Chapter 2

Literature Survey

2.1 Introduction

This chapter provides concepts, definitions and understandings related to Reliability which is one of the attributes of dependability. Hence, first we give a brief overview of dependability. A software system may contain both hardware and software components, such a system may also be termed as Computer Based System (CBS). There are various types of CBS deployed in the safety critical and safety related applications that need to be designed as per safety guides of the related authority for deployment. The classification of CBS is also described herein. The concept of software reliability has been inherited from hardware reliability. Therefore comparative study and concepts of hardware and software reliability is presented. Next, studies on software reliability are summarized in subsections according to their relevance to different aspects of software reliability: Assessment of Software Reliability, Quality, and Project Management. Finally, the studies on early prediction of software reliability are summarized.

2.2 Dependability

The migration from analog to digital systems for instrumentation and control (I & C) has increased the complexity of the instrumentation. The I & C systems being developed are computer-based consisting of embedded digital hardware and software components.

These systems are performing many varying and highly complex functions that are integral to the safety-critical requirements of a safety critical system, and the failure of an I & C system could lead to risk significant events. In order to prevent such situations from arising, there is a great need for these systems to be dependable; that is, these systems must provide a specified quality of service. It is the system designer's responsibility for demonstrating that a given system is dependable (Carter, 1987). Hence, there is a definite need for dependability analysis to be performed during the design phase.

A systematic exposition of the concepts of dependability consists of three parts: the threats to, the attributes of, and the means by which dependability is attained, as shown in figure 2.1.

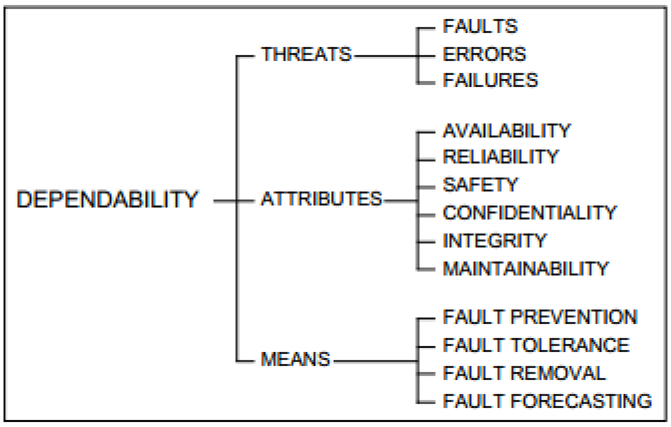

Figure 2.1: The dependability tree.

Computing systems are characterized by five fundamental properties: functionality, usability, performance, cost, and dependability. Dependability of a computing system is the ability to deliver service that can justifiably be trusted. The service delivered by a system is its behavior as it is perceived by its user(s); a user is another system (physical, human) that interacts with the former at the service interface. The function of a system is what the system is intended to do, and is described by the functional specification. Correct service is delivered when the service implements the system function. A system failure is an event that occurs when the delivered service deviates from correct service. A failure is thus a transition from correct service to incorrect service, i.e., to not implementing the system function. The delivery of incorrect service is a system outage. A transition from incorrect service to correct service is service restoration. Based on the definition of failure, an alternate definition of dependability, which complements the initial definition in providing a criterion for adjudicating whether the delivered service can be trusted or not: the ability of a system to avoid failures that are more frequent or more severe, and outage durations that are longer, than is acceptable to the user(s). In the opposite case, the system is no longer dependable: it suffers from a dependability failure, which is a meta-failure.

THE THREATS: FAULTS, ERRORS, AND FAILURES

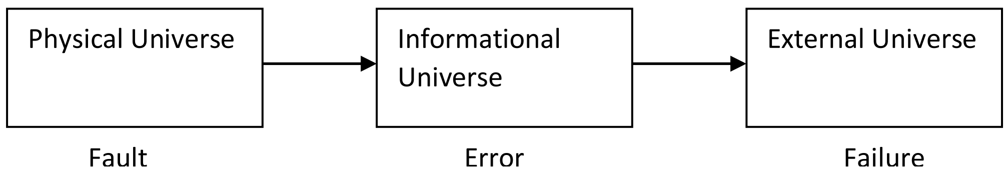

Figure 2.2: Three-universe model (Johnson, 1989; Laprie, 1985).

