# Software Reliability

Carnegie Mellon University
18-849b Dependable Embedded Systems
Spring 1999
Authors: Jiantao Pan
jpan@cmu.edu

---

## Abstract:

Software Reliability is the probability of failure-free software operation for a specified period of time in a specified environment. Software Reliability is also an important factor affecting system reliability. It differs from hardware reliability in that it reflects the design perfection, rather than manufacturing perfection. The high complexity of software is the major contributing factor of Software Reliability problems. Software Reliability is not a function of time - although researchers have come up with models relating the two. The modeling technique for Software Reliability is reaching its prosperity, but before using the technique, we must carefully select the appropriate model that can best suit our case. Measurement in software is still in its infancy. No good quantitative methods have been developed to represent Software Reliability without excessive limitations. Various approaches can be used to improve the reliability of software, however, it is hard to balance development time and budget with software reliability.

---

## Contents:



---

## Introduction: Embedded Software -- Embedded Disasters?

With the advent of the computer age, computers, as well as the software running on them, are playing a vital role in our daily lives. We may not have noticed, but appliances such as washing machines, telephones, TVs, and watches, are having their analog and mechanical parts replaced by CPUs and software. The computer industry is booming exponentially. With a continuously lowering cost and improved control, processors and software controlled systems offer compact design, flexible handling, rich features and competitive cost. Like machinery replaced craftsmanship in the industrial revolution, computers and intelligent parts are quickly pushing their mechanical counterparts out of the market.

People used to believe that "software never breaks". Intuitively, unlike mechanical parts such as bolts, levers, or electronic parts such as transistors, capacitor, software will stay "as is" unless there are problems in hardware that changes the storage content or data path. Software does not age, rust, wear-out, deform or crack. There is no environmental constraint for software to operate as long as the hardware processor it runs on can operate. Furthermore, software has no shape, color, material, mass. It can not be seen or touched, but it has a physical existence and is crucial to system functionality.

Without being proven to be wrong, optimistic people would think that once after the software can run correctly, it will be correct forever. A series of tragedies and chaos caused by software proves this to be wrong. These events will always have their place in history.

Tragedies in Therac 25 [Therac 25], a computer-controlled radiation-therapy machine in the year 1986, caused by the software not being able to detect a race condition, alerts us that it is dangerous to abandon our old but well-understood mechanical safety control and surrender our lives completely to software controlled safety mechanism.

Software can make decisions, but can just as unreliable as human beings. The British destroyer Sheffield was sunk because the radar system identified an incoming missile as "friendly". [Sheffield] The defense system has matured to the point that it will not mistaken the rising moon for incoming missiles, but gas-field fire, descending space junk, etc, were also examples that can be misidentified as incoming missiles by the defense system. [Neumann95]

Software can also have small unnoticeable errors or drifts that can culminate into a disaster. On February 25, 1991, during the Golf War, the chopping error that missed 0.000000095 second in precision in every 10th of a second, accumulating for 100 hours, made the Patriot missile fail to intercept a scud missile. 28 lives were lost. [Patriot]

Fixing problems may not necessarily make the software more reliable. On the contrary, new serious problems may arise. In 1991, after changing three lines of code in a signaling program which contains millions lines of code, the local telephone systems in California and along the Eastern seaboard came to a stop. [Telephone outage]

Once perfectly working software may also break if the running environment changes. After the success of Ariane 4 rocket, the maiden flight of Ariane 5 ended up in flames while design defects in the control software were unveiled by faster horizontal drifting speed of the new rocket. [Ariane 5]

There are much more scary stories to tell. This makes us wondering whether software is reliable at all, whether we should use software in safety-critical embedded applications. You can hardly ruin your clothes if the embedded software in your washing machine issues erroneous commands; and 50% of the chances you will be happy if the ATM machine miscalculates your money; but in airplanes, heart pace-makers, radiation therapy machines, a software error can easily claim people's lives. With processors and software permeating safety critical embedded world, the reliability of software is simply a matter of life and death. Are we embedding potential disasters while we embed software into systems?

---

# Key Concepts

## Definition

According to ANSI, Software Reliability is defined as: the probability of failure-free software operation for a specified period of time in a specified environment. [ANSI91][Lyu95]Although Software Reliability is defined as a probabilistic function, and comes with the notion of time, we must note that, different from traditional Hardware Reliability, Software Reliability is not a direct function of time. Electronic and mechanical parts may become "old" and wear out with time and usage, but software will not rust or wear-out during its life cycle. Software will not change over time unless intentionally changed or upgraded.

