Basic Concepts of Reliability

Reliability is a broad concept. It is applied whenever we expect something to behave in a certain way. Reliability is one of the metrics that are used to measure quality. It is a user-oriented quality factor relating to system operation. Intuitively, if the users of a system rarely experience failure, the system is considered to be more reliable than one that fails more often.

A system without faults is considered to be highly reliable. Constructing a correct system is a difficult task. Even an incorrect system may be considered to be reliable if the frequency of failure is "acceptable." The key concepts in discussing reliability are:

- Fault
- Failure
- Error
- Time

Failure

A failure is said to occur if the **observable** outcome of a **program execution** is different from the expected outcome. It is something dynamic. The program has to be executing for a failure to occur. The term failure relates to the behavior of the program. It includes such things as deficiency in performance attributes and excessive response time

Example: A failure may be cause by a defective block of code

Fault

The adjudged cause of failure is called a fault. A fault is the defect in the program that, when executed under particular conditions, causes a failure.

There can be different sets of conditions that cause failures, or the conditions can be repeated. Hence a fault can be source of more than one failure. A fault is a property of the program rather a property of its execution or behavior. It is what we are really referring to in general when we use the term "bug". A fault is created when a programmer makes an error

One fault can cause more than one failure depending upon how the system executes the faulty code. Depending whether fault will manifest itself as a failure; we have three types of faults:

a) Faults that are never executed so they don't trigger failures.
b) Faults that are executed but does not result in failures.
c) Faults that are executed and that result in failures.

Software reliability focuses solely on faults that have the potential to cause failures by detecting and removing faults that result in failures and implementing fault tolerance techniques to prevent faults from producing failures or mitigating the effects of the resulting failures.

Error

It can be defined as incorrect or missing human action that result in system/component containing a fault (i.e. incorrect system). Examples include omission of misinterpretation of user requirements in a product specification, incorrect translation, or omission of a requirement in the design specification.

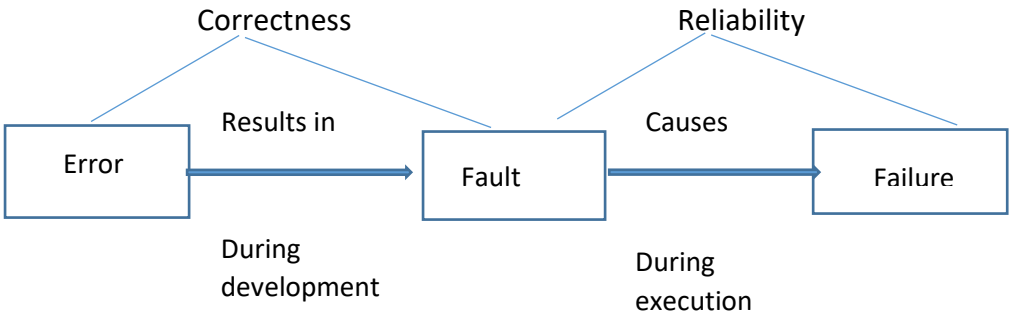

Figure: Relation of error, fault and failure

Time

Time is a key concept in the formulation of reliability. If the time gap between two successive failures is short, we say that the system is less reliable. Two forms of time are considered.

a) **Execution time (τ):**The execution time for a software system is the actual time spent by a processor in executing the instructions of the software system. Sometimes a processor may execute code from different programs, and therefore, their individual execution times must be considered independently

b) **Calendar time (t):** The calendar time is the time people normally experience in terms of years, months, weeks, days, etc. Calendar time is useful in order to express reliability suitable with the calendar time, because it offers to the managers and to persons that develop software the chance to see the date when the system attains the objectives of reliability.

   In other words, if you start a stopwatch when you start a program and check it when the program is done, you are measuring calendar time. This is equivalent to the "real" time

   If there are many things running on the machine at the same time, the Execution time could obviously be much less than calendar time.

   Also, if the program is sitting suspended while it waits for IO, the execution time might be low while calendar time might be high

Software Reliability

It is the probability that a system will operate without failure under *given environmental conditions* for a *specified period of time*.

It is expressed on a **scale from 0 to 1**:

- highly reliable system will have a reliability measure close to **1,** and

- unreliable system will have a measure close to **0**.

Reliability is measured **over execution time** so that it more accurately reflects system usage.

