Chapter 1
Introduction

The computer industry is booming exponentially. The computer revolution is fueled by an ever more rapid technological advancement. With a continuously lowering cost and improved control, processors and software-controlled systems offer compact design, flexible handling, rich features and competitive cost. Like machinery replaced craftsmanship in the industrial revolution, computers and intelligent parts are quickly pushing their mechanical counterparts out of the market. Software’s impact on our society and culture continues to be profound, as its importance grows, the software community continually attempts to develop technologies that will make it easier, faster, and less expensive to build high-quality computer programs.

Today, software takes a dual role. It is a product and, at the same time, the vehicle for delivering a product. As a product, it delivers the computing potential embodied by computer hardware or, more broadly, a network of computers that are accessible by local hardware. Whether it resides within a cellular phone or operates inside a mainframe computer, software is information transformer-producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as a single bit or as complex as a multimedia presentation.

Because of rapid technological advancement, the size and complexity of computer-intensive systems has grown dramatically and the trend will certainly continue in the future. Contemporary examples of highly complex hardware/software systems can be found in projects undertaken by ISRO, NASA, defense, telecommunications industry and a variety of other industries. Although advances have been made towards the production of defect free software, any software required to operate reliably must undergo extensive testing and debugging. This can be a costly and time consuming process, and managers require accurate information about how software reliability grows as a result of this process in order to effectively manage their budgets and projects. The effects of this process, by which it is hoped software is made more
reliable, can be modeled through the use of Software Reliability Growth Models, hereafter referred to as SRGMs. Ideally, these models provide a means of characterizing the development process and enable software reliability practitioners to make predictions about the expected future reliability of software under development. Such techniques allow managers to accurately allocate time, money, and human resources to a project, and assess when a piece of software has reached a point where it can be released with some level of confidence in its reliability.

In stark contrast with the rapid advancement of hardware technology, proper development of software technology has failed to keep pace in all measures including quality, productivity, cost and performance. The demand for complex hardware/software systems has increased more rapidly than the ability to design, implement, test and maintain them. When the requirements for and dependencies on computers increased, the possibility of crises from computer failures have also increased. The impact of these failures ranges from inconvenience (e.g., malfunctions of home appliances) to economic damage (e.g., interruptions of banking systems) to loss of life (e.g., failures of flight systems or medical software). Needless to say, the reliability of computer systems has become a major concern for our society. Recent literature is replete with horror stories of projects gone awry, generally as a result of problems traced to software (Lyu, 1996).

1.1. Software Reliability Engineering

(Musa, 1975) has defined Software Reliability Engineering (SRE) as: “SRE is a practice that helps one develop software that is more reliable, and helps one develop it faster and cheaper. It is a standard, proven, widespread best practice that is widely applicable to systems that include software. SRE is low in cost and its implementation has virtually no schedule impact”.

SRE works by quantitatively characterizing and applying two things about the product: the expected relative use of its functions and its required major quality characteristics. The major quality characteristics are reliability, availability, delivery date and life-cycle cost. While in the applying SRE, one can vary the relative emphasis on these factors. Once the expected future usage of the software
is thoroughly understood SRE tools can be applied for improving quality. It also maximizes test efficiency by making test highly representative of use in the field. SRE is centered around a very important software attribute: Reliability. Software Reliability is one of the attributes of software quality, a multidimensional property including customer satisfaction factors like functionality, usability, performance, serviceability, capability, installability, maintainability and documentation. As the definition of reliability is user oriented, it has become the fundamental quality attribute of any product, be it software or hardware.

Software Reliability is generally accepted as the key factor in software quality since it quantifies software failures—the most unwanted event. Software fails due to the presence of latent faults. IEEE has suggested the following definition for terms “software error”, “fault” and “failure” as:

**Error:** An error is a mental mistake made by the programmer or designer.

**Fault:** A fault is the manifestation of that error in the code.

**Failure** A software failure is defined as the occurrence of an incorrect output as a result of an input value that is received with respect to the specification.

A failure can occur before or after system delivery, during testing, or during operation or maintenance. Some faults may never turn into failures if faulty code is never executed or a particular state is never entered. Nevertheless software cannot be made fault free and these faults certainly lead to failures. This necessitates application of Software Engineering tools, techniques and procedures (Software Engineering is a discipline whose objective is to establish and use sound engineering principles to produce economically developed quality software). But the concern about failure free operation, has forced developers to lay special emphasis on minimizing the number of defects in the software. Defects can creep into the software during any stage of its development process. These faults need to be identified and removed. Software Reliability Engineering is a discipline that aims at ensuring failure free operation of software at the user end employs scientific tools and techniques during testing to remove the maximum number of faults.
1.2. Software Reliability

According to ANSI, Software Reliability is defined as: the probability of failure-free software operation for a specified period of time in a specified environment. Software Reliability is an important attribute of software quality, together with functionality, usability, performance, serviceability, capability, installability, maintainability, and documentation. Software Reliability is hard to achieve, because the complexity of software tends to be high. While any system with a high degree of complexity, including software, will be hard to reach a certain level of reliability, system developers tend to push complexity into the software layer, with the rapid growth of system size and ease of doing so by upgrading the software. For example, large next-generation aircraft will have over one million source lines of software on-board; next-generation air traffic control systems will contain between one and two million lines; the upcoming international Space Station will have over two million lines on-board and over ten million lines of ground support software; several major life-critical defense systems will have over five million source lines of software. While the complexity of software is inversely related to software reliability, it is directly related to other important factors in software quality, especially functionality, capability, etc. Emphasizing these features will tend to add more complexity to software.

1.3. The failure curve for Hardware and Software Reliability

Over time, hardware exhibits the failure characteristics shown in Figure 1.1, known as the bathtub curve with three periods, (a) burn-in phase, (b) useful life phase and (c) end-of-life phase.

![Figure 1.1: Failure curve for hardware reliability.](image-url)
Software reliability, however, does not show the same characteristics similar as hardware. A possible curve is shown in Figure 1. 2 if we project software reliability on the same axes. There are two major differences between hardware and software curves. One difference is that in the last phase, software does not have an increasing failure rate as hardware does. In this phase, software is approaching obsolescence; there is no motivation for any upgrades or changes to the software. Therefore, the failure rate will not change. The second difference is that in the useful-life phase, software will experience a drastic increase in failure rate each time an upgrade is made. The failure rate levels off gradually, because of the defects found and fixed after the upgrades.

![Failure curve for software reliability due to up-gradation](image)

**Figure 1.2:** Failure curve for software reliability due to up-gradation

The upgrades in Figure1.2 imply feature upgrades, not upgrades for reliability. For feature upgrades, the complexity of software is likely to increase, since the functionality of software is enhanced. Even bug fixes may be a reason for more software failures, if the bug fix induces other defects into software. For reliability upgrades, it is possible to incur a drop in software failure rate, if the goal of the upgrade is enhancing software reliability, such as a redesign or re-implementation of some modules using better engineering approaches, such as clean-room method. The Cleanroom Software Engineering process is a software development process intended
to produce software with a certifiable level of reliability. The focus of the Cleanroom process is on defect prevention, rather than defect removal.

1.4 Software Development Life Cycle (SDLC)

Software development process is often called Software Life Cycle, because it describes the life of a software product from its conception to its implementation. Every software development process model includes system requirements as input and a delivered product as output. Many life cycle models have been proposed, based on the tasks involved in developing and maintaining software, but they all consist of the following stages and faults can be introduced during any of these stages.

1.4.1. Stage 1: Requirement analysis and specification

Requirement analysis phase precedes all other activity for software development. During this phase the system service, constraints and goals are established. The user requirement are understood and specified. Though actual designing or coding is done, incorrect, missing or unclear requirements as specified by the user can lead to faults. In the Specification the translations of requirements into precise description of the externals of the software system are done. Hence defective software can result if user requirements are misinterpreted.

1.4.2. Stage 2: Design phase

In the Design phase the requirements and specifications are transformed into a structure that is suitable for implementation in some programming language. Here, overall software architecture is defined, and the high level and detailed design work is performed. Misinterpretations are common when the system design is translated into lower-level descriptions for program design specifications.

1.4.3. Stage 3: Implementation and Unit testing

During Implementation and Unit testing phase, the software system is created, which implements the design. Unit testing ensures that each unit meets its specifications. The programmers and designers of the development team may commit some logical
errors that cannot be pointed out by the compiler. The passing of parameters between modules is not considered during unit testing and can hide a fault.

1.4.4. Stage 4: Testing Phase

In system testing, the individual programs are integrated and tested to determine whether the implementation satisfies the requirement. Though this phase aims at removing all the faults included in the above phases, faults can be introduced into the software during the removal process. The theory developed in this thesis primarily addresses the testing phase.

1.4.5. Stage 5: Operational Phase

During Maintenance in the operational phase faults are corrected, which were not discovered during the previous stages, and enhances the performance of the software system. Incorrect user documentation, poor human factors and changes in requirement can lead to failure reports by the users.

The efforts to improve the software development process are accompanied with parallel efforts aiming at ensuring high quality software systems. The software quality assurance consists of those procedures, techniques and tools applied by professionals to ensure that a software product meets or exceeds pre-specified standards during software development cycles. The quality of the software system has many attributes such as complexities, maintainability, portability, usability, security, reliability, availability etc. (Pfleeger, 1998). Most of these attributes are subjective and can be useful to the software development team only. To express the quality of the software system to the end users, some objective attributes such as reliability and availability should be assessed. The software reliability is the most dynamic quality attribute (metric), which can measure and predict the operational quality of the software system. Higher reliability of software implies that the software is expected to meet the user requirements.

The issue of designing reliable software has acquired its importance due to the following reasons:
Chapter 1

Introduction

1. Systems are becoming software intensive.
2. Many software intensive systems are safety critical.
3. Software users are demanding reliable, warranted software systems.
4. The cost of software development is increasing.

The above mentioned reasons share one implied factor, the risk of facing a failure when the software is delivered to the end users and its consequent high cost. To minimize the risk of software failure in the field, the software is continuously verified and validated through each stage of software development process. The objective of the software validation is to ensure that we are developing the right product, whereas the software verification ensures that we are developing the software right (Pfleeger, 1998, Musa, 1987). In other words the verification process involves that the software conforms to specifications while validation involves checking that the software meets the user’s requirements.

