A SURVEY OF SOFTWARE RELIABILITY: MODELING, MEASUREMENT AND IMPROVEMENTS

*Ejem, Agbaeze¹, Diala, S. O¹, Okpalla C. L¹

¹Department of Computer Science, Federal University of Technology Owerri, Nigeria.

Abstract

Software Reliability is the probability of failure-free software operation for a predefined timeframe in a predetermined situation. Software Reliability is an important factor affecting system reliability. It is a noteworthy feature of Software quality. It contrasts from hardware reliability in that it mirrors the design perfection while hardware reliability ponders on manufacturing perfection. The high multifaceted nature of programming is the major contributing variable of Software Reliability issues. This paper provides a survey of Software Reliability which can be classified into: modeling, measurement and improvements. It then examines different modeling techniques and metrics for software reliability, although, there is no single model that is global to all the situations. The paper will likewise give a conceivable methodology of enhancing software unwavering quality in the life cycle of software improvement.

Keywords: Survey, Software Reliability, Software Testing, Modeling Techniques.

1. INTRODUCTION

The appearance of desktop computers as well as the program running on them, are playing a major role in our everyday lives. Electronic gadgets such as Cars, washer, TVs, fuel pumps, microwave ovens and so on have replaced their analog and mechanical factors by using digital instruments. The computer enterprise and companies are booming exponentially. With an unendingly decreasing cost and elevated control, processors and software controlled system provide compact design, versatile handling, wealthy features and competitive cost. Like machinery replaced craftsmanship in the industrial revolution, computers and smart materials are speedily pushing their mechanical counterparts out of the market (Jiantao, 1999). Reliability is the primary dynamic characteristic of almost all software systems. Unreliable software leads to excessive cost for end-users. The Reliability of an application or software system is a measure of how good customers think it provides the services that they want. This paper tries to give a survey of software reliability which has been classified into: modeling, measurement or dimension and improvements.

2. DEFINITIONS

Software Reliability is defined as the probability of the failure free software operation for a specified period of time in a specified environment (ANSI, 1991) (Lyu, 1995). Electronic and mechanical equipment might become "old" and wear out with time and usage, but software will remain same throughout its life cycle. Software will not change over time unless intentionally changed or upgraded. Software Reliability is an important attribute of software quality, together with functionality, usability, performance, serviceability, capability, installability, maintainability, and documentation.

3. HARDWARE AND SOFTWARE RELIABILITY CURVE

Software reliability is a function of the number of failures encountered by a certain consumer of that software or application. A software failure happens...
when the software is executing. It is a state where the software does not supply the service(s) expected by the consumer or user. Software errors have claimed many lives. For example, if an ATM machine malfunctions and miscalculates your money typically if it did not remember your last withdrawal there is a high probability that you will be elated, however in airplanes, heart pace-makers, radiation therapy machines, a software error can simply claim person's lives. Over time, hardware exhibits the failure features shown in figure 1, referred to as the bathtub curve. Point A, B and C stands for burn-in segment, priceless life phase and end-of-life or wear out phase.

The environmental factors that makes hardware to wear down does not have any effect on software reliability. A better curve is shown in figure 2 when software reliability is projected on the same axes. According to reliability evaluation center (RAC),1996, there are two primary variations between hardware and software curves. One change is that in the last section, software does not have a growing failure rate as hardware does. On this phase, software is becoming out of date; there are not any motivation for any enhancements or changes to the program. As a consequence, the failure rate will not alternate. The second difference is that in the valuable-life phase, software might experience a drastic broaden in failure rate each time an improvement is made. The failure rate levels off regularly, partly due to the defects discovered and fixed after the improvement.
4. REVIEW OF SOFTWARE RELIABILITY MODELS

A number of software reliability models have emerged as people try to understand the attributes of how and why software fails, and try to quantify software reliability. At least 200 models have been proposed by software experts, but how to estimate software reliability still remains unsolved. Some of the models proposed are presented in Table 1.

