Chapter-1: General Introduction

1.1 Motivation

Software has become significant both for industrial and scientific purposes. A primary objective in the software design is to minimize the number of software errors and increase the quality and reliability of the same. The software quality management is the process of managing the quality of software production successfully. It involves the continuous process of updating and enhancing the old softwares by releasing new versions. Many attempts have been made to define software quality by several software developers. Several commercial softwares suffer from significant design and implementation security vulnerabilities. A dilemma that numerous software organizations focus is to determine whether quality and quality programs should become part of the organizational strategy. Obviously, implementation of quality programs in software engineering organizations is of vital importance. The software management deals with coordinating, motivating, and guiding the professionals who are linked with various tasks on the related software project.

In recent years, many softwares have been developed in safety-critical applications in various fields such as medicine, transportation, nuclear energy, emergency operation theatre, etc.. To improve the software quality and reliability, the software reliability engineering (SRE) plays an important role in many aspects throughout the software life cycle. For example, to estimate the accurate reliability, it is recommended to verify the quality of the software before releasing the software in the market for operational purpose. Highly reliable software can be built through careful design and analysis under techno-economic constraints. The revolutionary social and technical changes that have occurred during last few decades, have contributed to the need for a quantitative treatment of the reliability of the system software. This approach has led to the rise in the literature on software reliability engineering (SRE) which is a scientific discipline focusing on safety, quality and reliability analysis. Software reliability provides the quantification of the problem of the failure of the software.
Fig. 1.1: Feedback loop for software development models
Chapter-1: General Introduction

The existing models demand initially the software developers in three different guises namely requirements & specification, design and coding as shown in fig.1.1. These three factors formulate the test plan and then after this test plan is tested. The testing goes according to the user’s need. When the software is tested perfectly and developer is satisfied, then software is released in the market for the customer’s use. High level quality software can be achieved by using well-established software reliability engineering methodologies. In the field of the software reliability engineering, many models have been developed to study the reliability improvement of software during the testing and operational phase. During the testing phase of the software development life cycle, faults are detected and removed and hence reliability of that software is increased. **Software reliability growth models (SRGMs)** are the mathematical tools which are used to develop the test cases, to detect the faults remaining in the software and to predict the reliability of the software during testing and operational phase. Various characteristics of SRGMs considering perfect and imperfect debugging, change point and error generation are also feasible to discuss in real time frames so as to suggest proper course of action. It is anticipated that the work done in the area of software reliability in the present thesis would be helpful to the software developers to attain a extensive understanding of the dynamics of software reliability growth model which would bring about endurable and cost effective designs that would address all maintainability and quality features.

1.2 Software Reliability

**Software reliability** is the probability that the software will not cause the failure for a product for specified time under specified conditions; this probability is a function of the inputs and use of the product, as well as a function of the existence of faults in the software. The studies on this subject were started in late 1960’s and now the same are growing exponentially. Software reliability is widely cited by many users and developers as one of the most important features of the software product. The estimation of the software reliability provides a quantitative assessment of the software failure before the system tests begin or at any point throughout. The measures of software reliability can be used for planning and controlling testing resources during software development. This also gives us confidence about the correctness of the completed software. Software reliability is a function of design,
environment, maintenance and many other factors. These factors may change in
different stages of the software life cycle. The life cycle of the software may also vary
with the type of the software and its applications.

To highlight the key factors behind the rapid development of the software reliability
growth model, we have to consider the following factors:

✓ Software is extremely complex.
✓ Software failures occur frequently.
✓ The consequences of software failures may be ruinous.
✓ The portion of software cost in the total cost of the system development
  increases steadily with the elapsed years.

The achieved reliability growth in the reliability improvement program of a
product is controlled by the reliability prediction. To deal with the software is more
difficult than the development and maintenance of the hardware. Microprocessor
chips and other associated electronic items are known as hardware elements but they
all run with the help of programs known as software. Unreliable software is a
consequence of unexpected results of the software operations. Software that fails
frequently is considered to have less quality than the software that fails less often.
Continuously improved and low cost softwares offer flexible handling, compact
design, rich factors and competitive cost.

In the software reliability modeling various assumptions are to be made to
formulate the reliability growth model. It is worthwhile to explain some common
terms which are frequently used in our thesis for the formulation of software
reliability growth models in different frameworks.
1.2.1 Error, Fault and Failure

- **Error**: An error is the programmer action or mission that results in the fault. This is a measure of the difference between the actual and the ideal results. For example, if an operator enters the wrong account number to be canceled in the power company computer then it is man made error. There may be machine made errors too.

- **Fault**: A fault (also called a bug) is an internal program error that causes a failure to occur. Any error in coding, logic or structure triggers a failure when such program is executed under specific operational conditions. According to the execution conditions different failures also may be triggered.

- **Failure**: A failure is a departure of observed results from program requirements or user expectation on a run. A departure is the occurrence of a discrepancy between the desired output state and actual output state specified by the requirements for the particular run. A failure occurs only when there is a fault in the system. Failure may be caused by the software, hardware or their interface.
1.2.2 Some Reliability Indices

(i) **Reliability** : Reliability is the probability that a program will run successfully in the time interval from time 0 to t. Then

\[ R(t) = 1 - F(t) \]  

…(1.1)

where \( R(t) \) is reliability and \( F(t) \) is defined as the probability that the system will fail by time t. Let \( f(t) \) be the probability density of the failure distribution, then

\[ R(t) = \int_{t}^{\infty} f(\tau) d\tau \]  

…(1.2)

(ii) **Mean Time to Failure (MTTF)** : If we have life-tests information on a population of N items with failure times \( t_1, t_2, t_3, \ldots, t_n \), then the MTTF is defined as

\[ \text{MTTF} = \frac{1}{N} \sum_{i=1}^{n} t_i \]  

…(1.3)

The expected failure time during which a component is expected to perform successfully is called system mean time to failure (MTTF) and is given by

\[ \text{MTTF} = \int_{0}^{\infty} t f(t) dt = \int_{0}^{\infty} R(t) dt \]

where \( f(t) = \frac{dF(t)}{dt} = -\frac{dR(t)}{dt} \)  

…(1.4)

(iii) **Mean Time to Repair (MTTR)** : MTTR measures the average time for the system to recover. By repair, we mean to track the errors causing the failure and remove them.

(iv) **Availability** : Availability is a measure that allows for a system to repair when the failure occurs; the availability of a system is defined as the probability that the system is functioning successfully at time t, Mathematically,

\[ \text{Availability} = \frac{\text{System up time}}{\text{System up time + System down time}} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} \]  

…(1.5)
1.3 Software Reliability Growth Models (SRGMs)

In spite of the diversity and elegance of many of SRGMs, there is still a need of those models which can be more readily applied in realistic situations. One aspect which should perhaps be taken into account, is the complex and challenging nature of the testing of the software. Under the study of SRGM, the developer or tester has the knowledge about the statistical probability that the software is working perfectly or not and the software has reached a certain quality level. These models may help in measuring and tracking the growth of reliability as software is being improved. There is a large volume of literature on SRG modeling with many detailed probability models to represent the probabilistic failure process. In this section, we give brief overview of the software reliability modeling and testing. Software developers release the software after perfect testing in the market but still more than half of all faults are found after the shipment. Motivated by the vital importance of the SRGMs for the improvement in equality and reliability of the softwares, we have dealt with reliability growth issues in the present thesis.

The reliability growth for the software system can be described as the process in which the number of detected software errors increases as time elapses through the stages of design, coding and testing. SRGMs are defined as the mathematical relationship between the number of software errors removed and the testing time. SRGMs can be used to monitor some other operational metrics such as the total expected cost and optimal release time of the software. It is worthwhile to describe some SRGMs related to our study.