There is a cause and effect relationship among these terms and this relationship is depicted in the three-universe model, as shown in figure 2.2. In this model, a fault occurs in the physical universe. The fault itself is a physical defect, imperfection or flaw that occurs within some hardware or software component (Johnson, 1989). An example of a fault could be a broken ground connection in a printed circuit board or an infinite loop in a computer program. The manifestation of a fault is an error, which occurs in the informational universe. An error is any deviation from correctness or accuracy. It can be represented by parity errors or by typographical mistakes. Finally, the third universe is the external universe that contains failures. A failure is simply the non performance of some expected action. Hence, a failure is the result of both a fault and an error. An example of a failure would be a quarter toll machine accepting dimes as payment in full. It is the cause and effect relationship among the components of the three-universe model that implies two important parameters: fault latency and error latency. Fault latency is simply the amount of time between the occurrence of a fault and its resulting error. Similarly, error latency is the amount of time between the occurrence of an error and its resulting failure.

In explaining the three-universe model, the causes of faults are described as are their differences with errors and failures. In order to completely understand the characteristics of faults, which is required for accurate dependability modeling, additional fault attributes must be examined.

THE ATTRIBUTES OF DEPENDABILITY

Dependability is an integrative concept that encompasses the following basic attributes:

1. availability: readiness for correct service;
2. reliability: continuity of correct service;
3. safety: absence of catastrophic consequences on the user(s) and the environment;
4. confidentiality: absence of unauthorized disclosure of information;
5. integrity: absence of improper system state alterations;
6. maintainability: ability to undergo repairs and modifications.

THE MEANS TO ATTAIN DEPENDABILITY

The development of a dependable computing system calls for the combined utilization of a set of four techniques:

1. fault prevention: how to prevent the occurrence or introduction of faults;
2. fault tolerance: how to deliver correct service in the presence of faults;
3. fault removal: how to reduce the number or severity of faults;
4. fault forecasting: how to estimate the present number, the future incidence, and the likely consequences of faults.

Dependability analysis can be done on hardware and software or hybrid system. Among its attributes availability and reliability can be quantified. Availability is useful for the repairable systems, while reliability is useful; for non-repairable systems. Traditional reliability concepts (for hardware) can be used for software reliability analysis and prediction.

2.3 Classification of Computer Based Systems

CBS can be classified into following [33]

1. Bespoke Systems- These systems are generally designed for a specific application. The software is developed from scratch and runs on raw hardware, which is configured using required hardware modules. The likely off-the-shelf software component these systems might employ is the real-time kernel or executive whose implementation details may not be available due to commercial reasons.

2. Embedded Systems- These are function-modules with embedded software, which provide limited flexibility to select functional parameters. These modules may have simple communication and analog and/or digital Input/Output (I/O) interfaces (e.g. single loop controllers, smart sensors).

3. Programmable Controller based Systems- The programmable controllers are general purpose, programmable process control and information systems which are available off-the-shelf. The hardware and software of these systems are designed to be configurable for various types of applications.

4. Systems based on General Purpose Computers- These can be either stand-alone systems or networked configurations for performing control, operator information functions etc.

2.4 Hardware and Software Reliability

Reliability in the field of engineering is defined as the ability of a system or component to perform its required functions under stated conditions for a specified period of time. Hardware reliability is defined as the probability of failure free operation of hardware for a specified time in a specified environment. Similarly Software reliability is defined as the probability of failure free operation of hardware for a specified time in a specified environment. The probabilistic definitions of software and hardware reliabilities are identical and the theories of probability and statistics are also similar. However the concept of reliability for hardware and software is different because of the dissimilarity in their environmental conditions, their failure causes and failure consequences [34- 36].

Software Reliability is not a function of time - although researchers have come up with models relating the two. It does not show the same characteristics similar as hardware as shown in Bathtub curve figure 2.3. In this, infant mortality is first being removed by doing burn-in test at factory and during this time the failure rate will keep on decreasing and reaches to a stability point after which hardware goes under operation, which is known as useful life in which failure rate is constant. After some period of time, hardware starts failing due to wear and tear out, also termed as aging [36, 37].

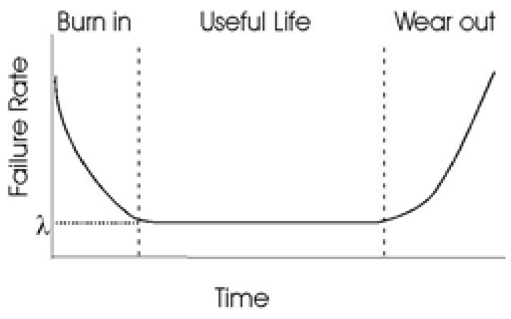

Figure 2.3: Bathtub curve for hardware reliability.

On the other side, the life of software is governed by different bathtub curve as shown in figure 2.4. In this the infant mortality time is similar to that of hardware i.e. failure rate keeps on decreasing while there is substantial difference in operation time onwards.