The only way to verify and validate the software is by testing. The software testing involves running the software and checking for unexpected behavior of the software output. The successful test can be considered to be the one, which reveals the presence of the latent faults. Therefore, the software should be thoroughly tested to expose as many software faults as possible. Obviously, testing the software through executing all its statements and paths (even at least once) is not practically possible in a large-scale software system. A compromise testing approach involves dividing the software into blocks and executing each block at least once. The potentially troublesome combinations of blocks are executed many times (Pfleeger, 1998). However, testing techniques can be categorized into two main techniques:

1. Top-down testing involves starting the test at the sub-system level. Then, the modules, which comprise the sub-system, are tested. The procedure is recursively repeated until the test reaches the lowest level software component (function, object).
2. Down-Top testing involves reversing the previous process.
Each testing technique has its merits and demerits. Choosing the testing approach mainly depends on the software development technique. The software testing is a destructive process as it aims at forcing the software to behave abnormally under some conditions. For this reason, the software programmers subconsciously avoid bringing their product into this stage. Therefore, it is preferred that an independent team tests the software. This testing method is called Independent Validation and Verification (IVV), which mean that the testing team is functionally independent from the development team and they test the software from the user’s point of view. This method is also called black box method. Many testing methods exist each with certain advantages and disadvantages. Generally, testing is the process of establishing the existence of the software faults.

The process of locating the faults and designing the procedures to remove them is called the debugging process. The process of fault removal (repair) involves rewriting the code if the fault is due to coding and design error or changing the requirements (which require doing major repair). The chronology of failure occurrence and fault removals can be utilized to provide an estimate of the software reliability and the level of fault content. In this light, there is a need to develop a tool that can utilize this information to help the software engineers and managers in monitoring the process of the testing. The Software Reliability Model (SRM) is the tool, which can be used to evaluate the software quantitatively, develop test status, schedule status and monitor the changes in reliability performance. Software reliability modelling and related issues have been in focus among software developers, managers, researchers and scientists which has led to the emergence of new discipline - Software reliability engineering.

1.5. The Software Process Models

Software engineering follows some structured models for software development. In this section, a generic overview of the different software development methodologies that are currently in use has been provided.

A Software Model describes the phases of the Software Life Cycle and the order in which those phases are executed. In the following, we discuss the important process
models with their advantages and disadvantages (Musa, 1975, Pham, 2006, Kapur et al., 1999):

- Waterfall Model
- Iterative Model
- Incremental Model
- Spiral Model
- V- Model

### 1.5.1. Waterfall Model

This is the most common, easy to implement and classic of all the life cycle models. It is also referred as a Classic Life Cycle Model or linear-sequential life cycle model. This model places a lot of emphasis on documentation, i.e. Requirements Specification and Design Document. In a waterfall model each phase must be completed sequentially in its entirety before the immediate next phase can begin. A review is done after each phase to analyze whether the project is running as per the required standards, specifications and timelines. The output of each phase is the input to the immediate next phase in this model. This model is well suited if the project requirements are static or have been clearly stated from the beginning. The Waterfall Model is great for specifying the individual tasks, roles, and deliverables in the project life cycle. It is a misleading model when used for project planning because it de-emphasizes iteration and incremental delivery. Advantages of the Waterfall Model are as follows:

1. It is very simple and easy to implement. It is well suited for small projects.
2. Testing is inherent to each of the phases of this model.
3. The model is rigid and each of the phases has certain deliverables and a review process immediately after a particular phase is over.

Also the disadvantages of the Waterfall Model can be counted as:

1. It is high risk.
2. It cannot be guaranteed that one phase of this model is perfect before we move on to the immediate next phase in the model.

3. It is not suited for long or complex projects or projects where the requirements can change.

4. The deliverable software is produced late during the life cycle.

1.5.2. Iterative Model

The Iterative Model addresses many problems associated with the Waterfall Model. In the Iterative Model, analysis is done the same way as it is done in the Waterfall model. Once this analysis is over, each requirement is categorized based on their priority as High, Low, Medium.

The advantages of the Iterative Model are:

1. Faster Coding, testing and Design Phases
2. Facilitates the support for changes within the life cycle

In addition the disadvantages of the Iterative Model given as:

1. More time spent in review and analysis
2. A lot of steps that need to be followed in this model
3. Delay in one phase can have detrimental effect on the software as a whole.

1.5.3. Incremental Model

The incremental model divides the software to be developed into modules which are then developed and tested in parallel. These modules or cycles are divided into smaller, easily managed iterations. Each iteration passes through the requirements, design, implementation and testing phases. The Incremental model allows full SDLC of prototypes to be made and then tested before moving to next level. In this model the functionality is produced and delivered to the customer incrementally. Starting from the existing situation, we proceed towards the desired solution in a number of steps. At each of these steps the Waterfall Model is followed. The advantages of the Incremental Model are as follows:
1. Deliverables are produced early in the software development lifecycle in each iteration.
2. It is flexible and easy to manage
3. Risk Management and Testing is easy

And the disadvantages of the Iterative Model are:

1. Each phase of iteration is rigid and does not overlap each other.
2. All the requirements are not gathered up front for the entire software life cycle which can create problems at the later stages in the design and development cycle.

1.5.4. Spiral Model

The Spiral Model or the Spiral Development Model combines the best of both top down and bottom up approaches and is specifically risk-driven. It combines the features of both the prototyping and the waterfall models. In essence the Spiral Model is a combination of the classic Waterfall Model and Risk Analysis. It is iterative, but each iteration is designed to reduce the risk at that particular stage of the project. The Spiral Model provides a rapid development and at the same time, incremental versions of the software application. The Spiral model is better than the Waterfall Model in the sense that it emphasizes more on risk management while the Waterfall Model emphasizes more on the project management aspects. The spiral model has four phases. These phases are as follows:

- Planning
- Risk Analysis
- Engineering
- Evaluation

The following are the advantages of the Spiral Model.

1. It has strong support for Risk Analysis.
2. It is well suited for complex and large projects.
3. The deliverable is produced early in the software development life cycle.
4. It uses prototyping as a risk reduction technique and can reduce risks in the SDLC process considerably.

Also the disadvantages of the Spiral Model are:

5. It is high in cost and Risk Analysis is also very difficult.
6. It is not suited for small projects.

1.5.5. V-Model

V-Model of testing incorporates testing into the entire software as well as application development life cycle. In the Figure 1.3, the V proceeds down and then up, from left to right depicting the basic sequence of development and testing activities. The model highlights the existence of different levels of testing and depicts the way each relates to a different development phase. Like any model, the V-Model has detractors and possibly has deficiencies and alternatives but it clearly illustrates that testing can and should start at the very beginning of the assignment. In the requirements gathering stage the business requirements can verify and validate the business case used to justify the assignment.

**Figure1.3:** The V-model of software process.
The model illustrates how each subsequent phase should verify and validate work done in the previous phase, and how work done during development is used to guide the individual testing phases. This interconnectedness lets us identify important errors, omissions, and other problems before they can do serious harm. (Kurundkar et al., 2010)

### 1.5.6. Which Model Should We Choose?

We discussed many models and their advantages and disadvantages. The question that now arises is, "Which model should we choose?" we should choose the right type of the Model to implement based on the scope of the software project. This depends on number of factors, some of which are given below:

- The Scope of the Project
- The Project Budget
- The organizational environment
- Available Resources

### 1.6. Software Testing

Software testing is an art that involve activities aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. It is a process of designing and executing test cases in a systematic fashion with the intent of detecting and correcting faults in minimum amount of time and effort. It is broadly deployed in every phase of the software development cycle. Typically, more than 40% percent of the development time is spent in testing. Testing as a process has economic, technical and managerial aspects. Economic aspects are related to the reality that resources and time available to the testing team are on a limited basis. In fact, complete testing in many cases is not practical because of these economic constraints. An organization must structure its testing process so that it can deliver software on time and within budget, and also satisfy the client’s requirements.
The technical aspects of testing relate to the techniques, methods, measurements, and tools used to insure that the software under test is as fault free and reliable as possible for the conditions and constraints under which it must operate. Testing is a process and as a process must be managed wherein testing must be planned, testers should be trained and testing procedures and steps must be defined and documented. Software testing indeed is a trade-off between budget, time and quality. The main purposes of testing are Quality Assurance, Verification and Validation and Reliability Estimation.

A strategy of Software Testing that integrates software test case design methods into a well-planned series of steps that results in the successful construction of software is illustrated in Figure 1.4 and are as follows:

- **Unit testing** tests the individual software component or module. Each unit of the software is tested to verify that the detailed design for the unit has been correctly implemented.

- **Integration Testing** exposes defects in the interfaces and interaction between integrated modules. Progressively larger groups of tested software components corresponding to elements of the architectural design are integrated and tested until the software works as a system.

- **System Testing** tests completely integrated system to verify that it meets its functional and non-functional requirements.

![Figure 1.4: Software testing levels](image-url)
Compatibility Testing

Compatibility test insures that the application works with differently configured systems based on what the users have or may have. When testing a web interface, this means testing for compatibility with different browsers and connection speeds. Before shipping the final version of software, alpha and beta testing are often done additionally:

- **Alpha testing** is a simulated or actual operational testing by potential users/customers or an independent test team testing at the developers' site.

- **Beta testing** comes after alpha testing. Versions of the software, known as beta version, are released to a limited audience outside of the programming team. The software is released to groups of people so that further testing can ensure the product has few faults or bugs.

- **Acceptance testing** is finally conducted by the end-user, customer, or client to validate whether or not to accept the product/software.

Software testing methods are traditionally divided into black box testing and white box testing. These two approaches are used to describe the point of view that a test engineer takes when designing test cases.

- **Black box testing** treats the software as a black-box without any understanding of internal behavior. It aims to test the functionality according to the requirements. Thus, the tester inputs data and only sees the output from the test object. This level of testing usually requires exhaustive test cases to be provided to the tester who can verify that for a given input, the output value (or behavior), is the same as the expected value specified in the test case.

