Metrics, Models and Measurements in Software Reliability

Sheikh Umar Farooq[*], SMK Quadri[**] and Nesar Ahmad[***]

[*] Research Scholar, P.G Department of Computer Sciences, University of Kashmir, Srinagar, India.
[**] Director, P.G Department of Computer Sciences, University of Kashmir, Srinagar, India.
[***] Department of Statistics and Computer Applications, T. M. Bhagalpur University, Bhagalpur, India.
* shiekh.umar.farooq@gmail.com, ** quadrismk@hotmail.com, *** nesar.bgp@gmail.com

Abstract—**Reliability is always important in all systems but sometimes it is more important than other quality attributes. Software reliability engineering approach is focused on comprehensive techniques for developing reliable software and for proper assessment and improvement of the reliability. Reliability metrics, models and measurements form an essential part of software reliability engineering process. We should apply appropriate metrics, models and measurement techniques in SRE to produce reliable software, as no metric or model can be used in all situations. So, we should have profound knowledge of metrics, models and measurement process before applying them in SRE. In this paper, we present an in-depth analysis of all metrics, models and measurements used in software reliability.**

I. INTRODUCTION

Software reliability is a key quality attribute as identified by quality models like ISO, McCall, and FURPS etc. Software reliability is beginning to be significantly considered in each new software development [1]. Reliability is a user-oriented view of software quality and is also most readily quantified and measured. Developing reliable software is one of the most difficult problems facing the software industry [2]. Schedule pressure, resource limitations, and unrealistic requirements can all negatively impact software reliability. To achieve reliability we should apply complete software reliability engineering practices which include appropriate assessment of reliability which can be accomplished by using proper metrics, measurements and models; as no metric or model can be used in all situations. In this paper we describe metrics, measurements and models and their use in assessment of software reliability. Section 2 describes reliability and its key facts. Section 3 describes key components of software reliability. Section 4 describes software reliability exclusively and also explains the need for software reliability. Section 5 describes software reliability measurement and reliability metrics. It also explains software reliability assessment in software development life cycle (SDLC). Section 6 presents the conclusions.

II. RELIABILITY

Reliability is the probability of a system or component to perform its required functions (output that agrees with specifications) without failure under stated conditions (operational profile) for a specified period of time. Informally reliability denotes a product's trustworthiness or dependability. Mathematically, reliability R(t) is the probability that a system will be successful in the interval from time 0 to time t [3]:

i.e. $\mathbf{R(t) = P(T > t), t \geq 0}$

Where T is a random variable denoting the time to failure or failure time. Unreliability F(t), a measure of failure, is defined as the probability that the system will fail by time t"

i.e. $\mathbf{F(t) = P(T \leq t), t \geq 0.}$

In other words, F (t) is the failure distribution function. The following relationship applies to reliability in general. The Reliability R (t) is related to failure probability F (t) by:

$\mathbf{R(t) = 1 - F(t).}$

This probability is, however, the object of interest mainly when the probability of failure-free survival of system/mission is of concern. Other measures are more appropriate in other situations, and include various reliability metrics like MTTF, ROCOF, and POFOD which are discussed later in the paper.

A. Criteria for Reliability

If the goal of reliability is the reduction of failures of a specified severity to an acceptable level of risk, then for system to be ready to deploy, after having been tested for total time (t_t), it must satisfy the following criteria:

Predicted remaining failures

$r(t_t) < rc$

Where, *rc is a specified critical value*, and

Predicted time to next failure

$TF\ (t_t) > t_m$

Where, *t_m is mission duration*

The total time (t_t) could represent a safe/unsafe criterion, or the time to remove all faults not considering their severity level. For systems that are tested and operated continuously like the shuttle, t_t, TF (tt), and t_m are measured in execution time [4]. By specifying criteria for reliability, the purpose is to reduce the risk of deploying the system to a required level but there is no assurance that the expected level will be attained.

III. KEY CONCEPTS IN RELIABILITY

It is imperative to define key elements of reliability before discussing finer details of software reliability. The key elements of the reliability are as follows:

1. *Probability of failure-free operation.*
2. *Duration of time of failure-free operation.*
3. *A given execution environment.*

As our work is actually focused on software reliability so key elements of software reliability will be defined mainly in the context of software reliability.