During the useful time, the software is under operational mode and keeps on undergoing faults, could be minor or major. Those faults are getting corrected and in this way the faults in the software keeps on vanishing. In this way, at one point of time, the failure rate becomes constant i.e. software is approaching Obsolescence; there are no motivation for any upgrades or changes to the software. Software never undergoes aging.

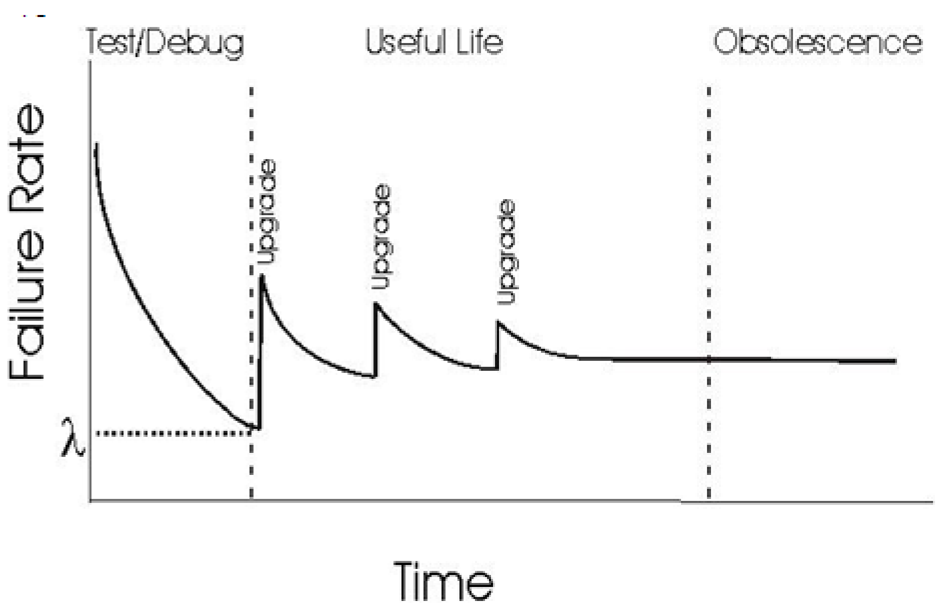

Figure 2.4: Bathtub curve for software reliability.

2.5 Fundamentals of Software Reliability

Software reliability theory deals with probabilistic methods applied to the analysis of random occurrence of failures in a given software system. Depending on the operating domain of the software, software reliability models have two classifications: (i) Data Domain: measures reliability as the ratio of successful runs to the total number of runs. (ii) Time Domain: models depends on time, whose main feature is that probability measures, such as the mean time between failures and the failure intensity function depend on failure time. We give a short review of some important terms, concepts, and notations which are frequently used in this dissertation.

2.5.1 Time to failure

Let a repairable system is observed until n failure times $t_1, t_2, \ldots, t_n$ occur, where $0 < t_1 < t_2 < \ldots < t_n$. Let $T > 0$ be the random variable representing the time to next failure. Then the time to failure denotes the probability that the time to failure T is in some interval $(t, t + \Delta t)$ as

$$P(t \leq T \leq t + \Delta t) \tag{2.1}$$

Given the cdf F(t) and pdf f(t),

$$P(t \leq T \leq t + \Delta t) = F(t + \Delta t) - F(t) \cong f(t)\Delta t \tag{2.2}$$

$$F(t) = P(0 \leq T \leq t) = \int_0^t f(x)dx \tag{2.3}$$

2.5.2 Reliability function

The reliability function is the probability of success at time t i.e. the probability that the time to failure exceeds t. Mathematically it is given by equation 2.4

$$R(t) = P(T > t) = \int_0^\infty f(x)dx \quad \text{(t>0)} \tag{2.4}$$

Therefore, cdf F(t) of T is given by equation 2.5

$$F(t) = \int_0^t f(x)dx = P(t \leq T \leq t) = 1 - R(t) \tag{2.5}$$

The reliability function is also known as survival function of T. R(t)decreases from 1 to 0, t=0 to t=∞. Hence f(t),F(t)and R(t) are equivalent representatives of the random variable T.

2.5.3 Hazard Rate

The hazard rate is defined as the limit of the failure rate as the interval Δt approaches zero. Mathematically it is given by equation 2.6.

$$\lim_{\Delta t \to 0} \frac{F(t+\Delta t) - F(t)}{(\Delta t)R(t)} = \frac{f(t)}{R(t)} \tag{2.6}$$

The hazard rate is an instantaneous rate of failure at time t, given that the system survives up to t. It is known as intensity function also.

