CHAPTER 2

NATURE OF SOFTWARE RELIABILITY

The first boom of Reliability Theory happened during the late 1950’s with hardware systems[1]. Then, mathematicians simply applied standard techniques of statistics as well as probability to engineer reliability issues [2-3]. In 1961, study of software reliability began as an independent discipline [2], after Birnbaum, Esary and Saunders work upon coherent systems [3]. Since then mathematical theory of reliability gained momentum as a separate subject. It is important to note here that these early studies of Reliability were carried on what we technically call as “Coherent Systems”[2] which is quite similar to runtime software representation as proposed in our work.

Technically, a Coherent System is defined in terms of $\Phi(x)$, where $\Phi(x) = \text{a system structure utility that is non-decreasing in every vector argument, such that every component is correct (i.e. working)}$ [3]. An example coherent system is represented as a directed graph in Figure 2.1 below:

![Fig. 2.1: Example coherent system](image)

As depicted in fig. 2.1, coherent systems are considered to be operational if and only if there exists a working path between the source and terminal. Arcs as depicted in fig. 2.1 may fail independently but are assumed to have different failure probabilities. As coherent systems can be very complex there has been
huge interest in their proficient probability computation \cite{2,3}. Formally, a coherent system of \( n \) components can be mathematically described as follows \cite{4}:

Let the state \( x_i \) of component \( i \) be defined by

\[
x_i = \begin{cases} 
1 & \text{if it is functioning} \\
0 & \text{if it has failed} 
\end{cases} \tag{2.1}
\]

Similarly, the state \( \phi \) of the system is a deterministic binary function of the vector \( x = (x_1, x_2, \ldots, x_n) \) of component states:

\[
\phi(x) = \begin{cases} 
1 & \text{if the system is functioning} \\
0 & \text{if the system fails} 
\end{cases} \tag{2.2}
\]

Past five decades have witnessed a sudden increase in technical complexity of both hardware as well as software alike \cite{1}. The sophistication of science and technology is also growing at an exponential rate. Hardware is categorized using physics and empirical data \cite{2}, provides system engineers ability to accurately assess the likely reliability. Software has on the other hand evaded such model development \cite{4}. This difference can be attributed to the fact that factors affecting software reliability are not quantifiable using material data \cite{5}. Software reliability has always been characterized through data accumulated during system integration and test. Software Reliability is much larger and important than the processes advocated by CMMI (Capability Maturity Model Integration) and is vulnerable to many obscure and harder parameters to measure \cite{6}. Since its inception software reliability has always been analyzed using techniques and assumptions suitable for the examination of hardware systems \cite{4}. Unfortunately the hardware world has little if any similarity to the software entity. Hence most of the assumptions about hardware reliability are useless in software as the characteristics of both the problem domains are entirely different \cite{5}. In the preceding subsections of this chapter we now shall highlight and analyze the theory of reliability and its application to software as software reliability.

2.1 DEFINITIONS

Theory of Reliability utilizes various concepts with unique vocabulary and numerous mathematical as well as statistical expressions \cite{7,8}. Traditional software reliability estimation is based on applying mathematics and statistics to reproduce ancient failure data to forecast future software performance. We
discuss some of the important terminologies before proceeding further with this study. Unless otherwise indicated, these definitions are followed throughout this thesis and carry the same meaning [6-7, 9-11].

**TABLE 2.1- Important Terminology in Reliability Theory**

<table>
<thead>
<tr>
<th>S.No</th>
<th>Terminology</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Life Distribution</strong></td>
<td>Belief about the likelihood of failure time</td>
</tr>
<tr>
<td>2</td>
<td><strong>Increasing Failure Rate Average (IFRA) Distributions Theorem</strong> [2]</td>
<td>Denotes the smallest class of life distributions with exponentials closed under the construction of coherent systems and limits in distribution. Theorem: Let ϕ be a coherent system with independent component performance processes. Assume: 1. Non-repairable components have increasing failure rate (IFR) distributions. 2. Repairable components have exponential failure distributions. 3. Repair distributions have decreasing repair rate i.e. Decreasing Failure Rate (DFR). 4. All components are new at t=0.</td>
</tr>
<tr>
<td>3</td>
<td><strong>Fault</strong></td>
<td>A defect in computer software that causes a failure.</td>
</tr>
<tr>
<td>4</td>
<td><strong>Failure</strong></td>
<td>A Departure of computer program’s operation from the user’s requirement; it may be a crash in which the system ceases to function or simply malfunction.</td>
</tr>
<tr>
<td>5</td>
<td><strong>Failure Intensity</strong></td>
<td>Number of failures in a given time period. An alternative way of expressing software reliability. Generally represented using Failure Intensity Function, ( \dot{\lambda}(t) = \frac{dR(t)}{dt} \cdot \frac{1}{R(t)} ) (2.3)</td>
</tr>
<tr>
<td>6</td>
<td>**Failure Rate ( \lambda $$k/T ) )</td>
<td>Ratio of the total failures to a given unit of measure. In a batch of N items if at time t, a number k has failed. Then for a fixed period T in the life of an item, the ratio of the total number of failures to the total cumulative observed time. Expressed as ( \dot{\lambda} = k/T ) (2.4)</td>
</tr>
<tr>
<td>7</td>
<td><strong>Probability/ Degree of Belief</strong></td>
<td>Likelihood of the occurrence of a given event or the theoretical relative frequency of an event. ( Prob(x) \in [0, 1] ) (2.5)</td>
</tr>
<tr>
<td>8</td>
<td><strong>Quality</strong></td>
<td>Conformance to Specification</td>
</tr>
<tr>
<td>9</td>
<td><strong>Reliability</strong></td>
<td>The conditional probability that an entity can perform its specified function for a particular interval under stated conditions (probability of non-failure in a given period). It is the extension of quality into the time domain.</td>
</tr>
<tr>
<td>10</td>
<td>**Time ( \tau $$t ) )</td>
<td>Reliability is the extension of quality into the time domain. It is defined as a function of the interlude of time when the</td>
</tr>
</tbody>
</table>
Wearout