A. Errors, Faults and Failures.

The concept of failure is fundamental to reliability. A system with occasional failures is considered to be highly reliable than the one that fails more often. A failure can be defined as the inability of a system to perform its required functions within specified requirements [5]. Failures are either random (in hardware) or systematic (in hardware or software). The adjudged cause of a failure is called a fault. A fault can be defined as abnormal condition that may cause a reduction in, or loss of, the capability of a functional unit to perform a required function. It can also be defined as a requirements, design, or implementation flaw or deviation from a desired or intended state [6]. Faults are initiated through a developer making an error [1] which can 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 [7]. The relation between error, fault and failure is depicted in figure 1. Software failures are always design failures. Often the system continues to be available in spite of the fact that a failure has occurred. The mere presence of faults in a system does not cause failure, as all faults do not produce failures as depicted in figure 2. It is important to understand what causes these faults to become failures. The first and most important starting point for determining if a fault is to become a failure is related to source code coverage. If a fault is never encountered during execution, it will never execute and as such can never become a failure. Rather, one or more faulty portions of a system must execute for a fault to manifest itself as a system failure. 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:

1. *Faults that are never executed so they don't trigger failures.*
2. *Faults that are executed but does not result in failures.*
3. *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 [8].

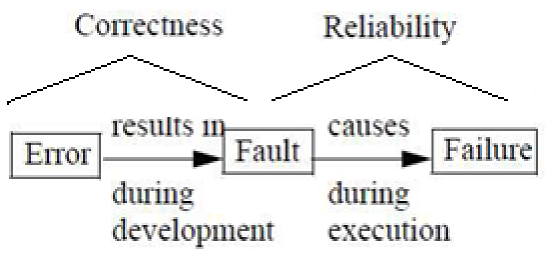

Figure 1: Relation of Error, Fault and Failure.

Any fault can potentially cause the failure of software. However, the probability for a fault manifesting itself as a failure is not uniform. Adams indicates that, on average, one third of all software faults manifest themselves as a failure [9]. Downtime is not evenly distributed either, as 90% of the downtime comes from less than 10% of the faults [1]. From this behavior, we can say that finding and removing a large number of faults does not necessarily yield a highly reliable product. Instead, it is important to focus on the faults that have the potential to cause immediate failures. A study has found that removing 60% of faults led to 3% reliability improvement only [10]. Two characteristics of failures are as follows:

1. *Failures are observable concepts.*
2. *Failures are associated with actual program executions.*

Software failures are usually systematic failures i.e. the mechanism whereby a fault reveals itself as a failure, and not to the failure process. If software failed once on a specific input it would always fail on that input until the underlying fault had been successfully removed. Failure processes are not deterministic for either 'systematic' faults or for random faults. The same probabilistic measures of reliability are appropriate in both cases (although the details of the probability models for evaluating software reliability generally differ from those used for hardware) [11].

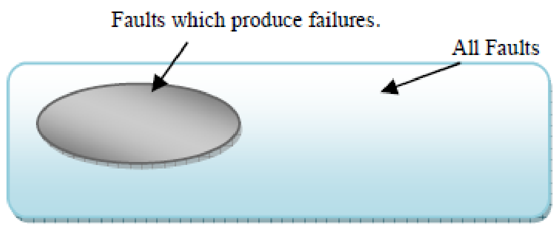

Figure 2: Relationship between Faults and Failures.

B. Time

The concept of time is important in modeling software reliability. Time units in reliability measurement must be carefully selected as time units are not the same for all types of systems. Three kinds of time units are relevant to software reliability:

1. *Execution time (τ)*
2. *Calendar time (t)*
3. *Clock time.*

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. The calendar time is the time people normally experience in terms of years, months, weeks, days, etc [11]. 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. The clock time is the elapsed time from start to end of execution of software by the processor. Clock time includes the wait time of the software system and execution times of other software systems. In measuring clock time, the periods during which the computer is shut down are not counted [11]. If execution time is not readily available, estimations such as clock time, weighted clock time, staff working time, or units that are natural to the application, such as number of transactions or number of test cases executed, may be used. If the fraction of time the processor is executing a program, is constant, clock time will be proportional to execution time. Nowadays execution time is used by most models to evaluate reliability. Execution time is shown to be superior to calendar time for software reliability growth models

[12].There is substantial evidence showing the superiority of execution time over calendar time for software reliability growth models [13][14][15]. [16] developed a modeling component that converts modeling results between execution time and calendar time. The execution time can be translated into calendar time if that is a more meaningful level for the analysis. Resource-limiting parameters such as testers, debuggers or computer time govern the relationship between calendar time and execution time during system test.

C. *Operational Profile*

The notion of operational profiles, or usage profiles, was developed at AT&T Bell Laboratories [17] and IBM [18, 19] independently. An operational profile is a quantitative characterization of how a system will be operated by actual users. It is a set of disjoint alternatives of system operations and their associated probabilities of occurrence [11]. 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. Software operating environments are extremely diverse and complex. To specify an operational profile, you must account for more than just the primary system user's [20]. The operating system and other applications competing for resources can cause an application to fail even under gentle use by a human. 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.