Converting,

$$z = \frac{f(t)}{R(t)} = \frac{dF(t)}{dt}\frac{1}{R(t)}$$
$$\frac{dF(t)}{dt} = -\frac{R(t)}{dt}$$

$$\Rightarrow \frac{dR(t)}{dt} = -z(t)dt \tag{2.7}$$

Integrating both sides w.r.t. t,

$$lnR(t) = -\int_0^t z(x)dx + c \tag{2.8}$$

$$\because R(0) = 1, c = 0$$

$$\therefore R(t) = \exp[-\int_0^t z(x)dx] \tag{2.9}$$

Differentiating,

$$f(t) = z(t)\exp[-\int_0^t z(x)\mathrm{d}x] \tag{2.10}$$

Suppose, the time is measured in terms of number of discrete inputs, k. Let p be the probability of failure on a given test input that no prior failure has occurred on prior inputs. If all the failure domain inputs are independent, the reliability function is given by equation 2.11.

$$R(k) = (1-p)^k \tag{2.11}$$

Let t_e be the time required to execute one test case, then

$$t = kt_e$$

Now, assume that there is a finite limit for $\frac{p}{t_e}$ as t_e becomes vanishingly small, then failure rate λ is given by equation 2.12.

$$R(t) = \lim_{t_e \to 0} (1 - p(t_e))^{\frac{t}{t_e}} = \mathrm{e}^{-\lambda x} \tag{2.12}$$

which is an exponential distribution.

The time interval under which the system software will be used is important. In order to achieve a required outcome, it is essential that the system software involves functions properly with an extremely high reliability during a short time interval, which is usually shorter than other phases of software development. For software in commercial and industry applications, the time interval for which software is designed is supposed to be much longer.

There are three synthetic measures of reliability, namely:

1. Mean time to failure (MTTF)
2. Mean time between failures (MTBF)
3. Median of random variable T

MTTF is the average interval of time expected to the next failure time, as given in equation 2.13:

$$MTTF = E(t) = \int_0^\infty tf(t)\mathrm{d}t = \int_0^\infty R(t)\mathrm{d}t \tag{2.13}$$

In other words given the reliability function R(t), MTTF is thus a measure of the average time to failure for software system with life distribution F(T). MTBF is the expected interval length from the current failure time,$T_n = t_n$, to the next failure time $T_{n+1} = t_{n+1}$. Let $f(t|t_1, t_2, \ldots, t_n)$ denote the conditional distribution of failure time T_{n+1} given $T_1 = t_1, T_2 = t_2, \ldots, T_n = t_n$, then the MTBF is defined by equation 2.14:

$$MTBF = \int_{t_n}^{\infty} f(t|t_1, t_2, \ldots, t_n)\mathrm{d}t - t_n \tag{2.14}$$

The reciprocal of the intensity function $\frac{1}{z(t)}$ is used to represent the expected time to the next failure time, given that the nth failure time occurred at time t. That is we consider $\frac{1}{z(t)}$ as the MTBF. In general, this representation is not accurate as it could be significantly different from the MTBF. In fact, under special conditions, MTBF can be approximated by $\frac{1}{z(t)}$

$$M\hat{T}BF \approx \frac{1}{z(t)}$$

An alternative measure of reliability is the median of the random variable T. It is defined by equation 2.15:

$$F(\tilde{t}) = R(\tilde{t}) = \frac{1}{2} \tag{2.15}$$

The median is always well defined. However, there exists random variable T whose distribution does not have a finite MTTF value.

Consider an increasing intensity function z(t). This equation indicates that the chance of failure over a specified short interval of time becomes greater as long as time proceeds. This trend is good whenever wear-out phenomena affect the operation of the system, such as in hardware products. If the hazard function is a decreasing function, then it is suitable for determining durability or the expected life-span of the product due to improper design and manufacturing defects. In software systems, it is reasonable to assume that the hazard rate may change only when the program undergoes some modification such as fault removal or new code addition where no physical deterioration effect occurs. Since the failure intensity function z(t) depends only on the cumulative failure time t and not on the previous pattern of failure times, then we can assume that a failed system is in exactly the same condition after a repair as it was just before the failure.

2.6 Probability Distributions and Reliability Functions

We describe some of the popular distributions that are commonly used in Software Reliability.