- **White box testing** is performed when the tester has access to the internal data structures, code, and algorithms. White box testing methods include creating tests to satisfy some code coverage criteria. For example, the test designer can create tests to cause all statements in the program to be executed at least once. Other
examples of white box testing are mutation testing and fault injection methods. White box testing includes all static testing.

Testing is a process of executing a program with the intent of detecting and removing faults. These faults may prove to be fatal to the users in terms of loss of time, money and even lives depending on criticality of the function as well as to the developers in terms of cost of debugging, risk cost of failure and goodwill loss. The faults in the software can be manifested in each stage of its development. Figure 1.5 shows factors contributing to fault manifestation in the various stages of SDLC.

![Diagram of Sources of Faults in Each Phase of SDLC]

**Figure 1.5:** Sources of Faults in Each Phase of SDLC

One of the key attribute of software testing is to administer *Software Quality* because of the fallibility of its human developers and its own abstract and complex nature. The Department of Defense (1985) in USA defines software quality as “*the degree to which the attributes of the software enable it to perform its intended end use*” which nearly combines the need to provide a good solution with the requirements. Producing and sustaining the high quality of software and processes in evolutionary systems are at the core of software engineering and it is only through a comprehensive
measurement program that a successful outcome can be assured. Cost and budget limitations, schedule due dates, all represent systems engineering constraints which impinge on the degree to which software development and maintenance professional can achieve maximum quality. Quality in turn has three sets of factors *functionality, engineering, and adaptability*. These three sets of factors can be thought of as dimensions in the software quality space. Each dimension may be broken down into its component factors and considerations at successively lower levels of detail. Table 1.1 illustrates some of the most frequently cited quality considerations.

**Table 1.1: Typical Software Quality Factors (Hetzel, 1988)**

<table>
<thead>
<tr>
<th>Functionality (exterior quality)</th>
<th>Engineering (interior quality)</th>
<th>Adaptability (future quality)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correctness</td>
<td>Efficiency</td>
<td>Flexibility</td>
</tr>
<tr>
<td>Reliability</td>
<td>Testability</td>
<td>Reusability</td>
</tr>
<tr>
<td>Usability</td>
<td>Documentation</td>
<td>Maintainability</td>
</tr>
<tr>
<td>Integrity</td>
<td>Structure</td>
<td></td>
</tr>
</tbody>
</table>

Software Reliability is generally accepted as the major factor in software quality among other attributes of software quality such as functionality, usability, capability, maintainability, etc. since it quantifies software failures, which can make a powerful system inoperative.

### 1.7. Software Reliability Growth Modelling

A software reliability model specifies the general form of the dependence of the failure process on the principal factors that affect it: fault introduction, fault removal and the operational environment.
Proliferations of SRMs have emerged as people try to understand the characteristics of how and why software fails, and try to quantify software reliability. Over 200 models have been developed since the early 1970s, but how to quantify software reliability still remains largely unsolved. Many models are there and many more emerging, none of the models can capture a satisfying amount of the complexity of software; constraints and assumptions have to be made for the quantifying process. Therefore, there is no single model that can be used in all situations. No model is complete or even representative. One model may work well for a set of certain software, but may be completely off track for other kinds of problems.

Most software models contain the following parts: assumptions, factors and a mathematical function that relates the reliability with the factors. The mathematical function is usually higher order exponential or logarithmic.

At any particular time, it is possible to observe a history of the failure rate of the software. Software reliability modeling forecasts the curve of the failure rate by statistical evidence. The purpose of this measure is twofold: (1) to predict the extra time needed to test the software to achieve a specified objective; (2) to predict the expected reliability of the software when the testing is finished.

Software generally enjoys reliability growth during testing and operation since software faults can be detected and removed when software failures occur. On the other hand, software may experience reliability decrease due to abrupt changes of its operational usage or incorrect modifications to the software. Software is also continuously modified throughout its life cycle.

- There are two main types of software reliability models: the deterministic and the probabilistic, more details may be found in (Kapur et al., 1999, Musa, 1987, Pham, 2006). The deterministic model is used to study the number of distinct operands in a program as well as the number of errors and the number of machine instructions in the program. Performance measures of the deterministic model are obtained by analyzing the program texture and do not involve any random event. The probabilistic model represents the failure occurrences and the fault removals as probabilistic events.
1.8. Non-Homogeneous Poisson Process (NHPP) based SRGM

The NHPP models provide the expected number of faults/failures at a given time. Schneidewind (1975) proposed an error detection model. Goel and Okumoto (1979) proposed the time dependent failure rate model. (Ohba, 1984) proposed the inflection S-shaped model. Musa (1975) and Musa (1999) proposed the basic execution time model and Log Poisson model respectively. Yamada, Ohba and Osaki (1983) proposed a model based on the concept of failure observation and the corresponding fault removal phenomenon. Goel (1985) modified his original model by introducing the test quality parameters. Yamada, Osaki and Narithisa (1985b) proposed a model with two types of faults. Yamada and Osaki (1985) also proposed two classes of discrete time models. One class describes an error detection process in which the expected number of errors detected per test case is geometrically decreasing while the other class is proportional to the current error content. Kapur et al. (1999) proposed a discrete time model based on the concept that the testing phase has two different processes namely, fault isolation and fault removal. Kapur et al. (1995) further proposed a discrete time model based on the assumption that the software consists of \( n \) different types of faults and on each type of fault a different strategy is required to remove the cause of the failure due to that fault. Ohba (1984) proposed the hyper-exponential model to describe the fault detection process in a module structured software. Khoshgoftaar (1988) proposed the K-Stage Erlangian model. Kapur and Garg (1990) modified G-O model by introducing the concept of imperfect debugging. Kareer et al. (1990) proposed two types of faults models where each fault type is modeled by an S-shaped curve. Kimura et al. (1992) proposed an exponential S-shaped model which describes the software with two types of faults.

In the real life software development projects, the non-uniform testing is more popular and hence the S-shaped growth curve has been observed in many software development projects. The cause of S-shapedness has been attributed to different reasons. Yamada et al., (1984) attributed it to time delay between the fault removal and the initial failure observation which is result of the unskilledness of the testing team at the early stages of the test. Also. Yamada et al. (1986) ascribed it to the non-uniform distribution of the testing-effort. Bittanti et al. (1998) accrued it to the
increased fault detection rate later in the testing phase. Kapur et al. (1999) ascribed it to the presence of different types of faults in a software system. As a result we have a large number of SRGM each being based on a particular set of assumptions that suits a specific testing environment.

The most important criterion in a model selection is the validity of the model assumptions and the relevance of these assumptions to the real testing environment. Besides, the performance of the model in terms of its ability to regenerate the past failure data and to predict the future of the failure observation process are two other important criteria.

1.9. A General Description of Continuous Time Model

Let \([N(t), t \geq 0]\) denotes a counting process representing the cumulative number of failures experienced (fault removed) up to time \(t\), i.e., \(N(t)\), is said to be an NHPP with intensity function \(\lambda(t)\), if it satisfies the following conditions:

i. There are no failures experienced at time \(t = 0\), i.e., \(N(0) = 0\) with probability 1.

ii. The process has independent increments, i.e., the number of failures experienced in \((t, t + \Delta t]\), i.e., \(N(t + \Delta t) - N(t)\) is independent of the history. Note this assumption implies the Markov property that is, \(N(t + \Delta t)\) of the process depends only on the present state \(N(t)\) and is independent of it is past state \(N(x)\), for \(x < t\).

iii. The probability that a failure will occur during \((t, t + \Delta t]\) is \(\lambda(t)\Delta t + o(\Delta t)\), i.e.,

\[
\Pr[N(t + \Delta t) - N(t) = 1] = \lambda(t) + o(\Delta t).
\]

Note that the function \(o(\Delta t)\) is defined as

\[
\lim_{\Delta t \to 0} \frac{o(\Delta t)}{\Delta t} = 0
\]

In practice, it implies that the second or higher order effects of \(\Delta t\) are negligible.
iv. The probability that more than one failure will occur during \((t, t + \Delta t]\) is \(o(\Delta t)\), i.e., \(\Pr[N(t + \Delta t) - N(t) > 1] = o(\Delta t)\).

Based on the above NHPP assumptions, it can be shown that the probability that \(N(t)\) is a given integer \(k\) is expressed by:

\[
\Pr[N(t) = k] = \frac{[m(t)]^k}{k!} \exp\{-m(t)\}, \quad k \geq 0
\]  

(1.1)

The function \(m(t)\) is called the mean value function and describes the expected cumulative number of failures in \((0,t]\). Hence, \(m(t)\) is a very useful descriptive measure of the failure behavior.

The function \(\lambda(t)\), which is called the instantaneous failure intensity, is defined as:

\[
\lambda(t) = \lim_{\Delta t \to 0} \frac{p[N(t + \Delta t) - N(t) > 0]}{\Delta t}
\]  

(1.2)

Given \(\lambda(t)\), the mean value function \(m(t) = E[N(t)]\) satisfies

\[
m(t) = \int_0^t \lambda(s)ds
\]  

(1.3)

Generally, by using different non-decreasing function \(m(t)\), we get different NHPP models.

Define the number of remaining software failure at time \(t\) by \(\overline{N}(t)\) and we have that

\[
\overline{N}(t) = N(\infty) - N(t)
\]  

(1.4)

where \(N(\infty)\) is the number of faults which can be detected by infinite time of testing.

It follows from the standard theory of NHPP that the distribution of \(\overline{N}(t)\) is Poisson with parameter \([m(\infty) - m(t)]\), that is
The reliability function at time \( t_0 \) is given by

\[
R(t \mid t_0) = \exp\left\{ -\left( m(t + t_0) - m(t_0) \right) \right\}
\]  

(1.6)

The above conditional reliability function is called a software reliability function based upon a NHPP for a continuous SRGM.