1.3.1 Non-Homogeneous Poisson Process (NHPP)

NHPP is important class of SRGMs that has been widely studied and used by many practitioners. Generally, an NHPP software reliability model can be constructed by using an appropriate mean value function $m(t)$ or failure intensity function $\lambda(t)$. While several NHPP models exist, it is our assertion that most NHPP SRGMs can be treated as special cases of following general model:

Let $\{N(t), t \geq 0\}$ denote a counting process representing the cumulative number of errors detected by time $t$. 
SRGMs based on an NHPP with the mean value function $m(t)$ can be formulated as
\[
\text{prob}[N(t) = n] = \frac{[m(t)]^n e^{-m(t)}}{n!}, \quad n = 0, 1, 2, 3, \ldots \quad \ldots(1.6)
\]
where $m(t)$ represents the expected cumulative number of errors detected in the time interval $(0, t]$. The mean value function $m(t)$ is non-decreasing in testing time $t$ with the boundary condition $m(0) = 0$ and $m(\infty) = a$, where $a$ is the expected number of errors to be eventually detected. Generally, we can get different NHPP models by using different non-decreasing mean value functions. The underlying assumptions of the NHPP are as follows:

(i) $N(0) = 0$,

(ii) $\{N(t), t \geq 0\}$ has independent increments,

(iii) $\Pr\{N(t + \Delta t) - N(t) \geq 2\} = 0(\Delta t)$, i.e., the system will not experience more than one failure at the same time.

(iv) $\Pr\{N(t + \Delta t) - N(t) = 1\} = \lambda(t) \Delta t + 0(\Delta t)$,

where, $0(\Delta t)$ has the usual meaning
\[
\lim_{\Delta t \to 0} \frac{0(\Delta t)}{\Delta t} = 0
\]

The failure intensity function at testing time $t$ can be formulated as
\[
\lambda(t) = \frac{d}{dt} m(t) \quad \ldots(1.7)
\]

The fault detection rate per fault at testing time $t$ is given by
\[
b(t) = \frac{m'(t)}{a - m(t)} = \frac{\lambda(t)}{a - m(t)} \quad \ldots(1.8)
\]

NHPP based SRGMs are generally classified into two groups. The first group contains models, which use the execution time (i.e., CPU time) or calendar time. Such models are called \textit{continuous time models}. The second group contains models, which use the test cases as a unit of fault removal period. Such models are called \textit{discrete time models}, since the unit of the software fault removal period is countable.
1.3.2 Generalized Continuous NHPP Model

A general class of NHPP SRGMs can be discussed by solving the following differential equation (Pham et al. 1999):

\[
\frac{dm(t)}{dt} = b(t)[a(t) - m(t)]
\]

\[\text{...(1.9)}\]

The solution of the above differential equation is

\[
m(t) = e^{-B(t)} \left[ m_0 + \int_{t_0}^{t} a(s)b(s)e^{B(s)} ds \right]
\]

\[\text{...(1.10)}\]

where \(B(t) = \int_{t_0}^{t} b(s) ds\) and \(m(t_0) = m_0\).

NHPP based SRGMs are considered under various environment according to the fault severity. Here we will discuss SRGMs including various phenomenon namely imperfect debugging, error generation, change point, time delay, etc. which are related to our research investigation.

1.3.3 Imperfect Debugging and Error Generation Phenomenon

In many realistic situations, mostly SRGMs assume that as soon as a failure is observed, the efforts are made to correct or remove the cause of failure. It is quite possible that testing team and efforts are not able to find the cause of failure to correct or remove the fault completely; this phenomenon is usually called imperfect debugging. Sometimes, there may be possibility of introducing new fault during correction or removal process. The situation when the old original fault is replaced by another fault, is called error generation. There is a big difference between imperfect debugging and error generation phenomenon. The total fault contents of the software are not changed during imperfect debugging because of the incomplete removal process. On the other hand, total fault content function increases in case of error generation. Fig 1.3 depicts the debugging process in SRGM.
1.3.4 Change-Point Phenomenon

Generally we test the software in specific environment and improve its quality by detecting and correcting/removing faults. But during testing phase in practical software development scenario, testing strategies, resource allocations, defect density, running environment may change. Once these factors are changed, this could result in a software failure intensity function that may increase or decrease non-monotonically. This phenomenon is called change-point problem. For detecting more faults for a short period of time, we have to develop new techniques or tools which have not been yet used. First time the change-point concept in software and hardware reliability was introduced by Zhao in (1993). In the present thesis, we have incorporated the change point concepts in chapters 3 and 5.

1.3.5 Fault Detection, Correction Process and Time Delay

In literature, a large numbers of SRGMs are based on NHPP, where detection/removal process is only dependent on the number of residual faults but in
real practice it is not so. Software testing and debugging are very complex and expensive processes. Once a fault is observed, a trouble ticket is created and assigned to one or more developers for analysis and code modification. It is not unusual for a software fault to occur multiple times in the field before it is finally removed. Therefore, the time delayed by the correction process should not be neglected. Detection and removal phenomenon is dependent on the testing time also. The removal of a detected fault always depends on the complexity of the fault, the available man-power and the software development environment. To detect a fault is one thing and remove that is another. In real time software, there may be a time delay between these two. Thus the time delay factor between the fault detection and correction should not be negligible. Mutually independent faults can be detected and removed directly; on the other hand mutually dependent faults can be removed iff the leading faults were removed.

In classical models, whenever failure occurs, the cause of fault is immediately removed perfectly and no new fault is introduced. But some researchers (cf. Huang et al., 2004; Huang and Lin, 2006) tried to remove this impractical assumption by developing the delayed fault detection models. The detected fault is not immediately removed, and it lags the fault detection process by a delay-effect factor $\varphi(t)$. Thus

$$m(t) = m_0(t - \varphi(t)) \quad \text{...(1.11)}$$

Many SRGMs can be described as a special case of the above general model. A brief summary of prominent NHPP SRGMs is presented in table 1.1.
<table>
<thead>
<tr>
<th>S. No.</th>
<th>Model Name</th>
<th>Initial Error Content Function $a(t)$</th>
<th>Error Detection Rate Function $b(t)$</th>
<th>Mean Value Function (MVF) $m(t)$</th>
<th>Model Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Goel – Okumoto</td>
<td>$a(t) = a$</td>
<td>$b(t) = b$</td>
<td>$m(t) = a \left[ 1 - e^{-bt} \right]$</td>
<td>Concave</td>
</tr>
<tr>
<td>2.</td>
<td>Delayed S-shaped</td>
<td>$a(t) = a$</td>
<td>$b(t) = b^2 \frac{1}{1 + bt}$</td>
<td>$m(t) = a \left[ 1 - (1 + bt) e^{-bt} \right]$</td>
<td>S-shaped</td>
</tr>
<tr>
<td>3.</td>
<td>Inflection S-shaped SRGM</td>
<td>$a(t) = a$</td>
<td>$b(t) = b \frac{1}{1 + \beta e^{-bt}}$</td>
<td>$m(t) = a \left[ \frac{1 - e^{-bt}}{1 + \beta e^{-bt}} \right]$</td>
<td>S-shaped</td>
</tr>
<tr>
<td>4.</td>
<td>Yamada Exponential</td>
<td>$a(t) = a$</td>
<td>$b(t) = \alpha \beta e^{-bt}$</td>
<td>$m(t) = a \left[ 1 - \gamma \left( 1 - e^{-bt} \right) \right]$</td>
<td>S-shaped</td>
</tr>
<tr>
<td>5.</td>
<td>Yamada Rayleigh</td>
<td>$a(t) = a$</td>
<td>$b(t) = \frac{\beta^2}{2}$</td>
<td>$m(t) = a \left[ 1 - e^{-\gamma \left( 1 - e^{bt} \right) \right]</td>
<td>S-shaped</td>
</tr>
<tr>
<td>6.</td>
<td>Yamada exponential imperfect debugging model</td>
<td>$a(t) = ae^\alpha t$</td>
<td>$b(t) = b$</td>
<td>$m(t) = \frac{ab}{(\alpha + b)} (e^{\alpha t} - e^{-bt})$</td>
<td>Concave</td>
</tr>
<tr>
<td>7.</td>
<td>Yamada linear imperfect debugging model</td>
<td>$a(t) = a(1 + \alpha t)$</td>
<td>$b(t) = b$</td>
<td>$m(t) = a \left[ 1 - e^{-bt} \right] \left[ 1 - \left( \frac{\alpha}{b} \right) \right] + \alpha at$</td>
<td>Concave</td>
</tr>
<tr>
<td>8.</td>
<td>Pham-Nordmann-Zhang (P-N-Z) Model</td>
<td>$a(t) = a(1 + \alpha t)$</td>
<td>$b(t) = b \frac{1}{1 + \beta e^{-bt}}$</td>
<td>$m(t) = \frac{a}{(1 + \frac{\beta}{b} e^{-bt})} \left[ (1 - e^{-bt}) \left( 1 - \left( \frac{\alpha}{b} \right) \right) + \alpha at \right]$</td>
<td>Concave and S-shaped</td>
</tr>
<tr>
<td>9.</td>
<td>Pham-Zhang (P-Z) Model</td>
<td>$a(t) = c + a(1 + \alpha t)$</td>
<td>$b(t) = b \frac{1}{1 + \beta e^{-bt}}$</td>
<td>$m(t) = \frac{a}{(1 + \frac{\beta}{b} e^{-bt})} \left[ (c + a) (1 - e^{-bt}) - (\alpha (b - \alpha)) (e^{\alpha t} - e^{-bt}) \right]$</td>
<td>Concave and S-shaped</td>
</tr>
</tbody>
</table>

Table 1.1: Some Software Reliability Growth Models
Chapter-1: General Introduction

Following are some notations which have been used for formulating the models described in table 1.1.

\[ m(t) \] : Expected number of faults detected or removed by time \( t \) or mean value function.

\[ a(t) \] : Initial number of faults lying dormant in the software when the testing starts.

\[ b(t) \] : Error detection rate.

\[ c, \beta \] : Constant parameter in the logistic function.

\[ \alpha \] : Constant rate of error generation where \( 0 < \alpha < 1 \).