Software Reliability is an important to attribute of software quality, together with functionality, usability, performance, serviceability, capability, installability, 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. For example, large next-generation aircraft will have over one million source lines of software on-board; next-generation air traffic control systems will contain between one and two million lines; the upcoming international Space Station will have over two million lines on-board and over ten million lines of ground support software; several major life-critical defense systems will have over five million source lines of software. [Rook90] While 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. Emphasizing these features will tend to add more complexity to software.

## Software failure mechanisms

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.[Keiller91] 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. [Lyu95] 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. Trying to achieve higher reliability by simply duplicating the same software modules will not work, because design faults can not be masked off by voting.

A partial list of the distinct characteristics of software compared to hardware is listed below [Keene94]:

- **Failure cause**: Software defects are mainly design defects.
- **Wear-out**: Software does not have energy related wear-out phase. Errors can occur without warning.
- **Repairable system concept**: Periodic restarts can help fix software problems.
- **Time dependency and life cycle**: Software reliability is not a function of operational time.

- **Environmental factors**: Do not affect Software reliability, except it might affect program inputs.
- **Reliability prediction**: Software reliability can not be predicted from any physical basis, since it depends completely on human factors in design.
- **Redundancy**: Can not improve Software reliability if identical software components are used.
- **Interfaces**: Software interfaces are purely conceptual other than visual.
- **Failure rate motivators**: Usually not predictable from analyses of separate statements.
- **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.

## The bathtub curve for Software Reliability

Over time, hardware exhibits the failure characteristics shown in Figure 1, known as the bathtub curve. Period A, B and C stands for burn-in phase, useful life phase and end-of-life phase. A detailed discussion about the curve can be found in the topic Traditional Reliability.

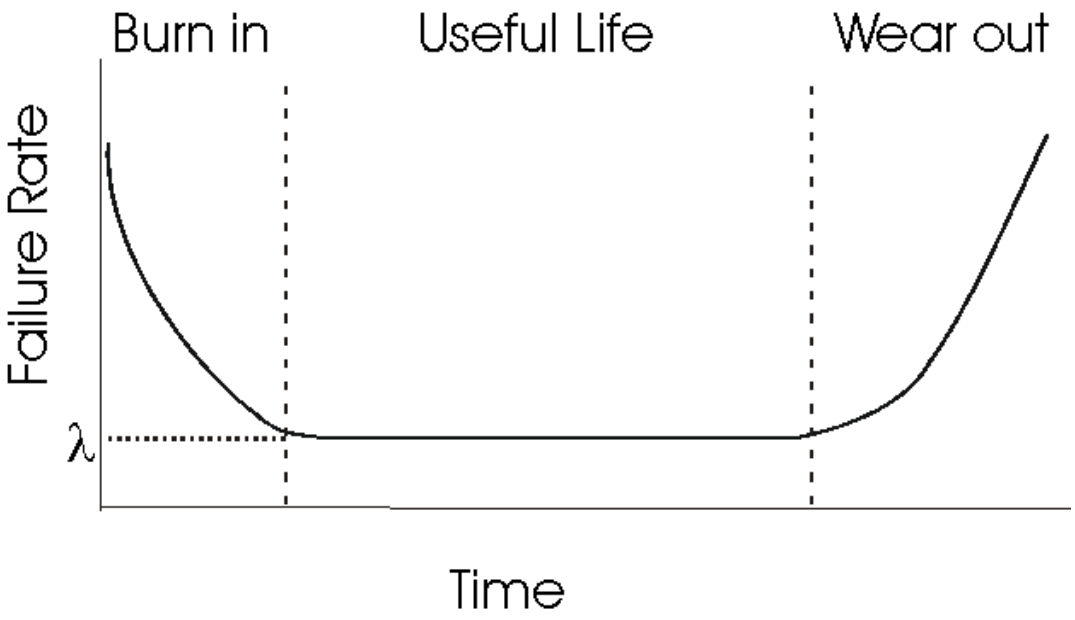



**Figure 1. Bathtub curve for hardware reliability**

Software reliability, however, does not show the same characteristics similar as hardware. A possible curve is shown in Figure 2 if we projected software reliability on the same axes. [RAC96] 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 motivation 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.