The goal is that reliability *must be quantified so that we can compare software systems*

- Examples
  - The probability that a PC in a store is up and running for eight hours without crash is 0.99.
  - An air traffic control system fails once in two years.

Factors Influencing Software Reliability

A user's perception of the reliability of a software depends upon two categories of information.

- The number of faults present in the software.
- The ways users operate the system.
  - This is known as the *operational profile*.

The fault count in a system is influenced by the following.

- Size and complexity of code
- Characteristics of the development process used
- Education, experience, and training of development personnel
- Operational environment

Hardware Reliability

It is the ability of **hardware** to perform its functions for some period of time. It is usually expressed as MTBF (mean time between failures).

Some of the important differences between software and hardware reliability can be listed in the following table:

Software Reliability	Hardware Reliability
Failures are primarily due to design faults. Repairs are made by modifying the design to make it robust against conditions that can trigger a failure	Failures are caused by deficiencies in design, production, and maintenance
There is no wear-out phenomena. Software errors occur without warning. “Old” code can exhibit an increasing failure rate as a function of errors induced while making upgrades.	Failures are due to wear-out or other energy-related phenomena. Sometimes a warning is available before a failure occurs
There is no equivalent to preventive maintenance for software.	Repairs can be made which would make the equipment more reliable through maintenance.
Reliability is not time dependent. Failures occur when the logic path that contains an error is executed. Reliability growth is observed as errors are detected and corrected.	Reliability is time related. Failure rates can be decreasing, constant, or increasing with respect to operating time.
External environment conditions do not affect software reliability. Internal environmental conditions, such as insufficient memory or inappropriate clock speeds do affect software reliability	Reliability is related to environmental conditions.
Reliability cannot be predicted form a knowledge of design, usage, and environmental stress factors	Reliability can be predicted in theory from physical bases.
Failure rates of software components are not predictable.	Failure rates of components are somewhat predictable according to known patterns
Software interfaces are conceptual	Hardware interfaces are visual
Software design does not use standard components	Hardware design uses standard components

System reliability

It is the probability that a system, including all hardware, firmware, and software, will satisfactorily perform the task for which it was designed or intended, for a specified time and in a specified environment.

Operational Profile

The notion of operational profiles, or usage profiles, was developed at AT&T Bell Laboratories and IBM independently.

An operational profile is a quantitative characterization of how a system will be operated by actual users. There are often different types of users for any relatively complex software system and these users may use system in different ways; this has the effect that different users of the same system might experience different levels of reliability. For one group of users, there could be operations that simply work very well and hence, they do not encounter many (or any) failures and as such, experiences a higher reliability. On the other hand, another group of users of the same system could encounter many more failures simply because they may use different operations of that system which contains many faults. This group then experiences a lower reliability. So it is necessary to test those operations which will be used most. It is operational profile which characterizes that actual system usage.

The operational profile acts as a guideline for testing, to make sure that when the testing is stopped, the critical operations are rigorously tested and thus, reliability has attained a desired goal. Using an operational profile allows us to quickly find the faults that impact the system reliability mostly. The notion of operational profiles was actually created to guide test engineers in selecting test cases, in making a decision concerning how much to test and what portions of a software system should receive more attention.

The operational profile of a system can be used throughout the life-cycle model of a software system as a guiding document in designing user interface by giving more importance to frequent used operations, in developing a version of a system for early release which contains the more frequently used operations.

Actual usage of the system is quantified by developing an operational profile and is therefore essential in any software reliability engineering process. For accurate measurement of the reliability of a system, we should test the system in the same way as it will be used by actual users. Ideally, we should strive to achieve 100% coverage by testing each operation at least once. Since software reliability is very much tied with the concept of failure, software with better reliability can be produced within a given amount of time by testing the more frequently used operations first. Use of the operational profile as a guide for system testing ensures that if testing is terminated, and the software is shipped because of imperative resource constraints, the most-used (or most critical) operations will have received the most testing and the reliability will be maximized for the given conditions/operations. It facilitates finding those faults early that have the biggest impact on reliability. Quality objectives and operational profile are employed to manage resources and to guide design, implementation, and testing of software.