Process that results in an enhancement of the failure rate or likelihood of failure as the number of life units increases.

Software Reliability

The possibility of failure-free function of a computer program/software for a particular period of time under some specified conditions. Represents an exponential function of failure intensity.

\[ R(t) = P(T > t) = \int_t^\infty f(x)dx = 1 - F(t) \quad (2.6) \]

Operational Profile

Set of functions a computer program is required to execute, further broken down by input data when they affect execution along with their associated probabilities of occurrence. It is actually a statistical explanation of the environment in which software is executed.

Software Reliability Model

A mathematical model that identifies the common form of the software failure procedure as a function of factors like fault introduction, fault removal and the operational environment.

Mean Time Between Failure (MTBF)

The normal or observed time between successive failures in a system or constituent. Calculated as the ratio of the total collective observed time to the total number of failures.

Expressed as: \( \bar{Q} = \frac{T}{K} \), hence, \( Q = 1/\lambda \) \quad (2.7)

Mean Time To Failure (MTTF)

A basic measure of reliability for non-repairable items. The collective time in the life of an item inhabitants divided by the amount of failures within that population, through a particular measurement under stated conditions.

Expressed as: \( \bar{t} = T/K \) \quad (2.8)

Hazard Rate

Specifies the alteration in the failure rate over the life of a population of components. Described via hazard function, \( h(t) \) which is defined as the limit of failure rate as the interval approaches 0.

\[ h(t) = \lim_{\Delta t \to 0} \frac{R(t) - R(t + \Delta t)/\Delta tR(t)}{f(t)/R(t)} \quad (2.9) \]

The terminologies explained in Table 2.1 above shall be used throughout this thesis with the same meaning, unless otherwise specified.

The common approach for estimating software reliability has been the use of some software reliability models applying varied software metrics \([12]\). Despite numerous available software reliability estimation models, ensuring reliability remains one of the most prominent problems of software development. With the current outburst of technology, any software produced is expected to execute failure free its anticipated tasks at a desired performance level for some given period. This set
of quality features have been united under the common notion of reliability \cite{6}. In general terms, reliability of a product is the probability that the underlying item will perform its intended functions. Hence, if $T$ variable represents the time-to-failure, then the reliability function associated with $T$ at time instance $t$, can be represented mathematically as given below in eq.11:

$$R(t) = \text{Prob}(T \geq t)$$  \hspace{1cm} (2.10)$$

Consequently, the complement of reliability stated as non-survival or non-reliability can be represented as in eq.12:

$$F(T) = 1 - R(t)$$  \hspace{1cm} (2.11)$$

From eq. 11, it is clear that at:

$$T = 0, R(0) = 1, F(0) = 0$$  \hspace{1cm} (2.12)$$

Hence at

$$T = \infty, R(\infty) = 0, F(\infty) = 1$$  \hspace{1cm} (2.13)$$

Eq. 13 and 14 above imply that any human engineered product is bound to fail after a certain period of time. Hence, completely reliable software which works for an indefinite time period is too high an expectation. However, accurate reliable operation for all software created at each runtime instance is something one can strive for.

2.2 RELIABILITY: HISTORICAL BACKGROUND

The first in print definition of the term “reliable” as relevant to electronic tools or components was by Rowe \cite{13}:

“A reliable valve (tube) is characterized by having a very high probability that it will operate normally when taken from stock and installed in equipment for which it was intended, and an extremely low probability that it will fail during subsequent operation in that equipment for some definite period of time”.

The term was later modified to “reliability” to give it a more quantitative expression.

The impetus to the science of reliability as a separate subject was due to the failure of American and British aircrafts around 1950’s \cite{2}. Naturally traditional reliability models were first developed for industrial, mass produced hardware
products such as electronic items and consumer goods. Hence, during the period of 1950’s to 1960’s many models for estimating reliability of electronic goods and components were suggested \cite{3,14}. Table 2.2 below lists some of these notable works:

