Exploitation of Quantitative Approaches to Software Reliability

Mariam Rahmani†, Azad Azadmanesh

Computer Science Department
University of Nebraska at Omaha, USA
{crahmani, azad}@unomaha.edu

Software reliability is one of the main factors to measure the quality of software. Since software errors cause spectacular failures in some cases, we need to measure the reliability factor to determine the quality of software product, predict reliability in the future, and use it for planning resources needed to fix failures. Software reliability models are applicable tools to analyze software in order to evaluate the reliability of software. During the past twenty five years, more than fifty different models have been proposed for estimating software reliability but many of software practitioners do not know how to utilize these models to evaluate their products. In this paper we will present a survey on different models of software reliability and their characteristics. We will propose taxonomy of different models and try to aid the comprehension of these models for practitioners, developers, and users. In the last section we apply some of these models on two different open source projects and compare the results.

* The work is the result of a class project. Because of some external requests who would like to cite this work properly, the report is converted into a technical report as is without any obligation until it can be formally organized and edited with adequate citations.

† Mariam Rahmani is a PhD student in the College of Information Science & Technology. She accomplished this work in Fall 2008 as a partial requirement of the course: Survivable Networked Systems (CIST-9900).
# Table of Contents

1. Introduction .................................................................................................................. 3
2. Software Reliability Definition ....................................................................................... 3
3. Taxonomy of Software Reliability Models ...................................................................... 5
4. Deterministic Reliability Model vs. Probabilistic Reliability Model .............................. 7
   4.1. Deterministic reliability models .................................................................................. 7
   4.1.1. Halstead’s Software Metric ..................................................................................... 8
   4.1.2. McCabe’s Cyclomatic Complexity Metric .............................................................. 9
   4.2. Probabilistic reliability models ............................................................................... 11
   4.2.1. Error Seeding Models .......................................................................................... 11
   4.2.2. Failure Rate Model ............................................................................................. 16
   4.2.3. Curve Fitting Models ......................................................................................... 19
   4.2.4. Non-Homogenous Poisson Process Models (NHPP) ............................................ 21
   4.2.5. Markov Structure Models ................................................................................. 25
   4.2.6. Reliability Growth Models ............................................................................... 26
5. Experimental Research ................................................................................................. 27
6. Conclusion ...................................................................................................................... 30
7. References ..................................................................................................................... 31
1. **Introduction**

After many years in computer era, software became as a necessary part of human life. They have been used in various areas from real-time applications and military service to business process and personal computing software. As computer software play an essential role in complex and critical applications, we would sense there is a greater need for more reliable software. Since faults in software products became more and more subtle, software reliability field turned to be an important issue. And the growing significance of software dictates the focus shift to this field more than before.

Though the software reliability approaches try to decrease the penalty of failure occurrence in software products, they are not easy to apply. Reliability as a factor of software quality needs to be defined as a group of requirements in the early phases of software development. Errors (design or implementation errors) need to be eliminated using various testing methods, verification and validation, and software fault tolerance approaches. Furthermore, some techniques for measuring and analyzing defects in software are needed.

We infer that to produce a reliable software product, we need to know more about reliability, reliability models, and how can we utilize them in our products. To receive this goal in this paper, we divided it to five further sections. In section two we will present some of the software reliability definitions. In the third section we give the taxonomy of software reliability models. Section four is a survey on different models of software reliability. Section five is about our experimental research on two different open source projects and the last section is conclusion part.

2. **Software Reliability Definition**

Musa and Okumoto in 1984 defined software reliability as the “probability of failure free operation of a computer program in a specified environment for a specified period of time.”

NASA Software Assurance Standard, NASA-STD-8739.8 [7], defines software reliability as a discipline of software assurance that (Fig.1):

1. Defines the requirements for software controlled system fault/failure detection, isolation, and recovery;
2. Reviews the software development processes and products for software error prevention and/or reduced functionality states; and,

3. Defines the process for measuring and analyzing defects and defines/derives the reliability and maintainability factors.

![Diagram](image)

**Figure (1) Software reliability definition from NASA Software Assurance Standard view**

As mentioned in NASA Software Assurance Standard, software reliability process consists of three main phases which are extended through the whole software life-cycle process. Define reliable requirements in initial phases of software lifecycle, review whole software process to prevent potential errors, and present some techniques to measure and analyze defects in software. In this paper we suppose that software process already supports the first and second phases and we try to study and categorize the techniques in the third section. In this research work, everywhere we talk about software reliability models we mean the models which can measure and analyze the defects in software to achieve two goals: representation of software reliability situation at present and estimation of software reliability condition in future.

Modeling the failure process in a piece of software is a very challenging exercise partially because of the diverse and interlocking nature of faults that may exist and the methods that may be used to detect or discover them [1].

Software reliability models are used to achieve the quality of software and also to plan for resources needed to fix the problems in the maintenance phase. They have been used as the most important and successful predictor of software quality when it hits the market. In general, most of these reliability
models use independent variables and show their modeling results in two-dimensional diagrams based on these independent variables.