### 1.9.1. Some Continuous Time Models

A very large number of Continuous Time Models has been developed in the literature to monitor the fault removal process and measure and predict the reliability of the software systems. During testing phase it has been observed that the relationship between the testing time and the corresponding number of faults removed is either exponential or S-shaped or the mix of two. The following are some of the well-established models:

1. Model due to Goel and Okumoto (1979) (purely exponential in nature)
2. Medel due to Yamada et al. (1983) (purely S-shaped)
3. Model due to Ohba (1984) (Flexible)

These models are briefly discussed below.

#### Models Assumptions

1. Software system is subject to failure during execution caused by faults remaining in the system.
2. Failure rate of the software is equally affected by faults remaining in the software.
3. The number of faults detected at any time instant is proportional to the remaining number of faults in the software.

4. On a failure, repair effort starts and fault causing the failure is removed with certainty.

5. All faults are mutually independent from failure detection point of view.

6. The proportionality of failure detection / fault isolation / fault removal is constant.

7. The fault detection / removal phenomenon is modeled by NHPP.

**Models Notations**

- **a**: Initial fault-content of the software.
- **b**: Fault removal rate per remaining fault per unit time.
- **m(t)**: The expected mean number of faults removed by time \( t \).
- **m_f(t)**: The expected mean number of failures occurred by time \( t \).

### 1.9.1.1. Goel-Okumoto Model

Goel and Okumoto (1979) assume that the faults are removed as soon as they are detected. Following differential equation results from above assumptions:

\[
\frac{d}{dt} m(t) = b[a - m(t)]
\]  
(1.7)

The above first order linear differential equation when solved with the initial condition \( m(0) = 0 \), gives the following mean value function

\[
m(t) = a(1 - e^{-bt})
\]  
(1.8)

The mean value function is exponential in nature and doesn't provide a good fit to the S-shaped growth curves that generally occur in Software Reliability. But the model is popular due to its simplicity. Now we briefly discuss below some S-shaped SRGM.
1.9.1.2. Delayed S-shaped SRGM due to Yamada et al. (1983)

Fault detection in this model is assumed to be a two-phase process consisting of failure detection and its eventual removal by isolation. It takes into account the time taken to isolate and remove a fault and so it is important that the data to be used here should be that of fault isolation. It is further assumed that the number of faults isolated at any time instant is proportional to the number of faults remaining in the software. Failure rate and isolation rate per fault are assumed to be same and equal to \( b \).

Thus

\[
\frac{d}{dt} m_f(t) = b[a - m_f(t)] \tag{1.9}
\]

\[
\frac{d}{dt} m(t) = b[m_f(t) - m(t)] \tag{1.10}
\]

\( m_f(t) \) is the expected number of failures in \((0, t]\). Solving (1.9) and (1.10), we get the mean value function as

\[
m(t) = a \left\{1 - (1 + bt)e^{-bt}\right\} \tag{1.11}
\]

Alternately the model can also be formulated as one stage process directly as follows:

\[
\frac{d}{dt} m(t) = \left(\frac{b^2t}{1+bt}\right)(a - m(t)) \tag{1.12}
\]

It is observed that \( \frac{b^2t}{1+bt} \to b \) as \( b \to \infty \). This model was specifically developed to account for lag in the failure observation and its subsequent removal. This kind of derivation is peculiar to software reliability only.
1.9.1.3. Inflection S-shaped SRGM due to Ohba (1984)

The model attributes S-shapedness to the mutual dependency between software faults. Other than assumption 3 it is also assumed that the software contains two types of faults, namely mutually dependent and mutually independent. The mutually independent faults are those located on different execution paths of the software, therefore they are equally likely to be detected and removed. The mutually dependent faults are those faults located on the same execution path. According to the order of the software execution, some faults in the execution path will not be removed until their preceding faults are removed.

Let \( r \) denote the ratio of independent faults to the total number of faults in the software. This ratio is called the inflection parameter \( (0 < r \leq 1) \). If all faults in the software system are mutually independent \( (r = 1) \) then the faults are randomly removed and the growth curve is exponential. According to the assumptions of the model, the fault removal intensity per unit time can be written as

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

(1.13)

\( b(t) \), the fault removal rate at time \( t \) is defined as

\[
b(t) = b\phi(t)
\]

where, \( \phi(t) \) the inflection function is defined as

\[
\phi(t) = r + (1 - r) \frac{m(t)}{a}, \quad \phi(0) = 0 \quad \& \quad \phi(\infty) = 1
\]

(1.14)

\( b \) is the fault removal rate in the steady state. Solving (1.13) under the initial condition \( m(0) = 0 \) we get

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

(1.15)
If \( r = 1 \), the model reduces to the Goel and Okumoto (1979). For different values of \( r \) different growth curves can be obtained and in that sense the model is flexible.

1.9.1.4. SRGM for Error Removal Phenomenon: Kapur and Garg(1992)

This model is based on the assumption that on a failure the detection of the error causing failure also results in the detection of some additional errors without these errors causing any failure. Hence the fault removal intensity per unit time for the model can be written as:

\[
\frac{dm_r(t)}{dt} = p(a - m_r(t)) + q \frac{m_r(t)}{a} (a - m_r(t))
\]

Solving above equation with initial condition \( m_r(0) = 0 \), we get the solution as:

\[
m_r(t) = a \left[ \frac{1 - e^{-(p+q)t}}{1 + \left(\frac{q}{p}\right) e^{-(p+q)t}} \right]
\]

If \( q = 0 \) the model reduces to G-O model and failure phenomenon is same as removal else the failure phenomenon is given as:

\[
m_f(t) = \int_0^t p \left(a - m_r(x)\right) dx = \frac{ap}{q} \ln \left[ \frac{p+q}{p+qe^{-(p+q)t}} \right]
\]

The model can be derived alternatively if we assume logistic learning fault detection rate given by \( b(t) = \frac{b}{1 + \beta e^{-bt}} \) with \( b = p + q \) & \( \beta = (q/p) \), where \( p, q \) are proportionality constants.

1.9.2. SRGMs under Imperfect Debugging Environment

Due to the complexity of the software system and incomplete understanding of the user’s requirements or specifications by the testing team it might happen that the software testing team is not be able to remove the cause of the failure perfectly on its detection and the original fault remains there in the code (known as imperfect
debugging) or they might introduce a new fault during its removal (known as fault generation). Thus it becomes crucial to incorporate the effect of imperfect debugging and fault generation into the software reliability growth modeling.

The concept of imperfect debugging was first introduced by (Goel and Okumoto, 1979). They introduced the probability of imperfect debugging in Jelinski and Moranda (1972) model. The integrated effect of two phenomenon has been discussed by (Kapur et al., 2011). We have discussed the effect of both of these phenomenon in Multi release SRGM in chapter 2 (section 2.2) of this thesis.

1.9.3. SRGM based on Imperfect Fault Debugging: Kapur and Garg (1990)

This Model introduced the probability of imperfect fault debugging on (Goel and Okumoto (1979) SRGM. It was assumed that on a failure the corresponding fault is identified but the detected fault is not removed completely and hence the fault content of the software remains unchanged on the removal action. Under the assumption that on an instantaneous repair effort either the fault content is reduced by one, with probability $p$ or the fault content remains unchanged with probability $1-p$, the rate of change of $m_r(t)$ with respect to time is given by:

$$\frac{dm_r(t)}{dt} = bp(a - m_r(t))$$  \hspace{1cm} (1.19)

Solving (1.19) under the initial condition $m_r(0) = 0$ we get:

$$m_r(t) = a \left[ 1 - e^{bpt} \right]$$  \hspace{1cm} (1.20)

Hence if, $p = 1$ i.e. perfect debugging. It is nothing but GO model.

Corresponding mean number of failures detected in $(0,t]$ is given by:

$$\frac{dm_f(t)}{dt} = b(a - pm_f(t))$$  \hspace{1cm} (1.21)

The solution of above equation under the initial condition $m_f(0) = 0$ is:
\[ m_f(t) = \frac{a}{p} \left[ 1 - e^{-bpt} \right] \]  (1.22)

1.9.4. SRGM based on Error Generation (Ohba and Chou, 1989)

(Ohba and Chou, 1989) formulated a SRGM incorporating the effect of error generation i.e. when a detected fault is removed, it is possible to introduce a new fault. Let \( \alpha \) be the constant fault generation rate, then the differential equation describing the failure phenomenon of the model may be given by:

\[ \frac{dm(t)}{dt} = b(t) [a(t) - m(t)] \]  (1.23)

Where \( b(t) = b, \ a(t) = a + \alpha m(t) \)

Solving the above system of equations with \( m(0) = 0 \) we get:

\[ m(t) = \frac{a}{1 - \alpha} \left[ 1 - e^{-b(1-\alpha)t} \right] \]  (1.24)

Further studies in modeling the error generation phenomenon was due to Pham (1996) in which, an exponential and a linear function is considered for the error content in the software \( a(t) = ae^{at} \) and \( a(t) = a(1+at) \) respectively. Pham (2006), Kapur et al. (1999) further incorporated the effect of learning phenomenon of the testing efficiency in imperfect debugging models due to Yamada since it is expected that the testing efficiency grows with time and hence reduces possibilities of imperfect debugging.

1.9.5. SRGM with respect to Testing Coverage

Testing coverage is defined as the extent or the degree to which software is executed by the test cases. With the running of test cases and corresponding failure-removal processes during testing phase, more and more portions of the software, paths, functions are tested. As testing coverage increases it can help software developer to evaluate the quality of the tested software and determine how much additional effort is needed to improve the reliability of the software. It is an important measure for both
software test-managers as well as users of the software. Several researchers (Pham and Zhang, 2003, Inoue and Yamada, 2004) have developed SRGM considering the relationship between testing coverage and software reliability in the literature.