**TABLE 2.2: NOTABLE HARDWARE RELIABILITY MODELS**

<table>
<thead>
<tr>
<th>S.No</th>
<th>Model</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Duane’s Power Model\cite{15}</td>
<td>$\mu(x) = ax^\beta$ (2.14)</td>
<td>Earliest reliability model, called power model; the mean function is defined as a power of $x$. Result is an NHPP.</td>
</tr>
<tr>
<td>2.</td>
<td>Crow’s Power Model \cite{14}</td>
<td>$\mu(t; \beta) = \beta_0 t^{\beta_1}$ (2.15) [ $\lambda(t; \beta) = \beta_0 \beta_1 t^{\beta_1-1}$ (2.16)</td>
<td>Estimates reliability of hardware systems through development testing. Models failure events as an NHPP process.</td>
</tr>
<tr>
<td>3.</td>
<td>Gompertz Reliability Growth Curve (1825)</td>
<td>$R = R_x^\alpha e^{\beta T}$ (2.17)</td>
<td>Basis of earliest known probabilistic mortality tables. Used to approximate number of outstanding faults in testing phase of software development. Formed the basis of stochastic models like Gompertz software reliability model \cite{105}.</td>
</tr>
<tr>
<td>4.</td>
<td>Gumbel (or Gompertz) Model / Log-Weibull Model/ Double Exponential Model \cite{16}</td>
<td>$cdf: F(x; \mu, \sigma) = \exp \left{ - \exp \left( - \frac{x-\mu}{\sigma} \right) \right}$ (2.18) [ $pdf: f(x; \mu, \sigma) = \frac{1}{\sigma} \exp \left( - \frac{x-\mu}{\sigma} \right) (2.19)$ [ $h(x) = \frac{1}{\sigma} \exp \left( - \frac{x-\mu}{\sigma} \right)$ (2.20)</td>
<td>Used to model the distribution of maximum or minimum of a number of samples of various distributions. First applied to engineering problems.</td>
</tr>
<tr>
<td>5.</td>
<td>Hyperbolic Reliability Model \cite{17}</td>
<td>$h(t) = \frac{\alpha}{t + 1}$ (2.22) [ $R(t) = e^{-H(t)}$ (2.23)</td>
<td>Based on “Stress-Strength,” and “Shocks” failure models. Used to estimate reliability of non-repairable systems.</td>
</tr>
</tbody>
</table>

It is notable here that above models for estimating reliability of hardware components were based on assumptions on nature of hardware failure and
operation. Using these assumptions certain mathematical functions were evaluated to estimate or optimize the life distribution, survival probability or mean life of hardware components.

From the above discussion, it is implicit that early reliability theory was majorly a direct application of the standard probability theory as it was dependent on probabilities; mean values, distributions etc [2]. This theory was successfully applied to hardware systems and was later extended to software also. However, the issue of software reliability prediction using the idea of logically repeatable component failure rate [3] as in hardware is dubious for software. The preceding sections validate this point.

2.3 HARDWARE VERSUS SOFTWARE RELIABILITY

There is a long-standing tradition for deciding reliability of hardware systems. Hence, it is natural to draw parallels between hardware and software reliability prediction [18]. Reliability studies actually began in late 1950’s with no established theory and little data about hardware component failures [18]. Following hardware reliability theory [3] and models [2-3, 14-15, 19], software reliability modelling also came into focus. The first software reliability model was a Markov birth-death model in 1967 [18]. However earliest software reliability models developed and published in open literature was by Jelinski and Moranda [20] in 1971. Hence, hardware reliability can be called as the precursor to the theory of software reliability. As a result it is natural that software reliability theory was influenced and developed as an extension to the mathematical theory of hardware reliability [1-2]. However now it is well-proven that the basic nature of hardware components is different from their software counterparts [6, 21]. Hence, we argue that similar treatment of two characteristically different entities is infeasible. To further support our argument Table 2.3 below examines the basic differences between hardware and software reliability [1, 22].
TABLE 2.3: HARDWARE VERSUS SOFTWARE RELIABILITY