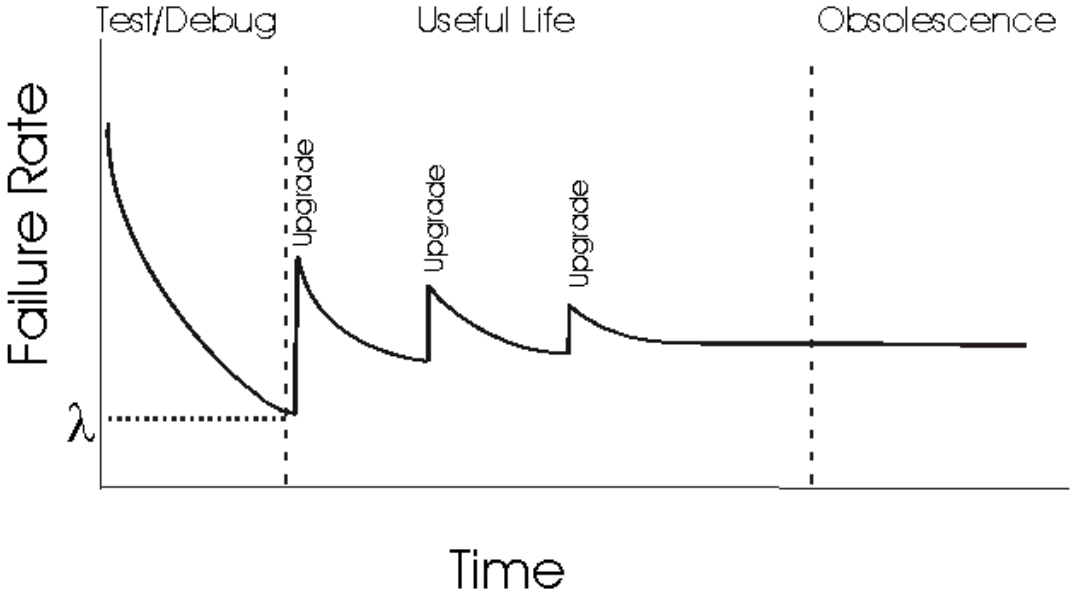
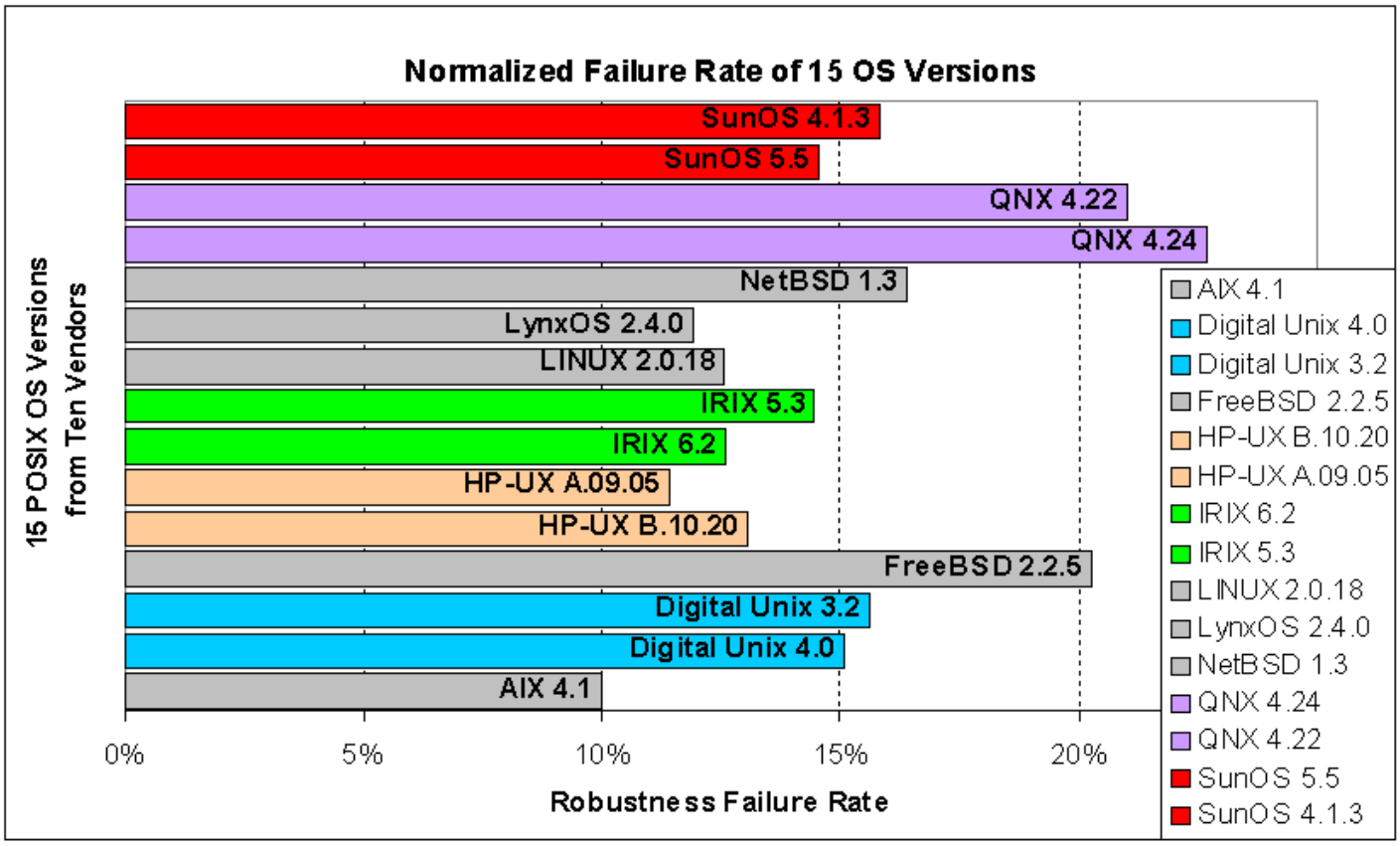