Some of the assumptions inherent in a SRGM are as follows:

- Failure observation/removal is modeled by NHPP.
- Testing occurs according to an operational phase.
- Time interval between failures is independent.
- Faults are removed or corrected immediately.
- Failures observation is mapped according to the execution time.
- Software is subject to failure during execution caused by faults remaining in the software.

1.3.6 General Discrete NHPP based SRGM

In discrete time model, the test cases are considered as a unit of fault removal period. A test case can be a single computer test run executed in an hour, day, week or even month. Because of difficulties in terms of mathematical complexity there are fewer discrete time models in comparison to continuous time models in software reliability literature. The utility of discrete software reliability growth models can not be under estimated.

We consider a discrete counting process \( \{N_n, n \geq 0: n = 0, 1, 2, \ldots\} \) which represent the cumulative number of faults detected up to \( r^{th} \) testing period. Thus

\[
P_r\{N_n = x \mid N_0 = 0\} = \frac{m(n)^x}{x!} \exp\{-m(n)\}, \quad n, x = 0, 1, 2, \ldots, \quad \ldots(1.12)
\]
where \( m(n) \) is a mean value function of the discrete counting process, i.e., expected number of cumulative faults detected up to \( n^{\text{th}} \) testing-period.

There are several well established NHPP based continuous SRGMs with different mean value functions (see table 1.1).

### 1.3.7 Discrete Exponential SRGM

Yamada and Osaki (1985a) proposed difference equation for this model which has the exact solution. A discrete analog of the exponential SRGM is as follows:

\[
m(n + 1) - m(n) = \delta b (a - m(n)) \quad \ldots(1.13)
\]

where \( \delta \) is a constant time interval. On solving the above equation, we obtain an exact solution as

\[
m(n) = a \left[ \left( 1 - \delta b \right)^n \right] \quad a > 0, \quad 0 < b < 1 \quad \ldots(1.15)
\]

where \( n = \frac{t}{\delta} \).

Eq. (1.14) converges to an exact solution of the Goel-Okumoto model (continuous model) as \( \delta \to 0 \) which is described by the differential equation

\[
m(t) = a \left[ 1 - \exp(-bt) \right] \quad \ldots(1.15)
\]

### 1.3.8 Discrete delayed S-shaped SRGM

In 1992a, Kapur et al. discussed that the testing phase is assumed to have two different processes, i.e., fault isolation and fault removal process. Based on this assumption, the two equations are constructed as:

\[
m_i(n+1) - m_i(n) = \delta b (a - m_i(n)) \quad \ldots(1.16)
\]

and

\[
m_r(n+1) - m_r(n) = \delta b (m_i(n+1) - m_r(n)) \quad \ldots(1.17)
\]

On solving these equations, we get

\[
m_r(n) = a \left[ \left( 1 + \delta bn \right) \left( 1 - \delta b \right)^n \right] \quad a > 0, \quad 0 < b < 1. \quad \ldots(1.18)
\]
As $\delta \to 0$, eq. (1.18) converges to continuous delayed S-shaped SRGM which is described by the differential equation given by

$$m(t) = a[1 - (1 + bt)\exp(-bt)]$$  \hspace{1cm} \text{(1.19)}

### 1.4 Testing Effort Function

For improving the efficiency and accuracy of the SRGMs, testing-effort functions have been used by some software developers. *These testing-effort functions (TEF) describe how an effort is distributed over the exposure period, and how effective it is.* The testing effort will depend heavily on many factors such as project, process and the personal. In the context of software testing, the concepts of testing effort and effectiveness of the test-cases are the key elements. Due to the complex dependence of testing effort on several factors, testing time may not provide an accurate measure of the testing effort. Although a large number of testing effort models have appeared in the literature, the development of these models appears to be ad hoc.

#### 1.4.1 Discrete Testing-effort Functions

A finite set of successive test occasions are described by discrete models. A set of test case is executed on each of these occasions and the number of faults induced is counted. We classify the discrete testing effort functions into three categories as follows:

- **Effort-as-time function:** In these functions, the testing-effort function is used into their binomial and Poisson distributions by explicitly relating the probability that a fault occurrence during a unit of testing to the amount of testing effort expended (*Brooks and Motley, 1980*).

- **Hypergeometric Distribution Function (HGDF):** For modeling the fault detection process, hypergeometric distribution functions are applied. First these functions were introduced by *Mills (1972)*. The notable feature of HGDF is that it relates the number of faults detected with testing-effort, both new and previously detected on any given test occasion associated.
Chapter-1: General Introduction

- **Composite Functions:** One of the first composite functions of discrete type was proposed by Kapur et al. (1994a). He developed a Rayleigh testing-effort function. The proposed models express fault detection to the amount of testing effort expended. The testing effort is

\[ W(n) = \alpha \left[ 1 - \prod_{i=0}^{n} (1 - i\beta) \right] \]

...(1.20)

where \( \alpha \) denotes the total eventual effort and \( \beta \) denotes the rate of effort manifestation.

### 1.4.2 Continuous Testing-effort Functions

Continuous testing-effort function provides a way to estimate the number of additional faults to be detected by some future point of view. Further these functions are classified into three categories.

- **Effort-as-time function:** In the continuous case, effort-as-time function is conceptually similar to the corresponding discrete testing-effort function. These functions use efforts to estimate the parameters in place of time data.

- **Composite Functions:** There are three broad categories to describe the composite testing-effort function. In the first category, known distribution functions are used. In the second category, general testing-effort functions are used. The combination of general and well known distribution functions is the used third category.

A classification of known testing effort models is given in fig. 1.4.

**Multiple Error Type Functions:** These functions classify errors into different categories based on the amount of effort required to fix them. Fault severity, new and reused components, attribution of the fault and fault complexity are various schemes to categorize the faults. Each classification of faults takes a unique approach for improving the accuracy of quantifying the testing-effort functions.

Let \( W(t) \) denote the cumulative testing effort and \( w(t) \) is defined as the current testing effort. The mathematical relationship between \( w(t) \) and \( W(t) \) is as follows:
\[ W(t) = \int_0^t w(t) \, dt \quad \text{...(1.21)} \]

<table>
<thead>
<tr>
<th>S.No.</th>
<th>Testing Effort (TE)</th>
<th>Function</th>
<th>Introduced by</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Exponential TE</td>
<td>[ W(t) = N[1 - e^{-\beta t}] ]</td>
<td>Yamada et al. (1986)</td>
</tr>
<tr>
<td>2.</td>
<td>Rayleigh TE</td>
<td>[ W(t) = \alpha \left(1 - e^{-\left(\frac{\beta}{2}\right) t^2}\right) ]</td>
<td>Yamada and Othera (1990)</td>
</tr>
<tr>
<td>3.</td>
<td>Weibull TE</td>
<td>[ W(t) = N[1 - e^{-\delta t^\gamma}] ]</td>
<td>Yamada (1991)</td>
</tr>
<tr>
<td>4.</td>
<td>Logistic TE</td>
<td>[ W(t) = \frac{N}{1 + Ae^{-\alpha t}} ]</td>
<td>Huang and Kuo (2002)</td>
</tr>
<tr>
<td>5.</td>
<td>Generalized logistic TE</td>
<td>[ W(t) = \frac{N}{\sqrt[3]{1 + Ae^{-\alpha t}}} ]</td>
<td>Huang and Lyu (2005)</td>
</tr>
<tr>
<td>6.</td>
<td>Log-logistic TE</td>
<td>[ W(t) = N\left(\frac{(\beta t)^\delta}{1 - (\beta t)^\delta}\right) ]</td>
<td>Bokhari and Ahmad (2006)</td>
</tr>
<tr>
<td>7.</td>
<td>Logistic-exponential TE</td>
<td>[ W(t) = \alpha \frac{\left(e^{\kappa t} - 1\right)^\beta}{\left(1 + \left(e^{\kappa t} - 1\right)^\beta\right)^\delta} ]</td>
<td>Lan and Leemis (2007)</td>
</tr>
</tbody>
</table>

