CHAPTER 1

INTRODUCTION

1.1 RELIABILITY

Software has become an essential part of industry, military systems, medical systems, and many other commercial systems. The application of software in many systems has led software reliability to be an important research area. Researchers worked to increase the chance that the software systems will perform satisfactorily. This process required the removal of faults during the testing phase. Software reliability deals with probabilistic methods applied to the analysis of random occurrences of failures in a software system.

In this study, the statistical description is concerned with the time-intervals between failures. Thus, software reliability can be defined as the probability that no failure occurs up to time ‘t’. A failure is the departure of software behavior from the user requirements. This phenomenon must be distinguished from the fault in the software code which causes the occurrence of failure as soon as it is activated during program execution. If each time after a failure has been experienced, the underlying fault is detected and fixed correctly, then the reliability of software will improve with time. As failure occurrences initiate the removal of faults, engineers reported failure times and time between failures (TBF). Both have been used to find the mean time between failures (MTBF), which is then used to investigate the reliability growth.

The software may fail when delivered to the end users, consequently increasing costs. To minimize the risk of the software failure, one should continuously verify and validate the software through each stage of the software development process. The failure intensity (FI) and the mean time to failure (MTTF) are two alternative ways of expressing software reliability. The FI is the expected number of failures per unit time while the MTTF is the expected value of the failure interval. The principal objective of a software reliability model is to forecast failure behavior that will be experienced by the time the program is operational. This expected behavior changes rapidly and can be tracked during the
periods in which the program is tested. In general, reliability improves within time as the failure intensity decreases. Previous research discusses various techniques used to determine software reliability. No matter how simple or complex a software program is, it is widely recognized that 100% percent reliability is impossible to obtain.

A software system is a repairable system. While software is being developed and tested, programmers detect and correct failures. After the corrections are made, programmers check the software again until another failure is observed. They continue this process until they have a reliable system. Before they put the system in the market, programmers would like to be sure that they have a desirable reliable system which rejects the quality and performance of the final design. This process of testing leads to the growth of the software system and maintains its reliability. For repairable systems, they observe failure times, let $T_1 < T_2 < \ldots < T_n$ denote the failure times of the software system, that is the time since the initial startup of the system operation. The times between failures will be denoted $Y_1, Y_2, \ldots$, and assigned the following: $Y_1 = T_1, Y_2 = T_2 - T_1; Y_3 = T_3 - T_2, \ldots$.

A repairable system is deteriorating if the times between failures tend to get shorter with advancing time and is improving or growing if the times between failures tend to increase.

**Hardware and Software Reliability**

It should be noted that software and hardware systems are different. Their environmental conditions vary, and their failure causes and failure consequences are dissimilar. However, the probabilistic definitions are identical and the theories of probability and statistics are also similar. After most engineering products have been completed, tested, and sold, it is expected that the products work reliably. But with software products, it is possible that one discovers that it has major bugs and the software system does not work reliably. It is possible that some software works well with one user while it does not work well with another user. The most competent programmers in the world cannot avoid error codes completely in the software products; it is not the maturity of the methods and tools used by software
professionals that make software production so critical. Rather, it is the conceptual complexity of software that makes software unreliable. While software products are less reliable than other engineering products, many real world applications depend on these products and their required quality level.

1. Failures are primarily due to design faults in the software. Modifying the design can make it robust.

2. No wear phenomena in the software. Software errors occur without previous warning. Old codes can exhibit an increasing failure rate as a function of errors induced while making upgrades.

3. Failures occur when the logic path that contains an error is executed. Reliability growth observed as errors in the software can be detected and corrected.

4. External environmental conditions do not affect the software reliability, while the internal environmental conditions affect the reliability, these internal conditions are insufficient memory and inappropriate clock speeds.

5. We can improve reliability by debugging and increasing the read access memory.

6. Reliability can be improved by diversity. In other words; making the software work with different systems.

7. Software interfaces are not visual but are conceptual.

8. Software design does not use standard components; it depends on the qualifications of a programmer.

**Fundamentals of Reliability**

Software Reliability Modeling plays an important role in developing software systems and enhancing computer software. In general, software reliability models fall into two categories depending on the operating domain. The most popular category of models depends on time, whose main feature is that probability measures, such as the mean time between failures and the failure
intensity function depend on failure time. The second category of software reliability models measures reliability as the ratio of successful runs to the total number of runs. Because the amount of current research is devoted to time-based models, the first category will be considered. The time domain employs two approaches, the observed time between failures and the number of discovered failures per time period. In this thesis, we will follow the time between failures. This section introduces some important terms, concepts, and notations which are frequently used.