Typical scopes of measurement (X axis) in reliability model diagrams include: 1- Calendar time (days of testing), 2- Cumulative testing effort (hours of testing); 3- Computer execution time (e.g. number of CPU hours). And typical dependent variables (Y) include: 1- Number of defects found per life cycle phase or total number ever, 2- Failure rate over time, 3- Cumulative number of failures over time, 4- Time between failures.

In this paper we extend some reliability models and study how they analyze the failure data and estimate the future reliability. In section three we will present taxonomy of different software reliability models and their characteristics. We try to present some examples for each category in software reliability models.

3. Taxonomy of Software Reliability Models

The taxonomy presented in this section gathered different types of reliability models from previous works in software reliability field. This taxonomy, presented in Figure 2, tries to show a common overview from different views existent in the previous research papers. Although it cannot be the only potential taxonomy of software reliability models but we tried to ease the understanding of communalities and discrepancies of these models.

Software Reliability models can be divided into two main categories: Deterministic or Static models and Probabilistic or Dynamic models.

A static model uses some attributes from project or program module other than failure data to estimate the number of failures in the software. Based on [6] a static model of software quality estimation has the following general form:

\[ y = f(x_1, x_2, ..., x_i) + e \]

Where the dependent variable \( y \) is the defect rate or the number of defect, and the independent variable \( x_i \) are the attributes of the product, the project or the process through which the product is developed. They could be line of code, complexity, skill level, count of decision, number of modules or in
object oriented languages number of classes, weighted method per class, coupling between object, depth of inheritance tree, and other meaningful measurements.

Figure (2) Taxonomy of software reliability models

- Shows an example of special model
- Shows is-a relationship
The error rate is $e$ (because models don’t completely explain the behavior of the dependent variable). Also estimated coefficients of the independent variable in the formula are based on previous software products. For the current software product or project, the value of independent variables are measured, then plugged into the formula to drive estimates of the dependent variable —the product defect rate or number of defect. This model supposes the existence of a sufficient number of software products; so it estimates the reliability of each product as a function of other products. In contrast, in Probabilistic reliability modeling data is gathered from the product of interest; therefore, the resulting model is specific to software product for which the projection of reliability is attempted. On the other hand, these reliability models tracks the failure data produced by the software system to develop a reliability operational profile of the system over a specified time. Because they are based on statistical distributions they are named dynamic models too. A common attribute of dynamic models is that they are expressed as a function of time.

### 4. Deterministic Reliability Model vs. Probabilistic Reliability Model

As shown in the presented taxonomy in previous section, for deterministic reliability models we will discuss about two popular models: Halstead’s Software Metric [10] and McCabe’s cyclomatic complexity (McCabe, 1976) [9].

Also probabilistic reliability models are categorized into six different branches: Error seeding, Failure rate, Curve Fitting, Reliability growth, Non-Homogenous Poisson Process, and Markov structure. Each of these software reliability models have some methods to estimate software reliability and other performance measures such as software complexity, software safety, and the number of remaining errors.

#### 4.1. Deterministic reliability models

Halstead’s theory is probably the best-known technique to measure the complexity in a software program and the amount of difficulty involved in testing and debugging the software [8]. We will discuss about this model in part 3.1.1. Another model presented in 3.1.2 is McCabe’s cyclomatic complexity which is a complexity measure for control flow representation of a program.
4.1.1. Halstead’s Software Metric

This model uses the number of distinct operators and the number of distinct operand in a program to develop expressions for the overall program length, volume and the number of remaining defects in a program.

These notations are used in Halstead model:

\[ n_1 = \text{number of unique or distinct operators appearing in a program}; \]

\[ n_2 = \text{number of unique or distinct operands appearing in a program}; \]

\[ N_1 = \text{total number of operators occurring in a program}; \]

\[ N_2 = \text{total number of operands occurring in a program}; \]

\[ N = \text{length of the program}; \]

\[ V = \text{volume of the program}; \]

\[ E = \text{number of errors in the program}; \]

\[ I = \text{number of machine instructions}. \]

In this model length and the volume measure of the program can be estimated:

\[ N = N_1 + N_2 \]

And

\[ V = N \log_2(n_1 + n_2) \]

Respectively, where

\[ N_1 = n_1 \log_2 n_1 \]
N2 = n2 \log_2 n2

Halstead proposed two empirical formulae to estimate the number of remaining defects in the program. These models named Halstead empirical model 1 and Halstead empirical model 2. They are presented respectively:

\[ \hat{E} = \frac{V}{3000} \]

And

\[ \hat{E} = \frac{A}{3000} \]

Where

\[ A = \left( \frac{V}{(2n_2/n_1N_2)} \right)^{2/3} \]

These coefficients are estimated empirically and examined by the author and also others [11]. In this model V is the volume of the assembly language version of the program. Halstead also showed that empirical model 2 is close to the number of observed defects than empirical 1.

Halstead examined his formulae by number of defects observed from an example program and found that the numbers of observed defects are closer to the calculated \( \hat{E} \) from empirical model 2. But he didn’t prove that these observed defects are all defects which are latent in software. This model also supposes that the number of faults in program is constant and all of them are indigenous errors.