Table 1.2: Testing-effort functions

Various testing effort functions used in literature are tabulated in table 1.2, some notations used in testing effort functions are listed below:

**Notations**

- \( \beta \) : Scale Parameter
- \( \delta \) : Shape Parameter
- \( N \) : Total amount of effort eventually spent
- \( A \) : Constant
- \( \kappa \) : Well-structured software development efforts
Fig. 1.4: Classification of Testing Effort Models
1.5 Testing Reliability

Reliability growth modeling is based on the testing during development. Software is tested with a set of test cases and the behavior of the system for the test cases is evaluated to determine that the system is performing as expected or not. The expectation is that the software will meet its reliability requirements in service, but sometimes the exposure of many copies of the software may reveal additional failures in the operational phase.

Software systems are developed in a heterogeneous environment according to the user’s environment, and hence it may be inappropriate to model the overall failure process using one of the several software reliability growth models. The SRGM have been made so that the design faults that lead to in-service failures are diagnosed and removed. There are two basic approaches of testing the software:

(i) Black Box Testing

Basically black box is an external assessment. In the black box testing, the structure of the program is not considered and the testers only know the inputs that can be given to the system. This type of testing is also called functional or behavioral testing.

(ii) White Box Testing

White box testing also called structural testing involves an assessment of software reliability and concerned with testing the implementation of the program. White box testing aims to achieve test cases that will force the desired coverage of different structures. Failures are analyzed during the testing, and corrective action is taken to prevent the effects of recurrence.

During the testing phase it may be possible that new faults may introduce into the software. So the new automated test tools and methods are employed to detect the additional faults. As time progresses, they can detect additional faults, which reduces the expenses of correcting faults during the operational phase. These approaches have improved software testing and productivity.

Software testing for a modular based software system is broadly classified into (i) component testing and (ii) integration testing. Testing carried out for a component
or module is designed to target a class of bugs or specific parts of code. Component testing alone does not deliver the type of testing required. Some faults emerge after integration as such the tests should be carried out over a period several times.

There are two important phases to analyze the software behavior (i) testing phase (ii) operational phase. One of the most challenging and controversial issue in SRGM is that software systems should be tested. The testing phase is followed by an operational phase according to the user’s intensity. If the software is used in different environment during testing, the results obtained may not be compatible for the software in the operational phase. The aim of testing is that the software should be statistically similar to the in-service environment. Tests can either be carried out with the software installed in the system according to the operational phase, or by means of a simulator. Therefore it is important and challenging issue for testers to establish an operational profile and test the software according to this profile. Software reliability can be expressed in terms of conditional probability given by Musa (1993).

\[
R \left( \frac{X}{t} \right) = \Pr \left( x_k > x | t_{k-1} = t \right) \quad \ldots (1.22)
\]

### 1.5.1 Testing Phase

*Testing reliability is the probability that any failure will not occur during the testing phase.* During the testing phase, software improves because identified faults are removed or corrected and similar failure will not occur again in the simulated user environment. Let the cumulative number of experienced failures up to time \( t \) be denoted by \( N(t) \). If the testing process follows the non-homogeneous Poisson process (NHPP), for any \( t \geq 0 \) and \( x > 0 \), then the mean value function (MVF) is given by

\[
\Pr \{ N(t+x) - N(t) = k \} = \frac{[m(t+x)-m(t)]^k}{k!} \exp \left[ -[m(t+x)-m(t)] \right], \quad k = 0, 1, 2, \ldots
\]

\ldots (1.23)

The reliability can be computed as

\[
R_{tc} \left( \frac{X}{t} \right) = \Pr \{ N(t+x) - N(s) = 0 \}
= \exp \left[ - \left( m(t+x) - m(t) \right) \right]
\]

\ldots (1.24)
Here $R_{te}(x/t)$ measures the software reliability during the testing phase and $(t + x)$ should not extend to the operational phase.

1.5.2 Operational Phase

During operational phase, the fault removal is not considered. The operational reliability is the probability that no failure will occur during the operational phase. Each software has a failure intensity associated with it. But failure intensity changes when software is released according to the environment. During operational phase, the failures and new factors occur continuously in the software system. If the time interval $x_k$ is for the operational phase and the software has been tested for $t$ units and then released to the customers, then the time to next failure will follow exponential distribution with parameter $\lambda_r(t)$, where $\lambda_r(t)$ is the failure intensity function of the NHPP calculated at time $t$. For two identical copies of the same software, the reliability may be different if they are used under different operational conditions (cf. Musa and Okumoto, 1984). The failure rate may depend on the testing time. The testing goes on till the management is satisfied with the quality and reliability of the software.

1.6 Expected Cost of the Software

In this section, we discuss total expected cost of the software. When the new software is released, various costs and profits are associated with the testing time decision. Various costs and benefits are associated with the right time of a new release. A more desirable approach would be to determine release times for new versions based on the total life-cycle cost of the software. The software testing process is time consuming and costly both. It may involve additional testing time with ineffective or redundant test excessiveness, leading to cost in the software maintenance. Software debuggers may use new automated tools or techniques to detect additional faults and a large number of bugs are not discovered during the testing to ensure the reliability of the software. The cost of new automated tools and techniques should be considered in the total expected cost of the software including their expenditure and benefits.
Fig. 1.5: Casual loop of the SRGM

In order to depict the total expected cost, a casual loop to portray the process of software reliability growth is shown in fig 1.5.

Some software cost models

In literature, there are various cost models involving different factors. To describe some cost functions used by several researchers, we use the following notations:

Notations

\( T \) : Release time of the software.

\( T_w \) : Warranty period.

\( T_{lc} \) : Software life-cycle.

\( \lambda(t) \) : Failure intensity function.

\( C_0 \) : Setup cost of the software.

\( C_t \) : Testing cost of the software per unit time.

\( C_m \) : Cost per unit time of removing an error during testing.

\( C_n \) : Cost per unit time of removing an error detected during warranty period.
Chapter-1: General Introduction

\( C_r \): Cost associated with the loss due to the software failure.

\( C_u \): Cost per unit of testing-effort consumption during testing.

\( C_w(T) \): Maintenance cost per fault during the warranty period.

\( E\{C(T)\} \): Total expected cost of the software.

\( w(t) \): Current testing-effort function.

\( \alpha \): Discount rate of testing cost.

\( u_y \): Expected time for removing an error during the testing period.

\( u_w \): Expected time for removing an error during the warranty period.

\( R(x/T) \): Reliability function.

\( x \): Operating time of the software.

Now we present some cost functions in terms of different cost factors associated with the software.

- **Software Cost Model**

\[
E\{C(T)\} = C_m m(T) + C_n \left[ m(T_{1c}) - m(T) \right] + C_i T
\]

\( \ldots (1.25) \)

- **Software Cost Model with Risk Factor**

\[
E\{C(T)\} = C_i T + C_m m(T) u_y + C_i \left[ 1 - R(x/T) \right]
\]

\( \ldots (1.26) \)

- **Generalized Software Cost Model**

\[
E\{C(T)\} = C_0 + C_i T^8 + C_m m(T) u_y + C_n \left[ m(T + T_w) - m(T) \right] u_w + C_i \left[ 1 - R(x/T) \right]
\]

\( \ldots (1.27) \)

- **Software Cost Model with Testing Effort Function**

\[
E\{C(T)\} = C_m m(T) + C_n \left[ m(T + T_w) - m(T) \right] + C_u \int_0^1 w(t) dt
\]

\( \ldots (1.28) \)
Software Cost Model with Warranty Cost

\[ E[C(T)] = C_0 + C_w \int_0^T e^{-\alpha t} dt + C_w(T) \]  \hspace{1cm} \text{...(1.29)}

Case 1: In this case, it is assumed that during the warranty period, the software reliability growth does not occur. Then \( C_w(T) \) is defined as

\[ C_w(T) = C_w \int_0^{T+T_w} \lambda(T) e^{-\alpha t} dt \]  \hspace{1cm} \text{...(1.30)}

Case 2: In this case during the warranty period, the software reliability growth occurs, even after the testing phase. Now \( C_w(T) \) is given by:

\[ C_w(T) = C_w \int_0^{T+T_w} \lambda(t) e^{-\alpha t} dt \]  \hspace{1cm} \text{...(1.31)}

After using the cost model, the software developer can assure that the software has achieved pre-specified goals, and get the information that when to stop the testing of the software. A software developer can determine whether more testing is required or whether the software is sufficiently tested to allow its release.