**Figure 2. Revised bathtub curve for software reliability**

The upgrades in Figure 2 imply feature upgrades, not upgrades for reliability. For feature upgrades, the complexity of software is likely to be increased, since the functionality of software is enhanced. Even bug fixes may be a reason for more software failures, if the bug fix induces other defects into software. For reliability upgrades, it is possible to incur a drop in software failure rate, if the goal of the upgrade is enhancing software reliability, such as a redesign or reimplementation of some modules using better engineering approaches, such as clean-room method.

A proof can be found in the result from Ballista project, robustness testing of off-the-shelf software Components. Figure 3 shows the testing results of fifteen POSIX compliant operating systems. From the graph we see that for QNX and HP-UX, robustness failure rate increases after the upgrade. But for SunOS, IRIX and Digital UNIX, robustness failure rate drops when the version numbers go up. Since software robustness is one aspect of software reliability, this result indicates that the upgrade of those systems shown in Figure 3 should have incorporated reliability upgrades.




## Available tools, techniques, and metrics

Since Software Reliability is one of the most important aspects of software quality, Reliability Engineering approaches are practiced in software field as well. *Software Reliability Engineering* (SRE) is the quantitative study of the operational behavior of software-based systems with respect to user requirements concerning reliability [IEEE95].

## Software Reliability Models

A proliferation of software reliability models have emerged as people try to understand the characteristics of how and why software fails, and try to quantify software reliability. Over 200 models have been developed since the early 1970s, but how to quantify software reliability still remains largely unsolved. Interested readers may refer to [RAC96], [Lyu95]. As many models as there are and many more emerging, none of the models can capture a satisfying amount of the complexity of software; constraints and assumptions have to be made for the quantifying process. Therefore, there is no single model that can be used in all situations. No model is complete or even representative. One model may work well for a set of certain software, but may be completely off track for other kinds of problems.

Most software models contain the following parts: assumptions, factors, and a mathematical function that relates the reliability with the factors. The mathematical function is usually higher order exponential or logarithmic.

Software modeling techniques can be divided into two subcategories: prediction modeling and estimation modeling. [RAC96] Both kinds of modeling techniques are based on observing and accumulating failure data and analyzing with statistical inference. The major difference of the two models are shown in Table 1.

| ISSUES | PREDICTION MODELS | ESTIMATION MODELS |
|---|---|---|
| **DATA REFERENCE** | Uses historical data | Uses data from the current software development effort |
| **WHEN USED IN DEVELOPMENT CYCLE** | Usually made prior to development or test phases; can be used as early as concept phase | Usually made later in life cycle(after some data have been collected); not typically used in concept or development phases |
| **TIME FRAME** | Predict reliability at some future time | Estimate reliability at either present or some future time |

**Table 1. Difference between software reliability prediction models and software reliability estimation models**

Representative prediction models include Musa's Execution Time Model, Putnam's Model. and Rome Laboratory models TR-92-51 and TR-92-15, etc. Using prediction models, software reliability can be predicted early in the development phase and enhancements can be initiated to improve the reliability.