2.6.1 Exponential Distribution

The exponential distribution becomes the most commonly applied distribution in life due to its important properties. It is the simplest model for failure times. Thus the one parameter exponential distribution is obtained by taking the intensity function to be constant $z(t) = \lambda > 0$, over the range of T. The exponential distribution with parameter λ denoted by $\exp(\lambda)$ is continuous, having a probability density function (p.d.f), shown in equation 2.16:

$$f(t) = \lambda e^{-\lambda t} \quad \lambda > 0 \tag{2.16}$$

$$\therefore F(t) = 1 - e^{-\lambda t} \Rightarrow R(t) = e^{-\lambda t} \tag{2.17}$$

The failure rate function of an exponential distribution with parameter λ is a constant denoted by equation 2.18:

$$z(t) = \lambda \tag{2.18}$$

Both the expected and the standard deviation of an exponentially distributed random variable T are equal to λ.

2.6.2 Weibull Distribution

It is a generalization of the exponential distribution. It is important because of the three reasons. (i) it is most commonly used for the distribution of lifetimes. (ii) it is related to the power law process and is used for repairable systems. (iii) if debugging the system, that is bringing it back to a new system, then the assumption that the times between failures $T_1, T_2, \ldots, T_n$ are independent identical distribution Weibull random variables may be reasonable. The Weibull distribution has the reliability function, given in equation 2.19

$$R(t) = \mathrm{e}^{-\lambda t^{\beta}} \tag{2.19}$$

If T is a random variable with this cdf, then X is distributed to Weibull(λ, β). The cdf, pdf, and the intensity function are given by equation 2.20, 2.21 and 2.22:

$$F(t) = 1 - R(t) = 1 - \mathrm{e}^{-\lambda t^{\beta}}, \quad t > 0 \tag{2.20}$$

$$f(t) = F'(t) = \lambda \beta t^{\beta-1} \mathrm{e}^{-\lambda t^{\beta}}, \quad t > 0 \tag{2.21}$$

$$z(t) = \frac{f(t)}{R(t)} = \frac{\lambda \beta t^{\beta-1} \mathrm{e}^{-\lambda t^{\beta}}}{\mathrm{e}^{-\lambda t^{\beta}}} = \lambda \beta t^{\beta-1}, \quad t > 0 \tag{2.22}$$

The probability density function (p.d.f) is a two parameter function. The parameters β and λ¸ are referred to as the shape and scale parameters, respectively.

The mean of the Weibull function can be expressed in terms of the gamma function.

2.6.3 Rayleigh Distribution

It is a special case of Weibull distribution. The pdf of Rayleigh distribution is given by equation 2.23:

$$f(t) = kt\mathrm{e}^{-\frac{kt^2}{2}} \tag{2.23}$$

The cdf is given by equation 2.24:

$$F(t) = -\mathrm{e}^{-\frac{kt^2}{2}} \tag{2.24}$$

$$\therefore R(t) = 1 - F(t) = 1 + \mathrm{e}^{-\frac{kt^2}{2}} \tag{2.25}$$

The intensity failure function is given by equation 2.26

$$z(t) = \frac{f(t)}{R(t)} = \frac{kt\mathrm{e}^{-\frac{kt^2}{2}}}{1 + \mathrm{e}^{-\frac{kt^2}{2}}} \tag{2.26}$$

$$M\hat{T}BF = \frac{1}{z(t)} = \frac{1 + e^{-\frac{kt^2}{2}}}{kte^{-\frac{kt^2}{2}}} = (kt)^{-1}e^{-\frac{kt^2}{2}} + (kt)^{-1}$$

It can be seen as a special case of Weibull distribution for $\beta = 2$ and $\lambda = (\frac{k}{2})^{\frac{1}{2}}$. In this case,

$$MTTF = (\frac{\Pi}{2k})^{\frac{1}{2}}$$

In 1980, Goel [3] criticized the realism of this model for pure software. The main problem of, is that it is accepted that the failure intensity function is independent of time, provided that the software code is not changed and the test environment is random. Therefore, the assumptions of a time-dependent failure rate are not theoretically justified. But, although this model has suffered from critique for its un-realism, it is still worth considering it in applications. An advantage of this model is that it can be useful in combined hardware and software system.

2.7 Classification of Software Reliability Models

The popular software reliability models may be classified as data-domain and time-domain models as shown in figure 2.5.