<table>
<thead>
<tr>
<th>S.No</th>
<th>Characteristics</th>
<th>Hardware Reliability</th>
<th>Software Reliability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Definition</strong></td>
<td>Probability that an electronic product gives acceptable performance for a preset period of time when used in the way and for the use planned(^{[13]})</td>
<td>The possibility of failure-free action of a computer program/software for a definite period of time under some specific conditions (^{[56]}). Represents an exponential function of failure intensity. [ R(t) = P(T &gt; t) = \int_{t}^{\infty} f(x)dx = 1 - F(t) ] (2.24)</td>
</tr>
<tr>
<td>2</td>
<td><strong>History</strong></td>
<td>Reliability studies began in 1950’s (^{[2-3]})</td>
<td>Late 1960’s saw the first efforts towards software reliability modelling.</td>
</tr>
<tr>
<td>3</td>
<td><strong>Pioneers</strong></td>
<td>Z.W. Birnbaum (^{[3]}), Richard E. Barlow (^{[2,10]}) etc.</td>
<td>Jelinski&amp;Moranda, John D. Musa M.L. Shooman, Hoang Pham etc.</td>
</tr>
<tr>
<td>4</td>
<td><strong>Cost</strong> (^{[18]})</td>
<td>Tremendous computing power is available with modest weight, volume and power requirements at low cost. Hence, advances in Integrated Circuit (IC) technology have caused relative decrease in hardware costs.</td>
<td>Software costs are primarily labor intensive, rather than technologically dependent. With increase in software complexity man-hours spent on software development have increased. Hence large portion of computer system development costs are due to software.</td>
</tr>
<tr>
<td>5</td>
<td><strong>Data</strong> (^{[65]})</td>
<td>Hardware Component Failure Rate Handbook, MIL-HDBK-217, a government standard is being published since 1962.</td>
<td>No comprehensive software error and reliability database.</td>
</tr>
<tr>
<td>6</td>
<td><strong>Causes of Failure</strong></td>
<td>Aging, Wearout etc or faults in design, manufacture, maintenance or misuse.</td>
<td>Design Defects, Bugs or Faults in the software, incorrect user input or hardware errors.</td>
</tr>
<tr>
<td>7</td>
<td><strong>Nature of Models</strong> (^{[18]})</td>
<td>Decompose a system into a structural model composed of system elements.</td>
<td>Regard the software as a black-box with no consideration to the software structure (macro model)</td>
</tr>
<tr>
<td>8</td>
<td><strong>Manufacturing</strong></td>
<td>Include parts, assembly.</td>
<td>Cost of storage media (disks or...</td>
</tr>
<tr>
<td>Costs (^{[18]})</td>
<td>inspection and test.</td>
<td>tapes), the computer time, cost of copying the master version of the program as well the printing of extra copies of the pertinent operating manuals.</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>Effect of Time</td>
<td>Hardware Reliability generally tends to reduce with time</td>
<td>Software reliability may increase or decrease with time (due to bug elimination) or may remain similar if no changes are made.</td>
<td></td>
</tr>
<tr>
<td>Effect Anomaly</td>
<td>Anomaly may lead to unpredictable failure or have no outcome.</td>
<td>One incorrect bit can lead to a disaster.</td>
<td></td>
</tr>
<tr>
<td>Predominant Phase</td>
<td>Design as well as production phases predominate</td>
<td>100% design (assembly is insignificant involving copy to CD or Diskette)</td>
<td></td>
</tr>
<tr>
<td>Events</td>
<td>Mathematical models can be applied to test all events.</td>
<td>Amount of events is huge and tends to be exclusive to each software instance.</td>
<td></td>
</tr>
<tr>
<td>Reliability Growth</td>
<td>Redundancy can be utilized to augment reliability. Maintenance may also improve reliability</td>
<td>Redundancy may not necessarily lead to enhancement. Maintenance may introduce new bugs.</td>
<td></td>
</tr>
<tr>
<td>Failure Description</td>
<td>Described through physical laws</td>
<td>No such equivalent laws exist.</td>
<td></td>
</tr>
<tr>
<td>Interfaces</td>
<td>Physical Structures</td>
<td>Conceptual Structures</td>
<td></td>
</tr>
<tr>
<td>Standards</td>
<td>Standard Parts used</td>
<td>No comparable standards, if available seldom used.</td>
<td></td>
</tr>
<tr>
<td>Safety Margins</td>
<td>Inbuilt in hardware design</td>
<td>No such safety margins found</td>
<td></td>
</tr>
<tr>
<td>Parts and Paths</td>
<td>Limited paths with finite number of parts</td>
<td>Many more parts and paths</td>
<td></td>
</tr>
<tr>
<td>Failure Modes</td>
<td>Well-Defined (^{[23]})</td>
<td>Not well-defined</td>
<td></td>
</tr>
<tr>
<td>MTTF/MTBF</td>
<td>MTTF commonly used in hardware when components are not repaired but directly replaced. MTBF used when hardware components are repaired.</td>
<td>MTTF is not completely applicable to non-redundant software systems (i.e. repairable). However it is applicable to redundant (e.g. fault tolerant) software systems. MTBF more common metric for repairable software systems.</td>
<td></td>
</tr>
</tbody>
</table>
Table 2.3 above highlights the major differences between hardware and software. Due to the large amount of variation, we conclude that the methods and models implemented for hardware components are inappropriate for software reliability analysis. However, many models for hardware reliability have been used for software as well. In sub-section 2.3.1 we further explore on previous observations in table 2.3. We explain how different input properties for hardware and software as well as non-analogous data from software testing in comparison to hardware testing differentiate software reliability composition and estimation from hardware.

2.3.1 Hardware versus Software Failure

Reliability for hardware system or component is controlled of four factors which may result in faults [24]:

i) Design Errors

ii) Manufacturing Errors

iii) Physical Wear-out and Aging

iv) Physical Defects

Varied hardware designs trade-off involvement of the above factors alongside qualitative elements like cost and reliability. The hardware failures resulting from any of the above defects is modelled as what is popularly called as the “bathtub curve” (fig. 2.2).

![Fig.2.2: Bathtub Curve for Hardware Reliability](image-url)
The bathtub curve in fig. 2.2 above indicates a high early failure rate which may be attributed to design or manufacturing errors or any physical wearout of parts. Once a system gets past this initial mortality period it tends to have low failures and maintains its performance for an extended interlude of time. Any failures during this period may be a result of environmental factors like heat, dust etc. Finally as aging sets in, the failure rate tends to rise. It is to be noted here that the bathtub predicts the failure history of a single product design over its life. However, in software if a design is modified or maintained over time, it shall result in a family of bathtub curves [24]. This family of curves may also differ from each other in terms of the length of the initial failure period and extended useful life period. In many cases the curve may not even report a final wearout phase and may give way to another curve. Hence, we argue that a bathtub curve is not possible to model software reliability as software failures do not comply with bathtub curve. Thus, bathtub curve cannot be used as a guide to software reliability. As the fundamental nature of software failure is unlike that of hardware (there is no wearout, aging etc) hardware reliability models should not be imposed on software.