4.1.2. McCabe’s Cyclomatic Complexity Metric

McCabe’s cyclomatic complexity metric is a software metric that provides a quantitative measure of the logical complexity of a program by counting the decision point. For example, one should start with 1 for the straight path through the module or subroutine. Then 1 should be added each time one of the following keywords appear: IF, REPEAT, WHILE, FOR, OR, AND. Also, 1 should be added for each case in a case statement lacks a default case. If the total score is less than 10, then by using the McCabe’s measure, the code is considered to be of high quality software. McCabe has suggested that a program
with a high level of the metric is very difficult to produce and maintain. He recommended a total score of 10 as an upper limit for the FORTRAN environment.

In a strongly connected graph the cyclomatic number is equal to the maximum number of linearly independent circuits. The linearly independent circuits from a basis for the set of all circuits in G, and any path passing through G can be expressed as a linear combination circuits.

The cyclomatic number \( V(G) \) of a graph G can be determined from the following formula:

\[
V(G) = e - n + 2p
\]

Where

\( e \) = number of edge in the control graph in which an edge is equivalent to a branching point in the program;

\( n \) = number of vertices in the control graph in which an edge is equivalent to a branching point in the program;

\( p \) = number of connected elements (usually 1).

The cyclomatic number of graph with multiple connected components is equal to the sum of the cyclomatic numbers of the connected components. In any program that can be represented by a state diagram, the McCabe cyclomatic complexity metric can also be used to calculate the number of control flow paths (FP). i.e.,

\[
FP = e - n + 2
\]

Where \( e \) is the number of edges and \( n \) is the number of nodes. Here, edges are represented on the diagram by arrows and nodes are shown on the diagram with circles.

Another way to calculate the cyclomatic number is as follows:

\[
V(G) = \Lambda + 1
\]
where \( \Lambda \) is the number of predicate nodes (decisions or branches) in the program. In other words, the cyclomatic number is a measure of the number of branches in a program.

A branch occurs in IF, WHILE, REPEAT, and CASE statements. The cyclomatic number has been widely used in predicting the number of errors and in measuring software quality. McCabe noted that, when used in the context of the basis path testing method, the cyclomatic complexity, \( V(G) \), provides an upper bound on the number of tests that must be conducted to ensure that all program statements have been executed at least once.

This metric is an interesting measure of whether the software designer has followed good cyclomatic complexity also estimates how difficult it will be to test all the paths in the program and thus, provide useful information of how difficult it will be to satisfy the software specifications and requirements. Also, McCabe’s measure will be less useful if the software developers and testing debuggers are interested in detecting all the faults [8].

### 4.2. Probabilistic reliability models

These types of reliability models rather than the deterministic ones which use some static parameters of program, apply some statistical methods to estimate measures such as the number of errors in a program, failure rate, relationship between software complexity and the number of faults in a program, and so on.

In this section we try to present some examples from each six types in probabilistic methods. In the next sections we will explain these models with more details.

#### 4.2.1. Error Seeding Models

Error seeding is a technique, which can be used to evaluate the amount of residual errors after the system software test phase. This method helps to decide how to release the software depending on results that fault -seeding technique can provide.

The error seeding group of models estimates the number of errors in a program by using the multistage sampling technique. Multistage sampling is a complex form of cluster sampling. Using all the sample elements in all the selected clusters may be prohibitively expensive or not necessary. Under these circumstances, multistage cluster sampling becomes useful. Instead of using all the elements contained
in the selected clusters, the researcher randomly selects elements from each cluster. Constructing the clusters is the first stage. Deciding what elements within the cluster to use is the second stage. This technique is used frequently when a complete list of all members of the population does not exist and is inappropriate [12].

In the error seeding models, errors are divided into indigenous errors and induced errors. The unknown number of indigenous errors is estimated from the number of induced errors and the ratio of the two types of errors obtained from the debugging data. By seeding errors to a document (for example: source code, requirements specification) and then let the document undergo testing of some kind it is possible to calculate how many “real” errors that exists. Models included in this group are:

- Mills’ error seeding model [13];
- Cai’s model [14]; and
- Hyper geometric distribution model [15]

Mills’ error seeding model is the first method proposed in 1970. It founded this type of models and uses seeded, induced, errors to calculate the number of errors in the document; Cai’s Model modified Mills’ model by instead of seeding errors divide the software (or document) into two parts to estimate the number of defects remaining; Hypergeometric Distribution model is also an estimation model to find out how many faults initially exists in a program. By not removing the errors when found and keeping track of the already found errors and newly found errors for each test instance it is possible to estimate the number of faults in the program. These models are explained in the next three sections.

4.2.1.1. Mills’ error seeding model
This model is one of the error seeding methods which estimate the number of errors in a program by introducing seeded errors into the program.

In this method some of known error types are inserted into the program, and the program is executed with the test cases under test conditions. The ratio of found seeded errors to the total number of seeded errors is approximate equal to the ratio of found real errors to total number of errors [16].

Assume the seeded faults (S) are of the same severity and are equally probable to be exposed than an equivalent latent fault (N) the relation:
$S/n/N \rightarrow N/nS/s$

$S$: number of seeded faults

$N$: number of original (latent) faults in the code (this is the number we want to estimate)

$s$: number of seeded faults found

$n$: number of initial faults found

Figure 3 shows a Venn diagram which presents the relationship between seeded and initial errors in a program.