Pham and Zhang (2003) introduced a model incorporating testing coverage measure into software reliability assessment assuming that failure intensity directly depends on coverage rate $c'(t)$ and the number of remaining faults at current time $t$ and inversely on the uncovered faults given by:

$$\frac{dm(t)}{dt} = \frac{c'(t)}{1-c(t)}[a(t)-m(t)]$$

(1.25)

where the testing coverage function $c(t)$ is assumed to be a non-negative and non-decreasing function of testing time

$$c(t) = \left(1-(1+bt)e^{-bt}\right)$$

(1.26)

and the model was defined under an imperfect debugging environment assuming faults can be introduced with a constant fault introduction rate $\alpha$ during the debugging process.

$$a(t) = a\left(1+\alpha t\right)$$

(1.27)

Solving (1.25) with (1.26) and (1.27) under the initial condition $m(0) = 0$ the mean value function is:

$$m(t) = a\left[1+\alpha t - \frac{(1+bt)}{e^{bt}}\right] - \frac{aa\left(1+bt\right)}{be^{1+bt}} \left(\ln(1+bt) + \sum_{i=0}^{\infty} \frac{(1+bt)^{i+1}}{(i+1)!((i+1))}\right)$$

(1.28)

1.9.6. Modeling Related to Faults Severity

Different faults may require different amount of testing efforts and testing strategy for their removal from the system. In the literature to incorporate this phenomenon, faults are categorized as of different types and are analyzed separately. The first attempt in this category was due to Yamada et al. (1985) who modified G-O exponential SRGM
assuming that there are two types of faults in the software. Pham (2006) incorporated the effect of error generation in GO model to analyze the reliability growth considering three level complexities of the faults. Both models assume different fault detection rate for each type of faults. Kapur et al. (1995) addressed three level complexities of faults considering the time delay between the failure observation and its subsequent removal.

Assuming \( a_1, a_2 \), to be simple and hard faults in a software system \( (a_1 + a_2 = a) \), the simple fault removal process is modeled as the following:

\[
\frac{d}{dt} m(t) = b_1 [a_1 - m(t)]
\]  

(1.29)

By solving the above differential equation, we get

\[
m_1(t) = a_1 (1 - e^{-b_1 t})
\]  

(1.30)

The hard fault removal process is modeled as a two stage process

\[
\frac{d}{dt} m_{j_f}(t) = b_2 \left( a_2 - m_{j_f}(t) \right)
\]  

(1.31)

\[
\frac{d}{dt} m_2(t) = b_2 \left( m_{j_f}(t) - m_2(t) \right)
\]  

(1.32)

solving, we get

\[
m_2(t) = a_2 \left( 1 - (1 + b_2 t) e^{-b_2 t} \right)
\]  

(1.33)

Assuming \( a_1 = \lambda.a \) and \( a_2 = (1-\lambda).a \), the mean value function of the proposed SRGM is

\[
m(t) = m_1(t) + m_2(t) = m_2(t) = \lambda.a.\left(1 - e^{-b_1 t}\right) + (1-\lambda).a.\left(1 - (1 + b_2 t) e^{-b_2 t}\right)
\]  

(1.34)
1.9.7. SRGMs using Change Point

In the software reliability growth phase, the software testing process in a sense, determines the nature of the failure data. There are many factors that affect software testing. These factors are unlikely to be kept stable during the entire process of software testing, with the result that the underlying nature of the failure process is likely to experience major changes. The fault detection rate strongly depends on some parameters like skill of testing team, program size, software testability, defect density and resource allocation. The fault detection rate for the all the faults lying in the software differs on the basis of their severity. In most of the NHPP based SRGMs, the fault detection rate is assumed to be constant. During a software testing process, there is a possibility that the underlying fault detection rate is changed at some time moment called Change Point. This would result in a software failure intensity function either increasing or decreasing monotonically, (Huang, 2005, Zou, 2003). The position of the Change Point can be judged by the graph of actual failure data. The work in this area started with Zhao (1993) who introduced the Change Point analysis in Hardware and Software reliability. Shyur (2003), Huang (2005b), Wang and Wang (2005), Kapur et al. (2006) made their contributions in this area. In this thesis a generalized framework is developed to derive various existing and new two dimensional SRGMs incorporating the effect of Change Point.


Let the parameter \( \tau \) be the Change Point that is considered unknown and is to be estimated from the data. The fault detection/removal rate function is defined as:

\[
b(t) = \begin{cases} 
    b_1 & \text{when } 0 \leq t \leq \tau , \\
    b_2 & \text{when } t > \tau 
\end{cases}
\]  

(1.35)

Under the assumptions described above, the fault removal process may be described by the following differential equation:

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

(1.36)
Solving equation (1.35) using (1.36), we get:

\[
m(t) = \begin{cases} 
a \left(1 - e^{-b_1 t}\right) & \text{when } 0 \leq t \leq \tau \\ a \left(1 - e^{-b_1 \tau - b_2 (t-\tau)}\right) & \text{when } t > \tau \end{cases}
\]

(1.37)

1.10. Two-Dimensional Software Reliability Growth Models

Recently, two dimensional software reliability models have been developed to assess the software quantitatively. The traditional one dimensional model has been dependent upon the testing time, testing effort or testing coverage. However if the reliability of a software is measured on the basis on the number of hours spent on testing the software or the percentage of software that has been covered then the results are not conclusive. The need for developing a two dimensional model is an ideal solution to the problem faced by software reliability engineers. To cater the need of high precision software reliability, we require a SRGM which consider not only the testing time but also the testing coverage of the software i.e. the percentage of code covered of the software. For this we develop two-dimensional SRGMs incorporating the joint effect of testing time and testing effort on the number of faults removed in the software in this thesis.

**Figure 1.6:** Two Metrics of Measuring Software Reliability

The two dimensional modelling framework is based on the Cobb Douglas production function (Cobb and Douglas, 1928). The functional form of production
functions is widely used to represent the relationship of an output to inputs. It was proposed by Knut Wicksell (1851–1926), and tested against statistical evidence by Cobb and Douglas (1928). The Cobb-Douglas function considered a simplified view of the economy in which production output is determined by the amount of labor involved and the amount of capital invested. While there are many factors affecting economic performance, their model proved to be remarkably accurate.

The mathematical form of the production function is given as follows:

\[ Y = AL^\alpha K^\beta, \]  

(1.38)

Where:

- \( Y \) = total production (the monetary value of all goods produced in a year)
- \( L \) = labor input
- \( K \) = capital input
- \( A \) = total factor productivity
- \( \alpha \) and \( \beta \) are the output elasticities of labor and capital, respectively. These values are constants determined by available technology.

Output elasticity measures the responsiveness of output to a change in levels of either labor or capital used in production, ceteris paribus. For example if \( \alpha = 0.15 \), a 1% increase in labor would lead to approximately a 0.15% increase in output.

Further, if: \( \alpha + \beta = 1 \), the production function has constant returns to scale. That is, if \( L \) and \( K \) are each increased by 20%, \( Y \) increases by 20%.

If \( \alpha + \beta < 1 \), returns to scale are decreasing, and

if \( \alpha + \beta > 1 \), returns to scale are increasing.

Assuming perfect competition and \( \alpha + \beta = 1 \), \( \alpha \) and \( \beta \) can be shown to be labor and capital's share of output.
(Inoue and Yamada, 2008.) proposed a two dimensional SRGM using the Cobb-Douglas production function. We derive various two dimensional SRGMs in this thesis using the Cobb-Douglas production function.

1.11. Optimization Problems in Software Reliability

In commercial software development organizations, increased complexity of products, shortened development life cycles, lower production costs, scarce resources and higher customer satisfaction of quality have placed a major responsibility on the management to have a closer look on their decision making strategies, future prediction and analysis in the areas of software debugging, testing and verification. The available resources are needed to be allocated efficiently without compromising with their effectiveness. Optimization Techniques play a significant role to help policy planners, managers and administrators in the analysis of many complex decisions or allocation problems. It offers an indispensable degree of operational simplicity. Skill in modeling to capture the essential elements of a problem and good judgment in the interpretation of results are required to obtain meaningful conclusions. The theory of optimization encompasses the quantitative study of optima and methods of finding them. The two important issues in optimization are the optimal point, which is the goal and seeking improvement, which drives the process towards the optimum solution. In optimization problems values of the decision variables produce the optimal result. The major steps required to perform for solving a typical optimization problem are as follows:

- The system-variables interaction must be known accurately and quantitatively.
- A single or multiple measure of effectiveness must be expressed in terms of system variables.
- The choice of these values of the system variables must yield optimum solution.

The term optimization is believed to be coined by a German mathematical philosopher Leibniz in his book “Essay on goodness of God, the freedom of man and
the origin of evil” (Leibniz et al., 1985). However the optimization problems and their solutions have made their footmarks in the literature as old as 1826 due to the work of (Fourier, 1831). Now a day’s various optimization models and techniques are used successfully in almost every industry, engineering system etc. Research in this field has grown enormously. Various new techniques and algorithms have been proposed in the field of mathematics, operational research and engineering by many scholars in their literature over the years.

1.11.1. Software Release Time Decision Problems

Importance of reliable software has escalated many folds. Notwithstanding its unassailable value, there is still no way to test whether it is completely fault free or not. It is owing to the prevailing paradox that software user’s requirements are conflicting with the developers. Software users demand faster deliveries, cheaper software and quality product whereas software developers aim at minimizing their development cost, maximizing the profit margins and meeting the competitive requirements. A vital decision problem that the management encounters is to determine when to stop testing and release the software system to the user. Such a problem is known as “Software Release Time Problem”. If the release of the software is unduly delayed, the manufacturer (software developer) may suffer in terms of penalties and revenue loss, while a premature release may cost heavily in terms of fixes (removals) to be done after release, which consequently might harm the manufacturer’s reputation.

The optimization problem of determining the optimal time of software release can be formulated based on goals set by the management in terms of cost, reliability and failure intensity etc subject to the system constraints. The use of SRGMs to depict software reliability provides a statistical foundation to establish optimal release time for software testing.