2.4 SOFTWARE RELIABILITY MODELS

Following Hardware Reliability, much progress was made in software reliability starting since early 1970’s. As hardware and software do not operate in vacuum [25], the mathematical theory of hardware reliability [2], had a profound effect on the software reliability models. Beginning since 1971, more than two-hundred different models have been proposed and used on different software applications till date [5, 26]. Carrying forward the hardware legacy all these models use parameters, functions and assumptions like their hardware counterparts [27]. Prior to discussing these models in detail, we first understand the common parameters, functions and assumptions that these models apply.

2.4.1 Parameters in Software Reliability

Many different parameters are used in software reliability estimation [6]. These parameters are generally related to the size of the program, amount of errors in the program or the nature of software failure. Some of these parameters like MTBF,
MTTF, Failure Rate and Hazard Rate have already been explained in Table 2.1 above. Table 2.4 below lists some other common metrics employed in mathematical or statistical functions for software reliability estimation.

**TABLE 2.4: Common Parameters in Software Reliability Estimation**

<table>
<thead>
<tr>
<th>S.NO</th>
<th>PARAMETER</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( N )</td>
<td>Original Amount of faults present in the software</td>
</tr>
<tr>
<td>2</td>
<td>Failure Rate, ( \lambda )</td>
<td>The ratio of the amount of failures to a specified unit of measure. Expressed as ( \lambda = k/T )</td>
</tr>
<tr>
<td>3</td>
<td>Mean Time Between Failure (MTBF)</td>
<td>The normal or actual time between repeated failures in a system or component. Calculated as the fraction of the total aggregate observed time to the total amount of failures. Expressed as: ( \bar{T} = \frac{T}{K} ), hence, ( Q = 1/\lambda )</td>
</tr>
<tr>
<td>4</td>
<td>Mean Time to Repair (MTTR)</td>
<td>Fundamental measure of the maintainability of repairable systems. Symbolizes the standard time required to restore a failed component or appliance.</td>
</tr>
<tr>
<td>5</td>
<td>Mean Time To Failure (MTTF), ( \bar{T} )</td>
<td>Fundamental standard of reliability for non-repairable items. The entire time in the life of an item population by the number of failures within that population, during a particular measurement under stated conditions. Expressed as: ( \bar{T} = T/K )</td>
</tr>
</tbody>
</table>

The above parameters form the base for quantitative estimation of software reliability. Parameter values are obtained using post-failure data along with model assumptions.

**2.4.2 Functions in Software Reliability**

Formally, software reliability is expressed as the conditional possibility of failure-free software execution. Popularly software reliability is expressed as an exponential function of time. Mathematically, a function say, \( f(x)^{[28]} \), is a special relationship between values \(^{[26]}\). Each of its input value gives back exactly one output value. Functions are important tool for quantitative estimation to represent dependency. Hence, they form the basic tool of software reliability estimation. Table 2.5 below lists some common functions in software reliability. It is important here to note that these functions are based on parameters which use certain assumptions regarding software nature to simplify reliability estimation.
### TABLE 2.5 Common Functions for Software Reliability Estimation

<table>
<thead>
<tr>
<th>S.No</th>
<th>FUNCTION</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$m(t)$</td>
<td>Mean Value Function</td>
</tr>
<tr>
<td>2</td>
<td>$\lambda(t)$</td>
<td>Failure Intensity Function</td>
</tr>
<tr>
<td>3</td>
<td>$R(t)$</td>
<td>Software Reliability</td>
</tr>
<tr>
<td>4</td>
<td>$d(t)$</td>
<td>Fault Detection Rate per Fault</td>
</tr>
<tr>
<td>5</td>
<td>$Z(\Delta t/t_{i-1})$</td>
<td>Hazard rate for software, the possibility of experiencing the $i^{th}$ failure at $t_{i-1} + \Delta t$ given that $(i-1)^{st}$ failure occurred at $t_{i-1}$</td>
</tr>
<tr>
<td>6</td>
<td>$Z(t)$</td>
<td>Per fault hazard rate</td>
</tr>
</tbody>
</table>

The functions in table 2.5 above have been used by different reliability estimation models discussed in Appendix A. It should be considered that different software reliability estimation models use similar functions but the parameters may differ depending upon the assumptions made.

The two most significant parameters being, the mean value, $\mu(t)$ and the failure intensity function, $\lambda(t)$ respectively. Table 2.6 lists these functions for some basic models:

### Table 2.6 Mean Value and Failure Intensity Functions of common Reliability Models

<table>
<thead>
<tr>
<th>S.No</th>
<th>Model</th>
<th>Mean Value $\mu(t)$</th>
<th>Failure Intensity $\lambda(t)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Musa’s Basic Execution Time</td>
<td>$V_0(1-\exp(-\lambda_0 t)/V_0)$</td>
<td>$\lambda_0 \exp(-\lambda_0 t)/V_0$</td>
</tr>
<tr>
<td>2.</td>
<td>Logarithmic Poisson Execution Time</td>
<td>$1/\theta \ln(\lambda_0 \theta t+1)$</td>
<td>$\lambda_0 \exp(-\theta \mu)$</td>
</tr>
<tr>
<td>3.</td>
<td>Jelinski – Moranda</td>
<td>$N(1-\exp(-\phi t))$</td>
<td>$N\Phi \exp(-\Phi t)$</td>
</tr>
<tr>
<td>4.</td>
<td>Goel-Okumoto NHPP Model</td>
<td>$a(1-e^{-bt})$</td>
<td>$abe^{-bt}$</td>
</tr>
</tbody>
</table>
2.4.3 Common Software Reliability Model Assumptions

Software reliability models generally make certain suppositions about the software and its failure method. Some of these common suppositions are listed below [25]:

- Failures are independent of each other.
- Every failure in a failure class has similar chance of being detected.
- Faults are corrected instantly.
- Software is executed under the same environment as that during true operation.
- Removal of a fault does not bring in new defects.

