Chapter 1

Introduction

Computers are bringing revolutionary changes to our life with their involvement in most human-made systems for sensing, communication, control, guidance, and decision-making. When the requirements for and dependencies on computers increase, the crises of computer failures also increase. The impact of hardware and software failures range from inconvenience (malfunctions of home appliances) and economic loss (interceptions of banking systems) to life-threatening (failures of flight systems and medical software).

As the functionality of computer operations becomes more essential and complicated in the modern society, the reliability of computer software becomes more important and critical. In fact, computer software had already become the major source of reported outages in many systems. This trend has been signified by hardware components of a system that has become increasingly reliable, and software starts to dominate the cause of computer system failures and outages. As the demand for software increases, its size, complexity, and criticality also increases. Today, the growth in utilization of software components is largely responsible for the high overall complexity of many system designs, since it is the integrating potential of software that has allowed designers to contemplate more ambitious systems encompassing a broader and more multidisciplinary scope.

Within the last decade of the 20th century and the first few years of the 21st century, many reported system outages or machines crashes were traced back to computer software failures. Consequently, recent literature is replete with horror stories due to software problem.

1. Even following the costly “Y2K” problem as a design failure, a problem that occupied tens of thousands of programmers in 1998-99 with the costs running to tens of billion of dollars, there have been many other critical failures.
2. Software failures have impaired several high visibility programs in space, telecommunication, and defense and other industries. The Mars Climate Orbiter crashed in 1999. The Mars Climate Orbiter Mission Failure Investigation Board (1999) concluded that “the root cause of the loss of the spacecraft was the failed translation of English units into metric units in a segment of ground-based, navigation-related mission software.


Man-rated software, software that is in control of systems and environments upon which human life is critically dependent, must receive special treatment throughout its lifecycle to assure demanded safety, reliability, and quality levels have been attained.

There are three major factors involved in "certifying" the safety and mission readiness of the space shuttle onboard man-rated software. The first involves proving that the actual process used to define, design, develop, test, and verify the software meets established standards for man-rated software, as specified by the procurer and that precisely "that" process has been followed without exception, unless formally approved deviations have been documented.

The second factor involves the maintaining of sufficiently detailed defect density history and failure history of software throughout multiple applications of the specified process over entire lifecycles to monitor, measure, and manage the quality of each system. Basic characteristics of software dictate that

1. Safety certification is currently based on "process adherence" rather than "product."
2. Assumption is that a known, controlled, repeatable process will result in a product of known quality.
3. This assumption requires constant statistical revalidation.
4. The relationship between quality and reliability must be established for each software system and statistically demonstrated for the required operational profile.
5. Quality must be built into the software, at a known level, rather than adding or determining the quality after development.

The third factor builds upon the results of the preceding two factors, combining sound engineering judgment and a systematic evaluation of the software failure probability. This should involve reliability modeling in addition to actual failure mode identification and risk assessment. Ideally, there is a specified reliability level the software must be "certified" to exceed for a specific set of required operational scenarios.

In practice, however, such a concise and deterministic quality assessment is very elusive. For example, there is not sufficient time realistically possible for most software systems to demonstrate a 0.0000001 failure probability. Software statements can be randomly accessed and classical analysis techniques such as "fault-free analysis" and "critical system identification" frequently breaks down due to the infinite possibilities for an "abstract entity" such as software to "behave."

The approach to reliability assurance for the space shuttle primary avionics software involves the systematic removal and elimination of all the known software failure modes and a common defect search and removal process that theoretically leaves the software "error free." The software is then executed through an extensive suite of nominal and off-nominal scenarios selected to cover the operational profile envelope required for shuttle missions. Since software cannot deteriorate, wear out, or fatigue once defects are removed, they cannot regenerate within the software until it is changed subsequently. This approach then relies on prevention of new defects and detection and permanent removal of latent defects.
Since there are virtually an infinite number of permutations and combinations of software paths in the shuttle avionics software, this approach is extremely inspection-dependent. Inspections employ detailed checklists and are automated where possible. Using the data accumulated by application of the second factor described above, statistical and analytical reliability models are employed to estimate the fault and failure densities that remain at the beginning of operational use to provide added confidence (allow for risk assessment) to the conventional engineering judgment that the system is "ready for flight." Fault density is much easier to determine than remaining failures. Accurate models for remaining failures as a function of both fault density and observed failure intensity have been validated. Reiterative revalidation of the model applications and calibration of the models to the process in use have produced reasonably accurate estimates of remaining failures when compared with actual system performance over multiyear periods of operational use.

1.1 Software Engineering

Software Engineering (SE) is the design, development and documentation of software by applying technologies and practices from computer science, project management engineering, and application domains, interface design, digital asset management and other fields. It is concerned with the conception, development and verification of a software system. This discipline deals with identifying, defining, realizing and verifying the required characteristics of the resultant software. These software characteristics may include functionality, reliability, maintainability, availability, testability ease-of-use portability and other attributes. Software engineering addresses these characteristics by preparing design and technical specifications that, if implemented properly, will result in software that can be verified to meet these requirements. Software engineering is also concerned with the characteristics of the software development process. In this regard, it deals with characteristics such as cost of development, duration of development, and risk in development of software. Hence choice of the Software Development Life Cycle (SDLC) model adopted is critical to the success of any software development project Lyu (1996), Musa (1999), Pfleeger (1998), Boehm (1989), Roger S. Pressman (1997).
1.2 Software Development Life Cycle (SDLC)

A Software Development Life Cycle is a set of activities, together with ordering constraints among them, such that if the activities are performed properly and in accordance with the ordering constraints, the desired result is produced. A Software Process Model is an abstract representation of a software process that can be used to explain different approaches to software development. Various software process models have been proposed by researchers based on different paradigms like: the Waterfall Model, the Boehm’s Spiral Model, Incremental model, Concurrent Development Model, etc. However, all these models are based on a set of related software engineering activities that form the generic process framework. Figure 1.1 depicts these process framework activities as the steps of a ladder that a software developer needs to climb in order to achieve the desired result. These process framework activities are described as follows:

(a) Requirement Analysis & Specification

Requirements specification is one of the most critical activities in the software development process. All the requirements of the system to be developed are captured in this phase. These requirements include attributes such as product reliability, availability, performance, constraints and goal that the end-user (who will be using the system) expects from the system. The requirements are gathered from the end-user by consultation. These requirements are analyzed for their validity and feasibility. Finally, a Requirement Specification document is created which serves the purpose of guideline for the next phase of the process model. The requirement document must specify all functional and performance requirements; the formats of inputs and outputs; quality standards and all design constraints. Incorrect, mislaid or unclear requirements specified by the user can lead to faults.

(b) Planning

This activity establishes a plan for software engineering work that follows. It defines the system to be developed based on the system priority according to the organization’s critical success factor (CSF). It describes the project scope, develops the projects plan-
that is all the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, cost and work schedule. Finally it manages and monitors the project plan that allows the organization to deliver the project on time.

(c) System & Software Design

This activity is concerned with a process of problem-solving and planning for a software solution. Here the requirements and specifications are transformed into a structure that is suitable for implementation in some programming language. The overall software architecture is defined and the high level and detailed design work is performed. The design engineers must take utmost care that the designed system architecture, program structure and algorithm design conforms to the specification document, since the design of a system is perhaps the most critical factor affecting the quality of the software. Any anomaly in the design phase could be very expensive to be solved in the later stage of the software development.

(d) Coding

The goal of the coding phase is to translate the design of the system into a high level programming language like C, C++, Pascal and Java. The right programming language is chosen according to the type of application. Programming tools like Compilers, Interpreters, and Debuggers are used for generation of code. The coding phase affects both testing and maintenance phase profoundly. Since the testing and maintenance cost of software is much higher than the coding cost, the main focus of this phase is to develop simple and clear programs in order to reduce the testing and maintenance cost.

(e) Unit, Integration & System Testing

Testing is the major quality control measure used during software development. The basic function of this framework activity is to uncover the faults in the software introduced during the requirement, design and coding phase, to demonstrate the presence of all specified functionalities and to predict the operational reliability of the product within the specified budget and optimal time. The starting point of testing is unit testing, where the
different modules or components are tested individually for their functionality followed by integration testing which focuses on testing the interconnection between modules. After the system is put together system testing is performed to see if all the requirements are met. Finally acceptance testing is performed to demonstrate the operation of the system to the client. It is the key method for dynamic verification and validation of a system. A number of testing tools and methods are available for testing purpose.

(f) Implementation & Maintenance

The software is delivered to client and deployed at its end for operational use. However, some problems with the system developed (which are not found during the development life cycle) may come up after its practical use starts, so the issues related to these need to be solved after deployment of the system. Not all the problems come in picture directly but they arise time to time and needs to be solved; hence this process is referred as Maintenance. In this phase the software is also updated to meet the changing customer needs hence enhancing the efficiency of the software.
1.3 Software Bugs

Reliability has become increasingly important, especially for server applications that require high availability. System failures can degrade system performance, crash systems and corrupt important data, which would significantly reduce system availability and lead to huge loss in productivity and business. According to a report by Gartner Group the average cost of an hour of downtime for a financial company is more than $6 million. Software bugs in critical systems have caused airplane crashes, shut down nuclear reactors, and resulted in many other disasters. Unfortunately, software bugs continue to be frequent. A number of studies on different types of systems have shown that software bugs are one of the major causes of system failures. For Tandem systems, Gray's report in 1986 shows that software bugs caused 25% of system failures. A report in 2000 shows that
software errors account for more than 40% of system failures. According to the National Institute of Standards and Technology (NIST), software bugs cost the U.S. economy about $59.5 billion annually, approximately 0.6% of the gross domestic product. In order to improve software quality, most software companies put a lot of effort on software testing and debugging. An exploratory study shows that the attempts to reduce the number of delivered errors are estimated to account for 50 - 80% of the development and maintenance effort.

1.3.1 Characteristics of Software Bugs

To design effective tools for improving software quality requires a good understanding of software error characteristics in representative software. Such characteristics include bug root causes, impact, resolution time, and correlations among them. For example, if many bugs are caused by simple typos or copying-and-pasting, software development tools can provide more support to help detect these automatically. In testing, developers can focus on bugs based on the severity of the impact so that resource can be utilized more effectively.

Many previous empirical studies, including a few classic ones, have been performed more than ten years ago to understand the characteristics of software bugs. For example, several researchers have studied software errors occurring during software development, testing and validation phases. Sullivan and Chillarege analyzed error type; defect type and error trigger distribution for shipped code of two IBM database management products and one IBM operating system. They found that undefined state errors dominated but did not have high impact on availability, while memory allocation errors, pointer errors, and synchronization errors had high impact. These studies provide useful insights and guidelines for software engineering tool designers and reliable system builders.
1.3.2 Detecting Software Bugs

Identifying and fixing software bugs is one of the most time-consuming and difficult tasks to reduce the errors in software. In order to improve programmers' efficiency, quite a few debugging techniques and tools have been proposed recently. The approaches can be
classified into two categories: dynamic checking and static checking. Some examples of dynamic tools are Purify, Valgrind, DIDUCE, Eraser, and CCured. They have more accurate information but can introduce overheads during execution. Moreover, most of them can only find bugs on the execution paths, so they depend on good test cases that are usually hard to provide. In contrast, static tools, including explicit model checking and program analysis, do not incur any overheads during execution. Further, they can find bugs that may not occur in the common execution paths.

