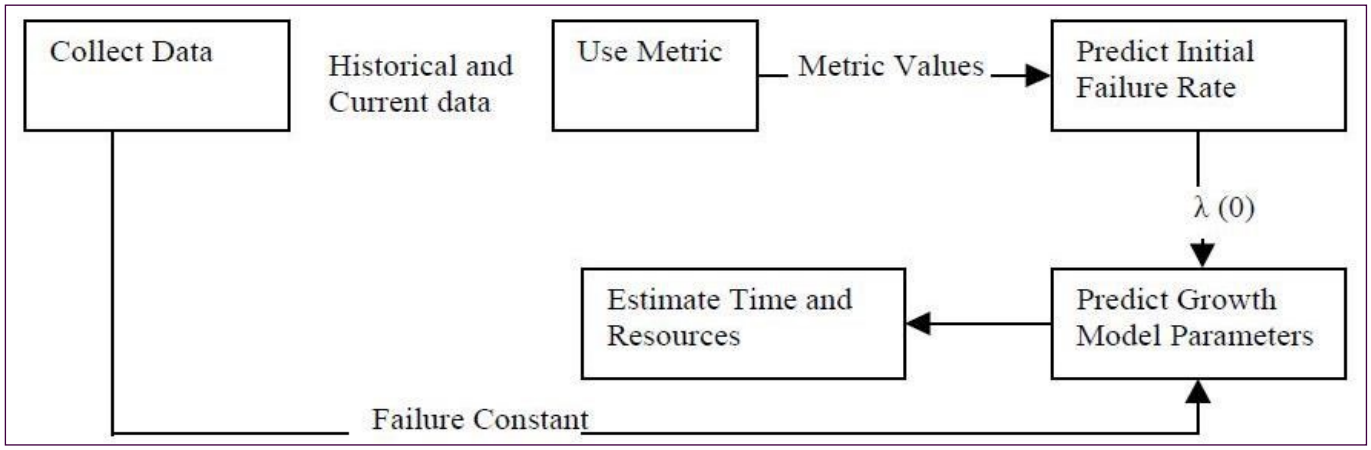

Fig. 4- Software Reliability Predictions

Software Reliability Models

Models have been developed to measure estimate and predict the reliability of computer software. Software reliability has received much attention because reliability has always had obvious effect on highly visible aspects of software development: testing prior to delivering, and maintains.

A model is an abstracted representation of the relationship among two or more variable attributes of an entity. A good model will incorporate the influence of all variables that affects the outcome. A useful model will have predictive capabilities given the values of some attributes we know, or can estimate reasonable well, it will determine the future values of other attributes with an acceptable degree of uncertainty.

Reliability models are mathematically intense incorporating stochastic processes, Probability and statistics in their calculation and relying on maximum likelihood estimate, numerical methods and confidence interval to model their assumptions. They express the general form of the relationship between the failures and the factors mentioned. The possibilities for different general mathematical forms to describe the failure process are almost limitless.

The specific form is determined by establishing the values of the parameters of the model through either:

Estimation: applying statistical inference procedures to the failure data, or

Prediction: relating parameter value to characteristics of the software product and the development process (Which can be done before failure data is available).

There will be uncertainty in the determination of specific form. This is generally expressed in terms of "confidence interval" for the parameters. A "confidence interval" represents a range of value within which a parameter is expected to lie with a certain confidence. For example, the 75% confidence interval of initial failure intensity may be 9 to 11 failures per hour. After the specific form has been established, many different properties of the failure process can be determined. For many models there is analytic expression for

The average number of failures experienced at any point in time,

The average number time failures in a time interval,

The failure intensity at any point in time, and

The probability distribution of failure intervals.

A good software reliability model has several important characteristics. It

- gives good prediction of future failure behavior,
- computes useful quantities,
- is simple
- is widely applicable,
- is based on sound assumptions.

However software reliability models must take the following properties of software error into account:

- The same program can have different reliability in different environments depending upon which portion of the code gets executed more often.
- There is no physical deterioration of the software.
- The software errors are usually correlated.
- New errors can be introduced during the correction of the previous error.
- Software reliability is dependent on the experience and educational level of the developer.

At present, almost all the software reliability models make simplifying assumptions that may not hold in practice. It is therefore important to check the validity of the assumptions made by the models before applying them in practice.

A set of desirable features for software reliability models was developed by Bastani [6] which includes language independent, methodology independent, test case selection criteria, correction errors, representativeness, input distribution, program complexity, model validation, time and use of data collection.

History of Software Reliability Model

The first known study of software reliability by Hudson in 1967 [7] viewed software development as birth-and –death process (a type of Markov process). Fault generation (through design changes, faults created in fixing other faults, etc) was a birth, and fault correction was a death. The number of faults remaining defined the Process State. The transition probabilities characterized the birth and death function. Hudson's analysis, limited to pure death process for reasons of mathematical tractability, assumed that the rate of detection to faults was proportional to the number of faults remaining and a positive power of the time.

Jelinski and Moranda [8] and Shooman [9] in 1972 made the next major steps. Both assumed a piecewise-constant hazard rate of failures that was proportional to the data of faults remaining. Moranda model that accounts for imperfect debugging. The view of debugging taken here is that of a Markov process, with appropriate transaction probabilities between states. Several useful quantities can be derived analytically, with the mathematics remaining tractable. Kremer in 1983 developed this idea further, including the possibility of spawning new faults due to the repair activity.

Goel and Okumoto [10] in 1979 depicted software failure as a non-homogeneous Poisson process with a exponentially decaying rate function. Maximum-likelihood methods for estimating the parameters were developed for two different situations; intervals between failures and per intervals a simple modification of this models was investigated by Yamada et al.[11] in 1983, where the cumulative number of failures detected is described as an S-shaped curve.

Crow [12] in 1984 proposed that a hardware model based on non-homogenous Poisson process with a failure intensity function i. e. a power function in time can be applied to software, using appropriate ranges of parameter values.

Thus, much of early history of software reliability modeling consisted of looking at different possible models. In the late 1970's and early 1980s, efforts started to focus on comparing software reliability models, with the objective of selecting the "best" one or ones. Initial efforts at comparison were made by Sukert, in 1979; and Schick and Wolvetron [13] in 1973 which suffered from lack of good failure data and lack of agreement on the criteria to be used in making the comparison. The publication by Musa of reasonably good-quality set of data stimulated comparison efforts. Lannino and other in 1984 worked out a consensus on the compression criteria to be employed. Examination on the basic concepts underlying software reliability modeling and development of classification scheme helped clarify and organized comparison and suggested possible new models. This efforts led to the development of the Musa and Okumoto logarithmic Poisson execution-time model in 1984 [14].

Ohba [15] in 1984 put forth the hyper exponential growth model based on the assumption that a program has a number of clusters of modules, each having a different initial number of errors and a different failure rate. Yamada and Osaki suggested a similar extension of the exponential growth model in 1985.

A major work was performed related to NHPP S-shaped model in the year 1983. Yamada proposed delayed S-shaped model and Ohba proposed the inflection S-shaped model [16]. Pham further enhanced this work in 1997.