IV. Software Reliability

Software reliability is the probability of failure-free software operation over a specified time within a specified environment for a specified purpose [21]. The probability is a function of the inputs and use of the system as well as a function of the existence of the faults in the software. The inputs to the system determine whether existing faults, if any, are encountered. Alternatively, we can define reliability in terms of failure intensity. Failure intensity takes into account number of failures per unit time and gives us a measure of the reliability of a software system operating in a given environment. The lower the failure intensity of a software system, the higher is its reliability. Ideally, failure intensity decreases with time. But, in reality, failure intensity can be a random function of time as developers may introduce more faults while attempting to fix a known fault. Mathematically, using failure intensity we can estimate reliability as:

$R = \exp(-\lambda t * t)$

Where R = reliability

λt = Number of failures/hour

t = Time period for which the reliability is to be calculated

As an example, consider we observed 22 failures in 4 days (i.e.; 96 hours) period. The reliability for periods of usage of 24 hours in length is desired.

We have: $\lambda t = 22/96 = .2291$ and $t = 24$

Therefore: $R = \exp(-\lambda t * t) = \exp(-0.2291)(24) = 0.00409$

What this tells us is that in 100 periods of time that are each 24 hours in length, this software will run failure-free (for all users) in 0.40 of those 24 hour periods.

Software reliability is usually expressed as a continuous random variable due to the fact that most large software systems do have some unknown number of faults, and they can fail anytime depending upon their execution pattern. Therefore, it is useful to represent software reliability as a probabilistic quantity. The range of values specified for software reliability always lies from **0.000 to 1.000**.

We defined reliability either as the probability of failure free operation or in terms of the failure intensity. One is probabilistic in nature, whereas the other is an absolute one. The first description of software reliability lays emphasis on the importance of a software system running without failure for a certain minimum length of time to complete an operation. Here, reliability quantifies what fraction of the total number of operations a software system is able to complete successfully. The second description of reliability simply requires that there be as little failures as possible. Such an expectation takes the view that the risk of failure at any instant of time is of significant importance. Here it does not matter for how long the software system has been operational without failure. Rather, the very occurrence of a failure is of much significance [22]. We can apply both measures simultaneously to the same software system without any inconsistency between them. User expectations from different systems may be different, and therefore it is more useful to apply one of the two to a given system.

A. *Why Software Reliability?*

Reliability is a quantitative measure of the failure-related quality aspect of a software system. It is an important attribute of quality as defined by almost all quality models like ISO 9126, FURPS, and McCall etc. Reliability is important due to following reasons:

1. *Increased role of software*

Software moves from a secondary to a primary role in providing critical services. Systems are becoming more and more software intensive rather than hardware intensive. Unreliable software costs us both in terms of money, time and other resources etc. Software becomes the only way of performing some function whose failures would deeply affect individuals or groups. So there is a real need for improving software reliability to improve system dependability.

2. Evaluation of Software Engineering Technologies

The concept of software reliability can be used to evaluate a technology in terms of its effectiveness to produce higher quality software. Consider a case where two technologies T1 and T2 are used to build a system S1 and system S2 to implement the same application. By monitoring the reliability levels of the two systems S1 and S2 for the same application, we can observe which technology is more effective in producing software systems of higher reliability.

3. Measuring the Progress of System Testing

Software reliability can also be used to measure how much progress has been made in system-level testing [23] [24]. We can monitor the failure intensity of the software under test to know reliability level of software by comparing the current failure intensity with the desired failure intensity.

4. Controlling the System in Operation

Altering or introducing new code can induce new faults and that will decrease the reliability of the system, and it will require prolonged system testing to increase system reliability by fixing those faults. A project manager may put a limit on the amount of change in system reliability for each kind of maintenance activity. Therefore, the size of a maintenance work can be determined by the amount of system reliability that can be sacrificed for a while [22].

5. Better Insight into Software Development Process

The concept of reliability allows us to quantify the failure-related quality aspect of a software system. Quantification of the quality aspect of software systems gives developers and managers a better insight into the process of software development.