![Venn diagram](image)

Figure (3) Seeded errors versus initial errors

There are some drawbacks with this model. It is expensive to conduct testing of the software and at the same time, it increases the testing effort. This method was also criticized for its inability to determine the type, location and difficulty level of the seeded errors such that they would be detected equally likely as the initial errors.

4.2.1.2. Cai’s Model

Cai modified Mill’s model by dividing the software into two parts: Part0 and Part1. There are following assumptions for this model:

1. The software can be divided into two parts: Part0 and Part1.
2. There are $N$ defects remaining in the software, where Part0 contains $N_0$ and Part1 $N_1$ remaining defects. That is $N = N_0 + N_1$.
3. At any time, that is, no matter how many remaining defects are contained in the software, each of the remaining defects has the same probability of being detected.
4. Upon being detected, the defect is removed.
5. Each time, one only one and remaining defect is removed and no new defects are introduced.
6. There are \( n \) remaining defects removed.

Figure 3 depicts the software defect removal process, where \( t_i \) represents the time instant of the \( i \)th remaining defect being removed. Let

\[
Y_i = \begin{cases} 
0 & \text{if the } i\text{th detected defect is remaining in part } 0 \\
1 & \text{if the } i\text{th detected defect is remaining in part } 1 
\end{cases}
\]

Suppose \( \{y_i\} \) is a realization of \( \{Y_i\} \). Let \( N_j(i) \) be the number of defects remaining in Part \( j \) in the time interval \( (t_i, t_{i+1}] \) for \( i = 0, 1, 2, \ldots, n \) and \( j = 0 \) or \( 1 \). Assume \( y_i \) are the observed values of \( Y_i \) and \( y_0 = 0 \). Therefore,

\[
N_0(i) = N - i + \sum_{j=0}^{i} y_j \\
N_1(i) = N_1 - \sum_{j=0}^{i} y_j \quad ; \quad i = 0, 1, \ldots, n
\]

Let \( p_{j}(i) \) be the probability of having a defect remaining in Part \( j \) detected in the time interval \( (t_i, t_{i+1}] \) where \( i = 0, 1, 2, \ldots, n \) and \( j = 0 \) or \( 1 \). Then we obtain

\[
p_0(i) = \frac{N_0(i)}{N_0(i) + N_1(i)} = \frac{N_0 - i + \sum_{j=0}^{i} y_j}{N_0 + N_1 - i} \\
p_1(i) = \frac{N_1(i)}{N_0(i) + N_1(i)} = \frac{N_1 - \sum_{j=0}^{i} y_j}{N_0 + N_1 - i} \quad ; \quad i = 0, 1, \ldots, n
\]
Now we wish to estimate $N_0$ and $N_1$. The likelihood function can be determined as follows:

$$L(y_1, y_2, \ldots, y_n) = \Pr\{Y_i = y_i, Y_j = y_j, \ldots, Y_n = y_n\}$$

$$= \prod_{i=1}^{n} \Pr\{Y_i = y_i | Y_1 = y_1, \ldots, Y_{i-1} = y_{i-1}\}$$

We note that

$$\Pr\{Y_i = y_i | Y_i = y_1, \ldots, Y_{i-1} = y_{i-1}\} = \begin{cases} p_0(i-1) & \text{if } y_i = 0 \\ p_1(i-1) & \text{if } y_i = 1 \end{cases}$$

Or

$$\Pr\{Y_i = y_i | Y_i = y_1, \ldots, Y_{i-1} = y_{i-1}\} = \left(\frac{p_0(i-1)}{p_1(i-1)}\right)^{y_i} \quad ; i = 1, 2, \ldots, n$$

Thus,

$$L(y_1, y_2, \ldots, y_n) = \prod_{i=1}^{n} \left(\frac{p_0(i-1)}{p_1(i-1)}\right)^{y_i}$$

Taking the ln of the likelihood function, we obtain:

$$\ln L(y_1, y_2, \ldots, y_n)$$

$$= \sum_{i=1}^{n} \left\{ (1 - y_i) \ln \left( N_0 - i + 1 + \sum_{j=0}^{i-1} y_j \right) - \ln (N_0 + N_1 - i + 1) + y_i \ln \left( N_1 - \sum_{j=0}^{i-1} y_j \right) \right\}$$

Substituting $p_0$ and $p_1$ from previous equations into the above equation and taking first derivatives with respect to $N_0$ and $N_1$ are determined by the following equations:

$$\sum_{i=1}^{n} \left\{ \frac{1 - y_i}{\hat{N}_0 - i + 1 + \sum_{j=0}^{i-1} y_j} - \frac{1}{\hat{N}_0 + \hat{N}_1 - i + 1} \right\} = 0$$

$$\sum_{i=1}^{n} \left\{ \frac{y_i}{\hat{N}_1 - \sum_{j=0}^{i-1} y_j} - \frac{1}{\hat{N}_0 + \hat{N}_1 - i + 1} \right\} = 0$$
The total initial number of defects remaining in the software is estimated as \( N = N_0 + N_1 \);
The drawbacks of this model are related to the key assumptions of 3 and 5. Sometimes these assumptions are not reasonable and applicable for real software.