Table 1: Software Reliability Model Classifications (Latha et al, 2012)

<table>
<thead>
<tr>
<th>S/N</th>
<th>Model Types</th>
<th>Model Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Time Between Failures</td>
<td>J-M De-Eutrophication, Schnick and Wolverton, Goel and Okumoto Imperfect Debugging, Littlewood-Verall Bayesian Models</td>
</tr>
<tr>
<td>3</td>
<td>Fault Seeding Models</td>
<td>Mills seeding model, Lipow model, Basin model.</td>
</tr>
<tr>
<td>4</td>
<td>Input Domain Models</td>
<td>Nelson Model, Ramamoorthy and Bastani Model.</td>
</tr>
<tr>
<td>5</td>
<td>Identical Error Behaviour Models</td>
<td>Shantikumar Model and Binomial Model.</td>
</tr>
<tr>
<td>6</td>
<td>Independently Distributed inter-failure models</td>
<td>JM Model, SchickWolertinModel, Moranda Model and Goel-Okumoto Model.</td>
</tr>
</tbody>
</table>

J-M De-Eutrophication

This model assumes that there are N software faults at the beginning of testing, each is independent of others which can cause failure during testing. A detected fault is deleted with certainty in a negligible time and no new faults are added during the debugging process. The software failure rate or hazard function during \( t_i \), the time between the \((i-1)st\) and \(i\)th failures, is given by

\[
Z(t_i) = \Theta [N - (i - 1)],
\]

Where \( \Theta \) is a proportionality constant. Remember that this hazard function is constant between failures but decreases in steps of \( \Theta \) following the deletion of each faults (Jelinski et al, 1972).

Musa Execution Time Model

This model assumptions are similar to that of JM model except that the process modelled is the number of failures in predefined execution time intervals. Its hazard function is given by
\[ Z(r) = \Delta f (N - n_e), \]

where \( r \) is the execution time utilized in executing the program up to the present, \( f \) is the linear execution frequency, \( \Delta \) is a proportionally constant, and \( n_e \) is the number of faults corrected during \((0, r)\). This model emphasizes the dependence of the hazard function on execution time (Musa, 1971).

**Nelson Model**

According to (Nelson, 1978), the reliability of the software is measured by running the software for a sample of \( n \) inputs. The \( n \) inputs are randomly selected from the input domain set \( E = \{E_i; i=1,\ldots, N\} \) where each \( E_i \) is the set of data values needed to make a run. If \( n_e \) is the number of inputs that resulted in execution failures, then an unbiased estimate of software reliability is \( RI = \{1 - (n_e/n)\} \) etc.

There's no single mannequin that can be utilized in the entire occasions. No model is complete; one mannequin may fit well for a suite of distinctive software but could also be totally off monitor for different kinds of problems. Most current analytical approaches to acquire reliability measures for application programs are founded on the Markovian model and they usually depend on the idea of exponential failure time distribution. The Markovian models are subject to the main issue of intractably large state space. Methods have been proposed to mannequin reliability development of components which cannot be accounted for by using the conventional analytical ways but they are also facing the state space explosion challenge. A simulation mannequin or model, alternatively offers an appealing substitute to analytical model as it describes a method being characterized in phrases of its artifacts, routine, interrelationships and interactions in such an approach that one may just perform experiments on the model, as an alternative than on the system itself, ideally with indistinguishable outcome (Aasia et al, 2010).

5. **SOFTWARE RELIABILITY METRICS**

Measurement is specified and exact in other engineering area however it is not definitely specified in software engineering. Although irritating, the hunt of quantifying software reliability has not ever ceased. Unless now, we still do not have any wonderful means of measuring application reliability. Measuring software reliability remains a difficult concern because we do not have a good way to understand the nature of software. There is not any clear definition to what facets are involving program reliability. We cannot discover asuitable solution to measure software reliability, and most of the features involving software reliability.