1.7 Optimization Issue in Software Reliability

Optimal testing time is the main issue in the software reliability modeling for software developers so that they can stop the testing at an appropriate time and software could be released in the market. The complete software is tested to determine its correctness. To gain high-quality, or high reliability software, a considerable amount of resources is invested in the tests. In the research environment, main issue for the software developer is to decide the optimal time to release. Testing is a key process in assuring quality by detecting and diagnosing the faults in software, and possibly removes them, before it is delivered to the customers. However, much more time and cost could be spent in the maintenance of the software later due to fixes of bugs which are not discovered during the testing phase. A major decision to make during this software testing is, to determine whether to continue testing and eventually releasing the software, or when to stop the test and release it in the market for the
users. Several factors influence the decision to release the software. Such a decision needs to be made to optimally balance the tradeoff between the cost of development and the reliability of the software.

Several researchers dealt with this optimal problem and developed some models that are basically related into two classes:

- Models that determine a optimal testing time, tells the developers to stop testing when a given condition is met.
- Models that perform cost-benefit analysis of the testing process.

The use of SRGMs to depict software reliability provides a statistical foundation to establish optimal stopping time for the software testing, which is a key decision criteria in the software engineering.

### 1.7.1 Optimization Problem

After reaching a certain level of software refinement, any effort to increase reliability will require an exponential increase in the cost.

The optimization problem can be formulated as

\[
\begin{align*}
\text{Minimize} & \quad C(T) \\
\text{Subject to} & \quad R(x/T) \geq R_0
\end{align*}
\]

### 1.7.2 Optimal Release Policies

The optimal release policies will allow software developers to release sufficiently reliable software prior to initially projected deadline. In a competitive software industry nowadays, implementing the optimal policies will give software developing companies a competitive advantage, by allowing them to cut down on the developmental cost and to release the product in the market. Future research could be directed toward the development of further general debugging models, where self-generating processes are of great interest, and optimal stopping formulation is likely to be applicable.

Release policies are classified into two categories: (i) static and (ii) dynamic. Under static release policies, the release time of the software is determined before
testing has begun, and is kept unchanged throughout the testing phase. The release time is independent of any discrepancy due to data collected during the testing phase. On the other hand, in the case of dynamic release policies, there is no preset release time. The release time is dynamically determined from failure statistics obtained during testing.

1.8 Reliability Modeling: A Methodological View

In this section, we discuss a brief account of methodological aspects of reliability theory related to our study including Markov modeling approach, fuzzy analysis, generating function, neuro-fuzzy method, etc..

1.8.1 Markovian Approach of Software Reliability

The markov model assumes that the future is independent of the past, given the present. There are following four possible combinations for expressing markov model:

(i) Continuous state, continuous-time;
(ii) Discrete-state, continuous-time;
(iii) Continuous-state, discrete-time;
(iv) Discrete-state, discrete-time.

In reliability analysis, we consider a discrete-state, continuous-time model, i.e., is called Markov Process. The concept of markov process is useful in the modeling of random behavior of the software in time such as faults remaining at time t and failure experienced by time t. In general, markov process is characterized by the amount of time spent in a state and the transitions between states. A markov process is completely characterized by its transition probability matrix.

1.8.2 Fuzzy Analysis

The concept of fuzzy logic (FL) was first introduced by Lotfi Zadeh (1978) who studied a way of processing data by allowing partial set membership. Professor Zadeh reasoned that people may not require precise, numerical information input, and yet they may be capable of highly adaptive control. Now fuzzy logic has become such a methodology which is used in systems ranging from simple, small, embedded
Chapter-1: General Introduction

micro-controllers to large, networked, multi-channel PC and control systems. Fuzzy logic can be implemented in hardware, software, or a combination of both. Fuzzy theory provides a simple way to arrive at a definite conclusion based upon vague, imprecise, ambiguous, or missing input information. Fuzzy Logic has emerged as a profitable tool for the modeling of software in particular when developing SRGM in different frameworks. Now we briefly discuss the triangular and trapezoidal fuzzy numbers which are used for formulating the SRGM in chapter 3.

1.8.1.1 Triangular fuzzy number- A fuzzy number ‘A’ having the shape of triangle as shown in fig. 1.7 and denoted by the triplet \( A = (a_1, a, a_2) \) is called a triangular fuzzy number (TFN).

The membership function of TFN is

\[
\mu_A(x) = \begin{cases} 
0, & x < a_1 \\
\frac{x-a_1}{a-a_1}, & a_1 \leq x \leq a \\
\frac{a_2-x}{a_2-a}, & a \leq x \leq a_2 \\
0, & x > a_2 
\end{cases} \quad \ldots (1.32)
\]

![Fig. 1.7: Triangular fuzzy diagram](image)

1.8.1.2 Trapezoidal fuzzy Number- A fuzzy number ‘B’ denoted by \( B = (a_1, a_2, a_3, a_4) \) is called a trapezoidal fuzzy number. Its membership function is
Chapter-1: General Introduction

The shape of its membership function is shown in fig 1.8.

![Trapezoidal fuzzy diagram](image)

1.8.2 Probability Generating Function

There are various techniques available to solve the difference equations dealing with probability distribution. One of the significant techniques is Probability Generating Function (PGF) method, as the name sounds uses convergent power series to solve the system of difference equations to generate probabilities. PGF is a powerful tool that simplifies computations involving integer valued, discrete random variables. Suppose $X$ is a random variable taking only non-negative integer values with $\Pr \{X=k\} = P_k$, $K = 0, 1, 2, 3, \ldots$; $\sum_{k} P_k = 1$.

If $P_0, P_1, P_2, \ldots$, is a sequence of real numbers and if

$$G_X(z) = P_0 + P_1z + P_2z^2 + \ldots = \sum_{i=0}^{\infty} P_i z^i \quad \ldots \text{(1.35)}$$

Then $G_X(z)$ is called the probability generating function of $X$. $G_X(z)$ is also known as $z$ transform of the random variable $X$, converges for any complex number $z$ such that $|z| = 1$, i.e., $z$ converges in some interval $-z_0 < z < z_0$, where the sequence is infinite.

$$\mu_B(x) = \begin{cases} 
\frac{x-a_1}{a_2-a_1}, & a_1 \leq x < a_2 \\
1, & a_2 \leq x \leq a_3 \\
\frac{a_4-x}{a_4-a_3}, & a_3 < x < a_4 \\
0, & x < a_1, x > a_4 
\end{cases} \quad \ldots \text{(1.33)}$$
1.8.3 Neuro-Fuzzy Method

The neuro fuzzy which enhances fuzzy logic gives the set values for the neural network controllers based on the process control expertise put in the fuzzy logic rules. The neuro fuzzy logic technologies enable the efficient and transparent implementation of the software systems. Neuro-fuzzy approach allows for the automated generation and optimization of fuzzy logic based software systems based on training data.

Many ways of integrating neural nets and fuzzy logic have been proposed in the scientific literature. Only a very few have already been successfully applied in software applications.

The neuro-fuzzy systems combine the design and optimization process of fuzzy controllers with learning capabilities derived from neural networks. The neuro-fuzzy technology is blessed due to following characterization:

- Fast computation using fuzzy number operations.
- Capable of handling any kind of information (numeric, linguistic, logical, etc.).
- No need of prior knowledge of relationships of data.
- Can manage imprecise, partial, vague or imperfect information.
- Emirates mimic human decision making process.
- Self-learning, self-organizing and self-tuning capabilities.

The rule structure is made up of neuro-fuzzy commands of “IF”, “AND, “Continues” and “THEN” carefully connected together to arrive at the specified desired parameter (output). All the commands are in such a way that a computer program can be written on them for computerization. The neuro-fuzzy commands “IF”, “AND”, “Continues” and “THEN” have significant meanings that account for the effectiveness of the model. The command “IF” means if the outcome of the relationship between input and output parameters is this, the command “AND” means “and this outcome”, the command “continues” means “the outcome continues over time, finally the command “THEN” means if all the previous commands hold “then the system should prompt the specified desired output”.