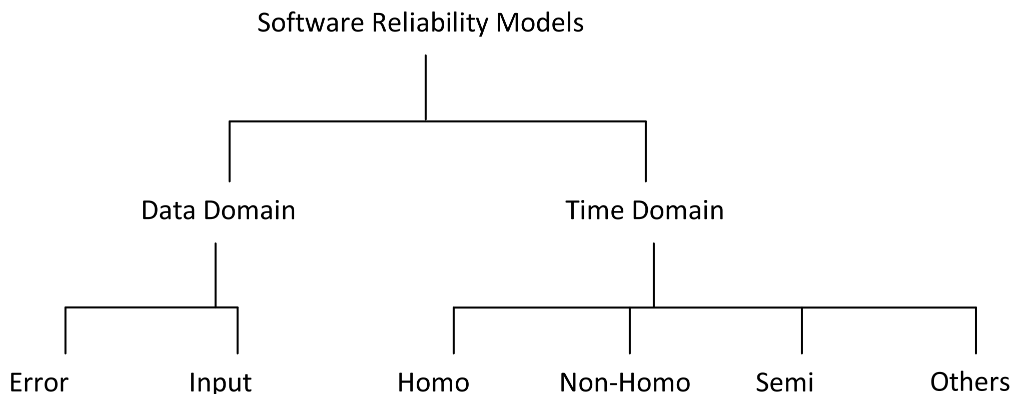

Figure 2.5: Classification of software reliability models.

2.7.1 Data-domain models

They are based on the philosophy that is the set of all input combinations to a computer program are identified, and then an estimate of its reliability can be obtained by exercising all the input combinations and observing the outcomes. In practice it is not feasible to identify the set of all input combinations and this approach is reduced to a method of selecting sample data sets representative of the expected operational usage for the purpose of estimating the residual number of faults in the software product under consideration. The data domain can be further classified as:

2.7.1.1 Fault-seeding models

Bebugging (or fault seeding) is a popular software engineering technique used in the 1970s to measure test coverage. Known bugs are randomly added to a program source code and the programmer is tasked to find them. The percentage of the known bugs not found gives an indication of the real bugs that remain.

The earliest application of bebugging was Harlan Mills's fault seeding approach [38] which was later refined by stratified fault-seeding [39] These techniques worked by adding a number of known faults to a software system for the purpose of monitoring the rate of detection and removal. This assumed that it is possible to estimate the number of remaining faults in a software system still to be detected by a particular test methodology. Mills's Hypergeometric Model is a popular model under this classification, which is described in Appendix A.

2.7.1.2 Input-domain models

In case of input-domain models, the reliability of the software is measured by exercising the software with a set of randomly chosen inputs. The ratio of the number of inputs that resulted in successful execution to the total number of inputs gives the estimate of the reliability of the software product. The model proposed by Nelson [40] and Bastani [32] belongs to this category. The basic approach taken here is to generate a set of test cases from an input distribution which is assumed to be representative of the operational usage of the program. Because of the difficulty in obtaining this distribution, the input domain is partitioned into a set of equivalence classes, each of which is usually associated with a program path. An estimate of program reliability is obtained from the failures observed during physical or symbolic execution of the test cases sampled from the input domain.

2.7.1.3 Time-domain models

They model the underlying failure process of the software under consideration, and use the observed failure history as a guideline, in order to estimate the residual number of faults in the software, and the test time required to detect them. The further classification of time-domain is given in Appendix A.

2.7.1.4 Bayesian software reliability growth models

In Bayesian models in the absence of data the parameters are not considered to be fixed at some unknown value, but they are assumed to follow a prior distribution. The models falls under this classification are explained in Appendix A.

2.7.1.5 Other models

These are the models, which do not fall into the category of fault seeding model and input domain model. The detail is given in the Appendix A.

2.8 An Overview of Software Reliability Early Prediction

This section includes an overview of the existing techniques. It also describes the similarities and differences among those approaches. It summarizes the limitations of those approaches. We also conclude the limitations of the current approaches and propose a roadmap to address them.

Researchers are continuously proposing approaches for prediction of software reliability in early phase of SDLC. These approaches are more or less similar in the sense that quantitative methods for reliability assessment depend on the availability of system usage information i.e. a system's operational profile. The operational profile information is combined with the non-probabilistic behavior models in order to obtain probabilistic models which can be used for reliability analysis. Error propagation can also take place so modeling approach to analyze the impact of error prorogation on reliability is also done [62]. So, these approaches have the common model as represented in figure 2.6.

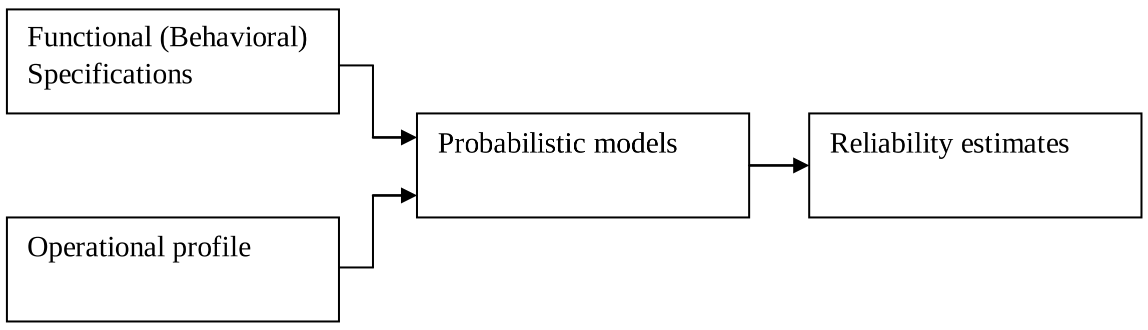

Figure 2.6: Probabilistic model for reliability analysis.