Pham [17] addressed the problems of multiple failure types and imperfect debugging based on NHPP for predicting software performance measure. Empirical evidences suggests that the higher the test coverage, the higher would be the reliability of the software product. Studies regarding test coverage and reliability were extensively worked out during the period of 1993 to 1996 by Wong [18], Chen [19], and Piwowarski [20], Malaiya [21], Frate [22], Jacoby and Masuzawa [23]. However most of the research till then focused on confirming the intuitive relationship between coverage and reliability, or enhancing a particular model to incorporate code coverage, for a specific experiment. They did not represent a framework, which will allow coverage measures to be accounted for, in a generalized fashion.

In 1996 Gokhale, Trivedi, Philip and Marinos [24] proposed an Enhanced Non Homogeneous Poisson Process model (ENHPP) as a unifying framework for finite failure NHPP models. It explicitly incorporates time varying test coverage.

In 1998 Rivers and Vouk [25] derived a hypergeometric model for the number of failures experienced at each stage of testing when the constructs tested are removed from the population. i.e. when there is no reset of the constructs covered before.

Thus, for about 25 years, software reliability modeling has been active process arena in software engineering environment and an attractive subject for technical publications in professional journals, trade magazines and SE symposium. Kan [26] has identified over 100 software reliability models and Neufelder who is the instructor for software reliability at Reliability analysis center has identified over seven hundred models that can generally be classified as software reliability models.

Classification of Software Reliability Models

The Software Reliability models can be classified as shown in [Fig-5]. Generally, a mathematical model based on stochastic and statistics theories is useful to describe the software fault-detection phenomena or the software failure-occurrence phenomena and estimate the software reliability quantitatively. During testing phase in the software development process, software faults are detected and removed with a lot of testing effort expenditures. Then, the number of faults remaining in the software system is decreases as the testing goes on. This means that the probability of software failure-occurrence decreases as the software reliability is increases and the time-interval between software failures becomes longer with the testing time.

The first classification into static and dynamic models reflects whether the reliability estimation is independent of time or has time base prediction capacity. In former case, reliability estimation is for a fixed point while in later, prediction into future is made based on stochastic model for the fault discovery history. The former models are useful only for estimation while the latter can be used both for estimation and prediction [4].

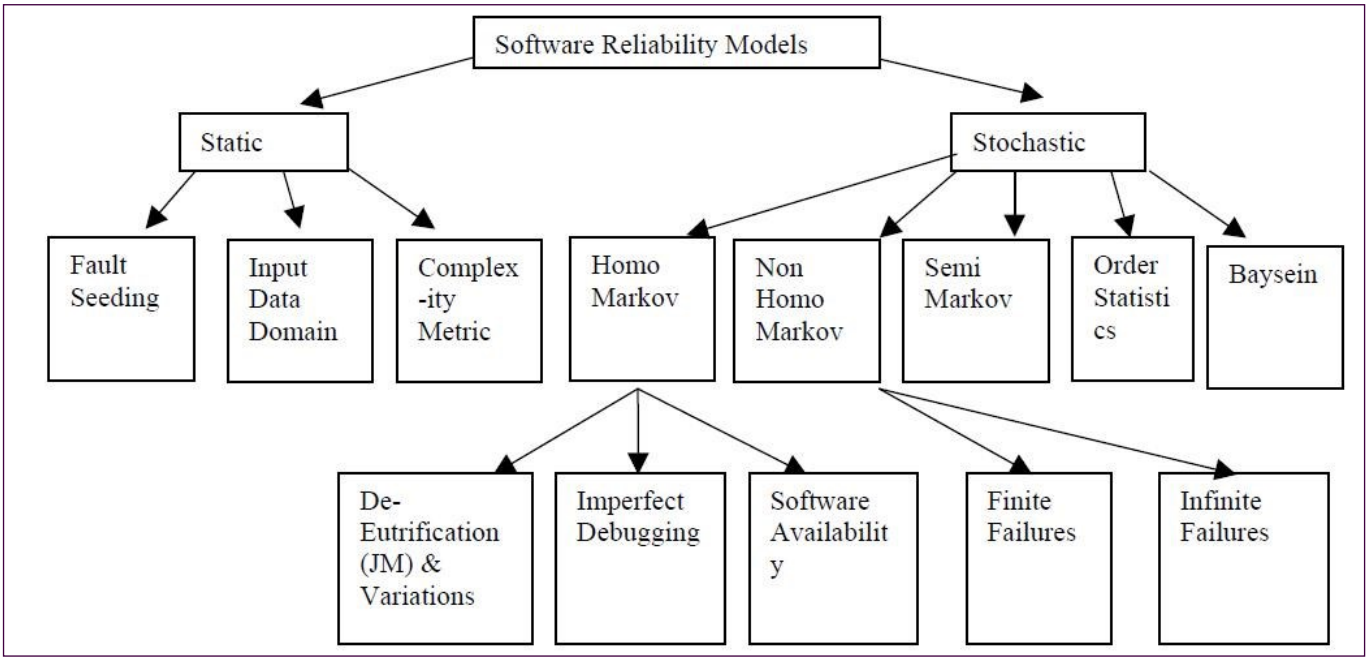

Fig. 5- Classification of Software Reliability Models

Static Models

Fault Seeding Models

The basic approach in this class of model is to "seed" a known number of faults in a program which is assumed to have an unknown number of indigenous faults. The program is tested and the observed number of seeded and indigenous faults is counted. From these, an estimate of the fault content of the program prior to seeding is obtained and used to assess software reliability and other relevant measures.

Mills' Hyper Geometric Model

The most popular and most basic seeding model is the Mill's hypergeometric model [27]. Assumptions:

- A known number of faults are planted in a program.
- Based on standard capture and recapture technique.
- Using combinatorics and maximum likelihood technique, the number of indigenous faults is estimated and the reliability of the software is then computed.

The procedure adopted in this model is similar to the one used for estimating population of fish in a pond or estimating wildlife. These models are also referred to as tagging models since a given fault is tagged as seeded or indigenous.

Lipow [28] modified this problem by taking into consideration the probability of finding a fault, of either kind, in any test of the software. Then, for statistically independent tests, the probability of finding given number of indigenous and seeded faults can be calculated. In another modification, Basin [29] suggested a two stage procedure with the use of two programmers which can be used to estimate the number of indigenous faults in the program.

Input Data Domain

In this reliability of the software is measured by exercising the software with a set of randomly chosen inputs. The ratio of inputs that resulted in successful execution to the total model of inputs gives an estimate of the reliability of the software product. Example: Nelson's model [30].

Nelson's Model

Assumptions:

The software is run for a set of n inputs chosen randomly from a set {Ei:i=1,2,3…N}

The random sampling of n inputs is done according to the probability vector p, which defines the operational profile

$$R = 1-(f/n) = (n-f)/n \quad (6)$$

is the estimate of reliability. Here the generation of test cases 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 usually partitioned into a set of equivalence class.

Complexity Metric

The complexity metric models employ a statistical model such as regression equation or principal component regression to estimate the number of faults or reliability as the function of relevant complexity metric [31,32].