Software reliability models serve to aid the software engineer by indicating the likelihood of system operation over a given time interval according to the stated specifications. Software reliability predictions can be used to judge the quality of a program. It is important to define the key terms in developing and describing software reliability models. The following definitions are used to distinguish between failures and faults throughout the thesis (Musa et al., 1987).

A **failure** is a departure from how software should behave during operation according to the requirements.

A **fault** is a defect in a program, that when executed causes failure(s).

The study of software reliability can be categorized into three parts: modeling, measurement and improvement. There are two general schemes to demonstrate whether the required software reliability has been achieved or not. They are formal verification and statistical testing.

Since 1970, scientists and engineers have been developing different models to analyze failure data in order to improve software reliability. These successive software failure times are assumed to be statistically independent.

It is now well-recognized that reliability growth models are better described by a Non-Homogeneous Poisson Process (NHPP). A Power Law Process (PLP) is a special case of a NHPP. Horigome et al. (1984) proposed a PLP model for describing the reliability growth, which relates the lifetimes of one stage to the next. The power model by Crow (1974) has closed form expressions for tail area probabilities and simple formulas for intensity functions.
Suppose that a repairable system is observed until \( n \) failure times \( t_1, t_2, \ldots, t_n \) occur, where \( 0 < t_1 < t_2 < \ldots < t_n \). Let \( T > 0 \) be the random variable representing the time to next failure. The reliability function \( R(t) \) is the probability that a system will achieve its mission through time \( t \). In other words, the probability of no failure occurs up to time \( t \) and is expressed by

\[
R(t) = P(T > t) = \int_t^\infty f(x)\,dx \quad t > 0,
\]

where \( f(t) \) is the probability density function (pdf) of the failure time \( T > 0 \). The cumulative distribution function (cdf) of the random variable \( T \) can be written in terms of \( R(t) \) as follows:

\[
F(t) = \int_0^t f(x)\,dx
\]

The reliability function is also called the survival function of \( T \). \( R(t) \) decreases from 1 to 0 as \( t = 0 \) to \( t = 1 \).

The probability density function \( f(t) \), the cumulative distribution function \( F(t) \), the reliability function \( R(t) \) and the failure rate function \( \lambda(t) \) are closely related to each other.

The MTBF is the expected interval length from the current failure time, say \( T_n = t_n \), to the next failure time \( T_{n+1} = t_{n+1} \). Let \( f(t | t_1, t_2, \ldots, t_n) \) denote the conditional distribution of failure time \( T_{n+1} \) given \( T_1 = t_1, T_2 = t_2, \ldots, T_n = t_n \), then the MTBF is defined by:

\[
MTBF = \int_{t_n}^\infty f(t | t_1, t_2, \ldots, t_n)\,dt - t_n
\]

A NHPP is described by the failure intensity function, which is denoted by \( \lambda(t) \).

If the intensity function has the form: 

\[
\lambda(t) = abt^{b-1}; \quad t > 0, a > 0, b > 0,
\]

then the process is called the power law process (PLP), where ‘a’ is the scale parameter and ‘b’ is the shape parameter of PLP (Rigdon, 1989 and 1998) and (Suresh,
This process is also called power law process as the intensity is the power of time.

The power law process is a special case of NHPP. The model \( \lambda(t) \) demonstrates whether a software system is improving or deteriorating when one chooses the appropriate parameters. When \( b > 1 \), the failure intensity increases at an exponential rate with time, and the PLP models the reliability of a repairable system with rapid deterioration. While, if \( b < 1 \), the intensity function is strictly decreasing. This corresponds to modeling the reliability of a repairable system with rapid improvement. For the PLP, when \( b = 1 \), mean time between failures is equal to a constant value.

The PLP has proved to be useful in reliability modeling for several reasons.

1. It can be used to model deteriorating systems as well as to model improving systems.

2. Duane, (Rigdon and Basu. 1989) showed that the failure data of many systems used at General Electric fit a model that is closely related to PLP, and Statistical inference procedures can be used easily and applied to PLP models.