But the approach to generate Probabilistic Models and to estimate the software reliability differs from one approach to other approach [63]. Probabilistic behavior of a software system is often modeled and analyzed with discrete-time Markov chains. DTMC [41] consists of a set of states; each state has corresponding probabilities of transitioning to other states. DTMCs embody a basic way of modeling, and reasoning about probabilistic behavior; further demands, including the need to more faithfully represent software systems, enabled a proliferation of probabilistic automata formalisms [18]. As an example DTMC modeling of Robot has been shown in figure 2.7. Using DTMC modeling there are two main paradigms for modeling probabilistic behaviors: the generative, which can generate some actions and the reactive paradigm, which can react to the external actions. In this case calculating the system-wide reliability is equivalent to computing the steady state probabilities of DTMC. To illustrate this approach, consider DTMC model of an example Robot, which is depicted in figure 2.7.

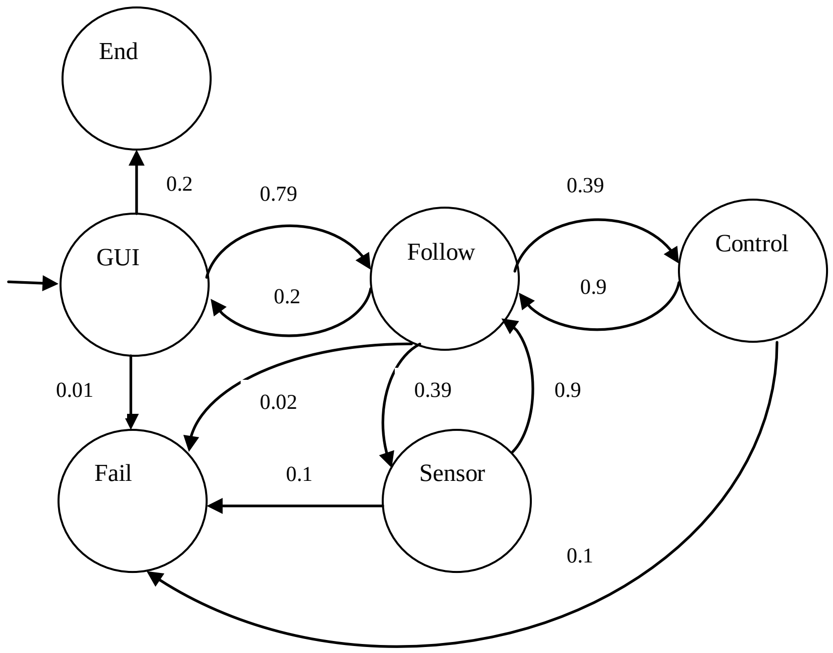

Figure 2.7: DTMC model of an example .

The architecture consists of Controller, Sensor, Follower, and GUI components. Several difficulties arise when deriving transition probabilities from an operational profile:

1. GUI can invoke different Controller's operations depending on the navigation model.

2. GUI and Follower may be concurrent to each other.

Some researchers also proposed Probabilistic model, based on the control flow graph as shown in figure 2.8, where $p_{1,2}$ is the probability of going from node 1 to node 2. The enumeration of paths could be conducted algorithmically, experimentally or by simulation. The reliability of each path is obtained as a product of the reliabilities of the components along path. For example application software shown in figure 3.3, possible execution path is 1, 3, 5, 6 and its reliability is obtained by R_1, R_3, R_5, R_6. The application reliability can be estimated by averaging path reliabilities. But the main drawback is that this approach doesn't give accurate reliability due to looping effects, like- in the path $1, 2, 4, 2^{1 \cdots *}, 6$. The subpath 2, 4, 2 can occur infinite number of times. The classification of architecture-based software reliability models can be understood by figure 2.9 [19].

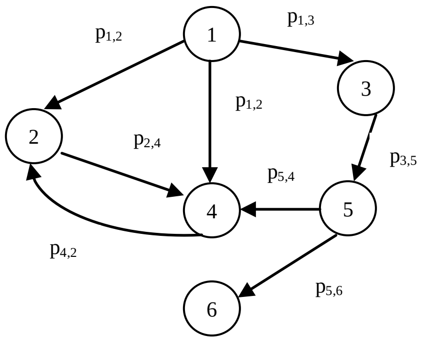

Figure 2.8: Probabilistic control flow graph of example application software.