However, most static tools require significant involvement of programmers to build checking models, write specifications, and annotate programs. It is difficult to build good checking models and specifications since it requires that programmers understand the target system thoroughly. Further, it is a tedious and time-consuming task for the programmers to write such specifications and annotations. Therefore, most of the existing bug detection tools are based on some general checkers that are applicable to any applications. For example, a program should never dereference a null pointer; a program should release memory that it no longer needs. Based on such well-known rules, quite a few memory-related bug detection tools have been proposed, which are quite effective in detecting the well-known bugs such as null pointer dereference, memory leak, buffer overrun, and so on.

### 1.3.3 Bug Classification

Most previous empirical studies on bug characteristics were based on a small set (usually with 60-500) of software bugs for the entire software revolution, which may result in a large experimental error and misleading results. For example, some bugs may not be included in the sampled set but could account for a significant fraction in the whole bug database of the evaluated software. Moreover, small size datasets also make it difficult to study the bug trend with the software evolution because some time intervals under studied may only contain very few or no bug samples. Fortunately, most of the open-source projects maintain bug tracking systems to track bugs and code changes. Some of them contain hundreds of thousands of bug reports, which are a valuable asset for us to...
understand the bug characteristics. However, it is challenging to obtain useful information from such large datasets written in human natural languages. For example, Bugzilla database for Mozilla contains more than 300,000 bug reports. It is impractical to verify every bug report by manual investigation. Text mining techniques such as document categorization and information retrieval is a suitable approach to analyze such large datasets that are written in human natural language. Using text mining, it can relieve heavy human work to classify and analyze such huge amount of data. This thesis proposes using natural language text classification and information retrieval techniques to automatically classify a large number of bug reports. Such a large dataset enables us to provide more accurate results such as trends of bug types with software evolution and the complexity of bug fixing, which are usually difficult to draw representative and accurate results from small datasets.

1.4 Software Testing

Software testing is the activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. Software testing is an important software quality assurance activity. A successful test should show that a program contains bugs rather than that it works fine. Testing is inherent to every phase of the Software Development Life Cycle (SDLC). It is an enforced disciplined approach. As the software is created and added to the developing system, testing is performed to ensure that it is working correctly and efficiently. The failure data during the testing process are taken down in order to estimate the software reliability. The testing process functions with regular feedback from the reliability analysis to the testers and designers, which is shown in Figure-1.3.
Although crucial to software quality and widely deployed by programmers and testers, software testing still remains an art due to limited understanding of the principles of software. The difficulty in software testing stems from the complexity of software, we cannot completely test a program with moderate complexity. Testing is more than just debugging. It is usually performed for the following purposes:

(a) To improve quality.
(b) For Verification and Validation (V and V)
(c) For reliability estimation (Lyu 1996)

As computers and software’s are used in critical applications, the outcome of a bug can be severe. Bugs can cause huge losses. In a computerized embedded world, the quality and reliability of software is a matter of life and death.

Quality means the conformance to the specified design requirement. Quality cannot be tested directly, but we can test related factors to make quality visible. Quality 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: Software quality characteristics

<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>

Good testing provides measures for all relevant factors. The importance of any particular factor varies from application to application. Any system where human lives are at stake must place extreme emphasis on reliability and integrity.

In the typical business system usability and maintainability are the key factors, while for a one-time scientific program neither may be significant.

1.5 Types of Software Testing

There is a plethora of testing methods and testing techniques serving multiple purposes in different life cycle phases. Classified by purpose, software testing can be divided into: correctness testing, performance testing, reliability testing and security testing. Classified by life-cycle phase, software testing can be classified into the following categories: requirements phase testing, design phase testing, program phase testing, evaluating test results, installation phase testing, acceptance testing and maintenance testing. By scope, software testing can be categorized as follows:
1.5.1 Black Box Testing

Black box testing takes an external perspective of the test object to derive test cases these tests can be functional or non-functional, though usually functional. The test designer select valid and invalid input and determines the correct output. There is no knowledge of the test object’s internal structure.

As a black box tester, one needs not to have the understanding of the program code. Tester doesn’t have to be able to read the code and one doesn’t need to be able to write the code. What one must be able to do is “test the black box”: the “box” being whatever program, widget, or doodad is placed in front of tester.

What is meant here is as a tester though one doesn’t have all the details of each nook and cranny of the code, one must gain an understanding of how the program works. One must be able to exercise the code through the user interface (UI) as thoroughly as one could if one had access to the code.

1.5.2 White Box Testing

White box testing (a.k.a. clear box testing, glass box testing or structural testing) uses an internal perspective of the system to design test cases based on internal structural. It requires programming skills to identify all paths through the software. The tester chose the test case inputs to exercise paths through the code and determines the appropriate outputs. As a white box tester, one will be expected to the parts of the code that one will be responsible for testing and the tools one will test with as soon as possible. Being able to read through the code well enough to spot weakness will allow one to quickly exploit the most vulnerable parts of the program.

1.5.3 Grey Box Testing

This type of software testing is exactly as it sounds: a mix of Black box and White box. It attempts to adapt the strength of each type and meld them into a “whole” testing that is greater than the sum of its parts. Gray box can take the easy-of-use, straight forward
1.5.4 Manual Testing

Manual testing requires a tester to execute each and every test step by hand. Every button to be pressed, each link to be selected, all assets to be verified are done by human. Thus the term ‘manual’ is justified.

Manual types of software testing can be very time consuming and quite laborious it is, however where the Quality Assurance Professional can, and does, most effectively advocate for the end-user. By looking at every page, by reading every help link, by analyzing every graphic, and by exercising all use paths available to the consumer, one will contribute one’s expertise to help create the highest quality product possible.

Manual testing is an absolute necessity if a human is necessary to judge whether or not the end-user experience is satisfactory. Certain types of software testing lend themselves more clearly to manual testing than others.

Types of software testing- manual testing focus:

(a) Game Testing- where human evaluation of the user experience is valued

(b) Internet Testing- where CAPTCHA (completely automated public turning test to tell computer and human apart) is used to prevent bots from attacking websites.

(c) End-User Testing- when design and development are happening so quickly and speed of feed-back from ad-hoc testing is at a premium.

1.5.5 Automated Testing

Automated testing is testing that is executed using a tool (either off-the-shelf or created in-house) to exercise the code. The tests are set up by a QA tester or QA engineer using the tool so that in the future a battery of tests can be run by pressing a single button or entering a single command.
The tool maybe complex or it may be simple point-and-click interface that records the actions the tester takes. Once the test steps and actions to be performed are defined, they are recorded so that they can be executed any time in the future.

This takes more than a couple of minutes. Once tester have defined and recorded the steps, one must run one’s automated test script and fix any issues that will produce a falsely negative result. What is meant by this is that one must ensure that all of the results one tests produce are reliable. Each failure tester’s test script reports should be an actual failure—not a false alarm. This is why tester must take time to adjust one’s test script.

After tester has set up one’s test automated test scripts, they must be maintained to keep up with the changes that are made in the product one is testing. If for instance tester’s scripts begin reporting that a link is not working, one has to be sure that the link itself is bad. If the placement of the link on the page has changed, but tester’s test script has not been updated to take this change into account, then one will see a failure. But this failure will be a false negative. If tester reports this false negative one will appear to know what one is doing.

1.6 Software Reliability

Reliability is an important attribute of software quality, together with functionality, usability, performance, serviceability, capability, install ability, maintainability, and documentation. According to ANSI (American National Standards Institute), 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 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. 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.
There are three major components in the definition of software reliability: time, failure and operational environment (Musa, Iannino and Okumoto 1987):

(a) Time

Reliability quantities are defined with respect to time, although it is possible to define them with respect to other bases like program runs. We are concerned with three types of time: the execution time for a software system is the CPU time that is actually spent by the computer in executing the software, the calendar time is the time people normally experience in terms of years, months, weeks, days, etc., and the clock time is the elapsed time from start to end of computer execution in running the software. In measuring clock time, the periods during which the computer is shut down are not counted.

b) Failure

A software failure is an incorrect result with respect to the specification or an unexpected software behavior perceived by the user at the boundary of the software system, while software fault is the identified or hypothesized cause of the software failure. When a time basis is determined, failures can be expressed in several ways: the cumulative failure function, the failure intensity function, the failure rate function, and the mean time to failure function. The cumulative failure function (also called the mean value function) denotes the average cumulative failures associated with each point of time.

The failure intensity function represents the rate of change of the cumulative failure function. The failure rate function (or called the hazard rate, or the rate of occurrence of failures) is defined as the instantaneous failure rate at a time \( t \), given that the system has not failed up to \( t \).

The mean time to failure (MTTF) function represents the expected time that the next failure will be observed. (MTTF is also known as MTBF, mean time between failures.) Note that the above three measures are closely related and could be translated with one another.
(c) Operational Profile

The operational profile of a system is defined as the set of operations that the software can execute along with the probability with which they will occur. An operation is a group of runs that typically involve similar processing.

1.7 Software Reliability Engineering

With the ever increasing role that software is playing in today's and tomorrow's world, the software developers and users are asking: "Just how 'good' is the software?" and "How much testing should be done before the software is released?" The software reliability methodology attempts to provide quantitative measures to help answer these questions. Software reliability is one of the important parameters of software quality and system dependability.

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 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.

Musa (1999) has defined Software Reliability Engineering (SRE) as: “SRE is a practice 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 works 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.
In applying SRE, one can vary the relative emphasis on these factors. Once the expected future usage of the software is thoroughly understood software reliability engineering tools can be applied for improving quality. SRE is centered on 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.

Achieving highly reliable software in the customer's perspective is a demanding job to all software engineers and reliability engineers. Four technical methods are applicable to achieve reliable software systems:

(a) **Fault Avoidance**

The interactive refinement of the user's system requirement, the engineering of the software specification process, the use of good software design methods, the enforcement of structured programming discipline, and the encouragement of writing clear code are the general approaches to avoid faults in the software. These guidelines have been, and will continue to be, the fundamental techniques in preventing software faults from being created.

Recently, formal methods have been attempted in the research community in attacking the software quality problem. In formal-methods approaches, requirement specifications are developed and maintained using mathematically trackable languages and tools. Current studies in this area have been focused on language issues and environmental supports, which include at least the following goals:

(a) Executable specifications for systematic and precise evaluation,
(b) Proof mechanisms for software verification and validation, and
(c) Development procedures which follow incremental refinement for step-by-step verification.

Every work item, be it a specification or a test case, is subject to mathematically verification for its correctness and appropriateness.
Another fault avoidance technique, particularly popular in the software development community, is software reuse. The crucial measure of success in this area is the capability to prototype and evaluates reusable synthesis techniques. This is why object-oriented paradigms and techniques are receiving much attention nowadays, largely due to their inherent properties in enforcing software reuse.

(a) Fault Removal

When formal methods are in full swing, formal design proofs might be available to achieve mathematical proof-of-correctness for programs. Also fault-monitoring assertions could be employed through executable specifications, and test cases could be automatically generated to achieve efficient software verification. However, before this happens, practitioners will have to rely mostly on software testing techniques to remove existing faults. Microsoft, for example, allocates as many software testers as software developers, and employs a "buddy" system which binds the developer of every software component with its tester for their daily work. The key question to reliability engineers, then, is how to derive testing quality measures (e.g., test coverage factors) and establish their relationships to reliability.