Stochastic Models

Homo Markov

The models belonging to this category assume that the initial number of faults in the software product under consideration is unknown but fixed. The number of faults in the system, at any time, forms the state space of homogeneous Markov chain. The failure intensity of the software or the transition rates of the Markov chain depend upon the number of residual faults in the software.

Example: Jelinski-Moranda [8], Goel Okumoto imperfect Debugging model [33]

De-Eutrophication

Jelinski-Moranda

The JM model is based on the following assumptions:

- The initial fault content of the tested software is an unknown fixed constant.
- The failure rate is proportional to the current fault content of the tested software, and remains constant between failures.
- All remaining software faults contribute the same amount to the failure rate.
- A detected fault is corrected and removed immediately.
- No new fault is introduced after a fault is corrected.

The expressions regarding Jelinski-Moranda [8] are

$$\mu(t) = No\,(1- \exp(-\varphi t)) \quad (7)$$

$$\lambda(t) = \varphi(No - \mu(t) \quad (8)$$

Where No is the initial number of faults and φ is the contribution of each fault to overall failure rate.

Imperfect Debugging

As a relaxation to assumption 5 of the JM model Goel and Okumoto proposed an imperfect debugging model[33][10] in which each detected fault is removed with the probability p, or remains in the software with the probability q = p-1.

In this model the number of faults in the system at time t, X(t) is related as a Markov process whose transition probabilitites are governed by the probability of imperfect debugging. Time between the transitions of X(t) are taken to be exponentially distributed with rates dependent on the current fault content of the system. The hazard function during the interval between the (i-1) and the i failures is given by

$$Z(ti)= [N-p(i-1)]\lambda \quad (9)$$

Where N is the initial fault content of the system, p is the probability of imperfect debugging and λ is the failure rate per fault.

Software Availability

These models are used for estimating and predicting software reliability and availability. In this model the system states are divided into distinct up and down states according to the number of faults remaining in the software weather the software is operating or not.

Trivedi and Shooman Model [2]

Assumptions:

- Assume that the software is in an up state at time t=0. When a failure occurs, the system is shut down and enters a down state. The fault which caused the failure is then detected and removed before the system begins to operate again.
- The operating time and the repair time are both assumed to be random variables described by failure rate

$$\varphi(No-i) \quad (10)$$

and repair rate

$$\mu(No-i) \quad (11)$$

where No is the total number of faults in the software and i is the no. of failures that have been occurred.

Non-Homo Markov

These models assume number of faults present in a software product to be a random variable most often assumed to display the behavior of a non-homogeneous Poisson process.

Finite Failures

They assume that the expected number of failure observed during an infinite amount of time will be a finite failure.

Example: Goel Okumoto NHPP model [1], Delayed S- Shaped NHPP model [15], Inflection S- shaped model [15], C1 NHPP model [15], pareto NHPP model [15], Little wood NHPP model [34],ENHPP [24].

Musa Basic Execution Model

Assumptions:

- The number of failures that can be experienced in finite time is infinite
- The distribution of the number of failures observed by time t is of Poisson type.
- The functional form of the failure intensity in terms of time is exponential.

Mean value function is given by

$$\mu(t) = \beta o(1-e-\beta 1t) \quad (12)$$

Failure intensity is given by

$$N(t) = \beta o\beta 1\ e-\beta 1t \quad (13)$$

Here the fault exposure ratio assumed is constant over time.

An interesting extension of this model is the logarithmic Poisson execution time model, where the expected number of failures is a Poisson random variable and a logarithmic function of the CPU time τ, a factor determining decay in failure intensity.

The Enhanced NHPP Model

The enhanced NHPP (ENHPP) model [24] is a unifying framework for finite failure NHPP models i.e. other NHPP models with bounded mean-value functions are special cases of the ENHPP model. The model explicitly incorporates time-varying test coverage and imperfect fault detection in its analytical formulation. Test coverage in this model is defined as the ratio of the number of potential fault sites sensitized by a test to the total number of potential fault sites. Potential fault sites refer to "the program entities representing either structural or functional program elements whose sensitization is deemed essential towards establishing the operational integrity of the software product" [24].

- The model makes the following assumptions:
- Faults are uniformly distributed over all potential fault sites. The probability of detecting a fault when a fault site is sensitized at time t is cd(t) = K, (a constant), the fault detection coverage.
- Faults are fixed perfectly.

The mean value function for this model is developed as

$$(t) = c(t)N \quad (14)$$

Where c(t) is the time variant test coverage function and N is number of faults expected to have been exposed at full coverage. This is distinguished from N, which is the expected number of faults to be detected after infinite testing time, perfect test and fault detection coverage. The failure intensity for this model then becomes

$$\lambda(t) = z(t)(N-\mu(t)) \quad (15)$$

Where z(t) = c'(t).(1 – c(t)) – 1 is the time variant per-fault hazard rate. The model allows the scenario of defective coverage to be incorporated in the reliability estimation. Different coverage function distributions result in the variations of the NHPP models i.e. the G-O model, the Yamada S-shaped model, etc. Reliability as obtained from this model is expressed as

$$R(t/s) = e – NK(c(s + t) – c(s)) \quad (16)$$

Where s is the time of last failure and t is the time measured from last failure. Grottke [35] observes correctly, that the main merit of this model is to serve as a unifying framework for NHPP models. Further, the dependence of the per-fault hazard rate solely on time-variant test coverage neglects other influencing factors such as the fact that full test coverage may not be successful in detecting all the faults and that failures may still occur without any gain in test coverage.

Infinite Failures

The mean value function of this class of models is unbounded, i.e., the expected number of failures experienced in infinite time is infinite.

Example: Musa Logarithmic models [2], Duane model [36].

Duane Model

The Duane model was originally proposed for hardware reliability studies. Its mean value function, m(t), and the failure intensity function, λ(t), are given by

$$m(t) = atb,\ b>0 \quad (17)$$

and

$$\lambda(t) = abt\ b-1\ b>0 \quad (18)$$

This model usually overestimates the cumulative number of failures. The main criticism of the model is that its mean value function approaches infinity very rapidly.

Semi Markov

Here the number of faults remaining in the software is modeled using a semi Markov process. e.g. Schick and Wolvetron Model [13]

Schick and Wolvetron Model

The model is applied to specify the number of residual errors, the mean time to detect the next error, the error detection rate.

The model is based on the following assumptions:

- Errors occur by accident.
- The error detection rate in the defined time intervals is constant.
- Errors are independent of each other.
- No new errors are generated.
- Errors are corrected after they have been detected.

In this model it is assumed that the error detection rate is proportional to the number of residual errors and the time passed since the detection of the preceding error. The transition rate, λ_I, during the test interval t_I, is assumed to be proportional to the current fault content of the system, and time lapsed since the last failure and is given by

$$\lambda_I = \varphi[N- (I-1)]t_I \quad (19)$$

Order Statistics

This model originated from the study of hardware reliability. Cozzolino [30] presented a model, which he called the initial defect model for a repairable hardware system.