Another model in this group is Hypergeometric distribution model which is proposed by Tohma et al. for estimating the number of faults initially resident in a program at the beginning of the test or debugging process based on Hypergeometric distribution. This model is also one of the error seeding models and it can be found in [15].

### 4.2.2. Failure Rate Model

This group of models is used to study the program failure rate per fault at the failure intervals. As shown in taxonomy, different models are proposed in these category and we explain three models in this category:

- Jelinski-Moranda Model – 1972. [18]
- Geol and Okumoto imperfect debugging Model – 1979. [20]

This group studies how failure rate change at the failure time during the failure intervals. As the number of remaining faults changes, the failure rate of the program changes accordingly.

#### 4.2.2.1. Jelinski-Moranda Model (1972)

Based on classification scheme for models described by Singpurwalla and Wilson in 1994 [2], most of the probabilistic software reliability models fall into one of two broad categories.

I: Modeling the times between successive failure of the software which is broken down into
   I-1. Those which use the failure rate as a modeling tool
   I-2. Those which model an inter-failure time as a function of previous inter-failure times

II: Modeling the number of failures of the software up to a given time.

Figure 5 shows this classification in Singpurwalla and Wilson (1994) work.
The Jelinski-Moranda (1972) model is a basic model of type I-1, where one assumes that in time 0 there are a finite and fixed number of bugs $N(0)$ in the software. The error process is a non-homogenous Poisson process which means it has Poisson function with a rate $\lambda(t)$ that may vary with time. The error rate $\lambda(t)$ at time $t$ is assumed to be proportional to $N(t)$,

$$\lambda(t) = cN(t)$$

$\lambda(t)$ has an initial value of $\lambda_0 = \lambda(0) = cN(0)$ and decreases by $c$ whenever an error occurs and the bug that caused it is corrected, and is constant between errors.

The (testing, not including fixing) time between consecutive errors (say $i$ and $i+1$) is exponentially distributed with parameter $\lambda(t)$, where $t$ is the time of the $i$th error. The reliability at time $t$, or the probability of a error free operation during $[0,t]$ is therefore

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

Given an error occurred at time $\tau$, the conditional future reliability, or the conditional probability that the following interval of length $t$, namely $[\tau, \tau + t]$ will be error free is

$$R(t | \tau) = e^{-\lambda(t) t}$$

Based on [3], as the software runs for longer and longer, more bugs are caught and purged from the system, and so the error rate declines and the future reliability increases.
One basic assumption in the Jelinski-Moranda model is that faults are perfectly repaired when encountered. Normally however, any valid software fault must pass a regression test before it is categorized as resolved, and in practice regression testing uncovers a significant number of imperfect fixes and in some cases new faults. Some modifications of the Jelinski-Moranda model have been developed to incorporate imperfect repair.

Another obvious objection to this model is that it assumes that all bugs contribute equally to the error rate, as expressed by the constant of proportionality number of \( c \). Actually not all bugs are found equal: some of them are exercised more often than others. Indeed, the more troublesome bugs are those that are not exercised often and they are extremely difficult to catch during testing.

4.2.2.2. Jelinski-Moranda Geometric Model (1979)

The J-M geometric model (1979) assumes that the program failure rate function is initially a constant \( D \) and decreases geometrically at failure times. The program failure rate and reliability function of time-between-failure at the \( i \)th failure interval can be expressed, respectively, as

\[
\lambda(t_i) = D R^{i-1}
\]

and

\[
R(t_i) = e^{-DR^{i-1}t_i}
\]

Where \( D \) = initial program failure rate and

\( k \) = parameter of geometric function \((0<k<1)\).

If multiple error removal be allowed in a time interval, then the failure rate function becomes

\[
\lambda(t_i) = D R_{i-1}^{k-1}
\]

Where \( R_{i-1} \) is the cumulative number of errors found up to the \((i-1)\)st time interval. The software reliability function can be written as

\[
R(t_i) = e^{-DR_{i-1}^{k-1}t_i}
\]

In the [17], authors examined that Jelinski-Moranda and J-M Geometric models failed the consistency test which they proposed. They challenged these models to take data which comes from a process which they have correctly modeled and to make predictions about the reliability of that process. The authors found that either model, given data precisely from a process it correctly models, will usually fail
to make good predictions. They attribute these problems to randomness in the data used as input to the models and indicate the remedy for this lack of robustness, namely replication of data.

4.2.2.3. Geol and Okumoto imperfect debugging Model
Goel and Okumoto in 1979 extended the J-M model by assuming that a fault is removed with probability p whenever a failure occurs. The failure rate function of the J-M model with imperfect debugging at the ith failure interval becomes

$$\lambda(t_i) = c \ [N-p(i-1)]$$

where c is the same as J-M model which is a proportional constant. Also N is the number of initial faults in the program.

