Software Reliability: To Use Or Not To Use?

Panel Chair: *Michael Lyu* (Bellcore)

Panelists:

Skeptics: *Fletcher Buckley* (Martin Marietta)
Robert Tausworthe (Jet Propulsion Laboratory)

Believers: *Ted Keller* (IBM-Loral)
John Musa (AT&T Bell Laboratories)

Introduction

Michael R. Lyu
Bellcore
445 South Street
Morristown, NJ 07962
<lyu@bellcore.com>

Computers are bringing revolutionary changes to our life with their involvement in most human-made systems for sensing, communication, control, guidance and decision-making. When the requirements for and dependencies on computers increase, the crises of computer failures also increase. The impact of hardware and software failures range from inconvenience (e.g., malfunctions of home appliances), economic loss (e.g., interceptions of banking systems) to life-threatening (e.g., failures of flight systems, medical software).

As the functionality of computer operations becomes more essential and complicated in the modern society, the reliability of computer software becomes more important and critical. In fact, computer software had already become the major source of reported outages in many systems. This trend has been signified by the fact that hardware components of a system become increasingly reliable, and software starts to dominate the cause of computer system failures and outages. As the demand for software increases, its size, complexity, and criticality also increases. Today, the growth in utilization of software components is largely responsible for the high overall complexity of many system designs, since it is the integrating potential of software that has allowed designers to contemplate more ambitious systems encompassing a broader and more multidisciplinary scope.

Research activities in software reliability engineering have been vigorous in the past two decades since Jelinski and Moranda proposed the first software reliability model in 1972˜[1]. Since then, numerous software reliability models and measurement procedures have been proposed for the prediction, estimation, and engineering of software reliability. However, there seems to be a gap among the software engineering practitioners regarding the use of software reliability. Believers advocate the use of software reliability and promote the stories of success, while skeptics doubt about the adequacy, validity, and consistency of software reliability both in terms of its concept and in terms of its practicality.

It is the purpose of this panel to bring together software reliability believers and skeptics to discuss, argue, debate various problems and issues in the practice of software reliability. The panel is expected to raise research, development and deployment issues concerning the use of software reliability, to address existing and potential problems, to resolve some misunderstandings and conflicts, and to reach a fundamental basis for the advancement/avoidance in this field.

The panelists are invited to discuss those topics including, but not limited to, the following:

---

[1] Z. Jelinski and P.B. Moranda, "Software Reliability Research," *Statistical Computer Performance Evaluation* W. Freiberber (editor), Academic Press, New York, 1972, pp. 465-484.

(1) Is reliability a critical and tangible metrics in software quality, or is it something superficial?

(2) Does quantitative reliability requirement make sense? Is it used often? Should it be promoted or discouraged?

(3) What is the advantages or risks in using reliability as a guideline for testing, shipping or maintenance?

(4) Is software reliability perceived as cost or value?

(5) Is reliability a believed and accepted measure for most software managers, or is it just a magic number most managers would not scrutinize.

(6) Do customers care about quantitative measure of reliability, or they only care about failure incidences?

(7) Is software reliability verifiable? Is the measurement of very high reliability (e.g., 0.999999) possible?

(8) Does the pursuit of reliability always delay product schedule and increase cost?

(9) Is a reliability engineer considered as a descent position in the software industry, or is it a position for second-class citizen?

(10) What are the main problems in software reliability? Can we overcome them, or they belong to our next generation (or the next one after)?

(11) Under what circumstances should we use reliability? Under what circumstances should we not?

(12) Are we using the right approach to software reliability? What are the alternatives if not?

(13) Does software reliability work overall? Why or why not?

The following sections consist the position statements written by each panelist under the panel title and the suggested topics. Fletcher Buckley of Martin Marietta and Robert Tausworthe of Jet Propulsion Laboratory will present their skeptical views on software reliability, and point out its potentials for misuse. Ted Keller of IBM-Loral and John Musa of AT&T Bell Labs. will present their supporting views on software reliability, and explain why they promote the use of software reliability. During the panel session, discussions, debates, arguments, or even crossfires are expected among the panelists and the audience.