Comprehensive details related to model assumptions may be referred from Farr (1995) and Xie (1991, 1993) and in the references listed for specific models [21, 29-30].

2.4.4 Software Reliability Models: Classification

Formally software reliability is the continuity of correct service delivery [31, 32]. Estimating reliability of software systems of varying size and complexity is essential to establish operational confidence [21]. To accomplish this challenge different software reliability models have been proposed since early 1970's [5-6, 11, 20, 24-25, 27, 30-31, 33-50]. Both white-box as well as black-box approaches have been used to model software reliability. The **black-box approach** is a simple external program measurement using rate of execution, time between failures and certain suppositions related to the related probability distributions [20, 25]. The **white-box approach** includes an evaluation of software reliability based on analysis of the complication of the program and composition of the code [38]. Many of the software reliability models suggested are primarily based on black-box approaches and hence called **Macro Models** [25]. The other set of white-box software reliability models are also known as **Micro models** for software reliability estimation [25].

Irrespective of the approach used, **assessment involves estimating a basis of assurance in the runtime correctness of the software** [5]. Varied numbers of models have been suggested for estimating the reliability of software systems. Currently the number of software reliability estimation models is well beyond two-hundred models as on record. Despite this software reliability theory is still
rudimentary [31]. The theoretical basis of conservative software reliability models is also viewed doubtfully [29, 45]. Many software reliability models endeavor to quantify reliability as a measure of its failure history or underlying faults in the system (black-box) [20, 41-45]. To analyze such a large, heterogeneous set of varied models is practically impossible. Hence, we first classify them on basis of their main criteria. Fig.2.3 below depicts the basic categorization of the existing conventional software reliability models. The categorization has been adopted from [2, 5, 21] and is primarily based on the basic nature of the model.

![Classification of Conventional Software Reliability Models](image)

**Fig.2.3: Classification of Conventional Software Reliability Models**

On basis of our study, we classify all the accessible software reliability models broadly as **black-box or non-architectural or conventional reliability models** and **architecture-based models**. Of these categories non-architecture based models do not regard the internal structure of the software and estimate software reliability using certain parameters, functions and assumptions regarding the software. This is a more popular class of models [21] which has been widely used since the inception of software reliability. On basis of our study, we further
classify all conventional software reliability models on basis of their underlying assumptions into two broad categories: the time-domain and data-domain models\textsuperscript{[21]}. The time-domain models are further classified as time-between failure and fault-count models. The time-between failure models are further classified into four classes namely: homogeneous Markov, semi-Markov, non-homogeneous Markov and other. The second category data-domain models are further classified as fault-seeding and input-domain models. The reason for adopting the above classification\textsuperscript{[21]} is discussed below:

- **Data-Domain Models**: These models work on the basic belief that if all input permutations to software can be determined, then its reliability estimate can be obtained by executing all the input combinations. The models under this category are further classified into:

  - **Fault-Seeding Models**: These models assume that a software product contains an unidentified number of native faults. Such software can be seeded with known amount of faults and submitted to testing. Using the ratio of discovered seeded faults and the discovered actual faults, one can estimate the actual number of native faults.

  - **Input-Domain Models**: Such models believe that the reliability of software can be estimated by executing a set of arbitrarily selected inputs. The ratio of the successfully executed inputs to the total amount of inputs gives an approximation of the software reliability.

- **Time-Domain Models**: These models represent the software failure process and use the software failure history to estimate the remaining amount of faults in the software and the time necessary to test them\textsuperscript{[21]}. The models are further classified into:

  - **Time-Between Failure Models**: This class of models presumes that software failures are independent of each other. Further each fault in the software has equal probability of executing into failure. Further once a
failure executes the system can recover from it in negligible time. This subclass of time-domain models is further sub-classified as follows:

- **Homogeneous Markov Models:** These models presume unknown but fixed initial amount of faults in the software under consideration. Further this class uses the quantity of faults in the software at any instance as the state space for a homogeneous Markov chain. Thus the failure intensity of the software or transition rate of the Markov chain depends on the number of remaining faults in the software.

- **Non-Homogeneous Markov Models:** These models presume the number of faults in software to be a random variable that follows the behavior of a Non-Homogeneous Poisson Process (NHPP). It is a popular class of models.

- **Semi-Markov Models:** These models like homogeneous markov models assume unknown but fixed initial quantity of faults in the software. However the failure intensity of the software or transition rate from a given state is a function of the quantity of residual faults in the software as well as the time elapsed in that state.

- **Other Models:** This class represents the models where the failure intensity follows a Bayesian Distribution and hence cannot be represented as a Markov process.

The above classification is an adoption from work of Swapna S. Gokhaleet. al.(1996) [21]. The advantage of the state-space view if the time domain models in this classification are that it can be easily extended to include imperfect detection/repair.

- **Fault Count Models:** This subclass of time-domain models assumes independent fault detection rate with independent homogeneous testing intervals.