V. SOFTWARE RELIABILITY MEASUREMENT

Reliability measurement is a set of mathematical techniques that can be used to estimate and predict the reliability behavior of software during its development and operation [25]. Software reliability measurement is the application of statistical inference measures to failure data taken from testing and operation of software to assess its reliability. There are four key elements in software reliability measurement process as shown in figure 3:

1. *Reliability objective.*
2. *Operational profile.*
3. R*eliability modeling and measurement*
4. *Reliability validation.*

A. Defining Reliability Objective

Quality should be quantitatively defined by determining a reliability objective and by specifying balance among major quality factors and resources. The first important phase in software reliability measurement process is specifying reliability requirements. The reliability requirements should be defined clearly in quantitative manner. In order to be able to do this, some metrics are needed to quantitatively express the reliability of a software product [26]. The reliability requirements for different categories of software products may be different. For this reason, it is necessary that the level of reliability required for a software product should be specified in the SRS (software requirements specification) document explicitly. When we specify the reliability for a system we should identify the failure type and consider whether it should be treated differently in specification. Once the class of failure is identified, the reliability objective can be defined according to a suitable reliability metrics. There are different types of reliability metrics used in software reliability process. Commonly used software reliability metrics are explained in subsequent section. Software reliability requirements are specified during earlier development phases, and later modeling techniques are used to estimate the resources that will be required to achieve those requirements. The resource requirements are translated into testing schedules, and budgets. Resource estimates are compared to the resources actually available to make quantitative, rather than qualitative, statements concerning achievement of the reliability requirements. It is easy and cheap to specify and assess the reliability requirements of subsystem individually as compared to the whole system altogether. Reliability and reliability-related requirements can be expressed in one of the three following ways [11]:

1. *Probability of failure-free operation over a specified time interval.*
2. *Mean time to failure (MTTF).*
3. *Expected number of failures per unit time interval (failure intensity).*

Reliability related requirements must be stated in quantitative term. Otherwise, it will not be possible to determine whether the requirements have been met. Confidence bounds should be associated with reliability or reliability-related requirements. We should express reliability requirement as the minimum value of the confidence interval. This will allow the end users to know the probability of the software meeting its reliability requirement, and permit them to plan accordingly.

It is possible to apply software reliability measurement techniques to predict future reliability of software without specifying the reliability requirement. However, the existence of a reliability requirement helps us in following ways [11]:

1. A reliability requirement will serve as a goal to be achieved during the development effort. During the testing phases, software developers and managers can estimate software reliability and determine how close they are to the required value. The difference between current and required reliability can be converted into estimates of the time and resources that will be required to achieve the goal.
2. Specifying a reliability requirement helps the users and developers focus on the components of the system that will have the most effect on the system's overall reliability. Potentially unreliable components can be re specified or redesigned to increase their reliability.

A.1 Reliability Metrics

No field can really mature until it can be described in a quantitative fashion. 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:

1. 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. To measure MTTF, we can record the failure data for n failures. Let the failures occur at the time instants t_1, t_2… t_n. Then, MTTF can be calculated as $\sum_{i=1}^{n} \frac{t_{i+1}-t_i}{(n-1)}$. 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 [26]. 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.

2. 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 trouble-shooting, dismantling, replacement, restoration, functional testing, but shall not include waiting times for resources [27]. 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.

3. 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.

4. 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 [26]. 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.

5. ROCOF (Rate of Occurrences of Failure)

ROCOF measures the frequency of occurrence of unexpected behavior (i.e. failures). It is measured by observing the behavior of a software product in operation over a specified time interval and then recording the total number of failures occurring during the interval [26]. 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 behavior 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.

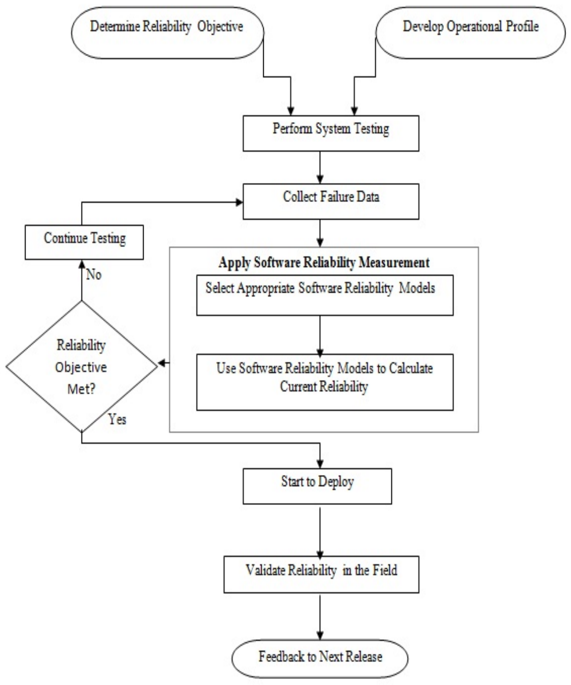
Determine Reliability Objective
Develop Operational Profile
Perform System Testing
Collect Failure Data
Continue Testing
No
Apply Software Reliability Measurement
Select Appropriate Software Reliability Models
Reliability Objective Met?
Use Software Reliability Models to Calculate Current Reliability
Yes
Start to Deploy
Validate Reliability in the Field
Feedback to Next Release

Figure 3: Software Reliability Measurement Process [29]

B. Define Operational Profile

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 [28]. 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.