There are two statistical descriptions, namely:

1. Time-interval between failures.

2. Number of failures experienced in a given period.

The PLP model can be applied in software reliability although some problems may arise, it has been used in many successful applications, particularly in the defense industry. To model the failure rate of repairable systems, the PLP model is adopted by the United States Army Materials System Analysis Activity and called the Crow-AMSAA model. Much theoretical work describing the PLP model was performed by Lee and Lee (1978), and Engelhardt and Bain (1986). The PLP model has been widely used in reliability growth, in repairable systems, and software reliability models (Ascher and Feingold, 1984).
1.2. APPLICATION OF SPC TO SOFTWARE

The influence of software technologies on our daily life has grown exponentially during the last two decades. From a simple digital clock to telecommunication networks, software is extensively used to ease our way of living. Thus software can be regarded as “the gate to future” as it provides the basis for much of the technological advance in this century.

If we look at the history of software at a glance, we see that there is a rapid improvement in this sector since 1960s. However, software industry is quite young compared to manufacturing industries. After the industrial revolution, the manufacturing industry has come up to a level of stability, and the idea of continuous process improvement has been accepted throughout the world. In software, however, the improvement trend is too steep and organizations still strive for achieving a high level of maturity. Moreover, the characteristics of software make it complex and invisible, which make it difficult to apply the practices that are in use in other industries.

One of these common practices in manufacturing industry is statistical process control. The investigation on quantitative mechanisms as an aid to control process variation gave rise to the application of SPC since 1930s. The idea of applying SPC to software development, however, is brought mainly by Capability Maturity Model (CMM) in mid 90s. Although its benefits are accepted for manufacturing companies, there have been many debates about its application in software development (Kan, 1995) and (Lantzy, 1992).

1.2.1 Difficulties and Benefits of SPC in Software Development

As Card (1994) points out, SPC is founded on the principle that a process will demonstrate consistent results unless it is performed inconsistently. Thus, we can define control limits for a consistent process and check new process outputs in order to determine whether there is a discrepancy or not.

In the manufacturing arena, it is not difficult to figure out the relationship between product quality and the corresponding production process. Therefore we can measure process attributes, work on them, improve according to the results
and produce high quality products. There is a repetitive fabrication of the same products in high numbers and this brings an opportunity to obtain high sample size for the measured attributes. Moreover, the product is concrete, and the attributes and variables to be measured are easily defined. Consequently, the only difficulty left is to define correct attributes and collect data for utilizing the tools of Statistical Process Control.

On the other hand, software product is difficult to characterize. As it is not concrete, it is difficult to recognize the correlation between a single software process and the quality of the related software product. Crosby (1980) defines quality as conformance to customer requirements. In addition, Lantzy (1992) indicates that a software process must be assessed based on its capability to develop software that is consistent with user’s requirements. Actually, there is no specific software measure showing the extent to which customer requirements are met. However, there are processes and products that influence production life cycle. By measuring specific characteristics of these processes and products, we can have an idea about the quality of the final product.

One other difficulty in software production is that, there is not a repetitive production. Each product is distinct and possesses different characteristics. For this reason, it is usually not possible to form a sample of n measurements. In this case, we shall take care of each single measurement, and perform the analysis accordingly. Beside these general software process characteristics, software measurement is also a very complicated process. Each metric requires different measurement techniques and the reliability of the metric data depends on how well the metric is defined, how properly the data collection procedures are performed and how robust the measures are with respect to varying environmental conditions. As the metrics are usually abstract, the interpretation of the data necessitates good judgment. Since the metric data are collected by regular project individuals a firm understanding of metrics should be provided for data validity.

Despite the mentioned difficulties, it may still be possible to apply Statistical Process Control to software processes. However, special consideration shall be given to the establishment of relevant mechanisms to gather beneficial outcomes. First of all, the key processes that need statistical control should be
identified. Not every process needs statistical control, nor is related to organizational goals and process performance objectives. The variability may be inherently existent in the process, undermining the idea of using SPC. Moreover, it may not be economically and technically feasible to apply statistical control to some of the processes. SPC necessitates process stability for meaningful data analysis; thus the selected processes should have well-prepared process definitions.

Secondly, relevant measures for the selected processes should be defined. These measures should be informative about the desired process/product characteristics. For this reason, the relationship between the measure and the process/product it represents should be clearly understood. Moreover, it is critical to provide repeatability in measurement practices and consistency in the measured data. Therefore, relevant data collection mechanisms should be described in detail; and manual or automated means for data collection should be constructed. A database should also be created to accommodate the data as a baseline for the derivation of control limits within the organization.