Cozzolino

Assumptions:

- Each new system has an unknown Poisson distributed number called initial defect.
- Each defect independently has a constant failure rate.
- When a failure occurs, the defect causing it will be discovered and repaired perfectly that is the defect will never reappear.
- The time to repair is negligible.

Baysein

In this, the model parameters are assumed to be random variables with some prior distributions. Based on failures observed, the posterior distributions of the model parameters are derived using Bayes theorem. These posterior distributions, together with other model assumptions, are then used to predict various reliability measures. Example: Littlewood Verall model [37].

Littlewood Verall Model

The models presented in the previous sections all assume that failure data is available. They also apply classical statistical techniques like maximum likelihood estimation (MLE) where model parameters are fixed but unknown and are estimated from the available data. The drawback of such an approach is that model parameters cannot be estimated when failure data is unavailable. Even when few data are available, MLE techniques are not trustworthy since they can result in unstable or incorrect estimations.

The bayesian SRGM considers reliability growth in the context of both the number of faults that have been detected and the failure-free operation. Further, in the absence of failure data, bayesian models consider that the model parameters have a prior distribution, which reflects judgment on the unknown data based on history *e.g.* a prior version and perhaps expert opinion about the software.

The Littlewood-Verrall model is one example of a bayesian SRGM that assumes that times between failures are independent exponential random variables with a parameter ξi, $i = 1,2, \ldots, n$ which itself has parameters $\psi(i)$ and α reflecting programmer quality and task difficulty having a prior gamma distribution. The failure intensity as obtained from the model using a linear form $\psi(i)$ function is

$$\lambda(t)=(\alpha-1)(N^2+2B\varphi(\alpha-1))^{-1/2} \qquad (20)$$

Where *B* represents the fault reduction factor, as in Musa's basic execution time model. This model requires tune between failure occurrences to obtain the posterior distribution from the prior distribution.

Other Reliability Models

There are several other models which has come up as the developmental approaches are changing up. Reliability Focused Quality model for Object oriented Design [38] is used to predict reliability of object oriented software in the early design phase. The concept of user-oriented reliability and user profile is presented by [39]. The reliability of a system is expresses as a function of the reliability of its components and the user profile. A Markov model is developed under the assumption of module reliability and the Markovian behavior of control transfer among module. The potential application of this module is for reliability estimation, testing strategy, maintenance philosophy and estimation of penalty cost.

[40] Suggested two new reliability models considering software and hardware faults as root causes of software failures for embedded software reliability estimation. The experimental results show that a Weibull based model, which takes characteristics of hardware degradation into account, has higher fitting-adequacy and superior accuracy for software reliability estimation.

In software industry, up gradations are made in the software at a very brisk speed. To capture the effect of faults generated in the software due to add-ons at various point in time [41] developed a multi up gradation, multi release software reliability model.

[42] Presented a prediction model of software reliability based on the modular. Here Markov analysis theory is applied for software reliability prediction based on module reliability and module importance in order to achieve the requirement of improving software quality.

[43] Proposes a multi-factor software reliability model based on logistic regression and its effective statistical parameter estimation method. The proposed parameter estimation algorithm is composed of the algorithm used in the logistic regression and the EM (expectation-maximization) algorithm for discrete time software reliability models. The multi-factor model deals with the metrics observed in testing phase (testing environmental factors), such as test coverage and the number of test workers, to predict the number of residual faults and other reliability measures.

As we have seen there are various models for software reliability, we find that the basic approach of modeling is to fit past data to a model data that describes the expected behavior of the data, and to use the model to predict future behavior. In order to provide good predictions, the model should accurately describe the observed process or at least be close to it. Predictions obtained using models that do not fit the observed data can be misleading. Therefore selecting the proper model is probably the most important part of modeling and one can use trend analysis for that purpose.

Trend Analysis

The task of verification process is to increase the reliability of the tested design or, in other words, to decrease the failure intensity of the design. This is done by detecting and removing faults that cause failures from the design. Therefore the first indication that can be obtained from the statistical analysis is whether there is a reliability growth in the tested design. This type of information can be provided by trend analysis.

Reliability growth means that the fault discovery intensity in the design is decreasing, and therefore the time between fault discoveries is increasing. This trend can be detected by looking at the cumulative number of fault function of the testing time. Since the failure intensity is the derivative of the cumulative number of faults, decreasing failure intensity means that the function of the cumulative number of faults vs. simulation time should be concave.

Taking a look at the raw data provides only a rough indication of reliability growth for two main reasons:

- The raw data of time between failures is only an instantiation of a random process. Therefore, the cumulative number of faults is not a smooth concave function, but with a function with many bumps and extreme points.

- The raw data information can very easily hide local changes in the trend of the reliability growth or strong local variations can hide the overall trend of reliability growth.

There are several analytical tests for reliability growth that are designed to overcome these problems. The idea behind these tests, known as trend tests, is to test a null hypothesis H0 versus an alternative H1. Usually H0 corresponds to the assumption that the reliability is not changing, while H1 corresponds to the assumption that reliability undergoes a monotonic trend.

The trend test is run against the failure history data to see if it exhibits reliability growth, decay followed by growth or stable reliability. Based on this information one can decide which reliability growth model to use [44]. The two trend tests that are commonly carried out are:

- Running Average
- Laplace test

Running Average

This test consists of computing the running average of the time between successive failures for time between failure data, or running average of number of failures per interval for failure count data. For time between failures, if the running average increases with failure number, this increases reliability growth.

The arithmetic mean T(i) of the observed interfailure times t_j, j = 1,2, …..i;

$$T(i) = 1/i \; \Sigma \; t_j \qquad (21)$$

An increasing sequence of T (i) indicates reliability growth and a decreasing sequence indicate reliability decay. For failure count data, if the running average decreases with time, reliability growth is indicated. [Fig-6] shows running average analysis using time between failure data (Sample data used).

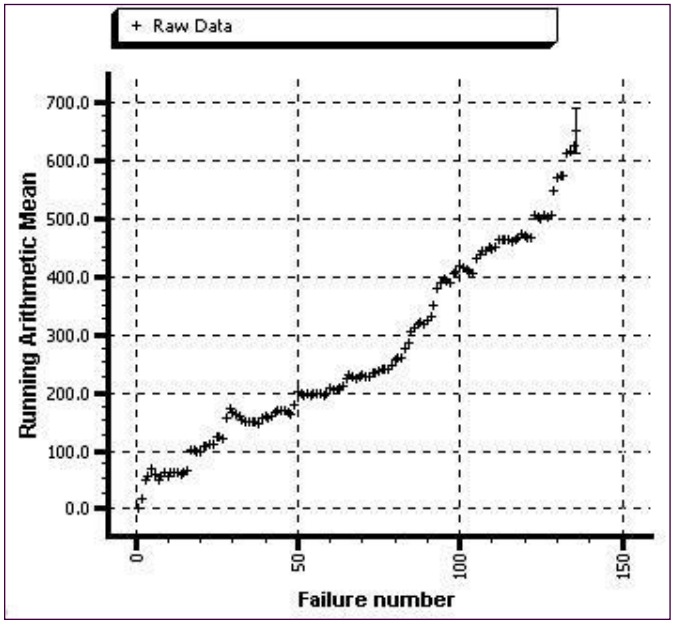

Fig. 6- Running Average Analyses for Time Between Failure Data

Now the s1 data set is converted type from time between failure to failure count data. To convert inter failure time data to failure frequency data the desired length of testing interval is specified to be 500 seconds. The running arithmetic mean for s1 data set is as shown in the [Fig-7] which shows decrease with the decrease in test interval number with time and hence reliability growth.