Software Reliability, Or One More Look At The Emperor's New Clothes

Fletcher Buckley
Martin Marietta
103 Wexford Drive
Cherry Hill, NJ 08003
<fbuckley@motown.ge.com>

One of the more successful problem-solving techniques (as my psychiatrist would say) is to look for analogies in other fields and use those solutions to resolve our problems. Sometimes this approach works -- sometimes you get the bear -- and sometimes it doesn't -- the bear gets you. One of the more recent attempts to use this transfer technique is to apply it to the field that has sprung up called "software reliability".

Now, reliability itself has a long and honorable history in the hardware world. In the hardware world, we can:

(1) receive a hardware reliability requirement at the start of a project (for example, 10,000 hours mean-time-to-failure (MTBF) at a 95% confidence level),

(2) compose a mathematical model of the system,

(3) decompose the overall reliability requirement and assign supporting reliability requirements to individual system components,

(4) determine the overall effect of these subordinate reliability requirements on the system,

(5) improve the supporting reliability requirements either through system redesign (e.g., placing a fan underneath the power supply to reduce the temperature -- thus increasing the MTBF), or through the use of higher-reliability parts,

(6) determine what the cost of meeting a system reliability requirement will be -- and determine what the additional incremental quantized cost would be to increase the MTBF by a quantized amount,

(7) monitor the construction of the hardware system to gain a reasonable degree of confidence that the system as it is being built, is acquiring the required reliability attributes,

(8) test the "as-built" system with well-accepted tests to determine whether or not the overall system reliability requirement has been met.

Now we software folks share a common need with the hardware people -- we need to have our systems work. So we looked at the hardware world, seized on the term "reliability" and applied it to our software world not realizing the contextual baggage associated with its former usage. Right now in looking at items (1) through (8) above, in the software world, we can:

(1) Construct a mathematical model.

(2) Test to see when the latest failure will occur.

(3) Apply the latest failure data to the model and obtain a projection of the new MTBF of the software.

Construction of software reliability models has been a enterprise going back to at least 1973 and these models come in various flavors, each with a different set of assumptions and with corresponding mathematical treatments. The majority of them are based on the concept of random failures -- and justify that assumption by saying the input data comes in at random times with random contents. Variants in these models include removal of the fault prior to continuing the test, and some even acknowledge that the act of removing the fault may cause other errors in the software.

Testing to see when the next failure will occur also has several variants as it involves a time variable. Now should this time variable be the number of instructions executed or wall-clock time? If it is wall-clock time then moving the program to a faster machine may result in a lower MTBF which is not a desired result.

This was first codified by T. Workman of HP who expressed it in "Workman's Law Of Software Reliability". While working at HP, he noted that despite their best endeavors, despite application of all the new software methodologies, tools, and techniques, that the reliability of the "as-built" software was not getting any better. Further thought on the problem recognized that the speed of the hardware was doubling every two years, the software was thus being executed twice as fast every two years. and therefore the software was failing twice as fast, every two years. Observing this, he postulated that for better software reliability HP should not build faster machines but rather they should build slower machines, with the corollary that ultimate software reliability would be achieved with the hardware powered down.

Given all of the above, the question then comes back, "What is to be done?" We still have a need for the software to work and the current work in the field does not inspire confidence in the industrial community. (I know of no one who will play the game called "Bet Your Company" on a fixed-price contract with a software reliability requirement. Now on a cost-plus contract, it is a different story ...) But, if we step back a bit and take another look at the situation, there are well-recognized solutions to the concerns of having the software work -- and the solution is not in software reliability but rather in software availability.

Consider, for example, a real-time system that is controlling anti-aircraft missiles in mixed airspace (both friendly and hostile aircraft.) To avoid fratricide, the missile will self-destruct if it does not receive a command-guidance message at least every 10 seconds. One approach to resolving this problem is a rapid restoration capability -- the software can fail all the time but as long as it can restart and get the next command to the missile within the 10-second window, no one cares.

In a similar manner, checkpoint-restart has been in the COBOL programs processing our payroll checks for the past 30 years -- and the wonderful world of fault-tolerant systems is becoming a reality.

In conclusion, while software reliability testing may be advantageous in some restricted specialized domains, from an overall industrial viewpoint:

(1) It is not comparable with the corresponding hardware reliability field, in that it provides no basis for early design and no ability to quantizing costs versus improvements.

(2) Not much progress has been made since 1973 when the cry was for "more data" at the 3rd Software Reliability Symposium.