Most of the times, the collected measure cannot be used directly in SPC analysis as some inherent parameters influence the generation of the data. Thus, it is essential to normalize the raw measures. These normalization procedures should be described in order to provide a consistent basis for meaningful comparison among different process instances. Meanwhile, relevant statistical techniques, control chart categories, and interpretation approaches should be identified. Finally, these techniques should be utilized first to find and correct special causes, and then to visualize and improve processes.

If supported with reasonable data, necessary infrastructure and human resources, Statistical Process Control can be appropriately applied to gather beneficial information for software processes. First of all, the processes can be monitored with the establishment of process capability baseline and the process outcomes become predictable. Unpredicted behavior can be detected and necessary corrective actions can be implemented on time. As the process describes the way a product is produced, the quality of the product can be deduced from the related process and this enables the organization to produce high quality
products by controlling the processes. After stability is achieved and special causes are removed, process improvement activities can be tracked through the analysis of common causes. The results of improvement actions can also be evaluated with the comparison of past and current conditions statistically.

1.2.2 Utilization of SPC in Software

The interest to apply SPC techniques in the software industry has been growing during the last decade as more organizations advance in maturity levels of process improvement models such as Capability Maturity Model (CMM) (Paulk et al. 1993), Capability Maturity Model Integration (CMMI) (CMMI,2001) and SPICE (Information technology, 1998 and ISO/IEC, 1998). These models implicitly direct software companies to implement SPC as a crucial step for achieving higher process maturity levels. They suggest control charts for both project level process control and organizational level process improvement purposes. In the literature, there are several resources on the usage of statistical techniques in software development (Jakolte and Saxena, 2002), (Radice, 1998), (Romine, 2002) and (Florac et al., 2000). Some researchers contribute to this trend by providing approaches to utilize SPC techniques for software industry. Moreover, most of the examples (Humphrey, 1989) exhibited in the studies refer to defect and inspection data. These studies however, focus on the potential benefits of SPC implementation, rather than providing a satisfactory guideline for software firms to implement SPC techniques with convincing information. We specifically lack knowledge on the applicability of different metrics, the means of reliable data collection mechanisms, meaningful analysis approaches and practical evidence.

In this study, we investigated how SPC can be effectively applied to the processes of a software firm by using its existing measures. As control chart is one of the most sophisticated data analysis tools within SPC, we demonstrated practical evidence on the utilization of SPC via control charts.

1.2.3 Statistical Process Control

Statistical Process Control (SPC) is a methodology that aims to provide process control in statistical terms. Since the great industrial revolution in Japan,
SPC has been widely used in manufacturing industries in order to control variability and improve processes (Sutherland et al. 1992). The basic tools used for statistical control are: Check Sheet, Cause-and-Effect Diagram, Scatter Diagram, Run Chart, Histogram, Bar Chart, Pareto Chart, and Control Chart (Ishikawa, 1982).

1.2.3.1 Shewhart Control Charts

During his studies at Bell Labs in 1920s, Shewhart proposed that it is possible to define limits within which the results of routine efforts must lie to be economical. Deviations in the process outcomes resulting in values out of these limits indicate that the process is not performed economically. In order to detect assignable causes, Shewhart utilized statistics and control charts.

Shewhart control chart model depends on hypothesis testing. First of all, a sample of data is collected for the subject measure. Then, its mean and variance are calculated. The lower and upper control limits are derived and data is analyzed using the statistical evidence on hand. By analyzing the data values with respect to upper and lower control limits together with their location in the zones, assignable causes are detected. Then necessary actions are taken and measurements are repeated. The charts are redrawn with the existing data values, and this process is repeated until no evidence remains for the existence of assignable causes. Once the process is brought under control, further improvement activities are implemented to minimize the effect of common causes.

The measurement can be performed by means of either variables or attributes. Burr and Owen (1996) define a variable as “measure of a product that can have any value between the limits of the measurement”, while an attribute as “count of things which may or may not be present in the product”. The nature of these two measurement categories necessitates different statistical analyses. A variable normally has normal probabilistic distribution, whereas it is likely to be binomial for an attribute. Control Charts are sophisticated statistical data analysis tools, which include upper and lower limits to detect any outliers. They are frequently used in SPC analysis and will be described in detail in the following sections.
1.2.3.2 Concept of Variability in Processes

In 1920s, W. A. Shewhart was working on the idea of quality control and he brought the idea that each process is driven by forces of variation (Shewhart, 1939). However, variation results in loss of quality by causing inefficiency and waste. If we can understand the sources of variation, we can take necessary actions to remove inefficiency, and increase quality.