Reliability Metrics

In software reliability engineering reliability metrics are used to quantitatively express the reliability of a software product. Besides being used for specifying and assessing software reliability, they are also used by many reliability models as a main parameter. Identifying, choosing and applying software reliability metrics is one of the crucial steps in measurement. Reliability metrics offer an excellent means of evaluating the performance of operational software and controlling changes to it. Following reliability metrics are used to quantify the reliability of software product:

a) MTTF (Mean Time To Failure)

The MTTF is the mean time for which a component is expected to be operational. MTTF is the average time between two successive failures, observed over a large number of failures.

It is important to note that only run time is considered in the time measurements, i.e. the time for which the system is down to fix the error, the boot time, etc are not taken into account in the time measurements. An MTTF of 500 means that one failure can be expected every 500 time units.

The time units are totally dependent on the system and it can even be specified in the number of transactions, as is the case of database query systems. MTTF is relevant for systems with long transactions, i.e., where system processing takes a long time. We expect MTTF to be longer than average transaction length.

b) MTTR (Mean Time To Repair)

MTTR is a factor expressing the mean active corrective maintenance time required to restore an item to an expected performance level. This includes activities like troubleshooting, dismantling, replacement, restoration, functional testing, but shall not include waiting times for resources. In software, MTTR (Mean time to Repair) measures the average time it takes to track the errors causing the failure and then to fix them. Informally it also measures the down time of a particular system

c) MTBF (Mean Time Between Failures)

MTTF and MTTR can be combined to get the MTBF metric:

MTBF = MTTF + MTTR

In this case, time measurements are real time and not the execution time as in MTTF. Thus, MTBF of 300 hours indicates that once a failure occurs, the next failure is expected after 300 hours

d) **Reliability** = MTBF / (1+MTBF)

e) Availability

- Measures the fraction of time system is really available for use
- Takes repair and restart times into account

Availability = MTBF / (MTBF+MTTR)

f) POFOD (Probability of Failure on Demand)

POFOD measures the likelihood of the system failing when a service request is made. Unlike the other metrics discussed, this metric does not explicitly involve time measurements. A POFOD of 0.005 means that five out of a thousand service requests may result in failure.

POFOD is an important measure and should be kept as low as possible. It is appropriate for systems demanding services at unpredictable or relatively long time intervals. Reliability for these systems would mean the likelihood the system will fail when a service request is made.

g) ROCOF (Rate of Occurrences of Failure)

ROCOF measures the frequency of occurrence of unexpected behaviour (i.e. failures). It is measured by observing the behaviour of a software product in operation over a specified time interval and then recording the total number of failures occurring during the interval. It is relevant for systems for which services are executed under regular demands and where the focus is on the correct delivery of service like operating systems and transaction processing systems. Reliability of such systems represents the frequency of occurrence with which unexpected behaviour is likely to occur. A ROCOF of 5/100 means that five failures are likely to occur in each 100 operational time units. This metric is sometimes called the failure intensity.

Markovian Models

The main function of most equipment and system depend more and more on software with the development of computer and information technology, but low reliability of software place a serious constraint on wide application of software, even lead to some disastrous result. Markov Model is used to represent the architecture of the software & provides a mean for analyzing the reliability of software **"A Markov Model is a mathematical system that undergoes transitions from one state to another, between a finite or countable number of possible states. It is a random process usually characterized as memory less: the next state depends only on the current state and not on the sequence of events that preceded it"**. By using Markov Model a statistical model of software is drawn wherein each possible use of the software has an associated probability of occurrence. Test cases are drawn from the sample population of possible uses according to the sample distribution and run against the software under test. Various statistics of interest, such as the estimated failure rate and mean time to failure of the software are computed.