Another practical fault removal scheme which has been widely implemented in industry is formal inspection. A formal inspection is a rigorous process focused on finding faults, correcting faults, and verifying the corrections. Formal inspection is carried out by a small group of peers with a vested interest in the work product during pretest phases of the life cycle. Many companies have claimed its success.

(b) Fault Tolerance

Fault tolerance is the survival attribute of computing systems or software in their ability to deliver continuous service to their users in the presence of faults. Software fault tolerance is concerned with all the techniques necessary to enable a system to tolerate software faults remaining in the system after its development. These software faults may or may not manifest themselves during system operations, but when they do, software fault tolerance
techniques should provide the necessary mechanisms to the software system to prevent system failure from occurring.

In a single-version software environment, the techniques for partially tolerating software design faults include monitoring techniques, atomicity of actions, decision verification, and exception handling. In order to fully recover from activated design faults, multiple versions of software developed via design diversity are introduced, in which functionally equivalent yet independently developed software versions are applied in the system to provide ultimate tolerance to software design faults.

The main approaches include the recovery blocks technique, the N-version programming technique, and the N self-checking programming technique. These approaches have found a wide range of applications in the aerospace industry, the nuclear power industry, the health care industry, the telecommunications industry, and the ground transportation industry.

(c) Fault\Failure Forecasting

Fault\failure forecasting involves formulation of the fault\failure relationship, an understanding of the operational environment, the establishment of reliability models, the collection of failure data, the application of reliability models by tools, the selection of appropriate models, the analysis and interpretation of results, and the guidance for management decisions. This has been the main focus of software reliability modeling.

Due to the intrinsic complexity of modern software systems, software reliability engineers have to apply a combination of the above methods for the delivery of reliable software systems. These four areas are also the main theme of the state of the art for software engineering covering a wide range of disciplines

1.8 Importance of Mathematical Modeling

Modeling is a strong tool to plan the steps in development of the system and to check the feasibility of ideas before final actions on it. It provides a base for testing and testability of
the system before the system is actually implemented. Moreover, Modeling reduces the complexity of the system and hence saves resources.

While designing a system, we need to analyze and understand it from different angles & view points. So many different conceptual, logical, physical and other types of complexities need to be solved. For this we use different design strategies and models. Modeling creates a base for implementation of the system and it’s testing. It gives specifications for its use by customer. It explains the design idea in a presentable form. A model explains the system at different levels of abstraction. At a time, it provides a selective aspect study of the system. The designer can use it to explain the architecture to the professionals and abstract and relevant details to others. It is important to mention here that the design is only a concept that describes the system understanding of the designer. It tells the thinking of an experience-holder about the strategy to be adopted for development of the system in hand. As we know that all human does not think in the same way. Some may have better understanding in textual form while others may prefer graphic notations of the design. Here the model efforts to bring them on an almost common platform where the conflicts of the design may be resolved and some consistent design for implementation may be proposed. Model is also important to remove the silly mistakes while developing the system and also creates a scale for evaluation while development. Its output is something more than a simple software system with its better understanding, documentation and maintenance tips.

1.9 Software Reliability Measurement

Software reliability measurement includes two types of activities, reliability estimation and reliability prediction:

(a) Reliability Estimation

This activity determines current software reliability by applying statistical inference techniques to failure data obtained during system test or during system operation. This is a
measure regarding the achieved reliability from the past until the current point. Its main purpose is to assess the current reliability, and determine whether a reliability model is a good fit in retrospect.

(b) Reliability Prediction

This activity determines future software reliability based upon available software metrics and measures. Depending on the software development stage, prediction involves different techniques:

When failure data are available (e.g., software is in system test or operation stage), the estimation techniques can be used to parameterize and verify software reliability models which can perform future reliability prediction. This definition is also referred to as reliability prediction.

When failure data are not available (e.g., software is in the design or coding stage), the metrics obtained from the software development process and the characteristics of the resulting product can be used to determine reliability of the software upon testing or delivery. This is referred to as early prediction.

Most current Software Reliability Models (SRMs) fall in the estimation category to do reliability prediction. Nevertheless, a few early prediction models were proposed and described in the literature. A SRM 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.

1.10 Software Reliability Models Classification

The Software Reliability Growth Model (SRGM) is the tool, which can be used to evaluate the software quantitatively, develop test status, schedule status and monitor the changes in reliability performance. There have been many Software Reliability models developed in the last two decades. Most of these are based upon historical failure data collected during
the testing phase. These models have been utilized to evaluate the quality of the software and for future reliability predictions. They have further been used in many management decision-making problems that occur during the testing phase. But none of these models can claim to be the best and hence there is a need for further research. In this section different modeling approaches are briefly discussed.

(a) Classification Schema

The software reliability models are broadly classified into black box (single system) and white box (multi component software) models. The black box models are classified further into models based on Inter Failure Times, Failure Count and Static Models. Markov assumptions are also made for some models and they are overlapping with the Failure Count and Inter Failure Time Modeling. All known models can be extended to be a Bayesian model. If the model is Bayesian then we are estimating the model parameters using Bayesian techniques. White box models are those that model modular software systems considering the architecture of the system. Popstajanova and Trivedi (2001) presented a classification of software reliability models that classified multi component software systems as stated in Figure 1.4. The white box models are broadly classified into State Based, Path Based and Additive Models. The State Based Models are further classified into Continuous Time Markov Chains (CTMC), Discrete Time Markov Chain (DTMC) and semi-Markov models. All these Markov models fall into either absorbing or irreducible class of Markov chains. Absorbing type represents terminating software and continuously operating system is represented by irreducible chain.
Figure 1.4: Classification schema
Bayesian models are extensions of Software Reliability Growth Models. In Bayesian models the stochastic model reflects the system accurately at that instant based on the latest failure data. In Bayesian models the distributions representing the failure behavior is represented by a prior and posterior distribution. The prior distribution reflects the expert knowledge available on the specific failure process, which is incorporated in the stochastic model. Bayesian models have wide acceptance due to these reasons in the software industry. In the following sections we describe each class of software reliability models with a representative reliability model.

(b) Single System Software

This model considers the whole software as a single monolithic system and the structure of the model is not considered in the process of reliability estimation of the software system. The reliability is solely estimated based on the failure history. Popular reliability estimation models include the Goel-Okumoto Goel and Okumoto (1979) model, and the Jelinski-Moranda model Jelinski and Moranda (1972). Thus, for example, these models do not account for the structure of the software. Single system software failure behavior is usually modeled using Software Reliability Growth Model and classified as times between failure and failure count models.

(c) Failure Rate Models

Inter failure times are the main modeling parameter for these models. Generally it is expected that the time for next failure increase as more bugs are detected and corrected. This may not be always correct as inter failure times are random variables and subjected to statistical fluctuations. The reliability function is a non-decreasing function of mission time. This demonstrates an increase in the software credibility.

(d) Markovian Models

When a markov process represents the failure process, the resultant model is called Markovian model. The software can attain many states at any particular time with respect to number of faults remaining or number of faults already removed. The transition between states depends on the current state of the software and the transition probability. The memory less property of the Markovian process implies that the time between failures follows an exponential distribution. Numerous attempts have been made to develop Markovian models; especially in earlier days due
to similar theory in hardware reliability was already well developed. One of the popular reliability models developed by Jelinski and Moranda Jelinski and Moranda (1972) is a Markov process model. Littlewood (1973,1979,1987) proposed a model based on semi-Markovian process to describe the failure phenomenon of software with module structure. Cheung Cheung (1980) has also proposed a Markovian model to describe module-structured software. Kremer Kremer (1983) proposed birth-death model, which incorporates the probabilities of fault removal and introduction. Goel (1985) modified Jelinski-Moranda model by introducing the concept of imperfect debugging. Kapur et al.(1999) proposed a model incorporating imperfect debugging and fault introduction with upper bound on the number of faults.

(e) Fault Counting Models

This category includes the models, which describe the failure phenomenon by stochastic processes like Homogeneous Poisson Process (HPP), Non-Homogeneous Poisson Process (NHPP), Compound Poisson Process (CPP) etc. The majority of these failure count models are based upon the NHPP. Schneidewind (1975) proposed a fault detection model based on NHPP. Different NHPP models are distinguished by their unique mean value functions. Yamada, Ohba and Osaki (1983) proposed the delayed S-Shaped model. Ohba (1984) proposed the inflection S-Shaped model. Musa, Iannino and Okumoto (1987) have proposed the basic execution time model and Log-Poisson model. Goel (1985) modified Musa’s original model by introducing the test quality parameter. Yamada, Osaki and Narithisa (1985) also proposed a discrete time model. The effect of testing effort on failure process was taken into consideration by Yamada, Ohtera and Narithisa (1986). Kapur, Garg and Kumar (1999) modified Goel and Okumoto (1979) model by introducing the concept of imperfect debugging. Kareer et al. Kareer (1990) proposed a model with two types of faults where each fault type is modeled as by an S-Shaped curve model. Xie and Zhao (1992) have illustrated how the Schneidewind model can be modified to result in many of the above SRGMs. Zeephongsekul P, Xia G and Kumar S (1993) proposed a model describing the case when a primary fault introduces secondary faults. Attempts as listed above for new models were made with the primary intention of getting flexible models that could describe a range of failure count curves or reliability growth curves like exponential curves and highly S-Shaped curves. Models with such property are termed as flexible SRGMs Kapur, Garg and Kumar (1999,2012). Recently efforts have been directed towards development of general
SRGMs Kapur, Garg and Kumar (1999, 2012). General SRGMs are flexible models and many of the above models can be derived from them.

Recently reliability modeling for distributed development environment has caught the attention of many researchers. Large software systems have modular design. A system is said to be modular when each activity of the system is performed by exactly one component, and when the inputs and outputs of each component are well defined Lyu (1996). Often different development teams develop such components of software separately. With availability of communication networks at cheaper rates some software components are developed at separate geographic locations also. Software developed under this distributed development environment has proved to be economical. Many a time components from other software projects are also reused. Therefore SRGMs for software developed under distributed environment needs to have different approach. But very few attempts have been made in this regard Yamada, Tamura and Kimura (2000).

(f) Models based on Bayesian Analysis

In the previous two categories the unknown parameters of the models are estimated either by the least squares method or by the maximum likelihood method (later in this chapter both these methods are briefly discussed). But in this category of models, the Bayesian analysis technique is used to estimate the unknown parameters of the models. This technique facilitates the use of information obtained by developing similar software projects. Based on this information the parameter of given model are assumed to follow some distribution (known as priori distribution). Given the software test data and based on a priori distribution, a posterior distribution can be obtained which in turn describes the failure phenomenon. Littlewood and Verral proposed the first software reliability model based on Bayesian Analysis. Xie (1993), Singpurwalla (1995, 1999) have proposed a number of Bayesian software reliability models for different testing environments.
The following four categories of Software Reliability models do not make any dynamic assumptions of the failure process. These are defined very briefly here.

(g) **Static Models**

These models use statistical techniques to evaluate software reliability and can be evaluated only if complete failure data is available. These models were developed in the initial stage of reliability model evolution and are now seldom used, as it cannot incorporate the structure of the software.