(3) Industry does not have the confidence in the models to accept fixed-price contracts that contain software reliability requirements.

(4) The problems (ensuring that the system will perform its intended function) are suspectable to design solutions when availability is considered instead of reliability.

Software Reliability Modeling: An Oversimplified Art

Robert C. Tausworthe
Jet Propulsion Laboratory
MS 525-3600
4800 Oak Grove Drive
Pasadena, CA 91109
<tausworthe@isd.jpl.nasa.gov>

There is no doubt in my mind that there is a great a need for reliable software. I am convinced of this by the concerns for poor software quality I see regularly expressed in the professional literature, where reliability is the first-cited offender. I am continually reminded of this as I sit in front of my terminal amid everyday instances of system crashes, lost email, and mysterious operational messages, such as ‘‘Both arguments of the arctangent function may not be zero!’’ (This a fact I have long known; its connection with a file-scanning program I wrote is mysterious, though!).

Reliability is a critical, yet tangible metric. We can measure it, if we try. Even without trying, we are aware of our degree of confidence in using various software packages. We are capable of articulating our needs in meaningful, quantitative terms, such as MTTF, MTBF, the probability of faithful operation for specified periods within given confidence limits, and the number and classifications of as-yet unrepaired faults. Deficiencies in any of these quantities are indications of needs for further work in maintenance and test. Readiness for shipping requires an assessment of risk and a judgment of propriety based on cost, schedule, ethics, and confidence.

The means to achieve prespecified reliability goals, however, are yet primitive. We try our best not to commit errors that put faults into our software. We analyze our code, inspect it, test it, and operate it in production. We use all this information to help us design our next programs. But, in the end, our systems still have bugs. And worse yet, merely measuring the reliability of our systems does not directly make the systems more reliable.

Reliability engineering is a profession that requires a certain set of skills, disciplines, and supporting technologies. The fundamental element in the art is the reliability model. Whether reliability engineering is an honest and honored practice, therefore, depends heavily on the accuracy and trustworthiness of its basic principles, as embodied in models.

Ideally, the least that a trustworthy reliability model would tell us is how many bugs we can expect to find and how much time and effort will be required to fix a designated number of them. Then, at least, we can make proper estimates of cost, schedule, and affordability. If a model is not trustworthy, however, its value to management and programmers alike is almost nil. If used, it is just one more thing they will have to worry about while building their system.

Some individuals and companies have been able to make their models work for them, to return a profit, and to keep them competitive in the marketplace. They profess confidence in the use and maintenance of these models. But other software professionals have had less success and bear less confidence in the application of reliability modeling to their tasks.

Most managers don’t understand averages and variances. If a model says to expect about n bugs, there had better be close to n bugs found, and no more or less. There should be no more, because test plans have limited contingencies; there should be no fewer, because testing would then be extended needlessly on in the quest of finding the missing bugs. Confidence in estimating is a critical requisite for making competitive fixed-price bids for systems that have stringent reliability requirements.

Even if an estimate of the number of bugs that will be found were to be correct, the manager and programmer are still left clueless as what to do to minimize costs to locate and correct them. Existence theorems may be elegant to mathematicians, but constructive methods are required in the practical world.

The public view of psychology for many years seemed to be that if you came to know, through psychoanalysis, that you were mentally ill, and that this illness was caused by a past event or behavioral fault, then this knowledge alone should be enough to cure you (it was always this way in the movies). But knowing you are buggy does not cure you. It still often takes years of therapy, and sometimes even that doesn’t work. The knowledge of how buggy you are may permit the analyst to predict grossly how long it will take to debug you; but the therapy depends on symptom-based diagnosis, training, and experience, not on the number of split personalities you have.

Programs, somewhat like humans, fail because they have diseased parts. We call them ‘‘faults,’’ but they are truly maladies because they hurt us through excess costs, wasted time, wasted effort, lost information, damaged lives, and missed opportunities. There are hundreds of maladies that we humans may inherit from birth, or subsequently fall prey to, that may be latent in our beings. There are perhaps an equal number of categories of software diseases, but these have not yet been cataloged very well.