C. Reliability Modeling and Measurement

Software reliability growth model is a mathematical model which depends on the use of statistical testing to measure the reliability of a system. Software reliability modeling techniques are used to predict a software system's failure behavior during test and operations. Moreover, reliability during testing is tracked to determine product release, using appropriate software reliability measurement models. This activity may be repeated until a certain reliability level has been achieved [29]. We have to measure reliability growth to discover reliability deficiencies through testing, analyzing such deficiencies, and implementation of corrective measures to lower the failure rate or to increase time between failures. Reliability growth measurement includes assessments of achievement and projecting the product reliability developments.

These reliability predictions and estimates can provide the following information:

1. The amount of (additional) test time and resources required to reach the product's reliability objective and to develop software worth release.
2. Identify elements in software systems that are leading candidates for redesign to improve reliability.
3. The reliability growth as a result of testing.
4. The reliability of the product at the end of system testing.
5. The predicted reliability beyond the system testing already performed.
6. Predict the occurrence of the next failure for a software system.
7. Determines the size and complexity of a software maintenance effort by predicting the software failure rate during the operational phase.
8. Assists in software safety certification.

Reliability modeling is an essential element of the reliability assessment process. It is required to use a reliability model to calculate, from failure data collected during system testing (such as failure report data and test time), various estimates of product reliability as a function of test time. Software reliability measurement includes two types of activities, reliability prediction and reliability estimation. So based on these activities we have two types of reliability models: *prediction based or estimation based*.

1. Software Reliability Prediction

Software reliability prediction is defined as the process of computing software reliability parameters from program characteristics (not failure data) [21]. Prediction activity employs different techniques in different development stages for reliability prediction. Prediction methods attempt to predict the future reliability of the software prior to the start of the test phase, using metrics and measures of software product and development process characteristics. Typically, software reliability prediction takes into account factors such as the size and complexity of a program. One of the major problems of software reliability prediction models is that they do not always predict the reliability accurately. The reason is that they assume limited historical data of special kind of organizations or of specific type of projects [7].

2. Software Reliability Estimation

Estimation determines current software reliability by applying statistical inference techniques to failure data obtained during system test or during system operation. This can be done after a program has executed long enough so that failure data are available. These methods tend to be used starting with subsystem integration and continuing through system integration, acceptance test, and operations [11]. This is a measure regarding the achieved reliability from the past until the current point. Its main purpose is to assess the current reliability, and determine whether a reliability model is a good fit in retrospect. Measurement of software reliability involves estimation of software reliability or its alternate quantities from failure data.

Measuring and projecting software reliability growth requires the use of an appropriate software reliability model that describes the variation of software reliability with time. After specifying the software reliability requirements and collecting failure data, we have to make selection of software reliability models. A software reliability model specifies the general form of the dependence of the failure process on the principal factors that affect it: fault introduction, fault removal, and the operational environment [30]. The sequence of failures is modeled as a stochastic process. Software reliability model specifies the failure behavior process. The model parameters are determined by fitting a curve to failure data. The parameters of the model can be obtained either from prediction performed during the period preceding system test, or from estimation performed during system test. This implies also a need for an inference procedure to fit the curve to data. The reliability model can then be used to predict or estimate the reliability. All software reliability models contain assumptions, factors & a mathematical function that relates the reliability with the factors. If the reliability improves over time, as faults are discovered and corrected, one would expect that the number of failures detected per unit of time would be decreasing and the time between failures would be increasing. It is this behavior that the software reliability models attempt to capture [31].

C.1 Model Classification

Software reliability growth models can be classified into two major classes, depending on the dependant variable of the model: models that use the time between successive failures (i.e., Type I models) and models that use the number of failures up to a given time (i.e., Type II models). Many of these models are either estimators i.e. they are used during software testing phase and are based on historical failure (fault) data. A few models are predictive i.e. they provide the reliability of software even before coding phase begins based on various program characteristics.

1. Time between Failure Models (Type I Models)

Different models exist where fault detection data is used in different ways to calculate an estimation of the statistical probability of the expected time until testers will detect the next error. The models that use the time between failures as a measure of progression in software testing projects are based on observing the mean time between failures over a specific time. The aim is to get an estimation of the expected time that will pass until the next fault will be found. The most common approach of this class of models is" to assume that the time between the $(i - 1)_{st}$ and the i_{th} failures follows a distribution whose parameters depend on the number of faults remaining in the program during this interval" [32]. This class of models often use exponentially growing/declining functions. Jelinski-Moranda model is the first time between failure model developed in 1972, where it is assumed that the times between failures are independently exponentially distributed. We can use different units of time to measure the mean time between failures depending on the requirement, size of the project and the testing rate although execution time is the best realistic time unit which can be used. In order to make the statistical estimations of time future faults, reliability growth models based on the time between failure metric often rely on a number of assumptions regarding the project characteristics. Following assumptions generally apply to most time between-failure models [32].