(h) **The Input Domain and Fault Seeding Models**

In fault seeding models, a known number of bugs are seeded (planted) in the program. The software is then tested for faults. The bugs detected would have a combination of inherent faults and seeded faults. The number of inherent faults in the software is calculated based on the inherent and seeded faults detected using maximum likelihood estimation and combinatorial. The drawback of this approach is that the seeded faults and the inherent faults must have the same detection probability. This is difficult to achieve. The basic approach in input domain based model is to generate a set of test cases from an input distribution. The reliability measure is calculated from the number of failures observed during symbolic or physical execution of sampled test cases. The test cases selected from the representative input space are executed recording the results. The probability of the success can be evaluated using statistical techniques. It is generally difficult to estimate the input distribution (operational profile); generally the input distribution is obtained based on the different paths that exist in the software.

(i) **Software Metrics Models**

The models in this class relate the fault content in the software to some features of the software program such as program length, complexity, volume etc. These models are empirically built and the result obtained by a model is dependent on the software development process environment, which may not be the same in the other projects.

Models other than those falling under above categories exist in the literature and more are being proposed. A number of review papers have been written, to name among others; Littlewood

(j) Multi Component Models

White Box model or Multi component system considers the Software Architecture of the system, its interactions with the different modules. White box models are used to model component based software. Such models seek to explicitly incorporate the testing method used during the testing phase, as well as the structure of the software being tested. Littlewood (1979), Kubat (1989), Cheung (1980), Krishnamurthy & Mathur (1997), Kyle Siegriest (1988) and Rajgopal & Mazumdar (1999) are some of the models in the modular software category. Modular systems have a general framework in which the system reliability is calculated. All modular systems are grouped as State Based, Path Based and Additive models.

(k) State Based Models

Architecture based models assume that components fail independently and that a component failure will ultimately lead to a system failure. Unlike hardware reliability every component is always in use, software components need a utilization factor in State Based Models. State Based Models are generally modeled using Markov models like CTMC, DTMC or semi Markov models. Models in this category are Littlewood (1979), Cheung (1980), Kubat (1989) and Laprie (1984). System reliability estimates are obtained using both architecture and failure model. This is achieved using two methods, Composite and Hierarchical solution approach.

(l) Composite-Hierarchical Approach

The State Based Models are further classified into composite and hierarchical based on the solution approach to obtain the reliability of the system. Composite method combines the architecture model with the failure model and then solved for reliability prediction. If the architecture model is first solved first and then superimposed on the failure behavior on the architecture model solution to predict reliability.
(m) Path Based Models

Similar to State Based Models, the Path Based Models consider software architecture with components and interfaces. Initially the different paths in system are obtained either experimentally or algorithmically. Path reliability is the product of all component reliabilities along the path. The system reliability is average of all the path reliabilities. State based models analytically account for the infinite loops in a path but path-based models terminate the loop to one or to an average execution time of the path. Shooman (1976) considers reliability of modular software introducing the path based approach by using the frequencies with which different paths are run. Krishnamurthy and Mathur (1997) developed a method to combine architecture and failure process by estimating the path reliabilities based on the sequence of components executed for a single test run and the average over all test runs to obtain the system reliability. State Based Models are an important category of models for modular systems.

(n) Additive Models

Additive models consider software testing phase and each component reliability is modeled by NHPP. This implies system failure rate is also NHPP with cumulative number of failures and failure intensity functions that are the sums of corresponding function of each component. Additive model do not consider architecture of the software. Xie & Wohlin (1995) developed an additive based architecture model. Semi Markov and Markov regenerative models attempt to relax the Markovian assumption of exponential failure and repair times description, in a restrictive manner. They are also exposed to the state space explosion problem. Discrete-event simulation on the other hand offers an attractive alternative to analytical models as it can capture a detailed system structure and facilitate the study of influence of various factors such as reliability growth, various repair policies. All software development teams have to answer an obvious question of when to stop testing and release the software, whether the software is single component or multi-component.

1.11 NHPP Based Software Reliability Growth Models

The NHPP models provide the expected number of faults/failures at a given time. Stochastic processes are used for the description of a system’s operation over time. There are two main
types of stochastic processes: continuous and discrete. Among discrete processes, counting processes in reliability engineering are widely used to describe the appearance of events in time (e.g., failures, number of perfect repairs, etc). The simplest counting process is a Poisson process. As a general class of well-developed stochastic process model in reliability engineering, they are especially useful to describe failure processes which possess certain trends such as reliability growth and deterioration. Therefore, an application of NHPP models to software reliability analysis is then easily implemented. The models provide the expected number of faults/failures at a given time.

Schneidewind (1975) proposed an error detection model. Goel and Okumoto (G-O) (1979) proposed the time dependent failure rate model. Ohba [1984] proposed the inflection S-shaped model. Musa (1999) and Musa and Okumoto (1984) 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. Yamada, Osaki and Narithisa (1984) 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. Yamada, Ohtera and Narithisa (1986) further proposed a testing effort dependent model which assumes the testing effort to follow either exponential, Weibull or Rayleigh distribution. Kapur et al (1999,2012) proposed a discrete time model based on the concept that the testing phase has two different processes namely, fault isolation and fault removal. Kapur, Younes, Sehgal and Yadav (1996) 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.

Yamada and Fujiwara (2001, 2002), Kapur, Khatri, Gupta and Singh (2006) have proposed several reliability growth models based on the concept of testing-domain.

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. Ohba and Yamada (1984) and Yamada, Ohba and Osaki (1983) 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.

We have a large number of SRGM each being based on a particular set of assumptions that suits a specific testing environment. The plethora of SRGM causes the problem of model selection. 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. The model selection problem is a tedious task in presence of a large number of SRGM. Besides, the performance of a model in terms of its ability to regenerate the past failure data and to predict the future of a failure observation process are two other important criteria.

1.11.1 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:

(a) There are no failures experienced at time \(t = 0\), i.e., \(N(t = 0) = 0\) with probability 1.
(b) 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 the \(N(t + \Delta t)\) of the process depends only on the present state \(N(t)\) and is independent of it is past state \(N(x)\), for \(x < t\).
(c) 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
A Generalized Modeling Framework in Software Reliability and Related Problems

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

(e) In practice, it implies that the second or higher order effects of \( \Delta t \) are negligible.

(f) 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 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)

Inversely, knowing \( m(t) \), the failure intensity function \( \lambda(t) \) can be obtained as:

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

(1.4)

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 \( \bar{N}(t) \) and we have that:
\[ \bar{N}(t) = N(\infty) - N(t) \]  \hspace{1cm} (1.5)

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 \( \bar{N}(t) \) is Poisson with parameter \( [m(\infty) - m(t)] \), that is:

\[ P[\bar{N}(t) = k] = \frac{[m(\infty) - m(t)]^k}{k!} \exp\{m(\infty) - m(t)\}, \hspace{0.5cm} k \geq 0 \]  \hspace{1cm} (1.6)

The reliability function at time \( t_0 \) is exponential given by:

\[ R(t \mid t_o) = \exp\{- (m(t + t_o) - m(t_o))\} \]  \hspace{1cm} (1.7)

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

1.11.2 Summary of NHPP Based Continuous Time Software Reliability Growth Models

A very large number of Continuous Time Models have 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. There exists another category of models, classified as flexible models. These models can depict both exponential and S-shaped failure growth phenomenon depending upon the values of their parameters. The following are some of the well-established models.

(a) Model due to Goel and Okumoto (1979) (purely Exponential)
(b) Model due to Yamada, Ohba and Osaki (1983) (purely S-shaped)
(c) Model due to Ohba (1984) (Flexible)
(d) Model due to Bittanti at al. (1998) (Flexible)
(e) Model due to Kapur and Garg (1992) (Flexible)
Generalized SRGM due to Kapur, Younes and Agarwala (1995) (Model for fault of different severity)

These models are briefly discussed below.

Models Assumptions

Some of the general assumptions (apart from some special ones for specific models discussed) for the above models are as follows:

(a) Software system is subject to failure during execution caused by faults remaining in the system.
(b) Failure rate of the software is equally affected by faults remaining in the software.
(c) The number of faults detected at any time instant is proportional to the remaining number of faults in the software.
(d) On a failure, repair effort starts and fault causing the failure is removed with certainty.
(e) All faults are mutually independent from failure detection point of view.
(f) The proportionality of failure detection / fault isolation / fault removal is constant.
(g) 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.

\[ p, q \] : proportionality constants.

\[ m_f(t) \] : expected mean number of failures occurred/observed by time \( t \).

\[ m_i(t) \] : expected mean number of faults isolated by time \( t \).

\[ m(t) \] : expected mean number of faults removed by time \( t \).
1.11.2.1 Goel-Okumoto Model (1979)

Following differential equation results from assumption-(c):

\[
\frac{d}{dt} m(t) = b[a - m(t)] \quad (1.8)
\]

The above first order linear differential equation when solved with the initial condition \( m(0) = 0 \) gives the following mean value function for NHPP (1.8):

\[
m(t) = a(1 - e^{-bt}) \quad (1.9)
\]

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.

1.11.2.2 Delayed S-shaped SRGM due to Yamada, Ohba and Osaki (1983)

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

Thus,

\[
\frac{d}{dt} m_f(t) = b[a - m_f(t)] \quad (1.10)
\]
\[
\frac{d}{dt} m(t) = b[m_f(t) - m(t)]
\]  
(1.11)

\(m_f(t)\) is the expected number of failures observed in \((0, t]\). Solving (1.10) and (1.11) under the initial condition, i.e. \(m(0) = 0\), we get the mean value function as:

\[
m(t) = a\left\{1 - (1 + bt)e^{-bt}\right\}
\]  
(1.12)

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

\[
\frac{d}{dt} m(t) = \left(\frac{b^2 t}{1 + bt}\right)(a - m(t))
\]  
(1.13)

It is observed that \(\frac{b^2 t}{1 + bt} \to b\) as \(t \to \infty\). This model was specifically developed to account for lag in the failure observation/detection and its subsequent removal by isolation. This kind of derivation is peculiar to software reliability only.

1.11.2.3. Inflection S-shaped SRGM due to Ohba (1984)

The model attributes S-shapedness to the mutual dependency between software faults. Other than assumption-(c) 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:

$$\frac{d}{dt}m(t) = b(t)[a - m(t)]$$  \quad (1.14)$$

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

$$b(t) = b\phi(t)$$  \quad (1.15)$$

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

$$\phi(t) = r + (1-r)\frac{m(t)}{a}, \quad \phi(0) = 0 \text{ and } \phi(\infty) = 1$$  \quad (1.16)$$

where $b$ is the fault removal rate in the steady state. Solving (1.14) under the initial condition $m(0)=0$ we get:

$$m(t) = a\left[ \frac{1-e^{-bt}}{1+r} \frac{1-r}{e^{-bt}} \right]$$  \quad (1.17)$$

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

1.11.2.4 SRGM for an Error Removal Phenomenon due to Kapur and Garg (1992)

This model is based upon the following additional assumption: On a failure observation, the fault removal phenomenon also removes proportion of remaining faults, without their causing any failure. Based on the assumption the fault removal intensity per unit time can be written as:

$$\frac{d}{dt}m(t) = p[a - m(t)] + q \frac{m(t)}{a}[a - m(t)]$$  \quad (1.18)$$
Solving equation (1.18) with the usual initial condition \( m(0)=0 \), the expected number of faults detected in \((0,t]\) is given as:

\[
m(t) = a \left[ \frac{1 - e^{-(p+q)t}}{1 + \frac{q}{p} e^{-(p+q)t}} \right] \quad (1.19)
\]

which is similar to equations (1.17) and (1.22), though they have been derived under different assumptions.