If we think of variation in software industry, source lines of code produced a day can be considered as a variable parameter. Either if the same person produces SLOC for the same component, the amount of time he spends will be different from one day to the other. This can be explained as the variation in a process attribute.

According to Shewhart, variation in a process has two types of causes: assignable causes and chance causes. Assignable causes appear in unexpected periods and can be fixed by immediate actions. For instance, if a new tool is being used for coding, the productivity of the coder may be lower during adaptation period. When this is realized, a training program can be implemented to improve productivity.

On the other hand, chance causes are the results of the system itself. They are naturally existent within the defined processes and can only be avoided by performing improvement programs. If we think of a software engineering firm that has no reusable code library, we witness that similar code pieces are written separately in each new application and this causes delays due to rework. Such a chance cause can be prevented by creating a reusable code library.

1.2.3.3 Statistical Control

If the variation in the behavior of a process is predictable in statistical terms, that process is said to be in control. This means that, we can expect what the outcome will be the next time we perform the same process. In this way, we can prepare more accurate project plans, do better cost estimations and schedule activities in more reasonable basis.
In order to calculate the variance in process behavior, several attributes or variables representing the outcomes of the process shall be defined. The number of defects found during unit testing, the number of requirements that are changed after requirements analysis phase, amount of CPU utilized to perform a specific application may all be used to understand the behavior of the processes they represent. The variability in process behavior, then, can be tracked through these measures.

The aim of Statistical Process Control is: firstly to detect assignable causes of variation in the processes and provide process control; secondly to enable monitoring of the improvement in processes by demonstrating the chance causes; and Shewhart Control Charts are good means to achieve Statistical Process Control.

SPC charts help to judge the state of the process correctly through the use of properly calculated control limits. They are based on the distribution of the rationally sampled observations and the cost of type I and II decision errors. The control limits for the chart are defined in such a manner that the process is considered to be out of control when the time to observe exactly one failure is less than $LCL$ or greater than $UCL$. A type I decision error (i.e. false positive) is made when the process is judged to be out-of-control when in fact it is in-control. A type II decision error (i.e. false negative) is made when the process is judged to be in-control when in fact it is out-of-control (Vanbrackle and Williamson, 1999; Naikan, 2008). Our aim is to monitor the failure process and detect any change of the intensity parameter. When the process is normal, there is a chance for this to happen and it is commonly known as false alarm. The traditional false alarm probability is to set to be 0.27% (Jacob and Sreejith, 2008) although any other false alarm probability can be used. The actual acceptable false alarm probability should in fact depend on the actual product or process (Gokhale and Trivedi, 1998). Both types of errors are typically associated with economic loss.
1.3 ORDER STATISTICS

To improve and understand the logic behind process control methods, it is necessary to give some thought to the behavior of sampling. If the length of a single failure interval is measured, it is clear that occasionally a length will be found which is towards one end of the tails of the process’s normal distribution. This occurrence may lead to the wrong conclusion that the process requires adjustment. If a sample of four or five is taken, it is extremely unlikely that all four or five failure interval lengths will lie towards one extreme end of the distribution. If we take the average or length of four or five failure intervals, we shall have a much more reliable indicator of the state of the process. Any change in the process mean, unless it is extremely large, will be difficult to detect from individual results alone. A large number of individual readings are necessary before such a change was confirmed.

The distribution of sample means reveals the change much quicker than individuals. Therefore, on a chart for sample means, plotted against time, the change in level would be revealed almost immediately. For this reason sample means rather than individual values are used to control the centering of processes. This provides a sound basis for the Mean Value Control Chart.

A subgroup or a sample is a small set of observations on a process parameter or its output, taken together in time. The size and the frequency of sampling are the two major problems in choosing a subgroup. The smaller the subgroup, there is less opportunity for variation within it. The larger the sample size the narrower the distribution of the means and they become more sensitive to detect change (Oakland, 2008).