Representative estimation models include exponential distribution models, Weibull distribution model, Thompson and Chelson's model, etc. Exponential models and Weibull distribution model are usually named as classical fault count/fault rate estimation models, while Thompson and Chelson's model belong to Bayesian fault rate estimation models.

The field has matured to the point that software models can be applied in practical situations and give meaningful results and, second, that there is no one model that is best in all situations. [Lyu95] Because of the complexity of software, any model has to have extra assumptions. Only limited factors can be put into consideration. Most software reliability models ignore the software development process and focus on the results -- the observed faults and/or failures. By doing so, complexity is reduced and abstraction is achieved, however, the models tend to specialize to be applied to only a portion of the situations and a certain class of the problems. We have to carefully choose the right model that suits our specific case. Furthermore, the modeling results can not be blindly believed and applied.

## Software Reliability Metrics

Measurement is commonplace in other engineering field, but not in software engineering. Though frustrating, the quest of quantifying software reliability has never ceased. Until now, we still have no good way of measuring software reliability.

Measuring software reliability remains a difficult problem because we don't have a good understanding of the nature of software. There is no clear definition to what aspects are related to software reliability. We can not find a suitable way to measure software

reliability, and most of the aspects related to software reliability. Even the most obvious product metrics such as software size have not uniform definition.

It is tempting to measure something related to reliability to reflect the characteristics, if we can not measure reliability directly. The current practices of software reliability measurement can be divided into four categories: [RAC96]

- Product metrics

Software size is thought to be reflective of complexity, development effort and reliability. Lines Of Code (LOC), or LOC in thousands(KLOC), is an intuitive initial approach to measuring software size. But there is not a standard way of counting. Typically, source code is used(SLOC, KSLOC) and comments and other non-executable statements are not counted. This method can not faithfully compare software not written in the same language. The advent of new technologies of code reuse and code generation technique also cast doubt on this simple method.

Function point metric is a method of measuring the functionality of a proposed software development based upon a count of inputs, outputs, master files, inquires, and interfaces. The method can be used to estimate the size of a software system as soon as these functions can be identified. It is a measure of the functional complexity of the program. It measures the functionality delivered to the user and is independent of the programming language. It is used primarily for business systems; it is not proven in scientific or real-time applications.

Complexity is directly related to software reliability, so representing complexity is important. Complexity-oriented metrics is a method of determining the complexity of a program's control structure, by simplify the code into a graphical representation. Representative metric is McCabe's Complexity Metric.

Test coverage metrics are a way of estimating fault and reliability by performing tests on software products, based on the assumption that software reliability is a function of the portion of software that has been successfully verified or tested. Detailed discussion about various software testing methods can be found in topic Software Testing.

- Project management metrics

Researchers have realized that good management can result in better products. Research has demonstrated that a relationship exists between the development process and the ability to complete projects on time and within the desired quality objectives. Costs increase when developers use inadequate processes. Higher reliability can be achieved by using better development process, risk management process, configuration management process, etc.

- Process metrics

Based on the assumption that the quality of the product is a direct function of the process, process metrics can be used to estimate, monitor and improve the reliability and quality of software. ISO-9000 certification, or "quality management standards", is the generic reference for a family of standards developed by the International Standards Organization(ISO).

- Fault and failure metrics

The goal of collecting fault and failure metrics is to be able to determine when the software is approaching failure-free execution. Minimally, both the number of faults found during testing (i.e., before delivery) and the failures (or other problems) reported by users after delivery are collected, summarized and analyzed to achieve this goal. Test strategy is highly relative to the effectiveness of fault metrics, because if the testing scenario does not cover the full functionality of the software, the software may pass all tests and yet be prone to failure once delivered. Usually, failure metrics are based upon customer information regarding failures found after release of the software. The failure data collected is therefore used to calculate failure density, Mean Time Between Failures (MTBF) or other parameters to measure or predict software reliability.

### Software Reliability Improvement Techniques

Good engineering methods can largely improve software reliability.