Table 2.7 below outlines the above classification with a few example models.
<table>
<thead>
<tr>
<th>Main Feature</th>
<th>Software Reliability Models</th>
<th>Assumptions &amp; Parameters</th>
<th>Critique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Domain Models: Identify set of all input combinations, reliability estimate obtained by exercising all combinations.</td>
<td>Mills Hypergeometric Model [51]</td>
<td>Seeded Faults: Randomly dispersed in the program; Both indigenous as well as seeded faults have independent probability of being detected.</td>
<td>Requires Large Number of test cases, Selection of all input combinations impossible in practice.</td>
</tr>
<tr>
<td>Error/ Fault Seeding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input Domain</td>
<td>Nelson Model [43]</td>
<td>Known Input set; Input set partitioned into equivalent classes; uses random testing.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ramamoorthy and Bastani Model [6]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time Domain Models: Model underlying failure process using observed failure history as a Markov, Poisson or Bayesian Distribution.</td>
<td>i) Homogeneous Markov: Jelinski &amp; Moranda [20], Goel-Okumoto Imperfect Debugging [46].</td>
<td>Fixed and unknown amount of initial faults; amount of faults in the system at an instance form state space of a homogeneous Markov Chain; failure intensity dependent on number of residual faults.</td>
<td>Debugging process is not perfect and may introduce new errors (imperfect debugging); state to state transition is not governed by residual faults.</td>
</tr>
<tr>
<td>Time Between Failure Models: independent time between failures; equal probability of fault exposure; perfect debugging.</td>
<td>ii) Non-Homogeneous Markov: Goel-Okumoto NHPP [47], Delayed S-Shaped [44].</td>
<td>Amount of faults in a software represented as a random variable with Non-Homogeneous Poisson Process (NHPP) behavior.</td>
<td>Requires test or field data to estimate model parameters; S-shaped curve observable only in</td>
</tr>
</tbody>
</table>
Table 2.7 by no means is an exhaustive classification of existing software reliability models. However, from Fig.2.3 and Table 2.7, we observe that conventional software reliability estimation models are based on either of two different approaches for calibrating software reliability. However, both time-domain as well as data-domain approaches have their own set of assumptions for estimating software reliability. Interestingly, none of the model assumptions actually represent the actual nature of software execution. As a result, accurate software reliability estimation still remains an open challenge. To meet this challenge, architecture-based reliability models are now being applied. These models enable what-if sensitivity analysis for software. They can be either state-based model which utilises the control flow graph to characterize software structure. The second category of path-based models computes reliability using the execution path of the program. These models have been found much more useful for software-related decision making at any point during the software life cycle. Our automata-based reliability model can also be included under the architecture-based state-based category of reliability model.
The above classification broadly classifies all the existing software reliability estimation models. For a detailed classification of individual software reliability estimation models refer Appendix A.

2.5 CRITICAL REVIEW

Current computer software is among the most intricate and unpredictable entity of the current times\(^5\). To achieve this several analytical models for software reliability and fault content estimation are available. The correctness of these models is however dubious \(^6\). Current software reliability estimation methods imitate their hardware counterparts \(^24\). Software reliability estimation challenge despite profound work remains an open ordeal. This weakness of the existing software reliability models can be accredited to a number of reasons like their inability to handle the, as yet, unidentified connection between defects and failures\(^32\). The existing software reliability models make certain assumptions to model the software failure process. The validity of these assumptions is questionable. We discuss these assumptions in light of the classification classes created for the on hand software reliability models in the above section. Table 2.8 below critically analyses the basic assumptions of the data-domain and time-domain models.

**TABLE 2.8: Critique of Data-Domain and Time-Domain Model Assumptions**

<table>
<thead>
<tr>
<th>SOFTWARE RELIABILITY MODEL CATEGORY</th>
<th>CRITICAL ANALYSIS</th>
</tr>
</thead>
</table>
| Data-Domain Models               | i) All possible input combinations to a program can be identified.  
**Reality:** Identification of all possible input combinations (valid as well as invalid) to a program is not feasible. Ex: Small program to add two 8-bit positive integers will have \(2^8 \times 2^8\) valid input combinations and many more invalid pairs. Determining all possible pairs is practically infeasible. As a result such models require lengthy testing times. |
| Time-Domain Models               | i) Model the fundamental failure process using observed failure history.  
**Reality:** Adequacy and availability of failure history |
data is questionable and may differ from software to software. Further the time-domain models underestimate the number of remaining errors. One should always remember that faults may not be uniformly distributed nor they may be of the same severity. Also unlike hardware repair, software fault repair may not always result in a better product as new faults may be initiated in the software during repair.

Despite numerous reliability estimation models, software reliability eludes a generic estimation model that can be applied to all software under all circumstances to accurately predict system reliability. The major underlying reason for this is the dubious assumptions and the fundamental statistical and data quality problems that undermine the validity of the existing models. Many researchers [53-54] have elaborately explained the reasons for the inaccurate estimates by the existing software reliability models.