1. Independent times between failures.
2. Equal probability of the exposure of each fault.
3. Embedded faults are independent of each other.
4. Faults are removed immediately after each occurrence.
5. No new faults introduced during correction i.e. perfect fault removal.

These models are often criticized for being too rigid and unrealistic. There exist a number of attempts to modify and complement the early models in order to make them allow for more flexible assumptions [33].

2. Fault-Count Models (Type II Models)

In this class of models the aim is to predict when the accumulated number of faults starts to level off. The property of interest is the number of faults in a specific time interval. The unit of time can be measured in CPU execution time, calendar time or number of test cases. [34] proposed the incorporation of testing effort into the modeling process, which can be measured by the human power, the number of test cases. Fault-count models make three assumptions [32]:

1. Testing intervals are independent of each other.
2. Testing during intervals is reasonable homogeneous.
3. Numbers of defects detected during non-overlapping intervals are independent of each other.

Relative large populations of fault-count models are those based on Non-Homogeneous Poisson Processes (NHPP). These are models that assume that the cumulative number of failures detected at any time follows a Poisson distribution. NHPP is constrained by the following assumptions [35]:

1. All faults in a program are mutually independent from failure detection point of view.
2. The number of failures detected at any time is proportional to the number of faults in a program. This means that the probability of the failure for faults actually occurring i.e., detected, is constant.
3. The isolated faults are removed prior to future test occasions.
4. Each time a software failure occurs, the software error which caused it is immediately.

C.2 Model Selection

Many models have been proposed in past decades, each with its own assumptions, applicability, and limitations. No model is complete. One model may work well for certain conditions, but may be completely unusable for other conditions. We have to take assumptions into consideration that the models make about the development method and environment to determine how well they apply to the effort at hand. Many models assume defects are fixed immediately when they are discovered. Many models assume that each unit of time (calendar, clock or execution) is equivalent, failures are independent and tests represent operational profile. Many models also assume perfect debugging. If previous experience on similar projects indicates that most repairs do not result in new faults being inserted into the software, choose from those models making this assumption (e.g. Goel- Okumoto model, Muss-Okumoto model). However, if a significant number of repairs result in new faults being inserted into the software, it is more appropriate to choose from those models that do not assume perfect debugging (e.g. Littlewood-Verrall model). Many models assume that the number of errors in the software has an upper bound. If software testing at the subsystem level does not occur until the software is relatively mature, and if there is a low probability of making changes to the software actually being tested, models making this assumption can be included, e.g. Goel Okumoto model, Musa Basic model). If, on the other hand, significant changes are being made to the software at the same time it is being tested, it would be more appropriate to choose from those models that do not assume an upper bound to the number of faults (e.g. Muss-Okumoto and Littlewood-Verrall models) [11]. It is important to note that there is currently no known method of evaluating these assumptions to determine a priori which model will prove optimal for a particular development effort [36]. Selection of models will be a qualitative, subjective evaluation. Most of the assumptions made by models are not realistic as in practice the scenario is completely opposite as shown in Table 1.

There are some criteria by which software reliability models can be evaluated which are defined in [11]. Six model criteria identified by LYU are:

1. Model Measurement

Includes measurement accuracy for current failure intensity, prediction of time to finish testing with associated date and costs, and prediction of the operational profile.

2. Ease of measuring parameters

Includes cost, schedule impact for data collection, and physical significance of parameters to software development process.

3. Quality of assumptions

Includes closeness to the real world, and adaptability to a specific development environment.

4. Applicability

Includes ability to handle program evolution and change in test and operational environment.

5. Simplicity

They are simple in concept, data collection, program implementation, and validation.

6. Insensitivity to noise

Minimal response to insignificant changes in input data and parameters without losing responsiveness to significant-differences (e.g. change in test method, changing test scenarios).

A good Software reliability models will have the following characteristics:

1. Sound theoretical foundation.
2. Realistic assumptions.
3. Valid empirical support.

4. As simple as possible.
5. Trustworthy and accurate.
6. Provide information that is useful to the decision making process.
7. Applicable across many environments.