Medical doctors would probably hedge if asked to predict if and when you will become ill without first knowing what illness you are talking about and what your family history is. They would probably cite actuarial studies that provide enough insight for insurance companies, which can count on averages, but even these statistics are likely to be very unreliable when applied to you, a single, random, particular case. Even if your mean-time-between-illness susceptibility were known, it would probably not be a number that would be accurate enough for you to plan your life by. I personally do not know what I would do with a deemed-accurate prediction that I would encounter 65 diseases during my lifetime (if that were the average based on all known factors about me)---except that I would perhaps establish a health-conscious life style and environment, and that after encountering 64 of the 65 illnesses, I would surely become an over-cautious, fearful, paranoid, nonfunctioning basket-case!

Medical statistics are much better documented than are software failure data. We know how many of cases per year there are of all the major diseases, and how these are correlated with body, environment, and event characteristics, such as gender, race, blood type, social habits, sexual preferences, and profession. They are published annually in the Encyclopedia Britannica yearbooks. Treatment cost statistics are tracked and reported by insurance companies in order to justify Medicare and other premiums.

But we know a lot less about software maladies. There are a few fault categories that sometimes get recorded in projects, such as ‘‘off-by-one error,’’ ‘‘design error,’’ ‘‘major failure,’’ ‘‘cosmetic fault,’’ etc. But I have yet to encounter a definitive listing of the major classifications of software faults. And certainly, there is very little correlative information on if (and how) gender, race, social habits, and sexual preferences enter into the statistics.

Computer programs of today are extremely intricate, complicated, and dispassionate servants of mankind. The way they fail is therefore likely to be complex. It has been adequately demonstrated that none of the traditional (simple, few-parameter) reliability models always predicts failures better than the others. Is this because of our inability to measure those things that the models need or predict, or is it because each of the models predicts a different set of maladies? Or is it just that the models themselves do not adequately accommodate the complexities of the entities whose performances they attempt to predict?

Most reliability models today are based on empirical data and perhaps on low-order heuristics of the underlying physical failure mechanisms, as well; these are expressed as simple equations involving only a very few parameters that are to be related to the project at hand. The historical data use to calibrate the model often exists only in the the form of invalidated, non-homogeneous, poorly-documented failure data from heterogeneous past projects. The models typically provide only estimates of mean project behavior, given values for the very few, usually guessed-at parameters. And even if the guesses were correct, the development process, being stochastic, is guaranteed not follow the mean exactly.

Simple reliability models can perhaps therefore foretell the behavior of application-specific software built by stable, well-measured, dedicated organizations that develop these applications on a regular basis for the same environment, better than they can for more volatile, ad-hoc organizations that develop a wide range of systems over varying customer environments.

People want simple models because they can mentally cope with them. But simple models do not adequately mirror complex systems. Wishing a complex thing to be simple does not make it simple. Ignoring the complexities of a system does not make it more predictable. Ignoring the inadequacies and imperfections of a model does not make it more applicable.

I understand why models have been kept simple. We don’t have enough validated empirical data to support more complex methods. Historical data is costly to gather, validate, and analyze. Managers traditionally have not been able to assess the quantitative values added to their projects by collecting and analyzing project parameters and failure data.

Modelers have bemoaned the lack-of-data problem for over a decade now. But the situation has not significantly changed since then. There is a ‘‘Catch-22’’ at work: Few projects are willing to pay for more reliability data until the benefits of reliability models can be cost-effectively and convincingly demonstrated; but the demonstration depends on first having the data. The reliability data that projects normally collect and use are sometimes not the data that conveniently fit the models assumptions.

Models also need to tell us other things in addition to just average numbers. They need to tell us the variances and skews of estimates. They should tell us which model, among those that exist, best fits our project situation. They need to take into account product-related, process-related, personnel, environmental, and situational characteristics of the development task. They need to relate effort, schedule, and other project resources, to faults detected. They need to show us how to trade off costs of inspections vs. those for testing. They need to predict the classifications and criticalities of failures that will be encountered. They need to relate failure symptoms to fault types, to causes of error, and to consequences of failures.

Don't get me wrong. I am not against the use of models. I have done a lot of modeling of my own in my day. My belief is merely that the models we have today are too simple to fit the complexities of the complexities of the tasks and environments they are applied to If today's models were truly of benefit to all managers and programmers, they would have caught on and would be in very widespread use by now; they have been around long enough to ''have been discovered.'' Entrepreneurs would be selling them at good profits. Arguments for their use would be convincing and comprehensive.

