ISSN (Online): 2230-7893
www.IJCEM.org

Software Reliability

Sushma Malik
Assistant Professor, FIMT, New Delhi
sushmalik25@gmail.com

Abstract

Unreliability of any product comes due to the failures or presence of faults in the system. As software does not "wear-out" or "age", as a mechanical or an electronic system does, the unreliability of software is primarily due to bugs or design faults in the software. Reliability is a probabilistic measure that assumes that the occurrence of failure of software is a random phenomenon. Randomness means that the failure can't be predicted accurately. Software reliability is the probability of the failure free operation of a computer program for a specified period of time in a specified environment. Software Reliability is dynamic in nature. It differs from the hardware reliability in that it reflects design perfection, rather than manufacturing perfection. Software reliability refers to the probability of failure-free operation of a system.

Keywords: *Software reliability growth models (SRGM), Software reliability engineering (SRE).*

1 Introduction

Software is a conceptual entity which is a set of computer programs, procedures, and associated documentation concerned with the operation of a data processing system. We can also say software refers to one or more computer programs and data held in the storage of the computer for some purposes. In other words software is a set of programs, procedures, algorithms and its documentation. Program software performs the function of the program it implements, either by directly providing instructions to the computer hardware or by serving as input to another piece of software. In contrast to hardware, software is intangible, meaning it "cannot be touched". Reliability is the ability of a person or system to perform and maintain its functions in routine circumstances, as well as hostile or unexpected circumstances. Reliability refers to the consistency of a measure. A test is considered reliable if we get the same result repeatedly.

Software Reliability is an important to attribute of software quality, together with functionality, usability, performance, serviceability, capability, install ability, maintainability, and documentation. Software Reliability is hard to achieve, because the complexity of software tends to be high. While any system with a high degree of complexity, including software, will be hard to reach a certain level of reliability, system developers tend to push complexity into the software layer, with the rapid growth of system size and ease of doing so by upgrading the software. The complexity of software is inversely related to software reliability; it is directly related to other important factors in software quality, especially functionality, capability, etc.

The Software failures may be due to errors, ambiguities, oversights or misinterpretation of the specification that the software is supposed to satisfy, carelessness or incompetence in writing code, inadequate testing, incorrect or unexpected usage of the software or other unforeseen problems. While it is tempting to draw an analogy between Software Reliability and Hardware Reliability, software and hardware have basic differences that make them different in failure mechanisms. Hardware faults are mostly ***physical faults***, while software faults are ***design faults,*** which are harder to visualize, classify, detect, and correct Design faults are closely related to fuzzy human factors and the design process, which we don't have a solid understanding. In hardware, design faults may also exist, but physical faults usually dominate. In software, we can hardly find a strict corresponding counterpart for "manufacturing" as hardware manufacturing process, if the simple action of uploading software modules into place does not count. Therefore, the quality of software will not change once it is uploaded into the storage and start running.

The Characteristics of software compared to hardware are

(i) Failure cause: Software defects are mainly design defects.

(ii) Wear-out: Software does not have energy related wear-out phase. Errors can occur without warning.

(iii) Repairable system concept: Periodic restarts can help fix software problems.

(iv) Time dependency and life cycle: Software reliability is not a function of operational time.

(v) Environmental factors: Do not affect Software reliability, except it might affect program inputs.

(vi) Reliability prediction: Software reliability cannot be predicted from any physical basis, since it depends completely on human factors in design.

(vii) Redundancy: Cannot improve Software reliability if identical software components are used.

(viii) Interfaces: Software interfaces are purely conceptual other than visual.

(ix) Failure rate motivators: Usually not predictable from analyses of separate statements.

(x) Built with standard components: Well-understood and extensively-tested standard parts will help improve maintainability and reliability. But in software industry, we have not observed this trend. Code reuse has been around for some time, but to a very limited extent. Strictly speaking there are no standard parts for software, except some standardized logic structures**.**

2. Software Reliability Models:

The **bathtub curve** is widely used in reliability engineering. It describes a particular form of the hazard function which comprises three parts:

a) The first part is a decreasing failure rate, known as early failures.

b) The second part is a constant failure rate, known as random failures.

c) The third part is an increasing failure rate, known as wear-out failures.