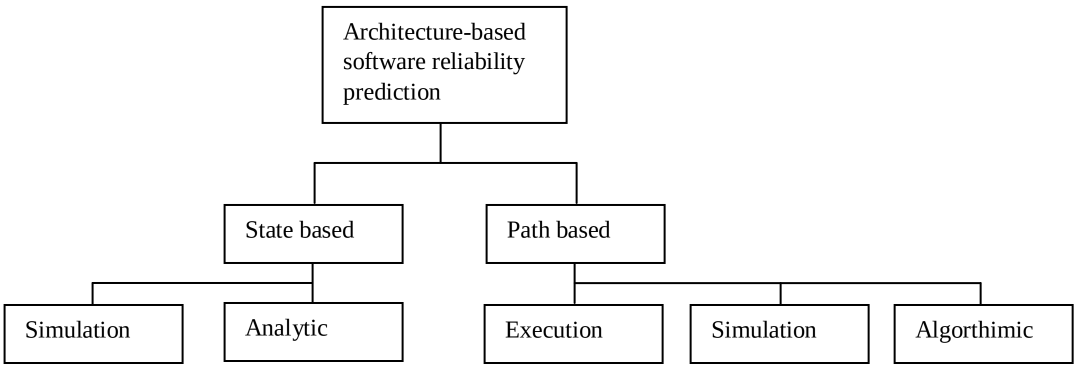

Figure 2.9: Classification of architecture-based software reliability models.

2.8.1 Limitations of existing approaches

In this section we discuss the limitations of the prevalent state-based architecture-based analysis techniques. The limitations of the existing approaches can be classified into–1) modeling, 2) analysis 3) parameter estimation 4)validation and 5) optimization. Usually modeling limitations are due to the assumptions we made to ensure model expansibility, which may lead to unreliable estimation. Analysis limitations are due to lacking of analysis techniques. Parameter Estimation limitations are due to non consideration of different software artifacts. Validation limitations are due to paying little effort. Optimization limitations are due to non consideration of complex interactions between components in the architectural design.

2.8.1.1 Modeling Limitations

1. A practice in which an engineer combines an operational profile and a non probabilistic specifications to directly produce an analysis-enabled generative model is tedious, non-intuitive and error prone.

2. The operational profile information that the existing approaches assume available is often just a subset of the available information.

3. Support for discovery and modeling of error states is not clear or accurate.

4. Existing DTMC based models assume that at a time the application can be in one state only which is not valid for today's complex systems.

5. Also the failure of one component can pass on its impact on other components as well which has not been taken care in anyone of the approaches.

6. None of the approach has taken into consideration, the nature of the interface between the components, as there is a possibility that components may be distributed with the advanced technologies like RMI, RPC, etc.

7. The architectural style of different components of the same application software may be different, for which we suspect to fit the common reliability approach on all the components. Also dynamically i.e. when application operates its architecture also changes dynamically.

2.8.1.2 Analysis Limitations

1. The reliability of the software system is a function of the reliabilities of each of its subsystems and their connectors. There should be a mechanism through which the impact of change of any of the component's or connector's reliability on the system reliability can be found to ensure the target reliability requirements of the system.

2. In some approaches Hidden Markov model has been used when it is difficult to have the surety of next probabilistic transition. But in this case the transition matrix and

observation probability matrix (which represents the probability of observing event in a particular state) has been initialized randomly, which may not be accurate.

2.8.1.3 Parameter Estimation Limitations

1. The reliability of the software system, based on Markov chain, is a function of transition probabilities in between the states of Markov chain. In the existing approaches these have been assumed analytically and hence the accuracy of the predicted values is not guaranteed.

2. The system-level model can be analyzed using traditional DTMC analysis [20], whose complexity is $O(n^3)$, where n is the number of states. Generally large complex software systems have thousands of states, which could be very expensive to solve the DTMC model.

2.8.1.4 Validation Limitations

The less effort has been paid to validate the predicted reliability, based on architectural design with the estimated reliability, just before product release to ensure the correctness of the predicted methodology so that, it can be applied to the future projects.

2.8.1.5 Optimization Limitations

Reliability prediction based on architecture can optimized if we success to optimize the software architecture. Sometimes it is noticed that architects design the software in a complex manner, full of tight coupling and low cohesion, which is a poor quality attributes of architecture [65]. For resource allocation optimizations refer [66].