But, to me, the approaches to reliability modeling today are still primarily of academic interest: alpha- and beta-test stuff. I can see statistical trends, but not detail. I can see generic behavior, but not particular causality. I cannot relate fault frequencies or classifications to the development process. While some practitioners extol the virtues of their models, others are adamant in their criticism of them. I am more neutral. I neither have a stake nor an axe, because my interest is academic: I am neither a (professional) programmer nor a software project manager.

As a technologist, however, it is my job to assess the readiness of certain technologies for JPL. Right now, with what I know about software reliability, I don't know how to get JPL's projects to pay for collecting, analyzing, and using software reliability data. I don't know how to convince them that reliability forecasting risks are offset by the benefits they provide. I don't know how to assure managers that results from simple models can be trusted. I don't know how to convince programmers that knowing how many bugs they may expect to create or find will help them make their software more reliable. If I knew how to do these things, I certainly would have done so by now.

In short, I am a skeptic. But I am also optimistic that the art will continually improve. Reliability is a problem that will not simply disappear of its own accord. Because of its critical relationship with quality, it will continue to be studied and modeled. Eventually, the industry will either come up with a trusted model (or models), or else, will discover the existence of a ''reliability uncertainty principle'' on modeling accuracy. Reliability engineering, as a profession, is not a dead end, provided its practitioners are willing to admit that their art is still in adolescence, and are willing to continue the quest for a richer and more cause-related understanding of the reliability process.

The Paradigm Transition from Qualitative to Quantitative Software Performance

Ted Keller
Space Shuttle Project Coordination
Loral Space Information Systems
Houston, Texas
<keller@houvmscc.vnet.ibm.com>

The ANSI/AIAA Standard (R-013-1992, "Recommended Practice Software Reliability") definition for Software Reliability Engineering is: "Software Reliability Engineering -- The application of statistical techniques to data collected during system development and operation to specify, predict, estimate, and assess the reliability of software-based systems."

The common applications of this technique include:

(1) Assist in determining whether a specific software process is likely to produce code which satisfies a given software reliability requirement

(2) Estimate the size of a software maintenance effort by predicting the failure rate expected during the operational phase

(3) Provide a metric for process improvement evaluation

(4) Assist software safety certification

(5) Determine when to release a software system or to stop testing it

(6) Estimate the occurrence of the next failure for a software system

(7) Identify components in a software system which are more likely to fail

(8) Measure reliability of a software system in operation and use this information to control change to the system

Man-rated software, "software which is in control of systems and environments upon which human life is critically dependent," must receive special treatment throughout its life-cycle in order to assure demanded safety, reliability, and quality levels have been attained. There are three major factors involved in "certifying" the safety and mission readiness of the Space Shuttle Onboard man-rated software. The first involves proving that the actual process used to define, design, develop, test, and verify the software meets established standards for man-rated software, as specified by the procurer, and that precisely "that" process has been followed without exception, unless formally approved deviations have been documented.

The second factor involves the maintaining of sufficiently detailed defect density history and failure history of software throughout multiple applications of the specified process over entire life-cycles in order to monitor, measure, and manage the quality of each system. Basic characteristics of software dictate that:

(1) Safety Certification is currently based on "Process Adherence" rather that "Product"

(2) Assumption is that a known, controlled, repeatable process will result in a product of known quality

(3) This assumption requires constant statistical revalidation

(4) The relationship between quality and reliability must be established for each software system and statistically demonstrated for the required operational profile

(5) Quality must be built into the software, at a known level, rather than adding or determining the quality after development

The third factor builds upon the results of the preceding two factors, combining sound engineering judgment and a systematic evaluation of the software failure probability. This should involve reliability modeling in addition to actual failure mode identification and risk assessment. Ideally, there is a specified reliability level the software must be "certified" to exceed for a specific set of required operational scenarios.

In practice, however, such a concise and deterministic quality assessment is very elusive. For example, there is not sufficient "burn-in" time realistically possible for most software systems to demonstrate a 0.0000001 failure probability! Software statements can be randomly accessed and classical analysis techniques such as "fault-free analysis" and "critical system identification" frequently break down due to the infinite possibilities for an "abstract entity" such as software to "behave".

