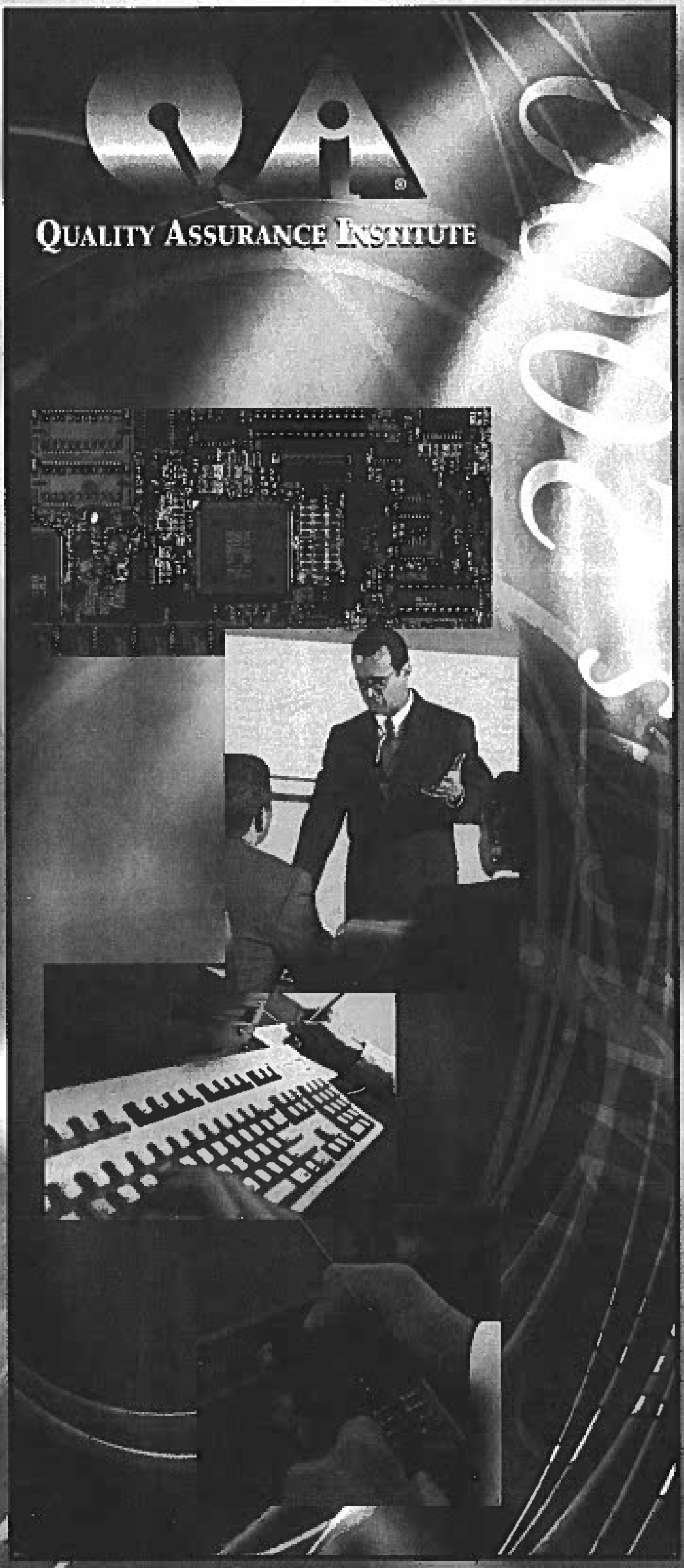
The Journal of the QUALITY ASSURANCE INSTITUTE

Solutions 2000 ❖ Volume 14, Number 4 ❖ October 2000



The Journal of the QAI CONTENTS

OCTOBER 2000
VOLUME 14, NUMBER 4

The Journal of the Quality Assurance Institute
(Copyright © 2000) is published quarterly by Quality Assurance Institute (QAI) with editorial and executive offices at 7575 Dr. Phillips Blvd., Suite 350 Orlando, FL 32819-7273; 407-363-1111; FAX: 407-363-1112, www.qaiusa.com. QAI members receive four issues per year at no additional cost, but may order additional subscriptions at $43 each; subscription price to nonmembers is $50. Individual copies of some back issues are available at $10 each; write or call QAI. Reprints of 500 or more of articles appearing in *The Journal* are available at a low cost. Requests for reprints or permission to reproduce may be made by writing or phoning QAI. Articles, letters to the editor, advertising, and suggestions are welcome. Address correspondence to QAI's office.

A Discussion of Software Reliability Modeling Problems

Originally published in The Journal of the RAC, First Quarter 2000

A quarter of a century has passed since the first software reliability model appeared. Many dozens more, of various types, have been developed since. Many practitioners still disagree on the practical uses of models in software managing, staffing, costing and release activities. The present article examines this situation, discusses some of its causes and suggests some approaches to improve it.

This author believes that the current user dissatisfaction stems from the manner in which reliability, as a concept, is applied to the software environment and on how the related models have evolved. This article begins by providing an overview of the characteristics of software reliability models and of their development efforts. This is followed by a discussion of the assumptions underlying software reliability models and other related problems. Finally, some suggestions on how to improve the situation are provided.

Software Reliability

Broadly speaking, reliability is the probability of satisfactory operation of a system or device, under specific conditions, for a specific time. In software systems, the concept of reliability is complicated by several factors. An operator and hardware subsystem is always associated with the software. Hence, documentation, training, and interface problems can (directly or indirectly) induce software failures, thus becoming part of the reliability assessment process.