The name is derived from the cross-sectional shape of a bathtub. The bathtub curve is generated by mapping the rate of early "infant mortality" failures when first introduced, the rate of random failures with constant failure rate during its "useful life", and finally the rate of "wear out" failures as the product exceeds its design lifetime. In less technical terms, in the early life of a product adhering to the bathtub curve, the failure rate is high but rapidly decreasing as defective products are identified and discarded, and early sources of potential failure such as handling and installation error are surmounted. In the mid-life of a product—generally, once it reaches consumers—the failure rate is low and constant. In the late life of the product, the failure rate increases, as age and wear take their toll on the product. Many consumer products strongly reflect the bathtub curve, such as computer processors.

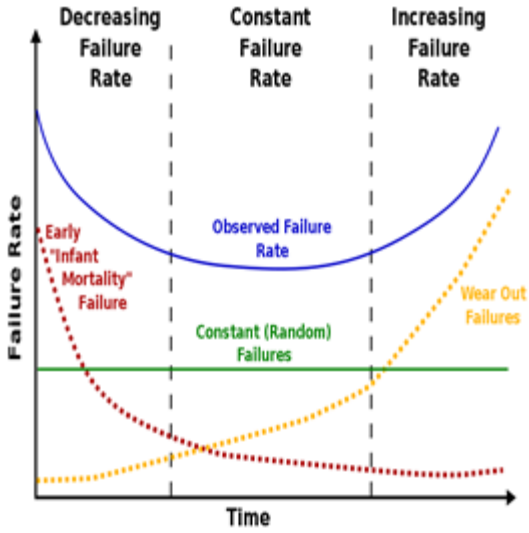

Figure 1: Bathtub curve for hardware reliability

Software reliability, however, does not show the same characteristics similar as Software reliability. There are two major differences between hardware and software curves. One difference is that in the last phase, software does not have an increasing failure rate as hardware does. In this phase, software is approaching obsolescence; there are no motivations for any upgrades or changes to the software. Therefore, the failure rate will not change. The second difference is that in the useful-life phase, software will experience a drastic increase in failure rate each time an upgrade is made. The failure rate levels off gradually, partly because of the defects found and fixed after the upgrades.

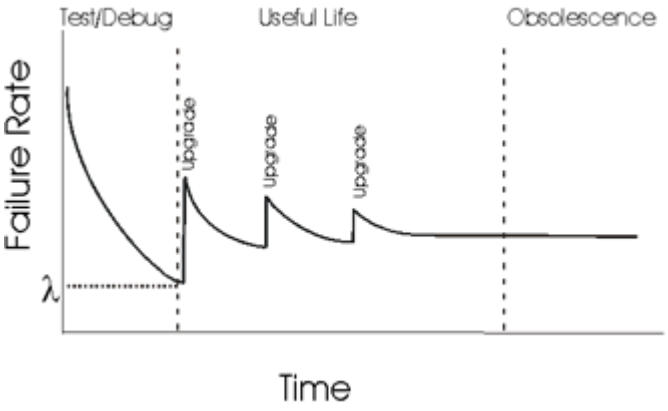

Figure 2:

The principal factors that affect the software reliability models are a) Fault introduction: It depends primarily on the characteristics of the developed code and development process characteristics, which include software engineering technologies and tools used and level of experience of personnel. b) Fault removal: It depends upon time, operational profile, and the quality of repair activity. c) The environment: It directly depends on the operational profile.