The reliability function is

$$R(t_i) = e^{-c[N-p(i-1)]t_i}$$

This model is the same as J-M model where

c = c'p and N = N'/p (where c' is constant number of J-M model and N' is the number of initial faults in the J-M model)

4.2.3. Curve Fitting Models
This group of models uses statistical regression analysis to investigate the relationship between program complexity and the number of faults, the failure rate and also the number of change. These models use linear regression, non-linear regression or time series analysis to find a functional relationship between dependent and independent variables. For example, dependent variables can be the number of errors in software. On the other hand, the independent variables may be the time between failures, the number of module changed in the maintenance phase, programmers’ skill, program size, and etc. Models which are presented in this group are [8]:
- Estimation of errors
- Estimation of complexity
- Estimation of failure rate

4.2.3.1. Estimation of errors Model
Using regression models we can estimate number of errors in a program. One of the simple non-linear regression model to estimate the total number of initial errors in the program, N, is presented as follows:
\[ N = \sum_i a_i X_i + \sum_i b_i X_i^2 + \sum_i c_i X_i^3 \]

Where \( X_i \) is the \( i \)th error factor and \( a_i, b_i, c_i \) are the coefficients of the model. Some of the error factors are software complexity metrics and environmental factors. Some of these environmental factors which are discussed in [8] are as follows:

- **Program Categories**
  PC indicates the system complexity. These categories can be operating systems, communication control program, data base systems, and etc.

- **Programmer’s Skill (PS)**
  PS can be defined as the average number of years of programming experience. It can be calculated as:

\[
PS = \frac{\sum_i Y_i^i}{n}
\]

Where \( Y_i \) = number of years of experience per programmer \( i \)

\( n \) = total number of programmers involved in the project.

- **Frequency of Program Specification and Requirements Change**
  The frequency of program specification change can be calculated by the total number of pages of problem reports generated to change the program design specifications and/or requirements during the software development.

Some of the other environmental factors also can be found in [8]. But most of the Curve fitting models involve only one error factor.

### 4.2.3.2. Estimation of complexity Model

This model is proposed as a model for estimation of the software complexity. This model uses time series approach to estimate software complexity which is shown by \( C_R \).

\[
C_R = a_0 + a_1 R + a_2 E_R + a_3 M_R + a_4 I_R + a_5 D + \epsilon
\]

Where

\( R \) = release sequence number

\( E_R \) = environmental factor(s) at release \( R \)
\( M_R = \) number of modules at release \( R \)

\( I_R = \) inter-release interval \( R \)

\( D = \) number of days since first release

\( \varepsilon = \) error

This model is usable for some software which have various release versions and evolving in long period of time.

### 4.2.3.3. Estimation of Failure Rate Model

This model tries to estimate the failure rate of a software product using previous failure times. This model assumes that the failure rate is monotonically non-increasing and can be earned using the least squared method.

Based on this model the failure rate at the \( i \)th failure interval is

\[
\lambda_i = \frac{1}{\tau_{i+1} - \tau_i}
\]

### 4.2.4. Non-Homogenous Poisson Process Models (NHPP)

These models are based on non-homogeneous Poisson process statistical model. A non-homogeneous Poisson process is a Poisson process with rate parameter \( \lambda(t) \) such that the rate parameter of the process is a function of time. An example of a non-homogeneous Poisson process would be the arrival rate of customers to a restaurant over a day. The restaurant would notice the arrival rate increase during the meal hours and slow down during other parts of the day. These models are used to provide a framework for describing the software failure phenomenon.

The major issue in these models is to determine a proper mean value function to designate the expected number of defects happened up to certain point in time. There are different models in this group and we will look at some of them such as:

- Goel and Okumoto NHPP (Goel, 1979)
- Musa exponential (Musa, 1985)
- S-shaped growth models (Ohba, 1984)
Goel and Okumoto NHPP model and Musa exponential model are in Concave shape. And Ohba model is S-shaped model. These two model types are shown in Figure 6. The most important thing about these two models is that they have the same asymptotic behavior. Failure detection rate decreases as the number of defects detected (and repaired) increases, and the total number of defects detected asymptotically approaches a finite value. The theory for this asymptotic behavior is that [5]:

I. A finite amount of code should have a finite number of failures. Repair and new functionality may introduce new failures which increase the original finite number of defects. Some models explicitly account for new defect introduction during test while others assume they are negligible or handled by the statistical fit of the software reliability model to the data.

II. It is assumed that the defect detection rate is proportional to the number of defects in the code. Each time a defect is repaired; there are fewer total defects in the code, so the defect detection rate decreases as the number of defects detected (and repaired) increases. The concave model strictly follows this pattern. In the S-shaped model, it is assumed that the early testing is not as efficient as later testing, so there is a ramp-up period during which the defect detection rate increases. This could be a good assumption if the first QA tests are simply repeating tests that developers have already run or if early QA tests uncover defects in other products that prevent QA from finding defects in the product being tested. For example, an application test may uncover as defects that need to be corrected before the application can be run. Application test hours are accumulated, but defect data is minimal because as defects don't count as part of the application test data. After the as defects are corrected, the remainder of the application test data (after the inflection point in the S-shaped curve) looks like the concave model.
4.2.4.1. Goel and Okumoto NHPP (Goel, 1979)
Non Homogeneous Poisson process models first are proposed by Goel and Okumoto in 1979 which has formed the basis for the failure count models. The idea is that the number of observed faults at time \( t \) is nonlinear because the failure intensity decreases when faults are found and corrected.

In general there are different types of NHPP process. Some of these categorizations are I–NHPP exponential model II– NHPP S-shaped model III– NHPP imperfect debugging model and IV- NHPP S-shaped imperfect debugging model.