It is tempting to measure whatever concerning reliability to reflect the features, if we cannot measure reliability immediately. The present practices of software reliability measurement can also be divided into four classes (Reliability analysis center (RAC), 1996)

5.1 **Product Metrics**

Software size is refers to be a reflective of complexity, development effort and reliability. Lines Of Code (LOC), or Kilo Lines Of Code (KLOC), is an intuitive first strategy to measuring software size. However there's no longer a typical means of counting. On the whole, source code is used and comments and other non-executable statements will not be counted. This procedure cannot faithfully
comparative program not written within the identical language. For example, software built with Java and C# will have different lines of code. The appearance of recent technologies of code reuse and code iteration technique also cast doubt on this straightforward approach.

Function point metric is a system of measuring the performance of a proposed software progress established upon a count of inputs, outputs, master documents, inquiries, and interfaces. The method can be used to estimate the software system size as quickly as these functions can be identified. It is a measure of the realistic complexity of the application. It measures the performance dropped at the user and is unbiased of the programming language. It is used above all for business programs; it is not established in scientific or actual-time functions.

Complexity is directly related to software reliability, so representing complexity is principal. Complexity-oriented metrics is a system of selecting the complexity of a software's control structure, with the aid of simplifying the code into a graphical representation. Representative metric is McCabe's Complexity Metric.

Test coverage metrics are a way of estimating fault and reliability with the aid of performing tests on program products, based on the assumption that software reliability is a function of the section of application that has been effectively verified or established.

5.2 Project Management Metrics

Researchers have realized that just right administration can give rise to a better product. Research has confirmed that a relationship exists between the development procedure and the potential to complete products on time and within the preferred satisfactory objectives. Cost increase when developers use inadequate methods. Higher reliability may also be actualized by means of utilizing higher progress approach, risk administration procedure, configuration administration approach, and so on.

Process Metrics

Centered on the assumption that the quality of the product is an instantaneous performance of the process, process metrics can be utilized to estimate, reveal and enhance the reliability and high-quality of software. ISO-9000 certification, is the accepted reference for a household of standards developed by using the International Standards Institution (ISO).

Fault and Failure Metrics

The intention of collecting fault and failure metrics is to be capable to check when the software is drawing near failure-free execution. Minimally, both the number of faults discovered in the course of checking out (i.e., before delivery) and the screw ups (or other problems) reported by customers after delivery are collected, summarized and analyzed to acquire this intention. Test technique is extremely relative to the effectiveness of fault metrics, given that if the checking out situation does not duvet the entire functionality of the program, the application may pass all tests and still be inclined to failure as soon as delivered. Most commonly, failure metrics are based upon consumer knowledge concerning screw ups determined after therelease of the application. The
failure information gathered is used to calculate failure density, Mean Time Between Failures (MTBF) or screw ups or different parameters to measure or predict program reliability. The alternative of which metric must be used is determined by the type of procedure to which it applies and the requirements of the appliance area. For some programs, it may be proper to use unique reliability metrics for one-of-a-kind sub-programs. Some of the vital metrics which had been used for program or software reliability specification are shown in Table 2.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Explanation</th>
<th>Example System</th>
</tr>
</thead>
<tbody>
<tr>
<td>POFOD Probability of failure on demand</td>
<td>This is a measure of the likelihood that the system will fail when a service request is made. For example, a POFOD of 0.0001 means that 1 out of 1000 service request may result in error.</td>
<td>Safety critical and non-stop systems, such as hardware control systems</td>
</tr>
<tr>
<td>ROCOF Rate of failure occurrence</td>
<td>This is a measure of the frequency of occurrence with which unexpected behavior is likely to occur. For example, a ROCOF of 2/100 means that 2 failures are likely to occur in each 100 operational time units. This metric is sometime called failure intensity.</td>
<td>Operating systems. Transaction processing systems</td>
</tr>
<tr>
<td>MTTF mean time to failure</td>
<td>This is a measure of the time between observed system failures. For example an MTTF of 500 means that 1 failure can be expected every 500 time units. If the system is not being changed, it is the reciprocal of the ROCOF</td>
<td>Systems with long transactions such as CAD systems. The MTTF must be greater than the transaction time.</td>
</tr>
<tr>
<td>AVAIL Availability</td>
<td>This is a measure of how likely the system is to be available for use. For example, an availability of 0.998 means that in every 1000 time units, the system is likely to be available for 998 of these</td>
<td>Continuously running systems such as telephone switching systems</td>
</tr>
</tbody>
</table>