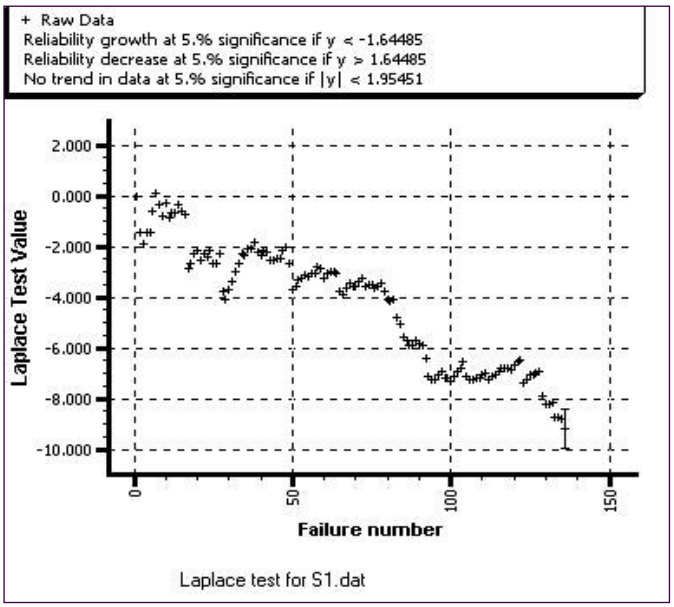

Fig. 7- Running Average Analysis For Failure Counts Data (S1 Data Set)

Laplace Test

Laplace test is superior from an optimality point of view and is recommended for use when the NHPP assumption is made [45]. The test procedure is to compute Laplace factor λ(t) given by [6]

The Laplace factor is evaluated step by step, after every failure occurrence. Here t is then equal to the time of occurrence of the ith failure, and the failure at time t is excluded.

Intuitively Laplace factor can be interpreted as follows:

- Negative values indicate a decreasing failure intensity, and thus reliability growth.
- Positive values indicate increasing failure intensity and thus a decre4ase in the reliability.
- Values between –2 and +2 indicate stable reliability.

[Fig-8] demonstrates the Laplace test for the s1 data set for the reliability growth at 5% significance.

Trend Analysis can provide information about trends that occurred during the verification process, and answer important questions, such as: does the reliability of the tested design grow, and how do specific events affect this growth? Still trend analysis cannot answer questions like: how many failures are left in the tested design, and when will be the desired level of reliability achieved? To answer such questions, predictions of the future behavior of the tested design and the verification process are needed. Usually, predictions of the future behavior of a process to some predefined model (or models) and using the properties of the model to describe predicted future behavior. Trend analysis can significantly help in choosing the appropriate model for a given sequence of inter failure times, so that they can be applied to data displaying trends in accordance with their assumptions rather than blindly. Using a model for the analysis of failure data set, without taking into consideration the trend displayed is different than assumed in the model. Also there are multiple models that can be use to fit the given set of data with more or less capability [44].

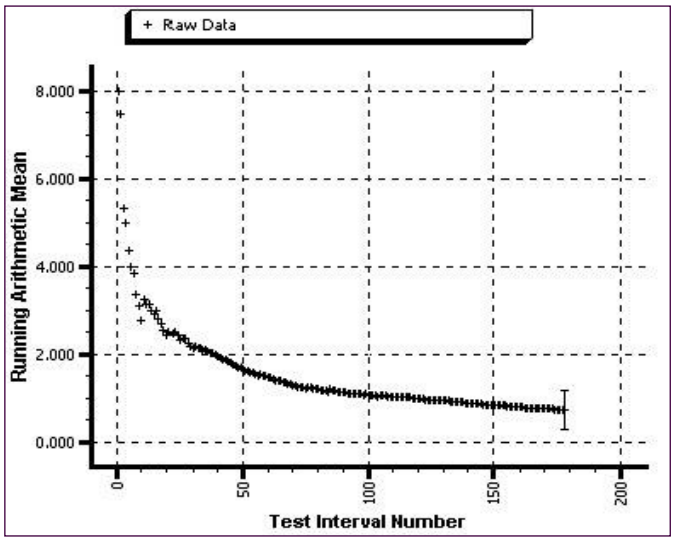

Fig. 8- Laplace Test

The predictive capabilities of the various models can be judged by estimating the following parameters for each of these models. Considering the Laplace test and running average test for the s1 data a increase in the software reliability growth is observed. Base on this result three models for the further demonstration of the parameter estimation are selected. The models used are:

- Jelinski Moranda Model
- Musa Basic
- Musa Okumoto

Parameter Estimation for Model Ranking

Prequential Likelihood

The parameter estimates and the actual observed failure times are used in the function to compute a value that can be used to determine how much more likely it is that one model will produce accurate estimates than another model. This likelihood is given by values of the ratio of Prequential likelihood for the two models being compared. [Fig-9] shows the Prequential likelihood for the three models.

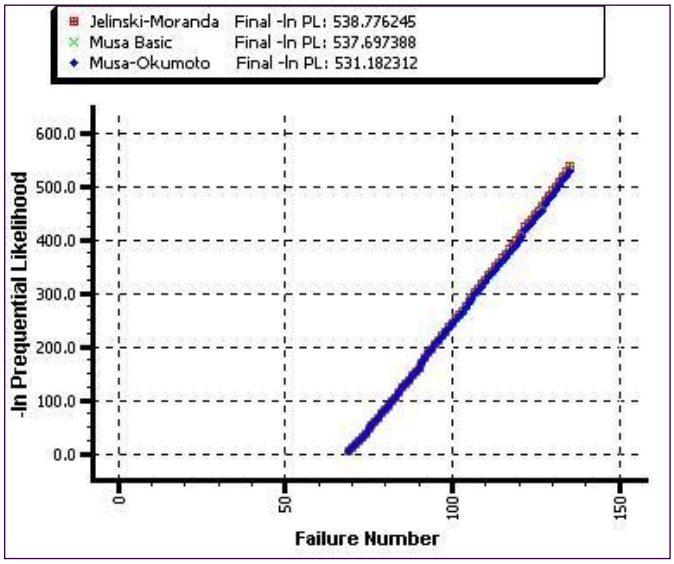

Fig. 9- Prequential Likelihood

Model Bias

Model bias determines whether the predictions are on an average close to the true distribution. The estimated probability of failure for each failure interval is used to determine the extent to which a model introduces bias into its predictions. If the model is optimistically biased, the estimates of time to net failure are higher than what is actually observed where as in case of pessimistic bias the estimates of times to next failure are lower than the observed one. Model biases are determined by u-plot [46], [Fig-10].