It is understood that, in any type of process control charting system, careful selection of subgroups is very important. The software failure data is in the form of <failure number, failure time>. By grouping a fixed number of data into one, the noise values may compensate each other for that period and thus the noise inherent in the failure data is reduced to great extent (Malaiya et al., 1990).
1.4 SPRT

The SPRT was initially developed for situations in which there is a random sample of a variable with a discrete or continuous distribution with one parameter variable and two simple hypotheses on the value of that parameter. The sequential approach is often used in many applications and quality control, because of two reasons. 1) For some observation schemes it is natural to follow the sequential approach to construct a probability model. 2) Sequential statistical procedures have some optimal properties, e.g. SPRT minimizes the expected sample size (Ghosh, 1970). Sequential analysis was first developed by Abraham Wald in the 1940’s and (Wald, 1945; Wald and Wolfowitz, 1948) and introduced the SPRT during that time. The SPRT was initially developed by Wald (1947) for quality control problems during World War II. Neyman and Pearson (1933) result inspired Wald to reformulate it as a sequential analysis problem. The original development of the SPRT is used as a statistical device to decide which of two simple hypotheses is more correct. It has been formulated for use in the computerized testing of human examinees as a termination criterion. The properties of it have been studied intensively by many researchers since Wald (1947). The likelihood based SPRT proposed by Wald is very general in that it can be used for many different probability distributions.

1.5. NHPP SRG MODELS

This section discusses stochastic reliability models for the software failure phenomenon based on NHPP. A NHPP is a realistic model for assessing software reliability and has a very interesting and useful interpretation in debugging and testing the software. There are two main types of software reliability models: the deterministic and the probabilistic. Performance measures of the deterministic type are obtained by analyzing the program texture and do not involve any random event. Two well-known models are: McCabe’s Cyclomatic complexity metric (McCabe, 1976) and Halstead’s software metric (Halstead, 1977). The probabilistic model represents the failure occurrences and the fault removals as probabilistic events. The probabilistic software reliability models can be classified into different groups (Pham, 2000) such as, Error seeding, Curve fitting, Failure rate, Reliability growth, Markov structure, Time-series and NHPP. In this thesis,
NHPP type of software reliability models and methods for estimating software reliability are used.

A process which develops in time in accordance with some probabilistic laws is called a stochastic process. They are used for the description of a system's operation over time. The two main types of stochastic processes are continuous and discrete. Counting processes in reliability engineering are widely used to describe the appearance of events in time, e.g., failures, number of perfect repairs, etc. The simplest counting process is a Poisson process. Poisson-type models assume that the number of failures detected within distinct time intervals is independent.

1) Homogeneous Poisson Process (HPP): with the same rate of failure.

2) Non-Homogeneous Poisson process (NHPP): with a varying rate of failure.

The Non-Homogeneous Poisson Process (NHPP) is a poisson process whose intensity function is non-constant. For more details about the NHPP theory, we refer the reader to Xie (1991) and Singpurwalla and Wilson (1999). In this section, we present briefly an introduction and present briefly some existing software reliability models for the failure process that are described by the NHPP. NHPP models have been studied and used successfully in hardware reliability systems. It describes failure processes which have certain trends such as reliability growth or deterioration. The applications of NHPP models have been implemented to software reliability. The cumulative number of failures up to time t, N(t), can be described by NHPP. Many software reliability models belong to this category. The Poisson process model for describing the uncertainty of the counting process \( N(t); t \geq 0 \) is the simplest of counting process models. The counting process modeled by NHPP, where N(t) follows a Poisson distribution with parameter m(t), which is the mean value function. The probability that N(t) is an integer is denoted by:

\[
P(N(t) = n) = \frac{(m(t))^n}{n!} e^{-m(t)}, \quad n = 0,1,2,...
\]
Where \( m(t) \) is the mean residual time or expected cumulative number of failures in \([0, t)\), the assumptions of NHPP are:

- \( N(0) = 0 \)
- \( (N(t), t \geq 0) \) has in dependent increments
- \( P(N(t+\Delta t) - N(t) = 1) = \lambda(t) + o(\Delta t) \)
- \( P(N(t+\Delta t) - N(t) \geq 2) = o(\Delta t) \)

Where \( o(\Delta t) \) approaches zero for small \( \Delta t \).

The mean value function is:

\[
m(t) = E(N(T)) = \int_0^t \lambda(x)\,dx
\]

If \( m(t) \) is known, the the failure intensity \( \lambda(t) \) is:

\[
\lambda(t) = \frac{d}{dt} m(t)
\]

If \( \lambda(t) \) is constant, then we have a Homogeneous Poisson Process (HPP).

In 1975, Schneidewind was the first to suggest the NHPP model. However, in 1979, Goel and Okumoto was the first who presented a simple model for the software failure process. He assumed that the cumulative failure is a NHPP with a simple mean value function. Later, the Goel-Okumoto model became very well known among software reliability. Goel and Okumoto proposed the time dependent failure rate model based on NHPP. Ohba and Yamada (1984) proposed some particular NHPP models such as the delayed S-Shaped software reliability models, and the inflection S-Shaped model. Musa and Okumoto (1986) proposed the Logarithmic Poisson execution time model. Littlewood assumed a modification of the Duane model based on NHPP. Kapur and Garg (1991) used a modified G-O model by introducing the concept of imperfect debugging.