30
The neuro-fuzzy models are gradually becoming established not only in the academia but also in the software applications. The tools for building neuro-fuzzy models are based on combinations of algorithms from the fields of neural networks, pattern recognition and regression analysis.

1.9 Statistical Approaches

Many software reliability growth models have been developed, in the recent years, which fit the experimental data and each model having its own strength in facilitating reliability estimations and other performance indices. These models use two or more parameters to get the best fit against the actual failure data. The validity of the model and to make the quantitative comparisons, are of vital importance. Statistical approaches examine whether the software model is fit for the observed software fault data. This will help us in predicting the future behavior of the software fault removal process satisfactorily. Now, we describe some statistical methods which are used in our research investigation.

1.9.1 Parameter Estimation

In literature most of SRGMs are studied analytically by making some assumptions about the software testing and the debugging process. The design of such models and the need for two or more parameters are based on the expectation of the trend in the failure data. The parameters of the model can be estimated from the available software failure data, and then the model is used to compute relevant reliability measures such as software reliability, availability, mean time to failure frequency, cost, optimal release times, etc..

There are many techniques to estimate the parameters of the software reliability growth model, based on NHPP. Maximum Likelihood Estimation (MLE) or Least Square Estimation (LSE) are the most widely used methods of estimation. With the help of statistical software package MATLAB, MAPPLE, MATHEMATICA, SPSS and SAS etc., the estimation of parameters of the models can be done with ease. It is worthwhile to review some basic statistical tools which have been used in the present study.
1.9.2 Maximum Likelihood Estimation (MLE)

Maximum likelihood estimation technique can be used for solving a set of simultaneous equations of the model. The set of equations may be very complex and usually must be solved numerically. The logarithmic maximum likelihood estimation (MLE) technique is very popular for estimating the unknown parameters.

Let \( t_1, t_2, \ldots, t_n \) be the random failure time of \( n \) items where \( 0 < t_1 < t_2 \ldots < t_n \). Then log likelihood function (LLF) is given by

\[
\prod_{j=1}^{n} \left( \frac{m(t_j) - m(t_{j-1})}{y(t_j) - y(t_{j-1})} \right) \exp \left( - \frac{(m(t_j) - m(t_{j-1}))}{(y(t_j) - y(t_{j-1}))} \right) \cdot t_{j-1}^{y(t_j) - y(t_{j-1})} \cdot (y(t_j) - y(t_{j-1}))! \]

... (1.36)

1.9.3 Least Square Estimation

Least square estimation technique minimizes the sum of squares of the deviation between observed and expected values. This approach is used in the software reliability to estimate the model parameters by fitting the functional relationship of the failure intensity with respect to the mean value function to the observed failure intensity.

Let \( (t_0, m_0), (t_1, m_1), (t_2, m_2), \ldots, (t_n, m_n) \) are \( n \) observed data pairs; where \( m_j \) is the total number of errors detected within time \( (0, t_j) \). Then

\[
\sum_{j=1}^{n} \left( m_j - m(t_j) \right)^2 \]

... (1.37)

where \( n \) is the number of total failure data. A series of differential equations can be obtained by taking the derivatives of the log likelihood function of equation (1.36) and partial derivatives of equation (1.37) with respect to each parameter of the model and equating them to zero. The parameters can be estimated by solving these differential equations simultaneously.

After estimating the parameters of the model, the results are evaluated on the basis of some comparison criteria. Here we describe below some commonly used criteria.
1.9.4 Accuracy of Estimation (AE)

AE is defined (cf. Musa et al., 1987) as

\[ AE = \left| \frac{m_a - \hat{a}}{m_a} \right| \]  

\[ \ldots(1.38) \]

where \( m_a \) is the actual cumulative number of detected faults after the test, and \( \hat{a} \) is the estimated number of initial faults.

1.9.5 Relative Error (RE)

The prediction of failure behavior for future with the help of present and past failure behavior, is called predictive validity and can be evaluated in terms of relative error (RE) for a data set. If \( q \) failures are observed by the end of test time \( t_q \), then RE is given by

\[ RE = \frac{m(t_q) - q}{t_q}, \]  

\[ \ldots(1.39) \]

1.9.6 Bias

Bias is the sum of the difference between the estimated curve and the actual data. The lower value of bias gives the better goodness of fit. The average of the prediction errors is called the prediction Bias. Bias is quantified by using

\[ \sum_{k=1}^{n} (m(t_k) - m_k)/n \]  

\[ \ldots(1.40) \]

Standard deviation is often used as a measure of the variance in the predictions. In order to evaluate variance, we use the following formula

\[ \text{Variance} = \frac{1}{n-1} \sum_{k=1}^{n} (m_k - m(t_k) - \text{Bias})^2 \]  

\[ \ldots(1.41) \]

1.9.7 Root Mean Square Prediction Error (RMSPE)

This is a measure of the closeness with which the model predicts the observation. We obtain the root mean square prediction error using
Chapter-1: General Introduction

\[
\text{RMSPE} = \sqrt{\text{Variance}^2 + \text{Bias}^2} \quad \ldots (1.42)
\]

1.9.8 Mean Square Error (MSE)

MSE is the sum of the square value of the deviation. MSE is obtained as

\[
\text{MSE} = \frac{1}{n} \sum_{k=1}^{n} (m(t_k) - m_k) \quad \ldots (1.43)
\]

where \( k \) is the number of observations.

A smaller MSE indicates a smaller fitting error and better performance.

1.10 Review of the Literature

In this section, we present a survey concerning our research topic on the software reliability growth models (SRGMs) by highlighting the historical advancement of the software reliability research. In the last two decades of 20\(^{th}\) century and in the beginning of 21\(^{st}\) century, the rise of information technology has gone far beyond. Software reliability is considered to be one of the important areas of Operations Research and Management Science as well as Mathematics, Statistics, and Computer Science, etc.. There are dozens of books and thousands of papers on software reliability and they continue to be published at an ever-increasing rate. But inspite of its apparent simplicity, the subject is one of the serious depth and delicacy while analyzing the real time embedded software systems. At this stage it appears to be appropriate to provide review of the literature related to our work over the past few decades.

The concept of reliability grew into a pervasive attribute worth of both qualitative and quantitative connotations. From 1800’s to 2000’s several revolutionizing social, and technological developments have occurred which have the need of a rational framework for the quantitative treatment of the software reliability. The developers need theories, concepts or mental ideas about how the software will work then the historical evidence and the overwhelming huge literature provide that help to the developers. ‘Software Quality’ means that the software product confirms to both explicit and implicit requirements. By the early 1980’s, “quality” became the popular theme in the computer industry. In this section, prominent recent past (1975
onwards) works done by various researchers in the field of software reliability have been reviewed. Several software reliability models have been used for Markov chain based testing. Markov process is used successfully to model system use, generate tests and compute statistics about system. Littlewood (1975) incorporated reliability model for markov structured software. Trivedi and Shooman (1975) predicted the performance parameters of computer software dependent on markov model. A versatile markovian point process was estimated by Neuts (1979). Cheung (1980) developed a user oriented software reliability model. Some important contributions in this field are due to Dhillon (1983), Currit et al. (1986), Mellor (1987), Cai et al. (1991), Lyu (1996). Mantere and Alander (2005) and Faqih (2009) measured and predicted the software reliability and its applications for a model studied by them. Govil (1984) incorporated the execution time concept in several software reliability models. Farr and Smith (1988) proposed a tool for statistical modeling and estimating of reliability functions for the software. Software engineering under statistical quality control was studied by Mills and Poore (1988) and Richard and Harlan (1990). In 1989, probability models were introduced by Ross. Krishnamurthy and Mathur (1997) estimated the reliability of a software system using reliability of its components. Xie et al. (1999) and Ho et al. (2003) incorporated software reliability prediction in their works. Rallis and Lansdowne (2001) estimated reliability for a software system with sequential independent reviews. Opiyo et al. (2002) studied quality assurance of design support software. Jeske and Zhang (2005) discussed approaches to software reliability modeling in industry. The fundamentals of reliability with compatibility and simplicity were given by Kuo (2007). Mohan et al. (2009) analyzed the impacts of faults in different software systems. Montesi and Lago (2008) and Beckhaus et al. (2009) suggested the applicability of software reliability growth modeling in the quality assurance phase of a large business software vendor.