Assumptions	Practice
Software does not change and defects are fixed immediately	Software changes rapidly and certain defects are scheduled to be fixed in a later date
Testing as per Operational Profile	Focus is on functional testing. It is difficult to determine operational profile and perform operational tests
All failures are observable	Testing in controlled environment may vary from actual execution environment.
Constant test effort and independent failure interval	Varying test effort and failure intervals vary.
All aspects of failures are collected	Not all reports address failure.
Remaining failures are either constant or decreasing	Remaining failures may actually increase due to imperfect debugging

TABLE 1: RELIABILITY MODELS ASSUMPTIONS VS PRACTICE

C.3 Problems with Reliability Models

1. Lack of clear understanding of inherent strengths and weaknesses.
2. Basic assumptions in reliability models are not realistic.
3. Not all models applicable to all testing environments.
4. The reliability models assume a typical operational profile of software is available, which is often not true, and hence models may not produce good results.
5. Predictions of growth models are typically not accurate, while as uncertainties associated with growth models are high and there is no effective mechanism to assess the uncertainties.
6. The type and amount of data required by most models is not readily and fully available.
7. Difficulty in validating models which makes the, unsuitable for software with high reliability requirements.

C.4 Software Reliability Assessment in SDLC

The methods and needs of software reliability assessment and prediction vary by the phase of software development lifecycle [37] [38]:

1. At the requirements and design phases, early prediction models can be used. Reliability must be analyzed based the architecture and stated requirements.
2. At the implementation and testing phases, software reliability growth models are used. Software reliability assessment is needed to make the stopping decision concerning testing and debugging: when the mean time to failure is long enough, the software can be released.
3. When the software is released, it is ordinary to assume that all observed faults have been debugged and corrected. Thus, after release, a reliability model is used to predict the mean time to failure that can be expected. The resulting reliability estimate may be used in system reliability estimation, as a basis of maintenance recommendations, and further improvement, or a basis of the recommendation to discontinue the use of the software.

Thus, when the software is in operational use, the model to be used depends on maintenance policies and occurrence of failures:

1. If no failures are detected in the software, or if the software is not maintained, a reliability model is most appropriate.
2. If failures are detected and the software is updated, a reliability growth model should be used

D. Reliability Validation.

Finally, reliability can be analyzed in the field to validate the reliability engineering effort and to provide feedback for product and process improvements. Reliability validation involves exercising the program to assess whether or not it has reached the required level of reliability. This cannot normally be included as part of a normal defect testing process because data for defect testing is (usually) atypical of actual usage data.

VI. CONCLUSION

Developing reliable software requires the best knowledge of software reliability techniques. In this paper we presented an in-depth study of software reliability metrics, models and software reliability measurement. We observed that these three things are essential part of software reliability engineering process and can yield excellent results if right metrics and models are selected during software measurement process. The key to success with SRE is to acquaint ourselves well with details of basic elements of SRE i.e. software reliability metrics, models and measurement process before using them in practice.

REFERENCES

[1] W.W, Schilling, "Modeling the Reliability of Existing Software using Static Analysis", In the Proceedings of IEEE International Conference on Electro/information Technology, May 2006.

[2] A. Wood, “Predicting Software Reliability”, IEEE Computer, vol. 29, no. 11, pp. 69-77, 1996.

[3] H. Pham., ‘System Software Reliability”, Springer Series in Reliability Engineering, Springer, London, pp.149–149, 2006.

[4] N.F, Schneidewind, “Reliability and Maintainability of Requirements Changes.” Proc. of the International Conference on Software Maintenance, Florence, Italy, 7-9 Nov. 2001: 127-136.

[5] Sheikh Umar Farooq and S. M. K. Quadri, “3W’s of Static Software Testing Techniques", Global Journal of Computer Science and Technology (USA), Volume 11 Issue 6 Version 1.0, pp. 77-86, April, 2011.

[6] Nancy, Leveson,”Safeware: System Safety and Computers” Addison-Wesley Publishing Company. 1995.

[7] Deniz Kaya, "Software Reliability Assessment", www.eee.metu.edu.tr/~bilgen/DKaya-thesis.pdf, Accessed on 25-09-2011

[8] Mike Silverman and George de La Fuente, "Software Design for Reliability",http://www.opsalacarte.com/pdfs/Tech_Papers/Software_Design_for_Reliability_-_Paper.pdf, Accessed on 25-09-2011.

[9] E. N. Adams, “Optimizing preventive service of software products”, IBM journal of Research and Development” Vol. 28, no. 1, pp. 2–14, January 1984.

[10] Ian Sommerville. "Software Engineering", Pearson Education India, 2008.

[11] MR Lyu, "Handbook of Software Reliability Engineering." McGraw-Hill, New York, 1996.

[12] M.R. Lyu, "Software Reliability Theory, Encyclopedia of Software Engineering", Wiley, New York, 2002.

[13] M. Trachtenberg, "The Linear Software Reliability Model and Uniform Testing", IEEE Transactions on Reliability R–34(1), 1985.