The NHPP models can be further classified as finite failure and infinite failure models.
Finite failure NHPP models

They assume that the expected number of failures during an infinite amount of time will be a finite number ‘a’. Some of the key models in this class are described below.

➢ Goel-Okumoto NHPP Model

The Goel-Okumoto model is considered to be one of the most influential NHPP based software reliability models. Its mean value function, \( m(t) \), and the failure intensity, \( \lambda(t) \), are given by

\[
m(t) = a \left(1 - e^{-bt}\right)
\]

and

\[
\lambda(t) = ab e^{-bt}
\]

Where ‘b’ is the failure occurrence rate per fault.

➢ Delayed S-Shaped NHPP model

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

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

and

\[
\lambda(t) = b^2 t e^{-bt}
\]

Where ‘b’ is the fault removal (failure detection and fault isolation) parameter.
**Inflection S-shaped NHPP model**

The inflection S-shaped model was proposed to analyze the software failure detection process where the faults in a program are mutually dependent. The mean value function, \( m(t) \), and the failure intensity, \( \lambda(t) \), are given by

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

and

\[
\lambda(t) = a \frac{be^{-bt}(1+\beta)}{(1+\beta e^{-bt})^2}
\]

Where ‘b’ is the failure detection rate and ‘\( \beta \)’ is the inflection parameter.

**Pareto NHPP Model**

This model assumes the failure occurrence rate per fault to be pareto. The mean value function, \( m(t) \), and the failure intensity, \( \lambda(t) \), are given by

\[
m(t) = a \left[ 1 - \left( 1 + \frac{t}{\beta} \right)^{-\alpha} \right] \quad \alpha, \beta > 0
\]

and

\[
\lambda(t) = \frac{a\alpha}{\beta} \left( 1 + \frac{t}{\beta} \right)^{-\alpha-1} \quad \alpha, \beta > 0
\]

**Infinite failure NHPP models**

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

**Logarithmic Poisson Execution Time model**

This model assumes the number of failures experienced by time ‘t’ (t is the execution time), is NHPP with the mean value function \( m(t) \) given by
\[ m(t) = \frac{1}{\theta} \ln \left( \lambda_0 \theta t + 1 \right) \]

Where \( \lambda_0 \) denotes the initial failure intensity, and \( \theta > 0 \), the failure decay parameter. The failure intensity function is given by

\[ \lambda(t) = \frac{\lambda_0}{(\lambda_0 \theta t + 1)} \]

Power Model

The power model was developed by Crow in 1974 as a model for hardware reliability. This model has the ability to be applied for the prediction of software reliability as well.

The mean value function is:

\[ m(t; a, b) = at^b \]

The failure intensity function is:

\[ \lambda(t; a, b) = abt^{b-1} \]

If \( b < 1 \), then the software reliability improves.

The main advantage of the Duane reliability growth model is the graphic representation of the cumulative number of failures versus the cumulative failure time on a log-log scale sheet. If the trend of the plotted graph is close to a straight line, then the model is valid. The main disadvantages of the Duane model are Rate of occurrence of failure becomes zero at time infinity and becomes infinite at time zero. Littlewood modified the Duane model by assuming the mean value function of the modified Duane model as:

\[ m(t) = k \left( 1 - \left( \frac{\alpha}{\alpha + 1} \right)^\beta \right) \quad \alpha > 0, \beta > 0, k > 0 \]

Where ‘k’ represents the number of failures to be detected.
Software Reliability Growth Model

A software is said to contain a fault if, for input data, the output result is incorrect. A fault is always an existing part in software codes. Therefore, the process of software debugging is a fundamental task of the life cycle of a software system. During this period, the software program is tested many times with the intent of discovering faults contained. When a failure is observed, the code is inspected to find the fault which caused the software failure. The fault is usually removed by correcting the software codes. As a result, one expects the software reliability to increase during the testing phase as more and more faults are removed. The reliability improvement phenomenon is called reliability growth. The size and the complexity of the software packages make it impossible to find and correct all existing faults. The best thing is to give software a reliability requirement and to try to attain a goal by testing the software and correcting the detected faults. However, obtaining the required software reliability is not an easy task. Thus, high reliability is usually estimated by using appropriate models applied on failure data from the software failure history. A Software reliability model is a mathematical description of the debugging and fixing process built in the following three different stages:

1. Model structure is selected.
2. The free parameters in the model are tuned on the basis of the experimental data.
3. A rule is given to use the estimated model for predictive purposes.