**1.11.2.5 Flexible SRGM due to Bittanti et al. (1998)**

The model is based on the following differential equation:

\[
\frac{d}{dt} m(t) = k(m)(a - m(t)) \quad (1.20)
\]

where
\[
k(m) = k_i + (k_f - k_i) \frac{m(t)}{a}
\]

(1.21)

Here \( k_i \) and \( k_f \) are initial and final values of Fault Exposure Coefficient. If \( k_i = k_f \), then it reduces to Exponential model. If \( k_f > k_i \), the failure growth curve takes S-shape. If \( k_f \) is very small as compared to \( k_i \) that it is almost equal to zero, the failure growth curve become flat at the end.

The solution of equation (1.20) with initial condition \( m(t = 0) = 0 \) is:

\[
m(t) = a \left( \frac{1 - e^{-k_f t}}{1 + \frac{k_f - k_i}{k_i} e^{-k_f t}} \right) \quad (1.22)
\]

For different values of \( k_f \) and \( k_i \), it describes different growth curves.
1.12 Some Concepts and Related SRGMs

The above described reliability growth models, though derived under different set of assumptions have one thing in common i.e. they relate reliability growth phenomenon with the testing time/ number of test cases. The ocean of software reliability engineering is full of a wide variety of models incorporating the concepts of ranging from perfect debugging environment to imperfect debugging (partial fault removal / fault generation), from sudden changes in the removal rate (change-point) to the application of testing coverage/effort in model building, from simple nature of the faults to severity of faults to the distributed software and many more. We have stated these concepts as they have been used in the formulation of the present work. We now describe briefly these concepts and some related SRGMs to provide a glimpse of the vivid and highly vibrant world of the ever-evolving subject of software reliability.

1.12.1 SRGMs under Imperfect Debugging Environment

Since the human factor is involved in debugging the software faults, we cannot deny the possibility of imperfection in testing and debugging. In practice the testing efficiency is usually imperfect. Sometimes the debugging team may not be able to remove the fault perfectly on the detection of failure and the original fault may remain (known as imperfect fault removal) or replaced by another fault (known as fault generation). Most of the existing SRGM describes either imperfect fault removal or error generation. The concept of imperfect debugging was first introduced by Goel and Okumoto (1978). They introduced the probability of imperfect debugging in Jelinski and Moranda (1972) model. The integrated effect of two phenomena has been discussed by Zang et al (2003), Kapur et al (2006c).

Model for Error Generation (Obha and Chou, (1989))

Obha and Chou proposed the first SRGM based on NHPP incorporating the effect of error generation. The following differential equations describes the failure phenomenon of the model

\[ \frac{dm(t)}{dt} = b(t)\{a(t) - m(t)\} \quad \text{with} \quad b(t) = b, \quad a(t) = a + \alpha m(t) \]  \hspace{1cm} (1.23)
solving equations (1.23) under the initial conditions \( m(0)=0 \) 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 are due to Yamada \textit{et al} (1992) and Pham (2006).

\textit{Imperfect Fault Debugging Model (Kapur and Garg, (1990))}

Kapur and Garg, (1990) introduced the imperfect debugging NHPP based Goel and Okumoto SRGM. They assumed that the fault removal rate (FRR) per remaining faults is reduced due to imperfect debugging.

\[
\frac{dm_r(t)}{dt} = pb[a - m_r(t)]
\]

(1.25)

\[
\frac{dm_f(t)}{dt} = b[a - pm_f(t)]
\]

(1.26)

solving (1.25) and (1.26) with the initial conditions \( m_r(0)=0 \) and \( m_f(0)=0 \) we get

\[
m_r(t) = a(1-e^{-bpt})
\]

(1.27)

\[
m_f(t) = \left(\frac{a}{p}\right)(1-e^{-bpt})
\]

(1.28)

If \( p = 1 \) the model reduces to GO model with \( m_r(t)=m_f(t) \).

\textit{Zang Testing Efficiency Model (Zang \textit{et al} (2003))}

The authors assumed that the fault generation rate is proportional to the failure intensity in the presence of possibility of imperfectly removing a fault. So the corresponding differential equation is given by:
\[
d\frac{d}{dt} m(t) = b(t)(a(t) - pm(t))
\]

(1.29)

where
\[a(t) = a + \alpha m(t)\]
\[b(t) = \left( c / (1 + \beta e^{-bt}) \right) \]

Mean value function of the failure phenomenon under the initial condition \( m(0) = 0 \) is
\[
m(t) = \frac{a}{p - a} \left( 1 - \left( \frac{(1 + \beta)e^{-bt}}{1 + \beta e^{-bt}} \right)^{\left( c/b \right) \left( p - \alpha \right)} \right)
\]

(1.30)

**Kapur Testing Efficiency Model (Kapur et al (2006c))**

In the above model by Zang et al (2003) assumed that the fault generation rate is proportional to the failure intensity in the presence of possibility of imperfectly removing a fault. However in practice a fault is generated while removing some fault and existence of a generated fault is known when original fault is removed perfectly and the same test case that has caused the failure is executed to test the corrected code. On the execution of this test case some other kind of failure is observed. Therefore the fault generation rate is proportional to the rate of fault removal. The number of failures is not same as the number of removals if we assume that the debugging team may not be able to repair a fault perfectly. The same fault may manifest on testing the corrected code for the same input on imperfect removal. Testing efficiency model due to Kapur et al (1999) corrected this ambiguity in modeling the two types of imperfect debugging.

Assuming that FRR per additional fault removed is reduced by the probability of perfect debugging, a constant proportion of removed faults are generated while removal and testing team gains efficiency with time, the general differential equation describing the removal phenomenon is given as
\[
\frac{d m_r(t)}{dt} = p b(t) [a(t) - m_r(t)]
\]

(1.31)

where
\[b(t) = \left( b / \left( 1 + \beta e^{-bt} \right) \right), \quad \text{and} \quad a(t) = a + \alpha m_r(t)\]
Solving the above differential equation under initial condition \( m(0)=0 \), we get mean value function for the removal phenomenon

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

(1.32)

The mean value function of the failure phenomenon can be derived from the relation \( pm_f(t) = m_r(t) \). It may be noted here that, \( m_r(\infty) = a/(1-\alpha) \) and \( m_f(\infty) = a/(p(1-\alpha)) \) whereas in Zang’s testing efficiency model \( m_f(\infty) = a/(p-\alpha) \), which implies that if testing is carried out for an infinite time more faults are removed as compared to the initial fault content because there are some faults added to the software due to error generation. The total number of generated faults by time infinity given by Kapur et al (1999) model is \( a(\infty) = a = (aa)/(1-\alpha) \) whereas for Zang’s model it is \( (a(1-p+\alpha))/(p-\alpha) \). It is important to note that imperfect repair of faults results in more number of failures than removals and has no contribution to increasing the fault content. Whereas, the result of Zang’s models yields the total number of generated faults is a function of both \( p \) and \( \alpha \), which is a contradiction.

The model reduces to the Kapur and Garg (1990) model if we set \( \alpha = \beta = 0 \). Similarly it yields the Obha and Chou (1989) model if \( p = 1 \) and \( \beta = 0 \). Incorporation of the effect of imperfect debugging in the reliability modeling helps the software developer in measuring the efficiency of their testing team.

### 1.12.2 SRGMs with Testing Effort

Most of these earlier SRGM use calendar (execution) time as the unit of fault detection/removal period and either assume that the testing effort consumption (TEC) rate is constant, or do not explicitly consider the testing efforts (TE) and its effectiveness. The achieved reliability during testing phase is highly related to the amount of development resources (test efforts) spent on detecting and correcting latent software faults. The TE are measured by the manpower spent during the testing phase, the number of CPU hours, the number of executed test cases, and so on.

The testing effort (resources) that govern the pace of testing for almost all the software projects are (Musa, Iannino and Okumoto 1987):

1. Manpower that includes
   i) Failure identification personnel
   ii) Failure correction personnel.

2. Computer time

These resources very closely interact during the system test phase. The function of the failure identification personnel is to run test cases and compare the test results against program requirement to establish the failures that have occurred. The failure correction personnel are the debuggers or developers available to repair the software. The Computer facility represents the computer(s) necessary for the failure identification personnel and failure correction personnel to do their tasks. Computer time is the measure that is used for allocating computer resources.

SRGMs have been proposed in the Software Reliability Engineering literature under sets of assumptions and for different testing environments. But most of them do not consider the consumption pattern of resources such as computer time and manpower during testing. In this thesis some new SRGMs incorporating testing effort have been proposed.

As the time increases, testing effort also increases. If testing time becomes quite large, testing effort also becomes quite large. The testing effort rate is proportional to the testing resources available i.e.,

\[
\frac{dW(t)}{dt} = v(t)[\alpha - W(t)]
\]  \hspace{1cm} (1.33)
where $\alpha$ is the total testing resource consumption and $v(t)$ is the time dependent rate at which testing resources are consumed, applied to remaining available resources. Solving equation (1.33) under the initial condition $W(0) = 0$ we get:

$$W(t) = \alpha \left( 1 - e^{-\int_0^t v(x) \, dx} \right)$$

(1.34)

Though various types of testing effort function have been studied in the literature. In this thesis, we have used four different curves functions- Exponentials, Rayleigh, Weibull and Logistic. The detailed description of these effort functions is given in chapter 4.

**Model using Testing Effort Function (Yamada et al 1983)**

Under the assumptions described above and using equation (1.34), the fault removal process can be described by the following differential equation:

$$\frac{dm(t)}{dt} = b(t) \left( a - m(t) \right)$$

(1.35)

Using the different forms of $b(t)$ several SRGMs can be formulated

Let the fault detection rate function is defined as:

$$b(t) = b$$

(1.36)

Here $w(t)$ is the current testing effort consumption at time $t$.

$$m(W(t)) = a \left( 1 - e^{-b \cdot w(t)} \right)$$

(1.37)

The above model is known as GO Model with respect to Testing Effort Function Yamada et al (1983).
1.12.3 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 and Osaki (1985) who modified G-O exponential SRGM assuming that there are two types of faults in the software. A study by Pham in 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 (1995b) addressed three level complexities of faults considering the time delay between the failure observation and its subsequent removal.