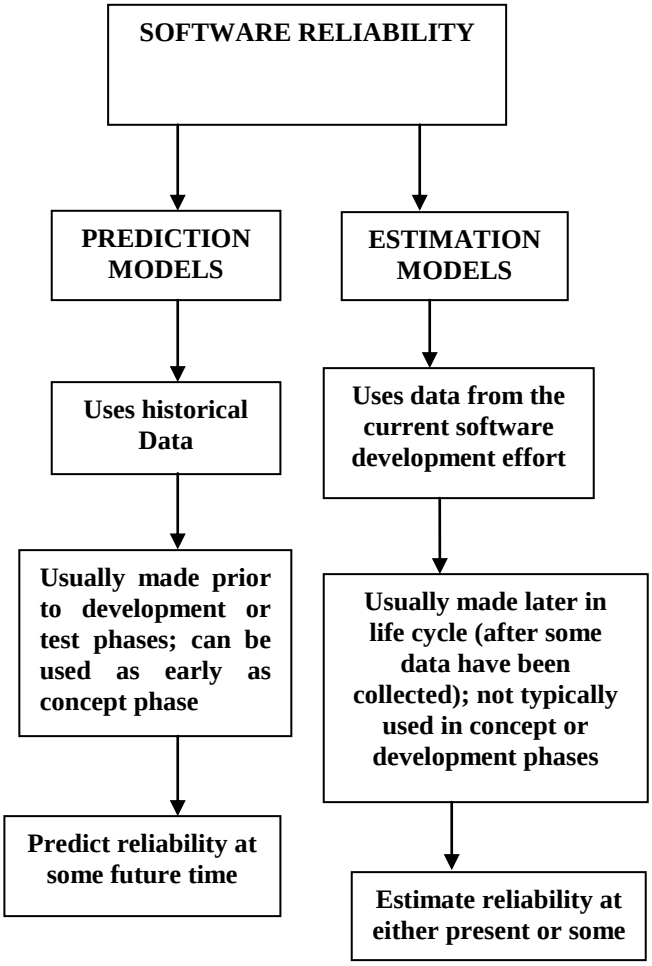

Figure 3: Software Reliability Models

Software reliability models usually make a number of common assumptions, as follows:
(1) The operation environment where the reliability is to be measured is the same as the testing environment in which the reliability model has been parameterized.
(2) Once a failure occurs, the fault which causes the failure is immediately removed.
(3)The fault removal process will not introduce new faults.
(4) The number of faults inherent in the software and the way these faults manifest themselves to cause failures follow, at least in a statistical sense, certain mathematical formulae. Since the number of faults (as well as the failure rate) of the software system reduces when the testing progresses, resulting in growth of reliability, these models are often called SRGM.

SRE is focused on engineering techniques for developing and maintaining software systems whose reliability can be quantitatively evaluated. In order to estimate as well as to predict the reliability of software systems, failure data need to be properly measured by various means during software development and operational phases. Software reliability engineering is centered on a key attribute, software reliability, which is defined as the probability of failure-free software operation for a specified period of time in a specified environment. Among other attributes of software quality such as functionality, usability, capability, and maintainability, etc., software reliability is generally accepted as the major factor in software quality since it quantifies software failures, which can make a powerful system inoperative. Software reliability engineering (SRE) is therefore defined as the quantitative study of the operational behavior of software-based systems with respect to user requirements concerning reliability. There are four major components in this SRE process are Reliability objective, Operational profile, Reliability modeling and measurement, and Reliability validation.

A reliability objective is the specification of the reliability goal of a product from the customer viewpoint. If a reliability objective has been specified by the customer, that reliability objective should be used. Otherwise, we can select the reliability measure which is the most intuitive and easily understood, and then determine the customer's "tolerance threshold" for system failures in terms of this reliability measure. The operational profile is a set of disjoint alternatives of system operational scenarios and their associated probabilities of occurrence. The construction of an operational profile encourages testers to select test cases according to the system's likely operational usage, which contributes to more accurate estimation of software reliability in the field. Reliability modeling is an essential element of the reliability estimation process. It determines whether a product meets its reliability objective and is ready for release. One or more reliability models are employed to calculate, from failure data collected during system testing, various estimates of a product's reliability as a function of test time. Several interdependent estimates can be obtained to make equivalent statements about a product's reliability.

These reliability estimates can provide the following information, which is useful for product quality management:

(1) The reliability of the product at the end of system testing.
(2) The amount of (additional) test time required to reach the product's reliability objective.
(3) The reliability growth as a result of testing (e.g., the ratio of the value of the failure intensity at the start of testing to the value at the end of testing).
(4) The predicted reliability beyond the system testing, such as the product's reliability in the field.