A software reliability model falls into two categories that depend on the operating domain. Thus, the most popular models are based on time. Their main feature of reliability measures, such as the failure intensity which is derived as a function of time. The second kind of software reliability models have a different approach. This approach is made by using operational inputs as their main features, which measure reliability as the ratio of successful runs to total runs. The second approach has some problems such as: many systems have runs of large lengths with output measures that are incompatible with the time-based measures. Due to these problems, the work of this dissertation has been devoted to time-
domain models. The time domain model employs either the observed time between failures or the number of discovered failures per time period. Thus, these two procedures were developed to estimate the model parameters from either failure count data or time between failures. Therefore, software reliability modeling and estimation can be grouped into two categories of general applicability:

1. Failure counting description (FCD).

2. Failure interval description (FID).

1.6. PARAMETER ESTIMATION METHODS

Two common ways for estimating the function’s parameters from data are the maximum likelihood and regression methods. The maximum likelihood technique consists of solving a set of simultaneous equations for parameter values. The equations define parameter values that maximize the likelihood that the observed data came from a distribution with those parameter values. Maximum likelihood estimation satisfies a number of important statistical conditions for an optimal estimator and is generally considered to be the best statistical estimator for large sample sizes. Unfortunately, the set of simultaneous equations it defines are very complex and usually have to be solved numerically. Usually numerically Parameter estimates may also be obtained using non-linear regression. This approach fits the curve to the data and estimates the parameters from the best fit. where, fit is defined as the difference between the data and the curve function fitting the data. For a general discussion of maximum likelihood theory and equation derivation, see (Mood et. al, 1974) and (Musa,1987).

Parameter estimation is of primary importance in software reliability estimation. The parameter estimation methods can be of two types 1) Point estimation 2) Interval estimation (Pham, 2006). Two most popular estimation techniques are MLE and Least Squares Estimation (LSE). The MLE technique estimates parameters by solving a set of simultaneous equations. It is the most widely used estimation technique. In many cases, the maximum likelihood
estimators are consistent and asymptotically normally distributed as the sample size increases (Zhao and Xie, 1996). In this thesis, the parameters are estimated by MLE technique for Time domain data.

**Maximum Likelihood Estimation Method**

The method of MLE is one of the most useful techniques for deriving point estimators. The idea behind maximum likelihood parameter estimation is to determine the parameters that maximize the probability of the sample data. A MLE method is versatile and applies to many models and to different types of data. Although the methodology for maximum likelihood estimation is simple, the implementation is mathematically intense.

1.7. **DATA ANALYSIS.**

Depending on the format in which test data are available, there are two common types of failure data: time-domain (i.e. ungrouped) data and interval-domain (i.e. grouped) data (Pham, 2006). The time-domain approach involves recording the individual times at which failure occurred. The interval-domain approach is characterized by counting the number of failures occurring during a fixed period (e.g., hour, week, day). These data are usually used by practitioners when analyzing, assessing and predicting reliability applications. Some software reliability models can handle both types of data.

**Type 1 Data: Time Domain Data**

Assuming that the data are given for the occurrence times of the failures or the times of successive failures, \( s_j \) for \( j = 1, 2, ..., n \). Given that the data provide \( n \) successive times of observed failures \( s_j \) for \( 0 \leq s_1 \leq s_2 \leq \cdots \leq s_n \), we can convert these data into the time between failures \( t_i \) where \( t_i = s_i - s_{i-1} \) for \( i = 1, 2, ..., n \). Given the recorded data on the time of failures, the Log Likelihood Function (LLF) takes on the following form:

\[
LLF = \sum_{i=1}^{n} \log [\lambda(t_i)] - m(t_i) \tag{1.7.1}
\]
**Type 2 Data: Interval Domain Data**

Assuming that the data are given for the cumulative number of detected errors $y_i$ in a given time-interval $(0, t_i)$ where $i = 1, 2, ..., n$ and $0 < t_1 < t_2 < ... < t_n$, then the LLF takes on the following form:

$$LLF = \sum_{i=1}^{n} (y_i - y_{i-1}) \log[m(t_i) - m(t_{i-1})] - m(t_i)$$

(1.7.2)