**Generalized Erlang SRGM (Kapur et al (1995b))**

Assuming $a_1$, $a_2$, $a_3$ to be simple, hard and complex faults in a software system ($a_1 + a_2 + a_3 = a$), the simple fault removal process is modeled as the following:

$$\frac{d}{dt} m(t) = b_1 [a_1 - m_1(t)]$$ \hspace{1cm} (1.38)

Solving, we get

$$m_1(t) = a_1 (1 - e^{-b_1 t})$$ \hspace{1cm} (1.39)

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

$$\frac{d}{dt} m_{2f}(t) = b_2 \left( a_2 - m_{2f}(t) \right)$$ \hspace{1cm} (1.40)

$$\frac{d}{dt} m_2(t) = b_2 \left( m_{2f}(t) - m_2(t) \right)$$ \hspace{1cm} (1.41)

Solving, we get

$$m_2(t) = a_2 \left( 1 - (1 + b_2 t) e^{-b_2 t} \right)$$ \hspace{1cm} (1.42)

The Complex fault removal process is modeled as a three stage process
\[
\begin{align*}
\frac{d}{dt} m_{3f}(t) &= b_3\left(a_3 - m_{3f}(t)\right) \quad (1.43) \\
\frac{d}{dt} m_{3i}(t) &= b_3\left(m_{3f}(t) - m_{3i}(t)\right) \quad (1.44) \\
\frac{d}{dt} m_3(t) &= b_3\left(m_{3i}(t) - m_3(t)\right) \quad (1.45)
\end{align*}
\]

Solving, we get
\[
m_3(t) = a_3\left(1 - \left(1 + b_3 t + \frac{b_3^2 t^2}{2}\right)e^{-b_3 t}\right) \quad (1.46)
\]

The mean value function of the given SRGM is
\[
m_r(t) = m_1(t) + m_2(t) + m_3(t) \quad (1.47)
\]

Assuming \(a_2 = p.a, a_3 = q.a\) and \(a_1 = (1 - p - q)a\),
\[
m(t) = a\left[\left(1 - e^{-b_1 t}\right)(1 - p - q) - p(1 + b_2 t)e^{-b_2 t} - q\left(1 + b_3 t + \frac{b_3^2 t^2}{2}\right)e^{-b_3 t}\right] \quad (1.48)
\]

Assuming \(b_1 = b_2 = b_3 = b\), we have
\[
m(t) = a\left[1 - e^{-b t}(1 + (p + q)bt + q\frac{b^2 t^2}{2})\right] \quad (1.49)
\]

The mean value function obtained in (1.49) can be generalized to include \(n\) different types of faults depending upon their severity. We may write
\[
m(t) = \sum a_i\left[1 - e^{-b_i t}\left(\sum_{j=0}^{i-1} \frac{(b_i t)^j}{j}\right)\right] \quad (1.50)
\]

**Flexible Generalized SRGM for Faults of Different Severity (Singh V.B. 2008)**

Complex software systems have different types of faults and failure/removal process is depicted by different types of curves. It has been also assumed that removal growth of type 1 fault which is simple in nature follows exponential curve. For other faults, which are more severe in nature, logistic learning has been incorporated during removal phenomenon and these faults are
A Generalized Modeling Framework in Software Reliability and Related Problems

depicted by different types of S-shaped curves. In the beginning, it is assumed that only three
types of faults exist in software type 1, type 2 and type 3 (simple, hard and complex namely) and
later, modeling has been extended to \( n \) types of faults.

Assuming, \( a_1 \), \( a_2 \) and \( a_3 \) to be simple, hard and complex faults in a software system
\((a_1 + a_2 + a_3 = a)\), the simple fault removal process is modeled by the following equation
\[
\frac{dm_1(t)}{dt} = b_1 (a_1 - m_1(t)) \tag{1.51}
\]

Solving equation (1.51) with the initial condition \( m_1(0)=0 \), we get:
\[
m_1(t)=a_1 \left(1 - \exp(-b_1 t)\right) \tag{1.52}
\]

The hard faults removal process is modeled as a two-stage process,
\[
\frac{dm_2(t)}{dt} = b_2 (a_2 - m_{2f}(t)) \tag{1.53}
\]

\[
\frac{dm_2(t)}{dt} = \frac{b_2}{1 + b_2 \exp(-b_2 t)} \left(m_{2f}(t) - m_2(t)\right) \tag{1.54}
\]

Here it is assumed that learning of removal team grows as testing progresses and follows logistic
removal rate.

Solving equation (1.53) and (1.54) with the initial condition \( m_{2f}(0)=0, m_2(0)=0 \), we get:
\[
m_2(t)=a_2 \frac{(1-(1+b_2 t) \exp(-b_2 t))}{\left(1 + b_2 \exp(-b_2 t)\right)} \tag{1.55}
\]

Here \( m_{2f}(t) \) denotes the number of failures observed in time \( t \) whereas \( m_2(t) \) represents the
number of faults removed in time \( t \).

The complex fault removal process is modeled as a three-stage process,
\[
\frac{dm_{3f}(t)}{dt} = b_3 (a_3 - m_{3f}(t)) \tag{1.56}
\]

\[
\frac{dm_{3is}(t)}{dt} = b_3 (m_{3f}(t) - m_{3is}(t)) \tag{1.57}
\]
\[ \frac{dm_3(t)}{dt} = \frac{b_3}{1 + \beta_3 \exp(-b_3 t)} (m_{3is}(t) - m_3(t)) \quad (1.58) \]

Here also, it is assumed that learning of removal team grows as testing progresses and follows logistic removal rate.

Solving equation (1.56), (1.57) and (1.58) with the initial condition \( m_{3f}(0) = 0, m_{3is}(0) = 0 \) and \( m_3(0) = 0 \), we get:

\[ m_3(t) = a_3 \left( 1 - \left( 1 + b_3 t + \frac{b_3^2 t^2}{2} \right) \exp(-b_3 t) \right) \left( 1 + \beta_3 \exp(-b_3 t) \right)^{-1} \quad (1.59) \]

It may be noted that \( m_2(t) \) and \( m_3(t) \) are expressed by delayed S-shaped and 3-stage Erlang growth curves with logistic removal rates. The removal rates per remaining fault for simple, hard and complex faults are given by \( b_1 \),

\[ d_2(t) = b_2 \left[ \frac{1}{(1 + b_2 \exp(-b_2 t))} - \frac{1}{(1 + b_2 + b_2 t)} \right] \]

and

\[ d_3(t) = b_3 \left[ \frac{1}{(1 + \beta_3 \exp(-b_3 t))} - \frac{(1 + b_3 t)}{(1 + \beta_3 + b_3 t + b_3^2 t^2)} \right] \]

respectively.

Note that \( d_2(t) \) and \( d_3(t) \) increase monotonically with \( t \) and tend to \( b_2 \) and \( b_3 \) respectively as \( t \to \infty \). Thus, in the steady state delayed S-shaped and 3-stage Erlang with logistic removal rate growth curves behave similar to the exponential growth curves and hence there is no loss of generality in assuming the steady state rates \( b_2 \) and \( b_3 \) equal to \( b_1 \). The removal rates for three types of faults become \( b_1 \),

\[ d_2(t) = b_1 \left[ \frac{1}{(1 + \beta_2 \exp(-b_1 t))} - \frac{1}{(1 + \beta_2 + b_1 t)} \right], \text{ and} \]

\[ d_3(t) = b_1 \left[ \frac{1}{(1 + \beta_3 \exp(-b_1 t))} - \frac{(1 + b_1 t)}{(1 + \beta_3 + b_1 t + b_1^2 t^2)} \right] \]

respectively.

It is also noted that
A Generalized Modeling Framework in Software Reliability and Related Problems

\[ b_1 > b_1 \left[ \frac{1}{1 + \beta_2 \exp(-b_1 t)} - \frac{1}{1 + \beta_2 + b_1 t} \right] > b_1 \left[ \frac{1}{1 + \beta_3 \exp(-b_1 t)} - \frac{(1 + b_1 t)}{1 + \beta_3 + b_1 t + b_1^2 t^2} \right], \]

Which is in accordance with the severity of faults.

The mean value function of the SRGM is

\[ m(t) = m_1(t) + m_2(t) + m_3(t) \]  \hspace{1cm} (1.60)

Where \( m_1(t) \) the mean value function of the simple faults is removed in time \([0,t]\), \( m_2(t) \) is the mean value function of the hard faults removed in time \([0,t]\) and \( m_3(t) \) is the mean value function of the complex faults removed in time \([0,t]\).

\[ m(t) = a_1 \left(1 - \exp(-b_1 t)\right) + a_2 \left(\frac{1 - (1 + b_1 t) \exp(-b_1 t)}{1 + \beta_2 \exp(-b_1 t)}\right) + a_3 \left(\frac{1 - \left(1 + b_1 t + \frac{b_1^2 t^2}{2}\right) \exp(-b_1 t)}{1 + \beta_3 \exp(-b_1 t)}\right) \]  \hspace{1cm} (1.61)

assuming \( b_1 = b_2 = b_3 = b \), we have

\[ m(t) = a_1 \left(1 - \exp(-bt)\right) + a_2 \left(\frac{1 - (1 + bt) \exp(-bt)}{1 + \beta_2 \exp(-bt)}\right) + a_3 \left(\frac{1 - \left(1 + bt + \frac{b^2 t^2}{2}\right) \exp(-bt)}{1 + \beta_3 \exp(-bt)}\right) \]  \hspace{1cm} (1.62)

assuming \( a_1 = a p_1, a_2 = a p_2 \) and \( a_3 = a \left(1 - p_1 - p_2\right) \)

\[ m(t) = a p_1 \left(1 - \exp(-bt)\right) + a p_2 \left(\frac{1 - (1 + bt) \exp(-bt)}{1 + \beta_2 \exp(-bt)}\right) + a \left(1 - p_1 - p_2\right) \left(\frac{1 - \left(1 + bt + \frac{b^2 t^2}{2}\right) \exp(-bt)}{1 + \beta_3 \exp(-bt)}\right) \]  \hspace{1cm} (1.63)

The mean value function obtained in above equation can be generalized to \( n \) different types of faults depending upon their severity. We may write

\[ m(t) = \sum_{i=1}^{n} m_i(t) = a_1 \left(1 - \exp(-b_1 t)\right) + \sum_{i=2}^{n} \frac{a_i}{1 + \beta_i \exp(-b_i t)} \left[1 - \exp(-b_i t) \sum_{j=0}^{i-1} \frac{(b_i t)^j}{j!} \right] \]  \hspace{1cm} (1.64)

Above model determines the type of faults present in software with a logistic removal rate and is abbreviated as GE-\( n \) (Logistic). If \( \beta_1 = 0 \), above model reduces to (Kapur, Younes and Agarwala 1995b).
The analysis followed in equation (1.62) can again be applied. Therefore, assume \( b_1 = b_2 = b_3 = \ldots = b_n = b \). It is also assumed that the value of \( \beta \) remain same for different types of faults. Accordingly equation (1.1.54) can be written

\[
m(t) = a_t \left(1 - \exp(-b_t t)\right) + \frac{1}{\left(1 + \beta \exp(-bt)\right)} \sum_{i=2}^{n} a_t \left[1 - \exp(-b_t \sum_{j=0}^{i-1} \frac{(bt)^j}{j!}\right] (1.65)
\]

### 1.12.4 General Model of Fault Detection and Correction Processes

In software reliability engineering, the correction of a fault is different from the detection. It will be more close to reality of the software testing if considered both the correction and the detection process. So the fault correction process may be different from the fault detection and it is important to have different models for these processes.

Assuming that the fault detection rate is proportional to the number of undetected faults, we have the following differential equation for the mean value function \( m_d(t) \) of the fault detection process

\[
\frac{dm_d(t)}{dt} = b(t)\left[a - m_d(t)\right] \quad (1.66)
\]

The proportionality \( b(t) \) may be a time-dependent function.