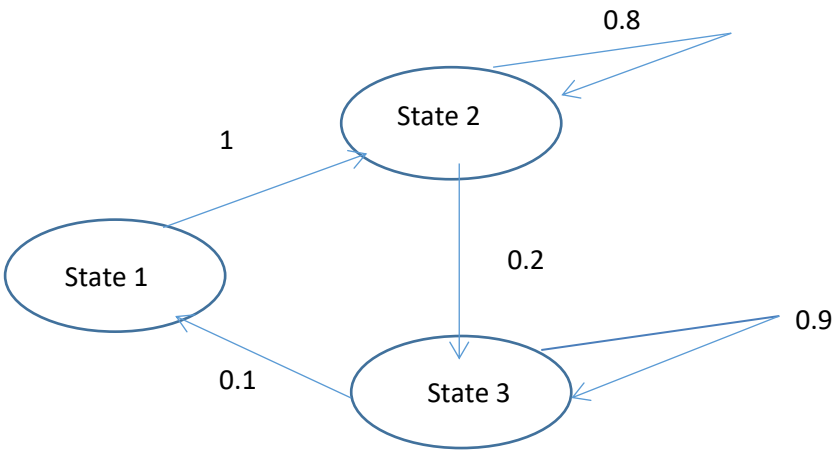

Figure: A Simple Three State Markov Model

A usage model for a software system consists of states, i.e., externally visible modes of operation that must be maintained in order to predict the application of all system inputs, and state transitions that are labelled with system inputs and transition probabilities. To determine the state set, one must consider each input and the information necessary to apply that input. It may be that certain software modes cause an input to become more or less probable (or even illegal). Such a mode represents a state or set of states in the usage chain. Once the states are identified, we establish a start state, a terminate state , and draw a state transition diagram by considering the effect of each input from each of the identified states. The Markov chain is completely defined when transition probabilities are established that represent the best estimate of real usage

Advantages of using Markov Model

- **Simplistic Modeling Approach**: Markov Models are simple to generate.
- **Redundancy Management Techniques**: System reconfiguration required by failures is easily incorporated in the model
- **Coverage**: Covered and uncovered failures of components are mutually exclusive events
- **Complex Systems**: Many simplifying techniques exit which allow the modeling of complex systems.
- **Sequenced Events**: It allow computing of an event resulting from a sequence of sub events. Disadvantage of Markov Model The major drawback of Markov Model is the explosion of number of states as the size of system increases. The resulting models are large & complicated.

Types of Markov Model:

The most common Markov models and their relationships are summarized in the following table:

	System state is fully observable	System state is partially observable
System is autonomous	Markov Chain	Hidden Markov Model
System is controlled	Markov Decision Process	Partially Observable Markov Decision Process

- **Markov Chain**: The simplest Markov model is the Markov chain. It models the state of a system with a random variable that changes through time. In this context, the Markov property suggests that the distribution for this variable depends only on the distribution of the previous state
- **Hidden Markov Model**: A hidden Markov model is a Markov chain for which the state is only partially observable. In other words, observations are related to the state of the system, but they are typically insufficient to precisely determine the state.
- **Markov Decision Process**: A Markov decision process is a Markov chain in which state transitions depend on the current state and an action vector that is applied to the system. Typically, a Markov decision process is used to compute a policy of actions that will maximize some utility with respect to expected rewards.
- **Partially Observable Markov Decision Process**: A partially observable Markov decision process (POMDP) is a Markov decision process in which the state of the system is only

partially observed. POMDPs are known to be NP complete, but recent approximation techniques have made them useful for a variety of applications, such as controlling simple agents or robots.

Binomial Type Model

In this type of model test data between failures is used instead of time as independent variable. Models based on the binomial distribution are finite failures models, that is, they postulate that a finite number of failures will be experienced in infinite time.

The failure process of software for a binomial type of model is characterized by the behavior of the rate of occurrence of failures during the execution of the software. The number of failures follows a binomial distribution of probabilities, this being the reason for its characterization as a binomial type model, according to Musa's classification.

Here, *time* is replaced as the control variable by the variable *number of test data* with the assumption that the execution of a test datum is equivalent to a unit of time of execution of the software, that is, the measurement unit of software testing is the test datum. This is necessary as reliability is a characteristic defined as a function of time. Thus, the failure rate of software is characterized by the number of failures by test datum or by test data set.

A number of test data executed by the software results in a measured coverage percentage of the software code. Moreover, the coverage percentage depends on the testing criterion used for evaluation of the test data. Hence, the failure rate of the software is also related to the coverage of the testing criterion achieved by execution of the test data.

In the initial stages of testing the failure rate is high and the test coverage is low as few test data have been executed; in the final stages of the testing, the failure rate is low and the test coverage is usually high.

Hence, the basic assumption of the Binomial model is that the failure rate of the software is directly proportional to the complement of the measured coverage achieved by execution of the test data.

The binomial model is based on the following assumptions:

I. The software is tested under conditions similar to those of its operational profile.
II. The probability of detection of any fault is determined by the same distribution.
III. Faults detected in each interval of test data are detected independently of each other.
IV. There are *N* faults in the software at the beginning of the test
V. The test data are executed and the coverage of the elements required by the selection criterion used in test data evaluation is calculated for each failure occurrence.