[14] J. D. Musa and K. Okumoto, "Comparison of Time Domains for Software Reliability Models", Journal of Systems and Software 4(4), 1984.

[15] H. Hecht, "Allocation of Resources for Software Reliability", Proceedings of COMPCON Fall 1981 Washington, DC, 1981.

[16] J. D. Musa, A. Iannino, and K. Okumoto, "Software Reliability: Measurement, Prediction, Application", McGraw-Hill, New York, 1987.

[17] J. D. Musa.," Software Reliability Engineering", IEEE Software, pp. 14–32, March 1993,

[18] R. Cobb and H. D. Mills, "Engineering Software under Statistical Quality Control", IEEE Software, pp. 44–54., November 1990,

[19] H. D. Mills, M. Dyer, and R. C. Linger, "Cleanroom Software Engineering", IEEE Software, pp. 19–24, September 1987.

[20] J. Whittaker, J. and J. Voas, 'Toward a More Reliable Theory of Software", IEEE Computer, Volume: 33 Issue 12, 2000.

[21] J. D. Musa and K. Okumoto., "A logarithmic poisson execution time model for software reliability measurement.", In Proceedings of the 7th international conference on Software engineering (ICSE '84). IEEE Press, Piscataway, NJ, USA, 230-238. 1984.

[22] Kshirasagar Naik, Priyadarshi Tripathy, "Software testing and quality assurance: theory and practice", John Wiley & Sons, 2008.

[23] S. Dalal and C. Mallows, "When Should One Stop Testing Software." Journal of the American Statistical Associations, Vol. 81, 1988, pp. 872–879.

[24] M. C. K. Yang and A. Chao, "Reliability-Estimation and Stopping-Rules for Software Testing, Based on Repeated Appearances of Bugs", IEEE Transactions on Reliability, June 1995, pp. 315–321.

[25] Chin-Yu Huang and M R Lyu, "Estimation and Analysis of Some Generalized Multiple Change-Point Software Reliability Models", IEEE Transactions on Reliability, Vol 60, issue 2, pp 498 – 514,June-2011

[26] Rajib Mall, "Fundamentals of Software Engineering", PHI Learning Pvt. Ltd., 2004 - 356 pages

[27] Guidelines to Understanding Reliability Prediction", http://www.epsma.org/pdf/MTBF%20Report_24%20June%202005.pdf, Accessed on 3 Oct 2011.

[28] J.D Musa, "Operational profiles in software-reliability engineering", IEEE Software, Volume 10, Issue 2, pp 14-32, March 1993.

[29] Michael R. Lyu., "Design, testing, and evaluation techniques for software reliability engineering", In Proc. of the 24th Euromicro Conf. on Engineering Systems and Software for the Next Decade (Euromicro'98), Workshop on Dependable Computing Systems, pages xxxix–xlvi, Vasteras, Sweden, August 1998. IEEE Comp. Soc. Press. (Keynote speech).

[30] Hoang Pham, "Handbook of reliability engineering" Springer Birkhäuser, 2003, 603 pages

[31] John J. Marciniak, "Encyclopedia of software engineering", Volume 2, John Wiley, 2002, 1929 pages

[32] A.L. Goel, "Software Reliability Models: Assumptions, Limitations, and Applicability," IEEE Transactions on Software Engineering, vol. 11, no. 12, pp. 1411-1423, Dec. 1985, doi:10.1109/TSE.1985.232177

[33] M. Engelhardt and B. J. Lee, "On the mean time between failures for repairable systems," IEEE Transactions on Reliability, 1986

[34] C. -Y. Huang, S. -Y. Kuo, and M. R. Lyu, "A Framework for Modeling Software Reliability Considering Various Testing Efforts and Fault Detection Rates", IEEE Transactions on Reliability, 2001.

[35] A. Pham, "Software Reliability", Wiley-IEEE Computer Society Pr, 1995.

[36] A. A. Abdel-Ghaly., P. Y. Chan, and B. Littlewood, ""Evaluation of Competing Software Reliability Predictions", IEEE Transactions on Software Engineering; vol. SE-12, pp. 950-967; Sep. 1986.

[37] Ch. Ali Asad, Muhammad Irfan Ullah, and Muhammad Jaffar-Ur Rehman, "An Approach for Software Reliability Model Selection", In Proceedings of the 28th Annual International Computer Software and Applications Conference - Volume 01 (COMPSAC '04), Vol. 1. IEEE Computer Society, Washington, DC, USA, 534-539.

[38] R. Hamlet, "Testing software for software reliability", Technical Report, TR – 91 – 2, rev. 1, Department of Computer Science, Portland, OR, USA, March 1992.