The fault correction process can also be modeled assuming that the fault correction rate to be proportional to the number of detected but not yet corrected faults. The following differential equation for the mean value function \( m_c(t) \) of the fault correction process can be formed.

\[
\frac{dm_c(t)}{dt} = c(t)\left[m_d(t) - m_c(t)\right] \quad (1.67)
\]

c(t) may also be time-dependent. The concept assumes more realistic situation and so has been incorporated while formulating the SRGM presented here.
1.12.5 Software Reliability Growth Model Using Unified Approach

Unified modeling approach is one of the recent topics of research in software reliability. Two research directions for SRGM are usually considered: unification and parameter estimation of SRGM. In fact, a unified modeling framework comprising some typical reliability growth patterns should be developed for robust software reliability assessment. There are some, but only a few, model unification schemes in the literature. Langberg and Singpurwalla (1995) showed that several SRGM can be comprehensively described by adopting a Bayesian point of view. Miller (1986) and Thompson, Jr. (1988) extended the Langberg and Singpurwalla’s idea and developed a generalized Order Statistic models (GOS). More precisely, they showed that almost all SRGM can be explained within the framework of GOS and record value and claim especially for the NHPP models that the model selection problem is reduced to a simple selection problem of fault detection time distribution. Based on their result the mean value function in NHPP models that can be characterized by theoretical probability distribution function of fault detection time. Chen and Singpurwalla (1997) proved that all SRMs as well as the NHPP models developed in the past literature can be unified by self-exciting point processes. Apart from the probabilistic approach, Huang et al. (2003) explained the deterministic behavior of the NHPP models, namely mean value function of time by introducing several kinds of mean operation. Pham et al. (1997) solved a generalized differential equation by which the mean value function in the NHPP model is governed and proposed an NHPP with a generalized mean value parameter.

Therefore, we see that during last three decades many SRGM have been proposed in the literature (Musa et. A1 1987, Kapur et. al 1999, Pham 2006) , which relates the number of failure (faults identified/corrected) and execution time. These SRGMs assumes diverse testing and debugging (T&D) environments like distinction between failure and correction processes, learning of the testing personnel, possibility of imperfect debugging and fault generation, constant or monotonically increasing/decreasing fault detection rate (FDR) or randomness in the growth curve. These SRGMs have been applied successfully in many real life software projects but no SRGM can claim to be the best in general as the physical interpretation of the testing and debugging changes due to numerous factors e.g., design of the test cases, defect density, skills and efficiency of the testing team, availability of testing resources etc. The plethora of SRGMs
makes the model selection a tedious task. As stated earlier, to reduce this difficulty unified modeling approach have been proposed by many researchers. These schemes have proved to be successful in obtaining several existing SRGMs by following single methodology and thus provide a insightful investigation for the study of general models without making many assumptions.

From the above literature it is clear that the work in this area started quite early. In fact we say it started as early as in 1980s with Shantikumar (1981) proposing a generalized birth process model. Gokhale and Trivedi (1996) used Testing coverage function to present a unified framework and showed how NHPP based models can be represented by probability distribution functions of fault–detection times. Dohi et al (2004) proposed a unification method for NHPP models describing test input and program path searching times stochastically by an infinite server queuing theory. Inoue (2006) applied infinite server queuing theory to the basic assumptions of delayed S-shaped SRGM (1983) i.e. fault correction phenomenon consists of successive failure observation and detection/correction processes and obtained several NHPP models describing fault correction as a two stage process.

Another unification methodology is based on a systematic study of Fault detection process (FDP) and Fault correction process (FCP) where FCPs are described by detection process with time delay. The idea of modelling FCP as a separate process following the FDP was first used by Schneidewind (1975). More general treatment of this concept is due to Xie et. al (1992) who suggested modelling of Fault detection process as a NHPP based SRGM followed by Fault correction process as a delayed detection process with random time lag. The recent unification scheme (due to Kapur et. al.(2004)) is based on Cumulative Distribution Function for the detection/correction times. The authors have used the concept of hazard rate in order to study fault detection rate. We are aware that the rate at which the failure occurs in a certain time interval \([t, t+\Delta t]\) is called failure rate. Thus hazard rate \(h(t)\) is defined as the probability that a failure per unit time occurs in the interval, given that a failure has not occurred prior to \(t\), the beginning of the interval. Thus the failure rate is

\[ \frac{R(t) - R(t + \Delta t)}{\Delta t R(t)}. \]

The hazard rate/function is defined as the limit of the failure rate as the interval approaches to zero. Thus .the hazard rate function \(h(t)\) is instantaneous failure rate, and is defined by
\[ h(t) = \lim_{\Delta t \to 0} \frac{R(t) - R(t + \Delta t)}{\Delta t R(t)} \]  
\[ = \frac{1}{R(t)} \left[ -\frac{d}{dt} R(t) \right] \]  
\[ = \frac{f(t)}{R(t)} = \frac{f(t)}{\int_{f(t)}^\infty} = \frac{f(t)}{1-F(t)} \]  
\[ (1.68) \]

\[ (1.69) \]

\[ (1.70) \]

In addition to above literature one more unification scheme has been discussed based on Queuing theory approach. Moreover, the hazard rate approach proved to be fruitful in obtaining several SRGMs by following single methodology and thus present a perspective investigation for study of general models without making any assumptions. We have therefore used this approach in modeling some new SRGMs in the present thesis.

1.12.6 Incorporating Change Point Analysis

In a number of actual testing scenarios, it is observed that the fault detection/ removal rate (FDR/FRR) may not be constant/continuous function of time and can change as the testing progresses. The changes in the FDR/FRR can be accounted due to the number of expected changes in the testing environment, testing strategy, complexity and size of the functions, defect density, skill, motivation and constitution of the testing and debugging team etc. The change in FDR or FRR can be analyzed using the change point concept. The idea behind the change point concept is that it divides the testing period into intervals and assumes that during a particular interval the testing strategy and environment are more or less similar and are slightly different from the other intervals. Some important contributions in this area have been made by Chang (1997) Haung (2005a), Shyur (2003) and Kapur et al (2006, 2007a).
**G-O Model with Change Point (Chang (1997))**

The time point where a change in the FDR/FRR is expected is termed as *change point*. The position of change point can be judged from the actual failure data curve. For the GO model with a single change point the FDR is defined as

\[
\frac{d}{dt} m(t) = \begin{cases} 
  b(t)[a - m(t)] & \text{for } 0 \leq t \leq \tau \\
  b(t)[a - m(t)] & \text{for } t > \tau
\end{cases}
\]  

(1.71)

Where \( b(t) = \begin{cases} 
  b_1 & 0 \leq t \leq \tau, \\
  b_2 & t > \tau
\end{cases} \) and \( \tau \) is the change point.

The mean value function for the SRGM is given as following

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

(1.72)


**GO Model using Change Point and Imperfect Debugging environment (Shyur 2003)**

On similar basis as discussed by Chang (1997), Shyur (2003) incorporated the concept of imperfect debugging. Consider the following equation in order to understand

\[
\frac{d}{dt} m(t) = \begin{cases} 
  b(t)[a(t) - m(t)] & \text{for } 0 \leq t \leq \tau \\
  b(t)[a(t) - m(t)] & \text{for } t > \tau
\end{cases}
\]

(1.73)

\[
b(t) = \begin{cases} 
  b_1 & 0 \leq t \leq \tau, \\
  b_2 & t > \tau
\end{cases}
\]
And \[
\frac{c\alpha(t)}{dt} = \alpha(t) \frac{dm(t)}{dt}
\] (1.74)

Where \(\alpha(t) = \alpha_1 \text{ and } \alpha_2\) i.e. the rate of error generation before and after change-point and \(\tau\) is the change point.

The mean value function for the SRGM is given as follows

\[
m(t) = \begin{cases} \frac{a}{1-\alpha_1} (1 - e^{-(1-\alpha_1)b_1 t}) & 0 \leq t \leq \tau \\ \frac{a}{1-\alpha_2} (1 - e^{-(1-\alpha_1)b_1 \tau - (1-\alpha_2)b_2(t-\tau)}) + m(\tau) \left( \frac{\alpha_1 - \alpha_2}{1-\alpha_2} \right) & t > \tau \end{cases}
\] (1.75)

**GO Model using change-point and Testing Efforts**

If having a measure of reliability is the major purpose of using an SRGM for practical applications then SRGM developed in respect to time can provide useful information. If however an SRGM used to measure the effectiveness of resources spent on testing or used in some optimization model for decision making purpose the models developed under testing effort utilized are more fruitful. Other considerations related to the use of testing effort have been discussed in the Sec. (1.12.2). Here we focus on an exponential model incorporating the change point phenomenon.

Consider the following equation in order to understand

\[
\frac{1}{w(t)} \frac{d}{dt} m(t) = \begin{cases} b(t) [a(t) - m(t)] & \text{for } 0 \leq t \leq \tau \\ b(t) [a(t) - m(t)] & \text{for } t > \tau \end{cases}
\] (1.76)

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

Where \(w(t)\) is density function of testing effort distribution and \(\tau\) is the change point.

The mean value function for the SRGM is given as follows

\[
m(t) = \begin{cases} a(1 - e^{-b_1 t(W(t) - W(0))}) & 0 \leq t \leq \tau \\ a(1 - e^{-b_1 (W(\tau) - W(0)) + b_2 (W(t) - W(\tau))}) & t > \tau \end{cases}
\] (1.77)

Or
In the work presented here, we have developed SRGMs based on change point and the concepts as described in Sec. (1.12). The phenomenon has been combined with concepts starting from fault severity to imperfect debugging. Further embodying testing effort functions some practicality is taken further. The concept of change point has been widely studied here and its impact on the unification scheme has been shown in the later chapters. It should be noted that the situation of multiple change points might exist in a testing process if changes are observed not only at one point of time rather at various different points of time and the fault detection/removal process between these change points is described by the parameter variation modeling approach, i.e. the process is described by the different set of parameters from a similar distribution. This modeling assumption has also been taken care of while formulating the SRGMs here.

### 1.13 Two Dimensional Modeling

In economics, The Cobb Douglas functional form of production functions is extensively used to characterize the relationship of an output to input. It was proposed by Knut Wicksell (1851–1926), and tested against statistical evidence by Charles Cobb and Paul Douglas in 1900–1928. The function of Cobb Douglas present a simplified outlook of the economy in which production output is obtained by the amount of labor occupied and the amount of capital invested. While there are many factors influencing economic performance, their model demonstrated remarkable accuracy. The mathematical form of the production function is specified as:

\[
Y = AL^vK^{1-v}
\]

Where:

\(Y\) = total production (the monetary value of all goods produced in a year),
\(L\) = labor input
$K =$ capital input,
$A =$ total factor productivity $v$ is elasticity of labor.
This value is constant and determined by available technology.
The assumptions made by Cobb and Douglas can be stated as follows:

1. If either labor or capital vanishes, then so will production.
2. The marginal productivity of labor is proportional to the amount of production per unit of labor.
3. The marginal productivity of capital is proportional to the amount of production per unit of capital.

The Cobb- Douglas function based on the above assumptions is very appealing. The basic characteristic of this function is linearly homogeneous with constant return to scale i.e. a proportion increase in all inputs leads to same proportion increase in output.

Fig. 1.5: A Two-Input Cobb Douglas Production Function

It is this function that we have utilised in framing some of the SRGMs in the thesis. Talking in terms of software reliability growth , time and resources are the two factors which are very
important for any organization, company and individual for optimization. The rationale being, both of these are limited and precious too. These two factors are essential in case of development of a software product.