Now we present a survey of literature on the specific topics related to our research investigations in the present study.
1.10.1 Software Reliability Growth Model

Over the last many years, a large number of mathematical models have been developed for reliability growth. Duane (1964) was first person to report the most commonly accepted pattern for reliability growth. Duane model is basically a graphical approach to perform analysis of reliability growth. After then since 1970’s, the traditional way of predicting software reliability has been the use of software reliability growth models. Many research activities in software reliability engineering have been conducted and software reliability growth models in different frameworks have been proposed to assess the reliability of the software. During the last three decades, many software reliability growth models (SRGMs) have developed for measuring the growth of reliability of the software. The most widely used models are developed by Jelinski and Moranda’s (1972), Shooman Model (1976) and many others. Littlewood and Verall (1973) used a bayesian approach to determine the reliability growth model for computer software. Musa (1975a) examined logarithmic poisson execution time model for software reliability measurement. Shooman (1987) and Schick and Wolverton (1978) analyzed the competing software reliability models. Time-dependent error detection rate model for software reliability and other performance indices were considered by Goel and Okumoto (1979).

The objective of software reliability testing is to determine probable problems with the software design and implementation as early as possible to assure that the system meets its reliability requirements. Several hundreds of statistical models used in software reliability testing have been developed over the years (cf. Dhillon, 1980; Ohba, 1984; Yamada et al., 1984a; Yamada et al., 1984b; Goel, 1985; Musa, 1988; Tohma et al., 1989; Xie, 1991; Shanthikumar, 1983). Yamada and Osaki (1983) and Hartler (1989) discussed reliability growth models for hardware and software systems based on non homogenous Poisson processes. Littlewood and Sofer (1987) proposed a bayesian modification to the Jelinski-Moranda software reliability growth model.


But last decade of twentieth century will be noted in the history for the incredible growth in information. The industrial applicability of SRGMs can be found in the literature by several case studies. Modeling the failure process in a piece of software is a very challenging exercise. Singpurwalla and Wilson (1994) analyzed the statistical methods in software engineering with reliability and risk. Tian (1999) described the recent work in establishing predictive linkage between software reliability and other indices that software developers can measure and control early in the development life-cycle of the software. Huang et al. (2003) examined several existing software reliability growth models based on non homogeneous poisson process. Further they proposed a more general NHPP model from the quasi arithmetic view point based on three weighted means. Some NHPP based software reliability growth models with environmental factors were studied by Pham (2000).

Over the past 30 years, most of the software reliability growth models have been proposed to estimate some important performance measures such as mean time to failure (MTTF), mean time to repair (MTTR), mean time between failures (MTBF), number of remaining faults, failure intensity, etc.. Besides these, software reliability growth models are also used to determine the behavior of fault detection and removal/correction process. It was assumed that detected faults were removed or corrected immediately but in real life situations there is a time gap between detection and correction processes. A general framework for modeling the software fault detection and correction process is discussed by Lo and Huang (2006). A SRGM describing the failure-occurrence or fault detection phenomena in the software testing phase was developed by Inoue and Yamada (2007). Huang and Huang (2008) testified the feature of correction lag function by experimental results using finite and
infinite queueing model. Shu et al. (2009) modeled the software fault detection and correction processes based on the correction lag.

It is common in all software reliability models that the occurrence of software failures is a fundamentally random process, which can be described as either a probability distribution or as a stochastic process. Utkin et al. (1996) analyzed reliability growth in the probability and possibility contexts. Kimura et al. (1995) discussed statistical methods in software engineering and its applicability. Stutzke and Smidts (2001) performed probabilistic analysis of stochastic model of fault introduction and removal during software development. Williams (2006) predicted capability analysis of two and three parameters software reliability growth models.

Software testing is a necessary and standard part of development needed to prove the merit of a design and the validity of the models. As far as reliability growth testing is concerned, much work has done into developing various statistical models for the purpose of planning and tracking reliability growth achieved through testing. Neufelder-Owner (2002) made prediction about the facts of software defects and reliability. Pham and Zhang (2003) studied NHPP software reliability and test models with testing coverage. Zhao et al. (2006) explored software reliability growth model with change point and environmental function. Luo et al. (2007) used a weighted laplace test statistic for modeling the software reliability growth. Su and Huang (2007) gave neural-network based approaches for software reliability estimation using dynamic weighted combinational models. Recently, Huang and Hung (2010) analyzed the software reliability by using queueing models with multiple change-points.

1.10.1.1 Flexible SRGM

Several software reliability growth models developed in the literature establish the relationship between the testing time and the corresponding number of faults detected. This phenomenon happens when software contains different type of faults and each fault requires different strategies and different amount of testing efforts for correcting or removing them. Bittanti et al. (1988) explored flexible modeling approach for software reliability growth. Kapur and Bhalla (1992) discussed optimal release policies for a flexible software reliability growth model. Kimura et al. (1992)
considered software reliability assessment for an exponential and S-shaped reliability growth phenomenon. Kapur et al. (2008a) suggested flexible software reliability growth model with testing-effort dependent learning process. Kapur et al. (2009a) studied the stochastic differential equation based flexible software reliability growth model.

### 1.10.1.2 Imperfect Debugging SRGM

Debugging process plays a very important role to determine the remaining fault contents and hence enhancing the reliability of the software. There are two processes during testing such as perfect debugging and imperfect debugging. If imperfect debugging phenomena is ignored, the SRGM may mislead the decision making process. Ohba and Chou (1989), and Leung (1992a) developed the software reliability growth model under imperfect debugging. Xia et al. (1993) evaluated optimal software release policy with a learning factor for imperfect debugging. Kapur et al. (1994b) described general imperfect debugging software reliability growth model. Kapur and Younes (1996) modeled an imperfect debugging phenomenon in software reliability. Zeephongsekul (1996) proposed software reliability growth model under imperfect debugging by considering that the primary-failures generate secondary-faults. Rattihalli and Zachariah (2002) developed NHPP models for the assessment of reliability of the software with imperfect debugging and testing effort. Xie and Yang (2003) studied the effect of imperfect debugging on the software development cost model. Tokuno and Yamada (2003b) defined relationship between software availability measurement and the number of restorations with imperfect debugging. Shyur (2003) analyzed the stochastic software reliability model with imperfect debugging and change point. Gokhale et al. (2006) incorporated fault debugging activities into software reliability models. In imperfect debugging environment, Pham (2007) discussed fault detection dependent parameter software. Cai et al. (2010) did the mathematical modeling of software reliability with imperfect debugging. S-shaped SRGM with imperfect debugging phenomenon and testing effort function was studied by Ahmad et al. (2010).
1.10.1.3 Module Based SRGM

Generally complex software systems consist of several modules or components; these are tested independently during the testing and finally in the last stage of testing, the complete software in which modules and components are interconnected is tested under the simulated environment according to the user Littlewood (1979) offered software reliability model for modular program structure. Kubat (1989) assessed the reliability of modular software system. Xie and Wohlin (1995) analyzed an additive reliability model for the analysis of modular software failure data. Kapur et al. (2004b) evaluated the software reliability growth model with testing effort dependent learning function for the distributed systems. Lo et al. (2005) gave reliability assessment and sensitivity analysis of software reliability growth modeling based on software module structure. Huang and Lo (2006) discussed optimal resource allocation for cost and reliability of the modular software systems in the testing phase. Hou et al. (2009) analyzed the reliability of component software in wireless sensor networks based on transformation of testing data. Module based testing approaches for software projects were selected by Dias and Travassos (2009). Software module testing and resource allocation dependent optimal release policies were given by Rafi and Akthar (2010).

1.10.1.4 SRGM with Testing Effort

The testing is technically and economically important for high quality software production. For developing the software, about half of the expenses have been estimated because of the testing expenditure. Software testing has become most important process in the software life cycle. SRGMs proposed by several authors incorporated traditional testing efforts (TE) like Exponential TE, Rayleigh TE, Wiebull TE functions. Many papers have been published based on TEF in NHPP models.