The approach to reliability assurance for the Space Shuttle Primary Avionics Software involves the systematic removal and elimination of all the known software failure modes and a common defect search and removal process which theoretically leaves the software "error-free". The software is then executed through an extensive suite of nominal and off-nominal scenarios selected to cover the operational profile envelope required for shuttle missions. Since software cannot deteriorate, burn out, wear out, or fatigue, once defects are removed, they cannot regenerate within the software until it is changed subsequently. This approach then relies on prevention of new defects and detection and permanent removal of latent defects.

Since there are virtually an infinite number of permutations and combinations of software paths in the Shuttle avionics software, this approach is extremely inspection dependent. Inspections are employ detailed checklists and are automated where possible. Using the data accumulated by application of the second factor described above, statistical and analytical reliability models are employed to estimate the fault and failure densities remaining at the beginning of operational use to provide added confidence (allow for risk assessment) to the conventional engineering judgment that the system is "Ready for Flight". Fault density is much easier to determine than remaining failures. Accurate models for remaining failures as a function of both fault density and observed failure intensity have been validated. Reiterative revalidation of the model applications and calibration of the models to the process in use have produced reasonably accurate estimates of remaining failures when compared with actual system performance over multi-year periods of operational use.

Our current research is attempting to relate the structure of individual software code modules and the dynamic operational complexity of that code to the fault density and failure intensities of the composite software systems. In each case adaptive, closed-loop calibration and model validation techniques ensure that updated model estimates are reasonable in order to retain previously established credibility.

Software Reliability Engineering: To Use

John D. Musa
AT&T Bell Laboratories
Room 2D-248
600 Mountain Avenue
Murray Hill, NJ 07974-0636
<j.d.musa@att.com>

The main argument for software reliability engineering is very simple: IT WORKS. It has been widely applied in AT&T as well as other companies worldwide. Our experience demonstrates at least three principal benefits:

(1) You gain a competitive edge in quality by *precisely supplying* customers with the balance of reliability, delivery date, and cost that they want.

(2) You reduce the risk of unsatisfactory reliability by engineering and tracking it during the development process.

(3) You increase development efficiency with quantitative objectives and focused effort, where the focused effort is based on the use and criticality of functions. Use is characterized by the operational profile.

Cost is relatively small. The benefit-to-cost ratio is generally 12 or greater.

On AT&T's International Definity˜[2] project (a PBX switching system), applying SRE and related technologies increased customer satisfaction significantly. This was demonstrated by a factor of 10 sales increase, primarily due to quality improvement, over the previous version. The system had increased reliability. Customer-reported problems dropped by a factor of 10. There have been no serious service-affecting outages in the first two years in the field, with thousands of systems installed worldwide.

---

[2] S. R. Abramson et al., "Customer Satisfaction-Based Product Development," *Proceedings International Switching Symposium, Volume 2,* Institute of Electronics, Information and Communications Engineers, Yokohama, Japan, 1992, pp. 65-69.

Extensive use of the operational profile in system testing along with quantitative reliability objectives enabled the project to reduce the system test interval by a factor of two. This resulted in a 30 percent reduction in the total project development interval, speeding up availability on the market.

Use of SRE reduced system test costs by a factor of 2 and program maintenance costs by a factor of 10. Thus there was a significant impact on overall project productivity.

DEFINITY is not an isolated example. SRE has been an AT&T Best Current Practice since May, 1991. A tool or technology is not considered for Best Current Practice status until it has been applied successfully on a number of projects, with documented evidence of a strong benefit/cost ratio. You have to make a written business case for a proposed Best Current Practice. The proposal undergoes probing reviews by two boards of high-level software managers. In 1991, only one of six proposals were approved.

AT&T Bell Laboratories has a President's Quality Award that recognizes products or services of outstanding quality. It is interesting to note that four of the five software winners of this award used SRE.

SRE is being integrated into the software development process of various AT&T business units. For example, the Operations Technology Center, an organization of around 1000 software developers, made SRE part of its standard development process in April 1992. This process is currently undergoing ISO certification.

I have given only examples of use at AT&T, because I am familiar with them. However, many other companies are applying SRE. You can see for yourself the rapidly growing use of SRE, merely by scanning software engineering literature and conferences such as this one. Certainly there are and will continue to be problems and challenges to be overcome, including those of technology transfer, but this is typical of any dynamic and useful technology.

References