Before the deployment of software products, testing, verification and validation are necessary steps. Software testing is heavily used to trigger, locate and remove software defects. Software testing is still in its infant stage; testing is crafted to suit specific needs in various software development projects in an ad-hoc manner. Various analysis tools such as trend analysis, fault-tree analysis, Orthogonal Defect classification and formal methods, etc, can also be used to minimize the possibility of defect occurrence after release and therefore improve software reliability.

After deployment of the software product, field data can be gathered and analyzed to study the behavior of software defects. Fault tolerance or fault/failure forecasting techniques will be helpful techniques and guide rules to minimize fault occurrence or impact of the fault on the system.

---

## Relationship to other topics

Software Reliability is a part of software quality. It relates to many areas where software quality is concerned.

- Traditional/Hardware Reliability

The initial quest in software reliability study is based on an analogy of traditional and hardware reliability. Many of the concepts and analytical methods that are used in traditional reliability can be used to assess and improve software reliability too. However, software reliability focuses on design perfection rather than manufacturing perfection, as traditional/hardware reliability does.

- Software Fault Tolerance

Software fault tolerance is a necessary part of a system with high reliability. It is a way of handling unknown and unpredictable software (and hardware) failures (faults) [Lyu95], by providing a set of functionally equivalent software modules developed by diverse and independent production teams. The assumption is the design diversity of software, which itself is difficult to achieve.

- Software Testing

Software testing serves as a way to measure and improve software reliability. It plays an important role in the design, implementation, validation and release phases. It is not a mature field. Advance in this field will have great impact on software industry.

- Social & Legal Concerns

As software permeates to every corner of our daily life, software related problems and the quality of software products can cause serious problems, such as the Therac-25 accident. The defects in software are significantly different than those in hardware and other components of the system: they are usually design defects, and a lot of them are related to problems in specification. The unfeasibility of completely testing a software module complicates the problem because bug-free software can not be guaranteed for a moderately complex piece of software. No matter how hard we try, defect-free software product can not be achieved. Losses caused by software defects causes more and more social and legal concerns. Guaranteeing no known bugs is certainly not a good-enough approach to the problem.

---

# Conclusions

Software reliability is a key part in software quality. The study of software reliability can be categorized into three parts: modeling, measurement and improvement.

Software reliability modeling has matured to the point that meaningful results can be obtained by applying suitable models to the problem. There are many models exist, but no single model can capture a necessary amount of the software characteristics. Assumptions and abstractions must be made to simplify the problem. There is no single model that is universal to all the situations.

Software reliability measurement is naive. Measurement is far from commonplace in software, as in other engineering field. "How good is the software, quantitatively?" As simple as the question is, there is still no good answer. Software reliability can not be directly measured, so other related factors are measured to estimate software reliability and compare it among products. Development process, faults and failures found are all factors related to software reliability.

Software reliability improvement is hard. The difficulty of the problem stems from insufficient understanding of software reliability and in general, the characteristics of software. Until now there is no good way to conquer the complexity problem of software. Complete testing of a moderately complex software module is infeasible. Defect-free software product can not be assured. Realistic constraints of time and budget severely limits the effort put into software reliability improvement.

As more and more software is creeping into embedded systems, we must make sure they don't embed disasters. If not considered carefully, software reliability can be the reliability bottleneck of the whole system. Ensuring software reliability is no easy task. As hard as the problem is, promising progresses are still being made toward more reliable software. More standard components, and better process are introduced in software engineering field.

---

# Annotated References

- [Lyu95] Michael R. Lyu , *Handbook of Software Reliability Engineering*. McGraw-Hill publishing, 1995, ISBN 0-07-039400-8. http://portal.research.bell-labs.com/orgs/ssr/book/reliability/introduction.html

This book gives a broad and in-depth overview of the Software Reliability Engineering(SRE) research. It is comprehensive and up-to-date, embracing both traditional mature modeling and prediction methods, and new emerging techniques. Chapter one can be served as the introduction to Software Engineering. The book contains 17 chapters, classified into three parts: Technical Foundations, Practices and Experiences and Emerging Techniques. Part one focuses on the traditional analysis, prediction, estimation, or simulation approaches. Part two introduces case studies and best current practices of SRE. Part three is devoted to

summarizing newly deployed techniques in SRE such as Software Reliability Simulation, Software Testing, Fault Tree Analysis and Neural Networks. Page 567, etc.

The author is also active in many research projects listed in http://www.cse.cuhk.edu.hk/~lyu/FYP.html.