Testing effort is defined as effort needed to detect and remove the errors during testing. Software testing consumes a testing effort during the testing phase. A large number of papers are published in this context. Several papers have appeared in software reliability literature by incorporating testing efforts like exponential, Rayleigh and Weibull type curve. Parr (1980) gave an alternative to the Rayleigh
Chapter-1: General Introduction

curve for software development effort. Yamada et al. (1987) explored software reliability growth models with testing–effort and its applications. Vouk (1990) analyzed back-to-back testing of the software. Yamada et al. (1993a) used weibull testing-effort function in software reliability growth model. Whittaker (1992) analyzed markov chain techniques for software testing and reliability. Whittaker and Poore (1993) and Whittaker and Thomson (1994) stated that Markov models provide a rich body of statistics useful in test planning and analysis of test results and also offer an effective means of test case generation. Joint effects of test effort and learning factor on the software reliability were analyzed by Chatterjee et al. (1997). Franid et al. (1998) equated testing methods by delivered reliability. Haung et al. (2000) integrated the logistic testing effort function into S-shaped model under imperfect debugging environment. They suggested that the logistic testing-effort functions are well suitable during the software development. Kuo et al. (2001) classified the framework for modeling the software reliability, using various testing-effort and fault-detection rates. Stringfellow et al. (2002) estimated the number of components with defects in testing. In 2007, Huang et al. incorporated the logistic testing effort function into both exponential and S-shaped reliability growth model under both type of conditions namely ideal and imperfect debugging. Kapur et al. (2009b) unified the framework for developing testing effort dependent software reliability growth models. Rafi et al. (2010) developed the software reliability growth model with logistic exponential test effort function. In 2010, Jain and Jain analyzed the testing effort dependent SRGM with multiple change-point concept.

1.10.2 Operational Reliability and Optimization Issues

The decision for software developer that what is the best time to stop software testing can be derived from the optimal release policies. Many mathematical models have been proposed to determine the optimal release strategies for computer software testing, based on optimal stopping formulation. The effects of reliability and cumulative costs of software testing have demonstrated that expected average costs of software testing can be reduced by restricting the scope of testing procedures (cf. Thayer et al., 1976). Forman and Singpurwalla (1977) and Forman and Singpurwalla (1979) suggested optimal stopping rule for debugging and testing computer software. Goel and Okumoto (1980) studied optimum release time for
software systems based on reliability and cost criteria. In addition to the optimal releasing problem, whether software should be released at all can be incorporated with the decision process.

Chapter-1: General Introduction

Ahmad (2010) gave optimal software release policies with new modified Weibull testing effort function.

1.11 Contents of the Thesis

Most of the current topics of software reliability consist of the formulation and analysis of various mathematical models, often in the form of differential equations, difference equations, differential-difference equations or stochastic differential equations etc.. The present thesis is devoted to the mathematical analysis and performance prediction of software systems in different frameworks for quantifying reliability growth. The thesis is organized into ten chapters counting the present chapter on “General Introduction” wherein some relevant fundamental concepts have been discussed. Numerical results are provided to authenticate the analytical results. These results are summarized in the form of tables and graphs as per requirements. The related references to our investigations have been provided in alphabetical order in the end of the thesis. The chapter wise organization of the thesis is as detailed below.

Chapter 1: General Introduction

The first chapter entitled “General Introduction” throws light on basic concepts and terminology of software reliability including the reliability, MTTF and availability. The concepts of imperfect debugging, error generation, testing effort, change point and relevant literature have been reported. Some significant software reliability growth models (continuous and discrete) are also discussed. Finally organization of the thesis and concluding remarks are given.

Chapter 2: Optimal Testing Time for Software Reliability Growth Models

In this chapter, component based software model has been discussed. We address the optimality issue of testing time with warranty. The total maintenance cost is minimized to achieve desired level of reliability. Warranty cost is also analyzed for two cases when reliability growth does not occur during the warranty period, and reliability growth occurs during the warranty period. The sensitivity analysis has also been facilitated to visualize the effects of certain key parameters on the performance indices.
Chapter 3: Modular Software Reliability Growth Model with Environmental Effects and Change-Point

Generally, it is assumed that during the testing phase, environment changes because of the complexity of the software. Therefore we have investigated the total cost and reliability of integrated software under the effects of changing environment. Maximum likelihood technique is used for estimating the parameters. We investigate the optimization problem, mean value function, total expected cost and reliability for a integrated software system.

Chapter 4: Fuzzy Reliability Model for Log-Logistic Testing-Effort Dependent Software Growth

This chapter deals with the cumulative number of errors removed for calculating the corresponding reliability. The mean value function is evaluated according to the fault’s severity such as simple, hard and complex. A special testing-effort function (lag-logistic testing effort) is described that enjoys the advantage of dealing with the proposed model. For checking the validity of the analytical results, fuzzy logic is applied and the results are compared with conventional results.

Chapter 5: Warranty Cost Analysis of Software Reliability Growth Model (SRGM) with Imperfect Debugging and Change-point

In this chapter, change-point concept is used with imperfect debugging for constructing the software reliability growth model. To evaluate the total expected cost, a warranty cost model with phases is considered for determining the optimal release time. The applicability of the proposed model is established via numerical illustration.

Chapter 6: Free Rectification Lifetime Warranty (FRLTW) Policy for Repairable Product

This chapter is devoted to the study of lifetime warranty of a repairable product. It is assumed that as time increases, the number of failures increases. For the free rectification lifetime warranty (FRLTW), the expected cost is evaluated and finally sensitivity analysis is facilitated for supporting the analytical results.
Chapter 7: Testing and Operational Reliability for a Distributed Software System with Correction Lag Constraints

This chapter is devoted to the study of testing and operational reliability for a distributed developed software system. Software system is distributed according to the fault severity namely reused components as well as newly developed components. Reused components have simple faults and newly developed components have hard and complex faults. A time dependent correction lag function is proposed for correcting the detected and isolated errors. Some performance measures such as total expected cost, reliability and optimal release policies are derived. Sensitivity analysis is carried out by using Adaptive Network-based Fuzzy Interference Systems (ANFIS). The neuro-fuzzy results are compared with the conventional results by taking numerical illustration.

Chapter 8: Discrete Reliability Growth Models for Flexible Softwares

This chapter is concerned with discrete flexible software reliability growth models based on NHPP. For predicting the error removal phenomenon, probability generating function (PGF) is used. It is assumed that the fault detection and removal show the flexibility of the model. Fault generating function and imperfect debugging with learning factors are also discussed. Numerical illustrations support the validity of analytical results.

Chapter 9: Markovian Software Reliability Model for Two types of Failures with Imperfect Debugging Rate and Generation of Errors

This chapter focuses on the study of software reliability model based on Markov process. Two types of software failures are considered for developing the model under the imperfect debugging environment and error generation phenomenon. For finding the solution of the system of differential-difference equations, Laplace transforms and matrix method are employed. Numerical results with the help of Runge-Kutta Method are given.
Chapter 10: NHPP Error Detection Software Model with Repair

This chapter is concerned with the optimal software release time with the goal that total cost of the software can not exceed a given budget and requirements. A non homogeneous software error detection model with repair is developed to study the reliability growth. From the manufacturer point of view, the optimal release time of the software is suggested.

1.12 Conclusion

Modern society almost relies on the complex systems which contain software. In fact dependency on the software is increasing, and with it amount of software is also increasing. Being a crucial point of concern of all, software reliability modeling has motivated us to analyze the software growth in different frameworks.

Stochastic modeling has been playing an important role in the study of software reliability growth. In this chapter, we have discussed various key factors of the software reliability growth models. An overwhelming number of continuous and discrete models have been reported in literature to address the issue of software reliability growth. The important works related to our work have also been reported in this chapter.

It is noticed, that the hardware reliability growth models have been developed during the last fifty years but software reliability growth models have been developing during the past thirty years. Based on different assumptions and philosophy of software phenomena, we have described various models for software reliability growth. In general, these models are based on non-homogenous poisson process (NHPP) and an analyst could choose the “best” model for his use. The optimal stopping formulation suggested may be helpful to determine the most favorable time to discontinue testing. The models developed in the present thesis exhibit the release options that can be chosen at any time during the software testing phase, in order to avoid wasting effort in the development process. To ensure ongoing software quality, the new release options of the software are suggested in different scenario. These release policies suggest improved and error-free versions and the process of providing new versions throughout the software life-cycle. Our study will provide an insight to the software developers and testers to agree with high quality software. Based on our
investigations, a strategic quality framework can be chalked out during the
development of the software life cycle. Software reliability measures suggested may
be embedded in an organizations and software engineering process to produce more
improved software as far as quality is concerned under techno-economic and
reliability constraints.