The wealth of research into release time problems is impressive and confirms its continuing importance as a research topic. Release time problems became a prime field of study for many eminent researchers around 1980s where they contributed immeasurably. Huang (2005a), Huang and Kuo (2002), Kapur et al. (2011e), Pham
(2006), Pham (1996), Pham and Zhang (2003), Okumoto and Goel (1980) did some of the pioneering work in this field. Keeping in mind the diverse complexity of a software system they determined Release Time Problems based on different criterion. Since the development of a software system is time-consuming and costly the foremost among all is to minimize the total software development cost because software cannot be tested for an infinite period as it increases its testing cost or it cannot be released prematurely as it increases the cost of fixing the faults in operational phase. On the contrary for safety critical systems it is imperative for the management to achieve prescribed level of reliability before releasing software to the user. As a consequence most of the software release time problems discussed in literature consider the minimization of total software development cost or maximization of reliability subject to budgetary constraint and/or minimum desired reliability level to be achieved by the release time. Models that minimize the number of remaining faults in the software or the failure intensity also fall under this category. Although there has been mammoth work done, there is much scope left in this domain due to global competition, rapid advancement in technologies, changing specifications from the client, increasing complexity of the software system to name a few. In this thesis we discuss optimal software release time in chapter 4.

1.12. SRGM Incorporating Enhancement of Features

Software technologies like all systems have a hierarchy. Any software developed for a major application undergoes several generations of change (see Figure 1.7). The changes improve the software and extend its useful life. In the last few decades it has been observed that the world of software development management (i.e. new product development, technology alliance etc.) has evolved rapidly due to the intensified market competition. In particular the use of continuous up-gradation model of software products is fast becoming commonplace due to the shrinking budget, expanding system requirements and on the other hand accelerating rate of software enhancement. The up-gradation of the system is done by extending it through add-ons, interfacing with other applications etc., e.g. windows operating system (Figure 1.7). The growing trend toward up-gradation of software has taken the original concept of reprocess it into a completely different arena and due to that
it has also presented many challenges to software developers attempting to enter this new arena.

Software companies plan successive releases of software by adding new features or new functionalities or try to improve performance of system as compared to previous releases. Microsoft Windows and Office, Adobe, Matlab represent good examples of such practice. This strategy provides some benefit for software companies. Few of them may be listed as:

- It delivers quicker return to investment, limits the impact of market uncertainty,
- Reduces the risk of project failure,
- Provides firms with opportunities to refine their development methodologies
- Improve product quality in later development cycles.

By this policy, company can use bugs reported from the operational phase of previous releases to removing bugs from the old code of the current release of the software (Garmabaki et al., 2011).

![Figure 1.7: Multi up-gradation of Microsoft Windows 7](image)
By introducing new add-ons or feature, the software will experience a drastic increase in failure rate each time an upgrade is made (see Figure 1.2). The failure rate levels off gradually, partly because of the defects found and fixed after the upgrades. Due to the feature upgrades, the complexity of software is likely to be increased as the functionality of software is enhanced.

Mathematical models have been proved to be useful tool for understanding the structure and functioning of software, predicting the reliability of software and prescribing the best course of actions under known constraints. Therefore, we capture the effect of upgradation with the help of mathematical modeling in this thesis.

1.13. Innovation Diffusion: An Overview

Modelling and forecasting of the diffusion of innovation has been a topic of practical and academic interest since the 1960s and it is applied in several areas like opening up of markets in developing countries, web-based services, virtual social networks, multi generation product and complex product-service structure. Diffusion of Innovations seeks to explain how innovations are taken up in a population. An innovation is an idea, behavior, or object that is perceived as new by its audience. The difference among individuals in their response to new ideas is called their innovativeness; it represents the degree to which an individual is relatively early or late in adopting a new product or services (Midgley and Dowling, 1987). Hence the diffusion process is the aggregate of all individual adoptions over time. There have been many studies on how individuals react to new ideas and products.

There are four key element of diffusion process

1. Innovation
2. Communication Channel
3. Social System
4. Time

1.13.1. Innovation
The term innovation may refer to something new - must be substantially different, not an insignificant change. There are three major types of innovations:

- **Continuous Innovation**: It is a simple change or improvement of an existing product where the adopter is expected to use the product in the same fashion as he/she did before.

- **Dynamically Continuous Innovation**: Here the innovations can either be a creation of a new product or a drastic change to an existing one which requires some change in the way the existing one is used.

- **Discontinuous Innovation**: Here the firm introduces a totally new product in the market. As the product or the innovation has never been seen before, consumer’s attitude towards it would be different in terms of how they use and buy it.

**1.13.2. Time**

It is very unlikely that all the individuals belonging to a social system will adopt the new innovation at the same time. Thus it is very interesting to know the time of diffusion or in other words when or how much time will be taken by an individual or unit in a social system to decide to adopt an innovation after its introduction.

**1.13.3. Communication Channel**

Communication is defined as: “the process by which participants of a process create and share information with one another in order to reach a mutual understanding.” Communication channels are most important elements of the diffusion as they represent the means by which information about an innovation reaches to the members of the social system. There are two broad types of communication channels—Mass media and Interpersonal communication channels. In the initial phase of diffusion only the mass media channels are operative. These are effective in creating knowledge of innovations. Mass media communication channels that includes:

Print media, Broadcast media and Display media. Later when the adoption process starts interpersonal communication channels also accelerates the process of
diffusion through the word of mouth communication. Interpersonal communication channels are more effective in forming and changing attitudes towards a new product. In addition Effect of both types of communication channels is different for the different products.

1.13.4. Social System

Social system in the new product context refers to the population of potential adopters of an innovation. It consists of individual product buyers, organizations or agencies that share a common culture. The role of members of the social system is most important in the diffusion process since they are the eventual adopters of the product. It is important to study the characteristics of the social system since the diffusion system is greatly influenced by the behavior pattern of its members. When all individuals or units in the targeted area have adopted the innovation, the diffusion process is said to be complete.

1.14. The Adoption Categories for New Products

French sociologist (Tarde, 1903) originally claimed that sociology was based on small psychological interactions among individuals, especially imitation and innovation. This process has been studied extensively in the scholarly literature from a variety of viewpoints, most notably in (Rogers, 1962) classic book, *The Diffusion of Innovations*. Rogers proposed that the distribution of adoption of any innovation over time approached normality. Roger on eight different studies found that in the early stage of a particular innovation, growth is relatively slow as the new product establishes itself. At some point customers begin to demand and the product growth increases more rapidly. New incremental innovations or changes to the product allow growth to continue.

Towards the end of its life cycle growth slows and may even begin to decline. In the later stages, no amount of new investment in that product will yield a normal rate of return. Based on a bell curve, Rogers has categorized that adopters of any new innovation or idea as given in the Figure 1.8 and assigning precise notional percentages for each segment as:
- Innovators: 2.5%
- Early Adopters: 13.5%
- Early majority: 34%
- Late majority 34%
- Laggards 16%

**Figure 1.8:** Adoption Categorization on the Basis of the Relative Time of Adoption of Innovations (Source: Rogers 1962)

Rogers further claimed that innovations would spread through society in an S curve (as shown in Figure 1.9), and the speed of technology diffusion is influenced by the product's perceived advantage or benefits, riskiness of purchase, ease of product use - complexity of the product, immediacy of benefits, observability, trialability, price, extent of behavioral changes required, return on investment in the case of industrial products. The s-curve or cumulative-curve maps growth of revenue or productivity in any given potential market against time.

The s-curve or bell-curve can be derived from half of a normal distribution curve. There is an assumption that new products are likely to have “Product Life”. i.e. a start-up phase, a rapid increase in revenue and eventual decline. In fact a great majority among innovations never gets off the bottom of the curve. Innovative companies will typically be working on new innovations that will eventually replace older ones. Successive s-curves will come along to replace older ones and continue to drive growth upwards.
1.15. Innovation Diffusion Modeling

A large number of innovation diffusion models have been developed in literature by Bass,(1969); Bass et al. (1994), Bass and Bass (2001); Mahajan et al. (1990); Norton and Bass (1987); Norton and Bass (1992); Kapur et al. (2011b); Kapur et al. (2010a); Kapur et al. (2010b); Kapur et al. (2010c); Peres et al. (2010); Van-Den-Bulte and Joshi (2007); Van-Den-Bulte and Stremersch (2004); Wilson and Norton (1989a), Jiang (2010). Models of innovation diffusion are used successfully by the various practitioners to evaluate the marked response over the product life cycles and make valuable decisions related to product modifications, price differentiation, etc (Rogers, 1962). The purpose of a diffusion model is to enact the successive increase in the number of adopters and predict the continued development of a diffusion process already in progress. In the year 1969, F.M. Bass proposed a mixed influence model, which later became most famous and widely accepted model of innovation diffusion which is briefly discussed in the following section.


F.M. Bass proposed the basic mixed innovation diffusion model in 1969. This model is appropriate for a single generation production. The model assumes that there exists a finite population of prospective adoption who with time increasingly adopt the
product. He categorized potential adopters in two categories as innovators and imitators depending on which way they get information about the product. Innovators make independent purchase decisions and previous buyers influence imitators by word of mouth. The model can be represented by the following differential equation:

\[
\frac{dN(t)}{dt} = (p + \frac{qN(t)}{N})(\bar{N} - N(t)) \quad (1.39)
\]

where \(N(t)\) represents the cumulative number of adopters by time \(t\), and the three constant parameters \(\bar{N}, p\) and \(q\) denote the potential market size, the coefficient of innovation, and the coefficient of imitation, respectively. Eq. (1.39) shows that the diffusion rate at a given time \(t\) equals the product of (i) the instantaneous probability of adoption at time \(t\), which increases linearly with the number of existing adopters, and (ii) the number of potential adopters who have not adopted by time \(t\). The cumulative number of adoptions and the non-cumulative diffusion rate at time \(t\), denoted by \(S(t)\), can be derived based on Eq. (1.40).

\[
N(t) = \bar{N} \left( \frac{1 - e^{-(p+q)t}}{1 + (q/p)e^{-(p+q)t}} \right) = \bar{N} F(t) \quad (1.40)
\]

\[
S(t) = \bar{N} \left( \frac{p + q}{p} \right) \left( \frac{1 - e^{-(p+q)t}}{1 + (q/p)e^{-(p+q)t}} \right)^2 \quad (1.41)
\]

Figure 1.10(a) shows the typical shape of a Bass diffusion curve (with \(q > p\)). The adoption rate is low when the product is first released. Due to both, the word-of-mouth effect and the external influences such as advertisement, the adoption rate gradually picks up until a peak is reached. Under this condition, the diffusion curve is symmetric with respect to the point \(t^*\), where

\[
t^* = \left[ \frac{1}{(p+q)} \right] \ln \left( \frac{q}{p} \right) \quad (1.42)
\]

After the peak time \(t^*\), the adoption rate decreases due to market saturation effect. Most product with perhaps a few exceptions such as movies, exhibit diffusion pattern
shown in Figure 1.10(a). If $q \leq p$, the diffusion rate decreases monotonically with respect to time $t$. We can put movie on this category and Figure 1.10(b) shows the penetration of diffusion in the market.