- [RAC96] Reliability Analysis Center, *Introduction to Software Reliability: A state of the Art Review.* Reliability Analysis Center (RAC), 1996. http://rome.iitri.com/RAC/

The RAC book has a broad range of short introductions to various Software Reliability disciplines such as Software Reliability models, the contrast of software issues to hardware, and various software engineering models and metrics. It is very good to be used as an introduction and starting point to arcane theories and abstract mathematics. It also has very thorough reference lists.

- [Rook90] Paul Rook, editor. Software Reliability Handbook. Centre for Software Reliability, City University, London, U.K. 1990.

This book is another handbook on software reliability. Although not as new as [Lyu95], it is also a good book.

- [Keiller91] Keiller, Peter A. and Miller, Douglas R., "*On the Use and the Performance of Software Reliability Growth Models*", Software Reliability and Safety, Elsvier, 1991, (pp. 95-117)
- [ANSI91] ANSI/IEEE, "*Standard Glossary of Software Engineering Terminology*", STD-729-1991, ANSI/IEEE, 1991

Terminology standard. Has many standard definitions.

- [Keene94] Keene, S. J., "*Comparing Hardware and Software Reliability*", Reliability Review, 14(4), December 1994, pp. 5-7, 21
- [IEEE95] IEEE, Charter and Organization of the Software Reliability Engineering Committee, 1995

- Gives the standard definition of Software Reliability Engineering, and others.

- [Sheffield] H. Lin, Scientific American, vol. 253, no. 6 (Dec. 1985), p.48.

- Sheffield hickups caused by software.

- [Patriot] http://www.math.psu.edu/dna/455.f96/disasters.html

- How patriot missile failed to intercept a scud in the gulf war.

- [Telephone outage] http://www.byte.com/art/9512/sec6/art1.htm#aircontrol

A lot of items about failures caused by unreliable software. Horror story 38, telephone. An example of chaos caused by erroneous software upgrades.

- [Neumann95] Neumann, P. Computer related risks. Addison Wesley, 1995.

The whole book summarizes many real events involving computer technologies and the people who depend on those technologies, with widely ranging causes and effects. In pp38, the defense system example can be found.

- [Therac25] http://ei.cs.vt.edu/~cs3604/lib/Therac_25/Therac_1.html. Nancy Leveson, Clark S. Turner, reprinted from "*An Investigation of the Therac-25 Accidents*". IEEE Computer, Vol. 26, No. 7, July 1993, pp. 18-41.

The classical software-related fatalities and the classical paper.

- [Ariane 5] Lions, J. (Chair), *Ariane 5 Flight 501 Failure*, European Space Agency, Paris, July 19, 1996. http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html. (Accessed May 1, 1999)

A typical example of how a not properly handled exception can cause a failed mission. Robustness can be expensive.

# Further Reading

- [Musa97] John D. Musa, *Introduction to Software Reliability Engineering and Testing,* 8th International Symposium on Software Reliability Engineering (Case Studies). November 2-5, 1997. Albuquerque, New Mexico.

This is the introduction to applying software reliability engineering to the testing area, so that reliability can be improved and assured after testing. This section of the ISSRE proceedings is composed by case studies covering software reliability engineering testing, operational profile, and several other military and aerospace cases which high software reliability is needed and software reliability engineering principles applied.

- [Musa87] John D. Musa, Anthony Iannino, and Kazuhira Okumoto, *Software Reliability: Measurement, Prediction, Application,* McGraw-Hill Book Company, 1987, ISBN 0-07-044093-X.

This is a classic work by John D. Musa et al. As the pioneer in Software Reliability Engineering, John Musa is rich in publications.

- [Musa93] Musa, J. "Operational Profiles in Software-Reliability Engineering." IEEE Software, March 1993.

An operational profile is a quantitative characterization of how a system will be used... To determine an operational profile, you look at use from a progressively narrowing perspective -- from customer down to operation -- and, at each step, you quantify how often each of the elements in that step will be used. More comments.

- http://www.byte.com/art/9512/sec6/art1.htm Alan Joch,

- Nine ways to make your code more reliable.

**Finally, sorted links:**

- John Musa has a collection of references for SRE practitioners.
- A very useful collection of reliability books from the Reliability Information Center, University of Maryland.
- A very useful collection of faces from the 11th INTERNATIONAL SOFTWARE QUALITY WEEK (QW'98), 26-29 May 1998, San Francisco, California USA. Abstracts also available.

---

Index of other topics

Home page

---