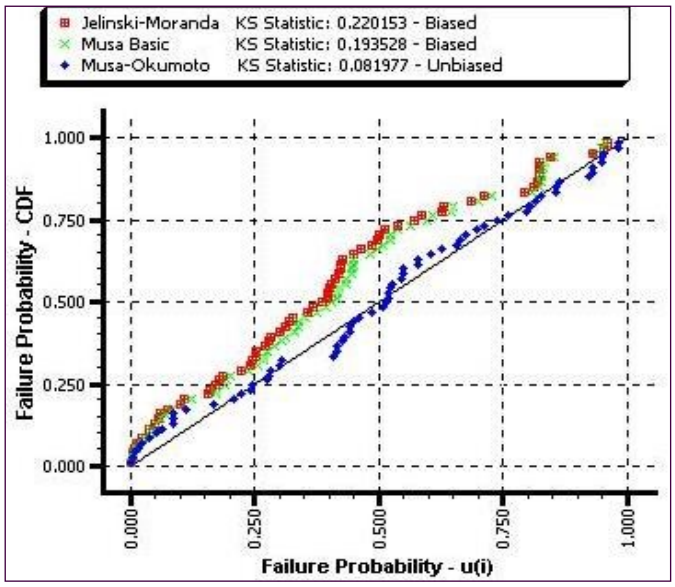

Fig. 10- Model Bias

Model Noise

This gives an indication of how much noise the model itself introduced into the predictions. In general, the higher this figure is, the less likely the model is to give accurate predictions. [Table-2] for model noise of three software reliability models.

Table 2- Model Noise

Model Name	Model Noise
Jelinski-Morando	5.58E+00
Musa Basic	5.35E+00
Musa -Okumoto	2.31E+00

Goodness of Fit Test

The ability of a model to reproduce the observed failure behavior of the software, also known as its retrodictive capability, is determined by the goodness of fit test. The observed failure data is used to estimate the parameters of the model. The estimated mean value function is computed and plotted along with the observed mean value function. The error sum of squares is then calculated to evaluate the goodness of fit. The lower is the error sum of square the better is the fit [Table-3].

Based on the above four parameter the models are ranked as per their predictive capabilities. [Table-4] gives the ranking details and [Table-5] shows the ranks given to these models.

Table 3- Goodness of fit test

Model Name	KS Distance	5.%Fit?
Jelinski-Morando	9.44E-02	Yes
Musa Basic	9.20E-02	Yes
Musa -Okumoto	8.79E-02	Yes

Table 4- Ranking Details

Model Name	-ln	Model Bias	Bias Trend	Model Noise
Musa -Okumoto	5.311823E+002[1]	8.197740E-002[1]	1.056971E-001[1]	2.307074E+000[1]
Musa Basic	5.376974E+002[2]	1.935277E-001[2]	1.556185E-001[3]	5.351962E+000[2]
Jelinski-Morando	5.387762E+002[3]	2.201526E-001[3]	1.542180E-001[2]	5.582339E+000[3]

Table 5- Model Ranking

Model Name	Rank	5.E+003 Sec Reliability after failure 136
Musa -Okumoto	1	1.04E-01
Musa Basic	2	3.39E-01
Jelinski-Morando	3	3.59E-01

Thus for the same set of data when the above two models are applied we find that Musa Okumoto model gives the better reliability prediction.

While considering software reliability models there are various aspects which needs be considered from various points of view for estimating and predicting software reliability. Next we concentrate on the details of these aspects and their impact on software reliability.

Parameters Affecting Software Reliability

Exponential Model parameters

While many of the reliability growth models are purely empirical, some of the models are based on some specific assumptions about the fault detection/ removal process. The parameters of these models thus have some interpretations and thus possibly may be estimated using empirical relationships using static attributes. The two parameters of the exponential model are the easiest to explain. Using this model the expected number of faults μ(t) detected in a duration t may be expressed as

$$\mu(t) = \beta_0^E (1 - e^{-\beta_0^E t}) \qquad (22)$$

Here β_0^E represents the total number of faults that would be eventually detected β_1^E and is the per fault hazard rate which is assumed to be constant for exponential model. The data collected by Musa shows that the number of additional faults introduced during the debugging process is only about 5% [47]. Thus β_0^E may be estimated as the initial number of faults.

The estimation of the other parameter β_1^E is more complex. Musa et al. have defined a parameter K, called fault exposure ratio (FER), which can be obtained by normalizing the per-fault hazard rate with respect to the linear execution frequency, which is the ratio of the instruction execution rate and the software size. For 13 software systems [48] it was found that the overall FER varies from 1.41 x 10^{-7} to 1 0.6 x 10^{-7}, with the average value equal to 4.2 x 10^{-7} failure/ fault. Once we know the value of *K,* β_1^E can be estimated using,

$$\beta_0^E = \frac{K}{T_L} \qquad (23)$$

Where T_L is the linear execution time, given by

$T_L = (I_s Q_r)$ 1/r I_s, is the number of source lines of code; Q_r is the average object instructions per source statement; r is the CPU instruction execution rate. If *N (t)* is the total number of defects still present at time t, it can be shown that [25],

$$\frac{dN(t)}{dt} = -\frac{K}{T_L} N(t) \qquad (24)$$

Thus the defect finding rate –dN(t)/ dt is proportional to the fault exposure ratio. Note that in the above equation, the effect of the software size and the instruction execution rate of the CPU has been taken into account separately in the term T_L. To characterize the defect removal process accurately, one need to identify the factors that control FER.

Once it has been done, one can empirically estimate FER and hence the failure intensity and reliability.

Logarithmic Model Parameters

Exponential model can be considered as an approximation of Logarithmic model [48]. Logarithmic model is found to have very good predictive capability in many cases. According to [48] the β0 and β1 parameters can be estimated using the logarithmic model, which is further designated as β_0^L and β_1^L.

Therefore

$$\beta_0^L = D_{min} * I_s \qquad (25)$$

and

$$\beta_1^L = (K_{min} / T_L) * e^{((D0 - Dmin) / Dmin)} \qquad (26)$$

Fault Exposure Ratio (FER (K))

FER plays an important role in software reliability growth. A manager can use it to plan the test resources need to achieve the desired quality level, even before testing begins. In the early stages of testing, only a limited number of data points are available, which are not often enough to establish the long-term trend. This makes parameter estimation for SRGMs unstable. Identifying what factors affect K is of considerable significance. If we can accurately model the behavior of K, there are three ways in which the software reliability engineering will be affected [49].

- When the process parameters are known a *priori,* optimal resource allocation can be done even before testing begins. Early planning can be crucial to the success of the project.
- In the early phases of testing, the failure intensity values observed contains considerable noise]. The use of reliability growth models in the early phases can sometimes result in grossly incorrect projection. The accuracy can be enhanced by using *a priori* parameter values in such cases.
- Residual defect density can be measured accurately.