Goel and Okumoto NHPP model and Musa exponential model are based on NHPP exponential model and Ohba S-shaped growth model is a model in NHPP S-shaped group. Here we present assumptions of NHPP exponential model which are common in Musa and Goel-Okumoto models.

a. All faults in program are mutually independent from the failure detection point of view.
b. The number of failures detected at any time is proportional to the current number of faults in a program. This means that the probability of the failures for faults actually occurring, i.e., detected is constant.
c. The isolated faults are removed prior to feature test occasions.
d. Each time a software failure occurs, the software error which caused it is immediately removed, and no new errors are introduced.

This is shown in the following differential equation:

\[
\frac{dm(t)}{dt} = b[a - m(t)]
\]
Where $m(t)$ is expected number of errors detected by time $t$ or mean value function. And $a$ is the expected total number of faults that exist in the software before testing and $b$ is the failure detection rate. The solution of the above differential equation is given by

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

This model is known as Goel-Okumoto model. We can see when time goes forward $e^{-bt}$ intends to become zero and $m(t)$ would be equal to $a$. And $a$ is the total number of defects discovered after an infinite amount of testing. Unknown parameters $a$ and $b$ can be estimated by parameter estimation techniques such as Maximum Likelihood Estimation (MLE) or Least Squares Estimation (LSE).

4.2.4.2. Musa exponential (Musa, 1985)

Musa in 1985 presented a model which is similar to Goel-Okumoto model. In this model $m(t)$ is the number of failure discovered as a result of test case runs up to the time of observation. This model presents the differential equation as follows:

$$\lambda(t) = \frac{\partial m(t)}{\partial t} = \frac{c}{nT}[a - m(t)]$$

Where

$a =$ number of failure in the program

$c =$ the testing compression factor

$T =$ mean time to failure at the beginning of the test.

$n =$ total number of failures possible during the maintained life of the program

$t =$ execution time or the total CPU time utilized to complete the test case runs up to a time of observation.

The mentioned equation will be solved as shown in the below:

$$m(t) = a(1-e^{-\frac{ct}{nT}}).$$

The reliability function would be as follows:
\[ R(t) = e^{-a(1-e^{-bt})}. \]

### 4.2.4.3. S-shaped growth models

The NHPP S-shaped model is based on the following assumptions:

i. The error detection rate differs among failure.

ii. Each time a program failure occurs, the software error which caused it is immediately removed, and no new errors are introduced.

This can be shown as the following differential equation:

\[
\frac{dM(t)}{dt} = b(t)[a - M(t)]
\]

Where

- \( a \) = expected total number of faults that exist in the program before testing
- \( m(t) \) = expected number of failures detected at time \( t \)
- \( b(t) \) = failure detection rate (failure intensity of a fault)

This equation can be solved as below:

\[ M(t) = a(1 - e^{-\int b(t) dt}). \]

Based on NHPP S-shaped model, different models are proposed like (Ohba 1984), (Yamada, 1984) and (Pham, 1997). These models have different assumptions and characteristics but all of them have S-shaped curve to model software reliability.

### 4.2.5. Markov Structure Models

A Markov model has a special property that the future behavior of the process depends only on the current state and it is independent of its past history. This type of models is a general way of representing the failure process of software. In these models it is supposed that failures of the modules are independent of each other. This assumption seems reasonable at the module level, because they can be designed, implemented, and tested independently, but may not be true at the system level.
4.2.6. Reliability Growth Models

Reliability growth models estimates the improvement of reliability measure through the testing process. These models represent the failure rate of a system as a function of time or the number of test cases. Two famous models in this group are:

- Coutinho model (Coutinho, 1973)
- Wall and Ferguson model (Wall, 1977)

4.2.6.1. Coutinho model

This model plots the cumulative number of failures discovered in the program and the number of correction actions made versus the cumulative testing weeks on log papers. N(t) shows the cumulative number of failures and t is the total testing time.

The failure rate can be expressed as:

\[ \lambda(t) = \frac{N(t)}{t} = \beta_0 t^{-\beta_1} \]

Where \( \beta_0 \) and \( \beta_1 \) are the model parameters and can be estimated by the least square method.

4.2.6.2. Wall and Ferguson model

This model is presented in 1977 and it is similar to Weibull growth model. This model also predicts the failure rate of software during the software process. The cumulative number of failures at time t, \( m(t) \), can be expressed as:

\[ m(t) = \alpha_0 [b(t)]^{\beta} \]

where \( \alpha_0 \) and \( \beta \) are the unknown parameters. \( b(t) \) is the function to obtain the number of test cases or total testing time.