We outline some of them below:

i) All existing software reliability models are confined to the application of software system during test. Most of the models being statistical models acknowledge some kind of failure history data during test (i.e. time since last failure MTBF/MTTF or number of failures discovered in an interval) as input and produce system reliability estimates for software systems completely ignoring their actual execution.

ii) Many software reliability models use the same mathematical techniques used to model hardware reliability. However, hardware reliability modeling predicts how system reliability decreases over time due to component wear-out. While software reliability predicts how system reliability keeps changing with additional testing and debugging[45].

iii) Hardware reliability can be maintained at a given level whereas software
reliability tends to alter during the software system life cycle.

iv) Software failures are not a result of physical system deterioration, rather they result from defects in software requirements, design or code\(^5\). Further most software reliability models make some set of assumptions about the software and the testing and debugging process. However the validity of these assumptions remains questionable. Table 2.9 below lists some common reliability model assumptions.

Table 2.9: Some Common Software Reliability Model Assumptions

<table>
<thead>
<tr>
<th>S.No</th>
<th>Assumption</th>
<th>Reality</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>During testing, software is executed similar to its actual operational usage.</td>
<td>Actual operational usage of a system is controlled by many human-controlled factors and hence may vary significantly from testing.</td>
</tr>
<tr>
<td>2.</td>
<td>There is an upper limit to the amount of failures experienced during testing.</td>
<td>There can be no upper limit on the number of failures observed during system execution. All system defects will also not be of the same size and severity.</td>
</tr>
<tr>
<td>3.</td>
<td>Perfect Debugging: No new defects are added into the code during debugging</td>
<td>There is always the possibility of introducing new defects during debugging.</td>
</tr>
<tr>
<td>4.</td>
<td>Independent Defect Detection: Defect Detection is independent of one another.</td>
<td>The assumption ignores the simple fact that a failure may be the result of more than one defect.</td>
</tr>
</tbody>
</table>

The above assumptions which are common to most reliability models have been made to make software reliability computations simpler and tractable. However, they contribute to the inaccuracy of the resulting estimates.

2.6 WHY A NEW APPROACH IS WARRANTED
As discussed in the previous section, fig.2.4 below models all the major limitations of software reliability modeling.

Methods and techniques for estimating reliability of software systems will be required as long as we carry on manufacturing fault-prone software. In this state of affairs, the existing reliability models fail to predict actual operational reliability of software \(^{[59]}\). The reasons for the same have been highlighted in fig. 4 above. As existing software reliability models are an imitation of their hardware predecessors, they suffer from four major limitations in the software context.

i) Applicability of assumptions made to simplify reliability calculations is debatable. The assumptions have already been criticized in the discussion above.

ii) Like hardware, software also requires post-failure history data in order to estimate reliability. However, unlike hardware there is no standardised, documented data available. Instead in many contexts for example, completely new software this data may not even be available.

iii) Existing software reliability models quantitatively estimate software reliability when in actual operational usage. Hence these models are simply
predictive models. No efforts are made at ensuring or tracking the altering software reliability through the software life.

iv) Conventional software reliability estimation models are either used through the testing or validation period of the software. There is no present model that can be applied across all stages of software life cycle.

All the above factors converge to a common point, namely, the need for more realistic modelling and accurate quantification of the software production process. Due to the basic differences in the nature of software as compared to its hardware counterpart along with the above reasons, software reliability is still a grey area of software engineering. The software reliability problem is no longer so insignificant so as to be ignored; neither can it be dealt with the current estimation approaches. Hence, it is time some reliable solution for software reliability problem be worked out.

What we require is a realistic model with respect to software execution. Such models can be used to predict the operational reliability of software system during execution. However there are no mature models of this type and hence this remains a topic of great interest to the software reliability community. Such a model should be able to:

- Control the reliability of the developed and tested software in a simple and well-organized way with a higher correctness than currently possible.

To solve this problem we first need to understand that software execution is actually a sequence of input $\rightarrow$ program $\rightarrow$ output. This implies that for some defined input the program processes the input to produce the desirable output. Hence, software reliability is basically a function of accurate I/O Pair $<i, o>$ at runtime. To elaborate the above statement further we explain the same in terms of Hoare’s rule as discussed in Chapter 1. Formally software execution can be represented as a set of logical rules which can help determine the correctness of software at runtime. Just like Hoare triple, software execution can be described as:

$$\{I\} C \{O\} \tag{2.28}$$

Where

$I=$ Precondition, which in case of software is input value $i$ at a given instance, where $i \in I$
O= Postcondition, which in case of software is output value o at a given instance, where o ∈ O

C= Command(function or method) that causes software transition.

Thus runtime reliability estimation is the true estimate for software reliability.Runtime reliability estimation focuses on software behavior at runtime. Formally, it is the use of runtime software representation for reliability estimation \(^{[38]}\). This kind of software analysis allows changes to be made to execution and thus allows prevention of fault execution. The technique implies enhancing runtime software to perform tasks like software state investigation and reliability estimation \(^{[57-60]}\).

As software reliability is a dynamic system attribute that alters at runtime. Thus, runtime software representation is the most appropriate source for software reliability estimation. The balance of this dissertation describes a novel runtime reliability estimation methodology that uses the probabilistic automata representation of runtime software to monitor and control software reliability. The resulting model is correct and formal enough to ensure an intelligent, self-healing software reliability control utility that can help control and tap the software reliability monster.

2.7 SUMMARY

This chapter throws light on the foundations of the theory of software reliability. Further important terminologies and parameters used in the estimation of software reliability have been defined. The chapter further marks out the major differences between hardware and software reliability. We also classify the numerous available software reliability models on basis of common characteristics. The chapter concludes with a discussion of the fact that why a new approach for software reliability estimation is still warranted.
REFERENCES