Musa, et al. have speculated that K may depend on program structure in some way. However, they suspected that for large programs, the "structured ness" (as measured by decision density) averages out and hence does not vary much from program to program [50]. Musa has also argued that K should be independent of program size [35]. Likewise Malaiya argue that K may be relatively independent of the program size [51].Mayrhauser and Teresinki [52] have suggested that K may depend on testability, as measured by static metrics like "loopiness" and "branchiness" of the program. However, because of lack of sufficient data, the results are not yet conclusive. Li and Malaiya suggested that K varies with the initial defect density and is given by the following expression.

$$K = (1.2 * 10E\text{-}06/Do)*Exp(0.005*Do) \quad (27)$$

Defect Density

Defect density is an important measure of software quality, one which is often used as an acceptance criterion for a piece of software. For this reason it is desirable to understand how various aspects of the development process impact defect density, so they can be controlled or at least used to gain a better understanding of product reliability. The maturity of the development process, the skill of the programmers involved, and the complexity of the program all play a significant part in the defect density of a program [51].

Leading edge software development organizations typically achieve a defect density of about 2.0 defects/KLOC [53] and some now use a lower target. The cost of fixing a defect later can be several orders of magnitude higher than during development, yet a program must be shipped by some deadline dictated by market considerations. This makes the estimation of the number of remaining defects a very important challenge.

One conceivable way of knowing the exact defect density of a program is to actually find all remaining defects. This is obviously infeasible for any commercial product. Even if there are resources available, it will take a prohibitive amount of time to find all bugs in a large program [54]. Sampling based methods have been suggested for estimating the number of remaining defects. They assume that the faults *found* have the same testability as faults *not found*. However, in actual practice, the faults not found represent faults that are harder to find [1]. Thus such methods are likely to yield an estimate of faults that are relatively easier to find, which will be less than the true number. It is possible to estimate the defect density based on past experience using empirical models like the Rome Lab model [55] or the model proposed by Malaiya and Denton [51].

The estimates obtained by such models can be very useful for initial planning; however these models are not expected to be accurate enough to compare with methods involving actual test data. Another possible way to estimate the number of faults is by using the Exponential (Software Reliability Growth Model) SRGM. In this model the parameter β0 represents the total number of defects that would eventually be found. We can estimate the number of remaining defects by subtracting the number of defects found from the value of β0 obtained by fitting. An SRGM relates the number of defects found to the testing time spent. As this work focuses on predicting the defect density at the early phase of object oriented software designs there is no working code in hand, so the use of SRGM fails.

Having an early warning system to estimate defect density would aid developers by giving them an indication as to potential problems in the system. We can leverage metrics, which are readily available in the system to help provide defect density estimate. The next section gives the overview of the previously suggested defect density model.

Defect Density Model

There has been considerable research to identify the major factors that correlate with the number of defects. The Malaiya and Denton [51] based on the data reported in literature, is given by

$$DD = C * Fph * Fpt * Fm * Fs \quad (28)$$

Where Fph = phase factor modeling dependence on software test phase.

Fpt = programming team factor [Table-7]

Fm = maturity of software development [Table-8]

Fs = structure of the software under development

C = constant of proportionality

DD = Defect Density

The number of defects present at the beginning of different test phases is different. Gaffney [16] has proposed a phase-based model that uses the Rayleigh curve. Malaiya [51] presented a simpler model using actual data reported by Musa and the error profile presented by Piwowarski. The first two columns of [Table-6] represent the multipliers suggested by the numbers given by Musa et al. and Piwowarski et al. The third column presents the multipliers assumed by Malaiya [51].

Table 6- Phase Factor

Test Phase	Multiplier		
	Musa et al.	Piwowarski	Malaiya et al.
Unit	3.28	5	4
Subsystem	Insufficient Data	2.5	2.5
System	1	1	1(default)
Operations	0.25	0.45	0.35

Table 7- Team Factor

Team's Average Skill	Multiplier
High	0.4
Average	1(default)
Low	2.5

Table 8- Maturity Factor

SEI CMM Level	Multiplier
Level 1	1.5
Level 2	1(default)
Level 3	0.4
Level 4	0.1
Level 5	0.05

The defect density varies significantly due to the coding and debugging capabilities of the individuals involved [14, 55]. The only available quantitative characterization is in terms of programmer's average experience in years, given by Takahashi and Kamayachi [14]. Their model can take into account programming experience of up to 7 years, each year reducing the number of defects by about 14%. The data in the study reported by Takada et al [55] suggests that programmers can vary in debugging efficiency by a factor of 3. In a study about the PSP process [54], the defect densities in a program written separately by 104 programmers were evaluated. For about 90% of the programmers, the defect density ranged from about 50 to 250 defects/KSLOC. This suggests that defect densities due to different programming skills can differ by a factor of 5 or even higher.

This factor takes into account the rigor of software development process at a specific organization. This level, as measured by the SEI Capability Maturity Model, can be used to quantify it.

The structure of development factor takes into account the dependence of defect density on language type (the fractions of code in assembly and high level languages), program complexity, modularity and the extent of reuse.

The model given equation 26 provides an initial estimate. It should be calibrated using the past data from the same organization. For this purpose the factor C is involved. Musa's data suggest that the

constant of proportionality C can range from 6 to 20 defects per KLOC.

Software Reliability Growth Problem

The basic problem of reliability theory is to predict when a system will eventually fail. Software fails because of design problems. Individuals based on their intellectual develop design and as intellectual itself varies from individual to individual, it is really not easy to predict from the design that when the system will fail. Various approaches have been proposed for modeling software reliability but no individual technique has singled out for unreserved recommendations from the many that have been proposed over years [56]. Empirical observations suggest that no such technique has consistently able to give accurate result over different data sources and we need to examine the accuracy of the actual reliability measures, obtained from several techniques in a particular case, with a view of selecting the one (if any) that yields trustworthy results [57].

But here it again gives rise to another problem that how to decide which model to use in the given circumstances as things are not as clear as they appear to be in theory. As noted above one should not trust a single approach for assessing reliability, but another problem that arises here is that, what about the economy of using multiple software reliability models for measuring the accuracy of the software?

The underlying problem with the software reliability is that you cannot firmly predict when it will fail. Suppose it is considered that the system will fail in its 10 hours of operation once. But in these 10 hours the system may fail in first two minutes or in the last two minutes. Also if the path containing error is not at all traversed in its 10 hours of working it will assure 100% reliability which may not be the fact.

The another issue that comes up is that, on often fixing a fault a general consideration is that it has led to the improvement of reliability however concerning software ineffective fixes or the introduction of novel faults will indeed decrease the reliability. Thus several questions come forward even after fixing the faults:

- How reliable is the software now?
- Is it sufficiently reliable that we can cease testing and ship it?
- How reliable it will be after we spend a given amount of further testing effort?
- How long are we likely to wait until the reliability target is achieved?

A crucial activity is the definition of operational profile and associated test cases. OP is used to select test cases and direct development, testing and maintenance [58]. Both the determination of OP and random sampling of test cases might be impractical in certain applications [59, 60]. In such a case one has little choice but to rely on educated guesstimates.

The problem of assessing the reliability of software is again a difficult one because claims for extremely high reliability appear to need extremely large amount of evidence. Thus direct evaluation of reliability, using statistical methods based upon operational data will only allow quiet modest claims to be made – putting it another way, to make claims ultra high reliability this way would require infeasible large amount of operational exposure [61].