Failure rate function at time t is equal to

\[ \lambda(t) = m'(t) = \alpha_0 \beta b'(t) [b(t)]^{\beta-1} \]

Wall and Ferguson tested this model using several software failure data and observed that failure data correlate well with the model.
5. Experimental Research

In this section we will present an experiment of comparing some of the presented reliability models on two open source project. We picked two open source projects from sourceforge.net named “MPlayer OS X” and “ClamWin Free Antivirus”. MPlayer OS X is a project based on MPlayer (The Movie Player for Linux) port to Mac OS X platform. This is an open source project and based on Sourceforge.net there exists 5,840,753 downloads. This project is launched in 2002 and it is one of the active projects in this website. The other project is called ClamWin Free Antivirus which is launched in 2004 and it is an antivirus software which includes scheduler, virus database updates, standalone scanner, context menu integration to MS Windows Explorer and Add-in to MS Outlook. We picked this project because it has a high percentage of activity in Sourceforge.net and also it has high number of downloads which is 17,361,767 up to now. The reason we selected the projects with high number of downloads is about a common view within open source community. Eric Raymond [21] states that “with enough eye balls, all bugs are shallow”, and it implicitly suggests a positive relationship between user numbers and bug number. So because we need to study on failure rate of projects we select some projects which are in a steady state of testing and already we have a good knowledge about their failure rate.

We used SMERFS3 [22] as a software reliability modeling tool which is an open source program for doing Statistical Modeling and Estimation of Reliability Functions for Systems. The program allows the user to perform a complete software reliability analysis.

This software supports eleven different models; six using as input data the time between error occurrences and five using the number of detected errors per testing period. The former include: Littlewood and Verrall's Bayesian Model, Moranda's Geometric Model, John Musa's Basic Execution Time Model and his Logarithmic Poisson Model, the Jelinski-Moranda Model, and an adaptation of Goel's Non-Homogeneous Poisson Process (NHPP) Model to time between error data. The latter models include: the Generalized Poisson Model, Goel's NHPP Model, Brooks and Motley's Model, Yamada's S-shaped Growth Model, and Norman Schneidewind's Model.

We gathered the number of failures and the time a failure happened from Sourceforge.net bug tracker section. From this section in sourceforge.net report, as shown in Figure 7, we can calculate the number of detected errors in each time interval.
We selected “one month” as a time interval in reliability models. And we used this information as an input file to SMERFS3 software. Before we use these data we removed some noise which existed in the bug tracker. Some of the bug reports are not real bugs and their status are changed to “Deleted” by developers; as we want to model real failure rate we deleted those bug reports in which their status was changed to “Deleted”.

<table>
<thead>
<tr>
<th>Request ID</th>
<th>Summary</th>
<th>Open Date</th>
<th>Priority</th>
<th>Status</th>
<th>Assigned To</th>
</tr>
</thead>
<tbody>
<tr>
<td>1570891</td>
<td>can't set a playback position in FLV videos</td>
<td>* 2006-10-04 18:43</td>
<td>5</td>
<td>Open</td>
<td>nobody</td>
</tr>
<tr>
<td>1565875</td>
<td>mencoder mp3lame option</td>
<td>* 2006-09-26 17:39</td>
<td>5</td>
<td>Open</td>
<td>nobody</td>
</tr>
<tr>
<td>1565868</td>
<td>mencoder -oac copy</td>
<td>* 2006-09-26 17:28</td>
<td>5</td>
<td>Open</td>
<td>nobody</td>
</tr>
<tr>
<td>1564167</td>
<td>Use second monitor bug</td>
<td>* 2006-09-23 18:16</td>
<td>5</td>
<td>Open</td>
<td>nobody</td>
</tr>
<tr>
<td>1531952</td>
<td>screen resizing</td>
<td>* 2006-07-31 19:37</td>
<td>5</td>
<td>Open</td>
<td>nobody</td>
</tr>
</tbody>
</table>

Figure (7) a part of bug tracker for “MPlayer OS X” project in the Sourceforge.net

We applied these data to some reliability models which use the number of detected errors per testing period. Four reliability models used in this experiment are the Generalized Poisson Model, NHPP Goel model, Yamada’s S-shaped Growth Model, and Norman Schneidewind’s Model. Figure 8 shows the results obtained from modeling process for ClamWin Free Antivirus project.
We can see that just three models, Generalized Poisson Model, NHPP Goel model, Yamada's S-shaped Growth Model could present estimation for failure data. And also calculations show that Yamada’s S-shaped model presents a better fit for this open source project.

Figure 9 presents reliability modeling for another open source project, MPlayer OS X.
Again we can see that the same as the previous project just three models, Generalized Poisson Model, NHPP Goel model, Yamada's S-shaped Growth Model could present a fit for presented failure data. And also calculations show that Yamada’s S-shaped model presents a better fit for this project.

6. Conclusion
In this research we presented some quantitative models in software reliability trend. We put these models in the software reliability models taxonomy. We tried to explain each model and find out their main characteristics. At last we modeled two open source software with some of the presented models. Based on results from these two projects we found that Yamada’s S-shape presented more accurate reliability model for these two projects. Although we cannot guarantee that this model is best for all open source projects but based on failure data from these two projects, Yamada’s S-shaped did a better job rather than the others NHPP Concave models. The results can be interpreted using the nature of open source projects which almost there are not a complete testing phase for these types of projects. Bugs can be detected after release and most of these projects have a pick in number of bugs after some
intervals after release. So rather than commercial software that we expect a smooth decreasing in number of faults, in open source projects we may encounter with a pick in number of failures after releases.

7. References

8. “Software Reliability”, Hoang Pham, 2000, Published by Springer-Verlag Singapore.