Table 2: Software Reliability metrics (Raghvendra, 2013)

In some instances, system users are most involved about how in general the system will fail, probably considering the fact that there's a big cost in restarting the system. In these instances, a metric established on a cost of failure occurrence (ROCOF) or the imply time to failure must be used.

In different instances, it is predominant that a system should always meet a request due to the fact there may be some cost in failing to give the service. The quantity of failures in a while period is less important. In these instances, a metric headquartering on the chance of failure on demand (POFOD) must be used. Sooner or later, users or system operators...
may be commonly worried that the process is on hand when a request for service is made. They will incur some loss if the approach is unavailable. Availability (AVAIL) takes into consideration the restore or restart time.

6. SOFTWARE RELIABILITY IMPROVEMENT TECHNIQUES

The development of software systems contain a sequence of production routine where possibilities for injection of human fallibilities are gigantic. Blunders may begin to arise at the very inception of the procedure where the objectives could also be erroneously or imperfectly exact. Due to human inability to participate in and keep up a correspondence with perfection, software development is accompanied by using a quality assurance undertaking. Just right engineering methods such as software testing or checking out, software validation and software verification can mostly strengthen software reliability.

6.1 Software Testing

Checking out or testing presents an exciting platform for reviewing the program developed with the aid of software engineers. For the duration of previous stage of application engineering pursuits, the engineer makes an attempt to build software from an abstract suggestion to a tangible product. The engineer creates a sequence of test instances which are supposed to 'damage' the application that has been developed. Correctly, trying out or testing is the one step in the application approach that could be seen as destructive rather than constructive. Software engineers by means of their nature are positive folks. Trying out or testing requires that the developer discard preconceived notions of the 'correctness' of software just developed and overcome a clash of interest that occurs when mistakes are uncovered. If checking out is conducted efficaciously it will find error in the application. Trying out demonstrates that software services appear to be working consistent with specification, that behavioral and efficiency standards appear to have been met.

6.1.2 Software Testing Principles

A software engineer needs to understand the following basic principles that guides software testing before applying methods to design effective software test cases:

(1) All tests should be traceable to customer requirements: As we have seen, the objective of software testing is to uncover errors. It follows that the most severe defects (from the customer's point of view) are those that cause the program to fail to meet its requirements.

(2) Tests should be planned long before testing begins: Test planning can begin as soon as the requirements model is complete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.

(3) The Pareto principle applies to software testing: Stated simply, the Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The probe of course, is to isolate these suspect components and to thoroughly test them.

(4) Testing should begin “in the small” and progress toward testing “in the large”: The first tests planned and executed generally focus on individual components. As testing progresses,
focus shifts in an attempt to find errors integrated clusters of components and ultimately in the entire system.

(5) **Exhaustive testing is not possible:** The number of path permutations even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.

(6) **To be most effective, testing should be conducted by an independent third party:** By most effective, we mean testing that has the highest probability of finding errors which is the primary objective of testing. The software engineer who created the system is not the best person to conduct all tests for the software.

### 6.1.3 Importance of Reliability Testing

The appliance of PC software has crossed into many special fields, with application being almost important part of industrial, commercial and military systems. Since its many functions in safeguard principal programs, software reliability is now a principal research subject. Despite the fact that application engineering is fitting the quickest constructing technology of the final century, there is not any complete, scientific, quantitative measure to assess them. Software reliability testing is being used as a device to aid examine these application engineering technologies. To toughen the efficiency of application product and software progress approach, an intensive evaluation of reliability is required. Trying out software reliability is important due to the fact it is of fine use for application managers and practitioners.