Although various software reliability models claims their efficiency in some way or the other they are not used as universally in software development as they should be. Some reasons that project managers give are following [9]:

- It costs too much to do such modeling and I can't afford it within my budget.
- There are so many software reliability models to use that I don't know which is best; therefore I choose not to use any.
- We are using the most advanced software development strategies and tools and produce high quality software; thus we don't need reliability measurements.
- Even if the model told me that the reliability is poor, I would just test some more and remove more errors
- If I release product with too many errors, I can always fix those that get discovered during early deployment.

Another issue that comes up is that assessing or predicting reliability of the software on the past experience and data of the similar type of software may not be relied upon. Reason, the past experiences has already thought us eliminate repetitive errors. Therefore by using past information and predicting the present software reliability status sounds abrupt. The other conditions like the external environment, the resources namely; human, other software and hardware may also vary from software to software development process.

Thus we see there are various aspects related to software reliability and an proper understanding of these aspects will provide better insight for effective software reliability estimation.

Acknowledgement: Dr. K V Kale, Professor and Head, Department of Computer Science, Dr. Babasaheb Ambedkar Marathwada University, Aurangabad for his valuable guidance.

References

[1] Goel A.L. (1985) *IEEE Transactions on Software Engineering*, 12, 1411-1423.

[2] Musa J.D. (1975) *IEEE Transactions on Software Engineering*, SE-1(3), 312-327.

[3] Wallace R.B. & Prabhakar Murthy D.N. (2000) *Reliability Modeling, Predition and Optimization*, Wiley Inter Science Publication 848.

[4] Goel A.L. & Yang K.Z. (1997) *Advances in Computers*, 45, 197-267.

[5] Mills H.D. (1972) *On the Statistical Validation of Computer Programs*, IBM Fedreal Syst.Div.,Gaitthersburg, MD, Rep. 72-6015.

[6] Ramamoorthy C.V. & Bastani F.B. (1982) *IEEE Trans. Software Eng*., 8(4), 354-371.

[7] Hudson G.R. (1967) *Program Errors as A Birth and Death Process*, Report SP-3011, System Development Corporation, Santa Monica, California.

[8] Jelinski Z. & Moranda P.B. (1972) *Software reliability research, Statistical Computer Performance Evaluation*, Academic Press, New York, 465-484.

[9] Shooman M.L. (2002) *Reliability of Complete System and Network*, Wiley InterScience, 552.

[10] Goel A.L. & Okumoto K. (1979) *IEEE Transactions on Reliability*, 28(3), 206-211.

[11] Yamada S., Ohba M. & Osaki S. (1983) *IEEE Transactions on Reliability*, 32(5), 475-484.

[12]Crow L.H. & Singpurwalla N.D. (1984) *IEEE Transactions on Reliability*, 33(2), 176-183.

[13]Schick G.J. & Wolverton R.W. (1973) *Annual Meeting* 1972 Physica-Verlag HD, 395-422.

[14]Musa J.D. & Okumoto K. (1984) *Journal of Systems and Software*, 4(4), 277-287.

[15]Ohba M. (1984) *IBM Journal of research and Development*, 28 (4), 428-443.

[16]Gaffney J. & Pietrolewicz J. (1990) *An Automated Model for Early Error Prediction in Software Development Process*, Proc. IEEE Software Reliability Symposium Colorado Spring.

[17]Pham H. (2000) *Software Reliability*, New York

[18]Abdel-Ghally A.A., Chan P.Y. & Littlewood B. (1986) *IEEE Trans. on Software Engineering*, 12(9).

[19]Grottke M. (2001) *Software Reliability Model Study*, Deliverable A-2 IST-1999-55017.

[20]Duane J.T. (1964) *IEEE Transactions on Aerospace*, 2(2), 563-566.

[21]Littlewood B. (1980) *IEEE Transactions on Software Engineering*, 5, 489-500.

[22]Desai C. (2008) *International Journal of Computational Intelligence and Telecommunication Systems*, 11-16

[23]Cheung R.C. (1980) *IEEE Transactions on Software Engineering*, 2, 118-125.

[24]Park J., Kim H.J., Shin J.H. & Baik J. (2012) *IEEE Sixth International Conference on Software Security and Reliability*, 207-214.

[25]Kapur P.K., Tandon A. & Kaur G. (2010) 2nd International Conference on Reliability, Safety and Hazard, 468-474.

[26]Zhu X.M., Guo Z.G. & Yuan C. (2012) *Fuzzy Engineering and Operations Research*, Springer Berlin, 485-492.

[27]Okamura H., Etani Y. & Dohi T. (2010) *IEEE 21st International Symposium on Software Reliability Engineering*, 31-40.

[28]Atole C.S. & Kale K.V. (2004) *An Analysis of Software Reliability Estimation for Model Ranking*, *Proceedings of National Conference on Software Engineering Principles and Practices*, Patiala, Punjab.

[29]Gaudoin O. (1992) *IEEE Transactions on Reliability*, 41(4), 525-532.

[30]Lyu M.R. & Nikora A. (1992) *CASRE: a computer-aided software reliability estimation tool*, Proceedings of Fifth International Workshop on Computer-Aided Software Engineering, 264-275.

[31]Naixin L. & Malaiya Y.K. (1996) *Fault exposure ratio estimation and applications*, Proceedings of Seventh International Symposium on Software Reliability Engineering, 372-381.

[32]Denton J.A. (1999) *Accurate Software Reliability Estimation*, Thesis.

[33]Malaiya Y.K., Von Mayrhauser A. & Srimani P.K. (1993) *IEEE Transactions on Software Engineering*, 19(11), 1087-1094.

[34]Musa J.D. (1991) *ACM SIGSOFT Software Engineering Notes*, 16(3), 78-79.

[35]Malaiya Y.K. & Denton J. (1997) *The Eighth International Symposium on Software Reliability Engineering*, 124-135.

[36]von Mayrhauser A. & Teresinki J.A. (1990) *Proc. Symp. On Software Reliability Engineering*, 19.1-19.13.

[37]Binder R. V.(1997) Six sigma: Hardware si, software no!

[38]Butler R.W. & Finelli G.B. (1993) *IEEE Transactions on Software Engineering*, 19(1), 3-12.

[39]Atole C.S. & Kale K.V. (2005) *Software Reliability Model Review: In Light Of Software Development Process*, Software Engineering Institute (SEI) CMM.

[40]Fenton N.E. & Pfleeger S.L. (1998) *Software metrics: a rigorous and practical approach*, PWS Publishing Co.

[41]Vouk M.A. (2000) *Software reliability engineering, In a tutorial presented at the Annual Reliability and Maintainability Symposium.*

[42]Horgan J.R. & Mathur A.P. (1996) *Software Testing And Reliability, Hand Book of Software Reliability Engineering*, Mc Graw Hill.

[43]Horgan J.R. & Mathur A P (1995) *Perils Of Software Reliability Modeling*, Technical Report, SERL-TR-160.

[44]Littlewood B. (2000) *The problems of assessing software reliability... when you really need to depend on it*.