Modern software systems pose challenging research issues in the following stages are:
1. **Fault confinement:** This stage limits the spread of fault effects to one area of the system, thus preventing contamination of other areas. Fault-confinement can be achieved through use of self-checking acceptance tests, exception handling routines, consistency checking mechanisms, and multiple requests/confirmations. As the erroneous system behaviors due to software faults are typically unpredictable, reduction of dependencies is the key to successful confinement of software faults. This has been an open problem for software reliability engineering, and will remain a tough research challenge.
2. **Fault detection:** This stage recognizes that something unexpected has occurred in the system. Fault latency is the period of time between the occurrence of a software fault and its detection. The shorter it is, the better the system can recover. Techniques fall in two classes: off-line and on-line. Off-line techniques such as diagnostic programs can offer comprehensive fault detection, but the system cannot perform useful work while under test. On-line techniques, such as watchdog monitors or redundancy schemes, provide a real-time detection capability that is performed concurrently with useful work.
3. **Diagnosis:** This stage is necessary if the fault detection technique does not provide information about the failure location and/or properties. On-line, failure prevention diagnosis is the research trend. When the diagnosis indicates unhealthy conditions in the system (such as low available system resources), software rejuvenation can be performed to achieve in-time transient failure prevention.
4. **Reconfiguration:** This stage occurs when a fault is detected and a permanent failure is located. The system may reconfigure its components either to replace the failed component or to isolate it from the rest of the system. Successful reconfiguration requires robust and flexible software architecture and the associated reconfiguration schemes.
5. **Recovery:** This stage utilizes techniques to eliminate the effects of faults. Two basic recovery approaches are based on: fault masking, retry and rollback. Fault-masking techniques hide the effects of failures by allowing redundant, correct information to outweigh the incorrect information. To handle design (permanent) faults, N-version programming can be employed. Retry, on the other hand, attempts a second try at an operation and is based on the premise that many faults are transient in nature. A recovery blocks approach is engaged to recover from software design faults in this case. Rollback makes use of the system operation having been backed up (check pointed) to some point in its processing prior to fault detection and operation recommences from this point. Fault latency is important here because the rollback must go back far enough to avoid the effects of undetected errors that occurred before the detected error. The effectiveness of design diversity as represented by N-version programming and recovery blocks, however, continues to be actively debated.
6. **Restart:** This stage occurs after the recovery of undamaged information. Depending on the way the system is configured, hot restart, warm restart, or cold restart can be achieved. In hot restart, resumption of all operations from the point of fault detection can be attempted, and this is possible only if no damage has occurred. In warm restart, only some of the processes can be resumed without loss; while in cold restart, complete reload of the system is performed with no processes surviving.
7. **Repair:** In this stage, a failed component is replaced. Repair can be off-line or on-line. In off-line repair, if proper component isolation can be achieved, the system will continue as the failed component can be removed for operation. Otherwise, the system must be brought down to perform the repair, and so the system availability and reliability depends on how fast a fault can be located and removed. In on-line repair the component may be replaced immediately with a backup spare (in a procedure equivalent to reconfiguration) or operation may continue without the faulty component (for example, masking redundancy or graceful degradation). With on-line repair, system operation is not interrupted; however, achieving complete and seamless repair poses a major challenge to researchers.
8. **Reintegration:** In this stage the repaired module must be reintegrated into the system. For on-line repair, Reintegration must be performed without interrupting system operation. Design for reliability techniques can further be pursued in four different areas: fault avoidance, fault detection, masking redundancy, and dynamic redundancy. Non-redundant systems are fault intolerant and, to achieve reliability, generally use fault avoidance techniques. Redundant systems typically use fault detection, masking redundancy, and dynamic redundancy to automate one or more of the stages of fault handling. The main design consideration for software fault tolerance is cost-effectiveness. The resulting design has to be effective in providing better reliability, yet it should not introduce excessive cost, including performance penalty

and unwarranted complexity, which may eventually prove unworthy of the investigation.

CONCLUSION:

Software reliability is a key part in software quality. SW reliability is similar to HW reliability but must be treated differently. Software reliability engineering is focused on engineering techniques for developing and maintaining software systems whose reliability can be quantitatively evaluated.

REFRENCES:

Given articles references

http://www.ece.cmu.edu/~koopman/des_s99/sw_reliabilit/
http://en.wikipedia.org/wiki/Reliability_engineering
http://www.cs.utexas.edu/~EWD/transcriptions/EWD06xx/EWD627.html
http://www.emeraldinsight.com/journals.htm?articleid=1775831
http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=5373372
http://en.wikipedia.org/wiki/Bathtub_curve

Books

Agarwal.K.K and Singh, Yogesh (2005), Software Engineering

Jalote, Pankaj (2005), Software Engineering: A Precise Approach