Relating the similarity of innovation diffusion with the spreading of an epidemic, imitation is often called a contagion effect. In a pure innovation scenario ($p > 0, q = 0$), diffusion follows a modified exponential distribution in a pure imitation scenario ($p = 0, q > 0$), diffusion follows a logistic curve. Other properties are that $(p + q)$ controls scale and $(q/p)$ controls shape (note that the condition $q/p > 1$ is necessary for the curve to be S-shaped). ([Islam and Meade, 2006, Zhang and Gao, 2011].

![Figure 1.10(a): A typical Bass diffusion curve ($q > p$).](image1)

![Figure 1.10(b): A typical Bass diffusion curve ($q \leq p$).](image2)

1.16. Successive Generation of Technologies

An important feature of most modern new technologies is that they come in successive generations. The clearest examples are from the area of information technology and telecommunication industry. Also the time gap between successive generations of products is reducing and the replacement of earlier technologies with the latest one is occurring quite frequently. With compatibility being no longer a problem area for major product categories, consumer today has more choices and has
the opportunity to choose from new as well as older generations of a technology after evaluating the prices, utilities, risk etc. Adopters of each successive generation consist of four categories of customers, namely, (i) Those who are switching from the earlier generation, (ii) those who adopt the new technology instead of the earlier one, (iii) potential customers who would only adopt the new technology and (iv) adoptions for each generation may also come from adopters of earlier generations and make decision to upgrade the product not exactly in the next generation of the product.

In today’s market, Majority of consumer durables have multiple technological generations (Figure 1.11). As a result it has become strategically more important to study the technological changes and the growth rate of consumer preference towards a generation and corresponding market behavior.

![Series of Technological Generations](image)

**Figure 1.11:** Series of Technological Generations (Source: Norton and Bass 1987)

Since the pioneering work of Bass (1969), lots of other models have been developed using the model to study the technological innovations. Norton and Bass (1987) model is a classic example of multiple generation model, which is again built upon the Bass model. In the model it is assumed that the coefficients of innovation and imitation remain unchanged from generation to generation of technology. But many authors have argued against this assumption. Islam and Meade (1997) have tested the hypothesis of coefficient constancy across generation of Norton and Bass model.
Their empirical work relaxed the assumption of constant coefficient. They proposed that the coefficients of later generation technology are constant increment/decrement over the coefficients of the first generation. Mahajan and Muller (1996) proposed a model, which is again an extension of Bass model to capture simultaneously both the substitution and diffusion patterns for each successive generation of technological products. Bayus (1992) developed a model for consumer sales of second-generation product by incorporating replacement behavior of first generation adopters and suggested different dynamic pricing policies of second-generation consumer durables. Speece and MacLachlan (1995) and Danaher et al. (2001) developed a model in a different way by incorporating price as an explanatory variable. Kim et al. (2000) in their model have tried to capture intergenerational linkages among successive generations within a product category. The authors proposed that the market potential of a generation of a product category not only affected by the technological substitution from another generation within the category but also affected by the adoption rate of the other categories. Recently Kapur et al. (2010b) proposed multigenerational diffusion model to study the marketing dynamics of Indian Television Market (both Black & White and Color Television). They considered the effect of repeat-adoption-substitution diffusion in their model.

1.17. Model Application

Mathematical models are applied on the actual data and model parameters are estimated. These estimates are used to generate information useful to the practitioners or fed to the optimization models to make decisions. It is a known fact that most of the times success of a model in a particular situation is due to judicious choice of situation, process, environment and time frame for evaluating the data. Generation of useful information depends heavily upon quality of collected data and selection of a model relevant to the process and environment. Usually there are two types of failure data: time domain data and interval domain data (Pham, 2006). The time domain data records the individual time at which failure occurred whereas the input domain data is characterized by counting the number of failures occurring during a fixed time period for example hours, weeks, and days. Most reliability
models can handle both types of data as both consider the cumulative occurrence time of the event. Time domain data usually provides higher accuracy in the parameter estimates but involve more data collection efforts than interval domain data that is more readily available and used for practical purpose. These data are used by practitioners for analyzing and predicting reliability applications.

1.17.1. Parameter Estimation

Parameter estimation is of primary importance in software reliability prediction. Once the analytical solution for $m(t)$ mostly described by the non-linear functions is known for a given model, the parameters in the solution are required to be determined. Parameter estimation is achieved by extensively using two estimation techniques for non-linear models namely Method of Non-linear Least Square (NLLS) and Maximum Likelihood Estimate (MLE) [(Kapur et al., 2011, Pham, 2006)].

Non-Linear Least Square Method

Consider a set of observed data points $(t_i, y_i); i = 1, 2,...n$, where $t_i$ is the observation time and $y_i$ is the observed sample value. A mathematical model of the form $m(x, t)$ is fitted on this data set. The model depends on the parameters $x = \{x_i; i = 1, 2,...m\}$, for some $\tilde{x}$ we can compute the residuals,

$$f_i(\tilde{x}) = y_i - m(\tilde{x}, t_i)$$

The method of least square determines the unknown parameters of the model by minimizing the sum square of these residuals for the observed data values. For small and medium size samples least square estimation is preferred (Musa et al., 1987). The method has also been applied for estimating parameter of testing effort curves.

Maximum likelihood Estimation Method

For the interval domain data points $(t_i, y_i); i = 1, 2,...n$, where $t_i$ is the observation time and $y_i$ is the cumulative number of failures by the time $t_i$, based on the NHPP assumptions the likelihood function is defined as
If the data set is time domain the likelihood function is defined as

\[
L = \prod_{i=1}^{n} \left[ \frac{(y_i - y_{i-1})!}{m(t_i) - m(t_{i-1})} \right] e^{-\lambda(t_i)dx}
\]  

Maximum likelihood estimation method yields the unknown estimates of the parameters maximizing the likelihood function. In most of the cases Log of the likelihood function is maximized as log function is monotonic and provides easy computations as compared to the actual likelihood function.

Both these methods have one thing in common that the nonlinear objective function (the sum square residuals in NLLS and the likelihood function in MLE) is optimized. For finding the optimal solution manually one needs to compute the first order partial differentiation equation corresponding to each parameter of the problem, equate them to zero and solve the resulting system of equations. In most of the cases solving this system of equations is difficult and contrary to the linear model fitting. Hence the analytical solution of this optimization problem is difficult to be expressed. Moreover it requires numerical methods and huge computation time to solve the problem, which is not favored by the management and software engineering practitioners. In order to overcome this problem, several Statistical software packages such as SPSS, SAS, R. can be used in which there are inbuilt software functions that can solve these kind of optimization problems for finding the estimates of nonlinear models. In our study we have used the Statistical Package for Social Sciences (SPSS) and Statistical Analyzer Software. SPSS/SAS Regression Models enables the user to apply more sophisticated models to the data using its wide range of nonlinear regression models.

In this thesis, nonlinear regression (NLR) and conditional nonlinear regression (CNLR) modules of SPSS have been used to estimate the unknown parameters. The modules use the iterative estimation algorithms namely sequential quadratic...
programming (SQP) and Levenberg-Marquardt (LM) method to find the least square estimates of the parameters. Both methods start with an initial approximation of the parameters and at each stage improve the objective value until convergence.

1.17.2. Model Validation

1.17.2.1. Comparison Criteria

The performance of selected SRGMs for an application are judged by their ability to fit the past software failure data and to predict satisfactorily the future behavior of the software fault removal process (predictive validity). Kapur et al. (2011e), Pham (2006), Musa et al. (1987) have suggested the following attributes for choosing an SRGM.

**Capability:** The model should possess the ability to estimate with satisfactory accuracy metrics needed by the software managers.

**Quality of assumptions:** The assumptions should be plausible and must depict the testing environment.

**Applicability:** A model can be adjudged as the better one if it can be applied across software products of different sizes, structures, platforms and functionalities.

**Simplicity:** The data required for an ideal SRGM should be simple and inexpensive to collect. The parameter estimation should not be too complex and is easy to understand and apply even for persons without extensive mathematical background. Other than the above qualitative aspects the many established criteria are defined in the literature to validate the *goodness of fit* of models on any particular data and choose the most appropriate one. Some of these criteria are given in following section:

**Goodness of Fit Criteria**

The term goodness of fit is used in two different contexts. In one context, it denotes the question if a sample of data came from a population with a specific distribution.
In another context, it denotes the question of “How good does a mathematical model (for example a linear regression model) fit to the data”?

**The Mean Square Fitting Error (MSE):** The model under comparison is used to simulate the fault data, the difference between the expected values, \( \hat{y}(t_i); i=1,2,...,k \) and the observed values \( y_i \) is measured by MSE as follows.

\[
MSE = \sum_{i=1}^{k} \frac{(\hat{y}(t_i) - y_i)^2}{k}
\]  

(1.46)

where \( k \) is the number of observations. The lower MSE indicates less fitting error, thus better goodness of fit (Kapur et al., 2011, Pham, 2006, Musa et al., 1987).

**Coefficient of Multiple Determination (R\(^2\)):** is defined as the ratio of the Sum of Squares (SS) resulting from the trend model to that from a constant model subtracted from 1, that is

\[
R^2 = 1 - \frac{\text{residual SS}}{\text{corrected SS}}
\]

(1.47)

\( R^2 \) measures the percentage of the total variation about the mean accounted for by the fitted curve. It ranges in value from 0 to 1. Small values indicate that the model does not fit the data well. The larger the value, the better the model explains the variation in the data (Pham, 2006).