To verify the reliability of the software through testing:

1. An adequate number of test circumstances must be implemented for enough amount of time to get an inexpensive estimate of how long the application will execute without failure. Lengthy duration exams are wanted to identify defects that take time to rationale a fault or failure to arise.

2. The distribution of test instances will have to match the planned operational profile of the program. The extra probably a function or subset of the application is executed, the better the percentage of scan cases that will have to be allotted to that function or subset.

### 6.1.4 Types of Reliability Testing

Software reliability testing includes feature testing, load testing, and regression testing.

**Feature Test:** Feature testing assesses the features furnished via the software and is carried out in the following steps:

- Every operation within the application is performed once.
- Interplay between the 2 operations is decreased and
- Each and every operation is checked for its suitable execution.

**Load Testing:** This experiment is carried out to determine the efficiency of the program under maximum work load. Any program performs better up to a few amount of workload, after which the response time of the program begins degrading. For illustration, a website online can be confirmed to look how many simultaneous users it will possibly support.
without efficiency degradation. This testing commonly helps for Databases and application servers. Load checking out also requires application performance testing, which tests how good some program performs beneath workload.

**Regression Testing:** Regression testing is used to investigate if any new bugs were offered through earlier malicious program fixes. Regression trying out is performed after every exchange or replace in the program elements. This trying out is periodic, relying on the size and elements of the program.

6.2 Software Verification and Validation

Verification refers back to the set of events that make sure that application appropriately implements a certain operation. Validation refers to an extra set of routine that make sure that the application that has been built is traceable to customer standards. Boehm states this in a further way:

Verification: "Are we building the product proper?"

Validation: "Are we building the right product?"

6.2.1 Validation Testing

At the culmination of integration testing, software is completely assembled as a bundle, interfacing mistakes were uncovered and corrected, and a final sequence of application checks – validation trying out – may begin.

Validation succeeds when the application features in a fashion that may be moderately anticipated by way of the customer. At this factor a battle-hardened program developer might protest: Who or what's the arbiter of cheap expectations?

Cheap expectations are outlined within the application requisites Specification – a file that describes all person-noticeable attributes of the software.

6.2.2 Validation Test Criteria

Software validation is finished through a series of black field checks that exhibit conformity with requisites. A test plan outlines the classes of checks to be conducted and a test process defines unique experiment instances so as to be used to illustrate conformity with requirements. Both the plan and the process are designed to ensure that every functional requirements are satisfied, all efficiency requirements are executed, documentation is proper and human-engineered, and other specifications are met (e.g., transportability, compatibility, error recovery, maintainability).

After every validation experiment case has been carried out, probably the most two feasible stipulations exists:

1. The efficiency characteristics conform to specification and are accepted, or
2. A deviation from specification is uncovered and a deficiency record is created.

**CONCLUSIONS**

This paper provided a survey, an evaluation, pointing-out the challenges and methods to support software reliability. Just like hardware, software reliability relies on just right specifications, design and implementation. An excellent software progress plan is a key part of the software reliability program. Software reliability will also be categorized into three: Modelling, dimension and upgrades. Many models of software reliability exist but there's no single model that can be used in every circumstances. No mannequin or model is whole or even consultative in its entirety. One model may match
good for a set of distinctive application, however could also be wholly off monitor for different kinds of problems. Testing or Checking out is even more colossal for software than hardware. Even the best program progress procedure result in some application faults that are practically undetectable unless verified. Software is verified at several stages, beginning with individual items, by means of integration and whole approach testing in a similar method with hardware. Unlike hardware, it is not recommended to skip levels of program testing. In the course of all phases of checking out or testing, software faults are discovered, corrected, and re-tested.

REFERENCES