Although testing helps in assessing and improving quality, but it cannot be performed indefinitely. The other factor which makes testing phase important is that it consumes major part of the resources available for the software development. Basically, testing activities account for 30 to 90 percent of labor expended to produce a working program (Beizer 1990). Therefore, the resources too play a chief role in determining the growth of testing progress. And thus it becomes important to study the impact of both of them simultaneously. Consideration of these factors in quantifying the reliability of a software system is made through the development of software reliability growth models. We have made use of the framework of Cobb Douglas production function and developed some SRGMs dependent on time and coverage concurrently.

1.14 Parameter Estimation of Analytical Models

The success of mathematical modeling approach to reliability evaluation depends heavily upon quality of failure data collected. The parameters of the SRGM are estimated based upon these data. Hence efforts should be made to make the data collection more explicit and scientific. Usually data is collected in one of the following two ways. In the first case the times between successive failures are recorded. Though this type of data collection is more desirable, it may not be simple. Complication can arise in measuring the testing effort for each fault and it may not be very convenient to note the time at each failure report. The other easier and commonly collected data type is known as the grouped data. Here testing intervals are specified and number of failures experienced during each such interval is noted. The studied analytical models presented in the thesis are non-linear and presents extra problems in estimating the parameters. Technically, it is more difficult to find the solution for non-linear models using Least Square method and requires numerical algorithms to solve it. Statistical software packages such as SPSS helps to overcome this problem. SPSS is Statistical Package for Social Sciences. It is a comprehensive, flexible statistical analysis and data management system. SPSS can take data from almost any type of file and use them to generate tabulated reports, charts, and plots of distributions and trends, descriptive statistics and conduct complex statistical analysis. SPSS Regression Models enables the user to apply more sophisticated models to the data using its wide
range of nonlinear regression models. For estimation of the parameters of the studied analytical models, Method of Least Square (Non Linear Regression method) has been used. Non Linear Regression is a method of finding a nonlinear model of the relationship between the dependent variable and a set of independent variables. Unlike traditional linear regression, which is restricted to estimating linear models, nonlinear regression can estimate models with arbitrary relationships between independent and dependent variables.

1.14.1 Method of Least Squares

In this method the square of the difference between observed response and value predicted by the model is minimized. If the expected value of the response variable is given by \( m(t) \), then the least square estimators of the parameters of the model may be obtained from \( n \) pairs of sample values \((t_1, y_1), (t_2, y_2), \ldots, (t_n, y_n)\) by minimizing \( J \) given by:

\[
J = \sum_{i=1}^{n} [y_i - m(t)]^2
\]  
\[(1.79)\]

where \( t_i \) and \( y_i \) observed values of explanatory and dependent variables respectively. For small and medium size samples least squares estimation is preferred Musa, Iannino and Okumoto (1987).

1.14.2 Maximum Likelihood Estimation

Maximum Likelihood Estimation (MLE) method has been extensively adopted for estimation of parameters of SRGM based upon NHPP Musa, Iannino and Okumoto (1987). We briefly discuss below the MLE procedure for two types of software failure data discussed above.

For the first type of data, suppose that estimation is to be performed at a specified time \( t_k \), not necessarily corresponding to a failure, and with total of \( m_k \) failures being experienced at time \( t_1, t_2, \ldots, t_{m_k} \). Then the likelihood function for the NHPP discussed above is:

\[
L = \left[ \prod_{i=1}^{k} \lambda(t_i) \right] e^{-\int_{0}^{t_k} \lambda(x)dx}
\]  
\[(1.80)\]
The MLE of the Parameters can be obtained by maximizing Likelihood function or its log likelihood function ($\log L$).

If the software failure data is grouped into $k$ points $(t_i, y_i)$; $i = 1, 2, \ldots, k$, where $y_i$ is the cumulative number of failure reports at time $t_i$. Then the Likelihood function $L$ is given as follows:

$$L \equiv \prod_{i=1}^{k} \frac{[m(t_i) - m(t_{i-1})]^{y_i - y_{i-1}}}{(y_i - y_{i-1})!} e^{-[m(t_i) - m(t_{i-1})]}$$  \hspace{1cm} (1.81)

Taking natural logarithm of (1.81) we get the log likelihood function:

$$LogL = \sum_{i=1}^{k} (y_i - y_{i-1}) \ln [m(t_i) - m(t_{i-1})] - m(t_k) - \sum_{i=1}^{k} \ln [(y_i - y_{i-1})!]$$  \hspace{1cm} (1.82)

The MLE of the parameters of SRGM can be obtained by maximizing (1.82) with respect to the model parameters.

Maximum likelihood estimators possess many desirable properties such as consistency, efficiency, asymptotic normality and the invariance property Musa, Iannino and Okumoto (1987). Hence it is the most preferred estimation procedure for relatively large sample size.

1.15 Comparison Criteria for SRGM

The performance of SRGM are judged by their ability to fit the past software fault data (goodness of fit) and to predict satisfactorily the future behavior of the software fault removal process (predictive validity) Musa (1999), Kapur et al (1999,2012), Musa, Iannino and Okumoto (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 following indices help in comparing SRGM.

1.15.1 Goodness of Fit Criterion

(a) The Mean Square Fitting Error (MSE): The model under comparison is used to simulate the fault data, the difference between the expected values, $\hat{m}(t_i)$ and the observed data $y_i$ is measured by MSE as follows:

$$MSE = \frac{1}{k} \sum_{i=1}^{k} (\hat{m}(t_i) - y_i)^2$$

where $k$ is the number of observations, and Sum of Square Fitting Error (SSE) is given by:

$$SSE = \sum_{i=1}^{k} (\hat{m}(t_i) - y_i)^2$$. The lower SSE, and hence, MSE indicates less fitting error, thus better goodness of fit (Kapur et al (1999, 2012)).

(b) The Akaike Information Criterion (AIC): It is defined as $AIC = -2(\text{The value of the maximum log likelihood function}) + 2(\text{The number of the parameters used in the model})$.

This index takes into account both the statistical goodness of fit and the number of parameters that are estimated. Lower values of AIC indicate the preferred model, i.e. the one with the minimum number of parameters that still provides a good fit to the data Kapur, Garg and Kumar (1999, 2012), Akaike (1974).

(c) Coefficient of Multiple Determination ($R^2$): This Goodness-of-fit measure can be used to investigate whether a significant trend exists in the observed failure intensity. We define this
coefficient 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:

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

(1.84)

\( R^2 \) 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. Larger the value of \( R^2 \) better the model explains the variation in the data Kapur, Garg and Kumar (1999, 2012).

(d) **Prediction Error (PE):** The difference between the observation and prediction of number of failures at any instant of time \( i \) is known as \( PE_i \). Lower the value of Prediction Error, better is the goodness of fit Pillai and Nair (1997).

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

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

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

(1.85)

Lower the value of Variation, better is the goodness of fit Pillai and Nair (1997).

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

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

(1.86)

Lower the value of Root Mean Square Prediction Error, better is the goodness of fit Pillai and Nair (1997).
1.16 Structure of Thesis

The work done is divided into five chapters. A brief summary of each chapter is given below. However, for detailed discussion, one may refer to the relevant chapter.

**Chapter-1** gives introduction to the research work presented in the thesis. Nature and importance of software testing are highlighted. It also contains exhaustive literature survey relevant to the thesis. Modeling in software reliability and evolution of SRGMs are briefly discussed along with the model validation techniques.

**Chapter-2:** During the software testing on the detection of a failure the fault that has caused the failure is isolated and removed. Most of the existing research in this area considers that similar testing efforts and strategy are required on each debugging effort. However this may not be true in practice. Different faults may require different amount of testing efforts and testing strategy for their removal. In this chapter we have made use of the fact that faults are classified into different categories as simple, hard and/or complex faults. This categorization is also extended to n-types of faults. Some of the existing research incorporates this phenomenon considering that the fault removal rate is different for different types of faults and remains constant during the overall period of testing. However this assumption may not apply in general testing environment in practice. It is a common observation that as the testing progresses the fault detection and/or removal rate changes. This change can be due to a number of reasons. The changing testing environment, testing strategy, skill, motivation and constitution of the testing and debugging personnel etc. are some of the major reasons behind this change. Using this fact we have propose a general software reliability growth model considering three types of faults in the software system incorporating the effect of changing fault-debugging rate using the change point concept. The general framework of the model can be reformulated for specific applications and testing environment with ease. Further we have formulated the model for the software system developed for critical application under a specific testing environment. The model is validated on real life data sets.
Chapter-3: Model unification is an insightful investigation for the study of general models without making many assumptions. In the literature various software reliability models have been proposed incorporating change-point concept. To the best of our knowledge these models have been developed separately. This chapter is based on unification schemes involving change in fault removal rate at some point of time. In Section 1 we have proposed a general framework for deriving several software reliability growth models with change-point concept based on non-homogeneous Poisson process (NHPP). The proposed framework helps in assessment of already existing change-point models along with three new models. Real data sets have been used for the validation of the models presented. In Section 2 we have considered fault detection-correction as separate processes in software reliability growth modeling. We have proposed a general framework for deriving several software reliability growth models for fault detection-correction process incorporating change-point concept based on non-homogeneous Poisson process (NHPP). Some existing change-point models along with new models have been derived from the proposed general framework. The models derived have been validated and verified using real data sets. Estimated Parameters and comparison criteria results have also been presented.

Chapter-4: Various authors have tried to develop a unifying approach so as to capture different growth curves, thus easing the model selection process. The work in this area done so far relates the fault removal process to the testing / execution time and does not consider the consumption pattern of resources such as computer working time, manpower and number of executed test cases etc. More realistic modeling can result if the reliability growth process is studied with respect the amount of expended testing efforts. Due to the complexity of software system and incomplete understanding of software, the testing process may not be perfect or the fault detection /correction rate may change at any time. In this chapter, we propose a generalized framework for deriving several existing as well as new testing effort dependent software reliability growth models incorporating change point and the possibility of imperfect debugging. The proposed framework is based on standard probability distribution functions. The developed models have been validated and verified using real data sets. Estimated Parameters and comparison criteria results have also been presented.
Chapter-5: In this part of the thesis we have discussed important management problems related to software. The chapter is divided into two sections. In Section -1 we have made use of the concept of features addition done in the software regularly. The up-gradation model has been characterized by increasing the number of features in the software that gives the firm competitive edge in the market. On the other hand, continuous up-gradation of software’s also brings complexity in the systems once it fails to work properly. Many software reliability growth models (SRGM) have been proposed over past three decades that estimate the reliability of a software system as it undergoes changes in removing the faults. But unfortunately most of the models do not consider anything about the increase in failure rate once an up-gradation is made in the software. The objective of this part of the chapter is to propose software reliability growth model that incorporates the effect of enhancement of features on software during testing and debugging process. Results have been supplemented with numerical examples. In Section -2 we have directed our work from single dimension to multi-dimensional framework. The new outlines of competition and collaboration that have arisen in software engineering as a result of the globalization process have an impact on the entire software process. Therefore to capture the combined effect of testing time and coverage we have proposed a two dimensional software reliability growth model. We have made use of the famous cobb-douglas production function to develop the two dimensional model. We have presented two software reliability growth models incorporating logistic distribution function and Exponentiated Exponential distribution function respectively. The proposed models are validated on real data set.

Conclusion of the work done, Scope for future research and an elaborate list of References are presented at the end of the thesis.

The work in this thesis is based on the following Research Papers:

1. P K Kapur, Archana Kumar, Kalpana Yadav, Jyotish Kumar, “Incorporating errors of different severity and change point in software reliability growth modeling” Quality