**Prediction Error (PE):** The difference between the observed and predicted values at any instant of time \( i \) is known as \( PE_i \). Lower the value of Prediction Error better is the goodness of fit(Kapur et al., 1999, Musa, 1999).

**Bias:** The average of PE is known as bias. Lower the value of Bias better is the goodness of fit [Pillai and Nair, (1997)].

\[
Bias = \frac{1}{k} \sum_{i=1}^{k} (\hat{m}(t_i) - y_i)
\]

(1.48)
**Variation**: The standard deviation of PE is known as variation.

\[
\text{Variation} = \sqrt\left(\frac{1}{N-1}\sum (PE_i - \text{Bias})^2\right)
\]  

(1.49)

Lower the value of Variation better is the goodness of fit (Kapur et al., 1999, Musa, 1999).

**Root Mean Square Prediction Error**: It is a measure of closeness with which a model predicts the observation.

\[
\text{RMSPE} = \sqrt{\text{Bias}^2 + \text{Variation}^2}
\]  

(1.50)

Lower the value of Root Mean Square Prediction Error better is the goodness of fit (Kapur et al., 1999)

### 1.17.2.2. Predictive Validity Criterion

Predictive validity is defined as the ability of the model to determine the future behavior from present and past behavior of a process. This criterion was proposed by (Musa et al., 1987). Suppose \(t_k\) be the time, \(y_k\) is the observed value of the event during the interval \((0, t_k]\), and \(\hat{y}(t_k)\) is the estimated value determined using the actually observed data up to an arbitrary time \(t_k(0 < t_k \leq t_k)\), in which \(t_k / t_k\) denotes the process progress ratio. The difference between the predicted value \(\hat{y}(t_k)\) and the reported value \(y_k\) measures the prediction fault. The ratio \(\left\{\left(\hat{y}(t_k) - y_k\right) / y_k\right\}\) is called Relative Prediction Error (RPE). If the RPE value is negative / positive the model is said to underestimate / overestimate the future process. A value close to zero for RPE indicates more accurate prediction, thus more confidence in the model and better predictive validity. The value of RPE is said to be acceptable if it is within ±10% (Musa, 1987, Kapur et al., 1999). A particular model can also be judged to fit to a given data if the parameter estimates are relatively stable over some particular intervals for the various truncations.
In this thesis parameter estimation and model validation applied to the proposed SRGMs. In this thesis we have proposed some of the NHPP based SRGM models. For each of the models we have provided the motivation behind it, the assumptions on which it is based, and data required for its implementation. We have also carried out the resulting formation of model, the resulting estimates, comparison with other models and goodness of fit.

1.18. Structure of Thesis

The work presented in this thesis focuses on several aspects of multi-release problems in software and successive generations of technologies. The performance of the proposed models is shown with the parameter estimation and model validations on real life data sets existing in software reliability literature and sale data of high technology products. The results obtained are encouraging. The thesis provides an insight into modeling adoption process of multi generation products. Also, it discusses optimal release time policies in software and optimal introduction timing for new generation in marketing.

Chapter 1 is introductory and explains the basic concepts of software development process, software reliability modeling, release time problems, marketing models, multi generation modeling and data analysis related to the understanding of the research work. The existing research and applicability of the research carried in the thesis in modeling software reliability and successive generations of technologies is highlighted in this chapter. Rest of the thesis is organized into five chapters as follows:

Chapter 2 is divided in three sections. In Section 1, a multi up-gradation reliability model is developed for the software with faults of different severity and imperfect debugging environment. The proposed model is based on the assumption that the overall fault removal of the new release depends on the reported faults from the just previous release of the software and on the faults generated due to adding some new functionalities (add-ons/up-gradations) to the existing software system.
Section 2 discusses about change of nature of fault during successive releases of the software. In the proposed model, we consider that undetected simple faults of old code may be removed as simple fault in new release or it may also happen that simple faults of old code are removed as hard fault during new release. But the hard faults of old code are assumed not to change their nature during testing of next release and they are removed as hard only. Section 3 includes a unified modeling framework based on hazard rate function to develop various multi up-gradation model using different failure distributions under imperfect debugging phenomenon. Parameter estimation, model validation have been done for all the models discussed in the chapter using real data sets cited in literature.

Chapter 3 discusses about multi release SRGMs for fault detection-correction processes and the effect of reported bugs. In some cases, fault is not corrected immediately once a failure is detected. For each detected fault, it is reported, diagnosed, verified, and then corrected. The time between detection and correction should not be neglected in practical software testing process. The time to remove a detected fault depends on the complexity of the fault, the skill and experience of the debugging team, the available manpower, the software development environment, and so on. Also due to successive release policy, company can use bugs reported from the operational phase of previous releases to remove bugs from the old code of the current release of the software. During testing of multi release software, it is quite possible that some of faults of old code are removed directly by testing team of new release (without using any bug reports of operational phase of the previous release of software) and some others are removed on the basis of bug reported during operational phase.

This chapter is divided in two sections. In first section, fault detection and correction process have been discussed and a unified framework for fault detection and correction processes for successive release of software is proposed. In second section, we consider the combined effect of bugs encountered during testing of present release and user reported bug from operational phase. The model developed takes into consideration the testing and the operational phase where fault removal phenomenon
follows Kapur-Garg model and Weibull-model, respectively. All models are validated on real datasets for four release software system.

**In chapter 4** we have discussed two-dimensional modeling technique in area of software reliability engineering. In order to assure software quality and assess software reliability, many SRGMs have been proposed. In One-Dimension SRGMs researcher used one factor such as Testing-Time, Testing-Effort or Coverage, etc for designing the model but in Two-Dimensional software reliability growth model, process depends on two-types of reliability growth factors like: Testing-Time and Testing-Effort or Testing-Time and Testing-Coverage or any combination between factors. The two dimensional models are developed using Cobb-Douglas production function. We have also considered the case when failure distribution gets affected by factors, such as the running environment, testing strategy and resource allocation. Once these factors are changed during testing phase, it could result in failure intensity function that increases or decreases non-monotonically and the time point corresponding to abrupt fluctuations is called change point. In Section 1 we have developed a unified framework for two-dimensional SRGM with change-point for software reliability assessment. In Section 2, a two-dimensional multi release SRGM has been discussed. Also we propose a software cost model for multi up-gradation software reliability model in two-dimensional environments. In order to determine optimal time and effort for the release of a new version of the software, multi attribute utility theory has been used. This technique attempts to identify relevant objectives for any given decision making problem, where a decision is typified by multiple objectives. It can be difficult to quantitatively compare these objectives one against another. In order to provide insight into this problem, a utility function is assessed for each of the relevant objectives. This allows for an appropriate multiple-objective utility function that is used to identify trade-offs and compare the various objectives in a consistent manner. Numerical illustrations are given to justify the release time problems and lastly sensitivity analysis is given for few parameter based on time and effort, separately. The last two chapter of thesis extend the modeling framework to the marketing research. This extension of the work from software reliability to marketing research establishes the interdisciplinary nature of the work compiled in the thesis.
The first section of **chapter 5** is based on the behavioral assumptions of diffusion theory, we propose an extension of the Bass diffusion model that separates substitution from switching as well as leapfrogging. The proposed methodology is built upon the assumption that adopters of each successive generation consist of four categories of customers, namely, (i) Those who are switching from the earlier generation, (ii) those who adopt the new technology instead of the earlier one, (iii) potential customers who would only adopt the new technology and (iv) adoptions for each generation may also come from adopters of earlier generations and make decision to upgrade the product not exactly in the next generation of the product. Moreover, we categorize switching adopters in two groups, i.e. those who previously purchased earlier generation products and proceed to the adoption of the new generation without discarding the earlier ones, or those who discard the earlier generation product and adopt only the new. We also present a relationship between our model and Norton-Bass model.

In section 2 of **chapter 5**, we have discussed about the introduction time of a new product to the market which is in development phase that commands a large commitment of time, money, and managerial resources. Determination of optimal introduction time is especially critical for high-technology products, where the introduction of each successive generation of a product requires the firm to explicitly consider its impact on the demand for preceding generations. Each generation has unique expectations, experiences, generational history, lifestyles, values, and demographics that influence behavior of potential buyers. Accordingly, many companies are reaching out to multi-generational consumers and trying to understand and gain the attention of these diverse buyers. The timing decision depends on whether companies invest more time for product design or push the product to market before maturity. The study identifies attributes such as *Customer’s Adoption Indicator* and *Cost* that affect the introduction time of new generation. To trade-off between two decision factors, multi-attribute utility theory (MAUT) is applied in our decision space. We examine the case where a firm introduces successive generations of a durable product for which demand is characterized by an innovation diffusion
process. In both cases the empirical implications of the proposed models have been validated on data collected from two industries.

In chapter 6 we propose a modeling framework which accounts for the interactions between different dimensions of adoption i.e. time and price. Incorporating the dynamics of continuation time of the product in the market and the price has allowed us to model the diffusion process in two-dimensional framework. Technological adoptions and the role of other dimensions are explicitly taken into consideration using the Cobb-Douglas production function. For many technology products customer interactions through word-of-mouth can reduce the product-uncertainty and can play an effective role in the acceptance of the new product. The other marketing-mix variable viz price also play important role in diffusion of a technology product. A potential individual would become a purchaser after satisfying himself of the utility of the product vis-à-vis the unit price. The model proposed in this chapter is for successive generations of product. It decomposes the total sales into first time purchasers, switchers and substitutes. Substitution and switching are taken to be the two main factors of multi generational modeling. These assumptions are intuitively consistent with the view that attitudes change towards technology as it evolves from one generation to the next generation. The proposed model has been compared with a well established model from the literature. The result obtained are encouraging and the findings are consistent with the idea that attitudes of purchasers towards new generations change from the previous one and it is imperative to identify a trend.

Conclusion of the work done and the scope for future research has also been briefly discussed. An elaborate list of references is presented at the end of thesis.

The thesis is based on the contents of following research papers, which have been published/ communicated/ presented at international/ national conferences.


Management (IEEM), 1539-1543.


* Not included in the thesis.