It is also important to understand the origins of the concept of software reliability. It evolved as a result of the increasing use of embedded software in already existing hardware systems. Hardware reliability had been successfully developed and understood since the early 50's. It was only natural that the same type of hardware professionals (e.g., systems and electronical engineers) would develop the first software models by extending and adapting their previously successful hardware ones. But the hardware modeling techniques, as we will later see, did not always work well in the software environment.

After its development stage, hardware is mass-produced and the resulting industrial product is used in relatively similar environments. For example, after the prototype has been developed and tested, a military helicopter is mass-produced and flown by similarly trained pilots in attack and rescue missions.

Software, on the other hand, always remains a "prototype" on its own, of which exact copies are made and used by a wide variety of people with very different interests and applications. For example, matrix-inversion software may be used by a high school student to solve a 2x2 system of linear equations or by a Ph.D. candidate to invert thousands of multivariate correlation matrices in a simulation study.

The modeling stage is permanent for the (prototype) software product. Hence, its characteristics and problems have a more significant impact in this environment than in the hardware one, as we will see next.

Modeling Problems

A major problem encountered in the software reliability modeling activity arises from the involvement of two different groups of individuals, modelers and practitioners, each having a different product and process in mind and seeking a different result. The fundamental differences between these two groups make the existing software reliability models a professional success for many modelers but unsatisfactory working tools for many practitioners.

Modelers are usually researchers or academicians while most practitioners are software developers. Academicians and researchers rely on publishing papers, which are peer reviewed and assessed for their theoretical value by other academicians and researchers, to obtain their tenure, promotion or doctorates. To publish their work, modelers use sophistication statistical theories that require strict (and sometimes unrealistic or unjustified) underlying assumptions.

Practitioners (managers, developers) on the other hand, need to staff, cost, and release software products. Practitioners work with programmers, under time constraints and must rely on insufficient and sometimes deficient information. Software practitioners need models and approaches that are feasible (implemented without incurring exorbitant costs or excessive burden) and practical (can be used to staff, cost, release the software, etc.)

Theoretical models are based on many mathematically driven software assumptions that, in practice, do not hold or are weak. In addition, many models of the Black Box (e.g., time-based) class fail to capture several other important factors that affect software reliability.

Validity of Software Reliability Model Assumptions

Some software reliability model assumptions do not hold or are weak because they have a purely theoretical (mathematical) origin. Note that not all model assumptions are invalid all the time or in all the models. Some (Black Box) model assumptions and related topics and the reasons for their possible lack of validity are:

Definition and Criticality of Failures: In many cases, failures are user dependent and poorly defined. This makes their identification in the field also difficult.

Definition of Time Units: Include calendar time, execution time, etc., which may differ substantially or may not always be accurately recorded. Some models (Musa) have found ways to deal with this by converting units from one time domain to another. The assumption also implies that testing intensity is time homogeneous.

Fixed Number of Faults: Assumes that no additional faults are introduced and that every debugging attempt is successful. Some models (e. g., imperfect debugging of Goel) attempt to address these issues.

All Faults Have the Same Failure Rate: This implies that all faults have the same probability of occurrence. But failure probability is in fact associated with input domain and user profile. Hence, all failures are not equally likely. For example, a software failure occurs only when a specific input is given. But some users may provide such input very frequently. For this user, the program will have a high failure rate. Another consequence of failures not being equally likely is that reliability will be affected by the order in which faults are discovered. Say two different testing teams have uncovered two different sequences of "n" faults. They may obtain two different reliability estimates (and this is complicated by the specific user profile).

All Software Faults are Always Exposed: Faults are encountered only if that part of the software where they reside is exercised. If there is fault that prevents the execution of some part of the software until it is removed, the faults that exist downstream, in that part of the code, are not exposed until the initial one is uncovered and removed.

Faults are Immediately Removed: Testing will not usually be stopped once a fault is uncovered. Adaptive procedures (removing/patching part of the code; restricting the input) will be used to continue the testing, while the fault is uncovered and fixed.

Only One Failure at a Time Occurs: This is a required Poisson Process assumption and not all software necessarily complies with it. There may occur multiple failures simultaneously (and not all faults will be corrected before restarting the testing).

Testing is Homogeneous: Testing effort is not always the same; personnel may vary as well as time dedicated to it and to other concurrent functions. In addition, if a critical fault is uncovered or a deadline approaches, testing may become more intensive.

Failure Rate is (only) Proportional to Error Content: There are many other factors, such as user profile, complexity of the problem, language, programming experience, etc.

Number of Failures in Disjoint Intervals is Independent: This is also another key Poisson Process assumption. There is a finite number of faults in the software, which are sequentially removed. If we encounter and remove a large number of faults in one interval, then in the next time interval there will be less faults to find, and vice versa. Hence, the number of faults uncovered in two disjoint, adjacent time intervals are affected by the number of faults previously uncovered.

Times Between Failures are Independent: Since the number of failures encountered in disjoint intervals is not independent, this associated assumption is not true either.

Testing Proceeds Only After a Fault is Removed (corrected): This is an ideal situation that does not occur in practice. Adaptive procedures are used to proceed with testing.

All the Code is Tested, All the Time: Some testing may occur before all the code is completed. Then, if a fault is encountered and located in a given module, this fault may be removed or patched (adaptive procedure) to proceed with the testing.

Run Time versus Think Time: Run (test) time models penalize development strategies that spend more desk (think) time analyzing the program than in testing. Calendar time captures both of these activities (think and run times) but this time measurement is weak.

"I believe the current user dissatisfaction stems from the manner in which reliability, as a concept, is applied to the software environment and how the related models have evolved."
Jorge Romeu

Specific Prior Distribution: Some modelers have attempted to deal with the reality of different failure rates for different faults by assumption a prior distribution and then using a Bayesian model for the reliability. The form of such a prior is selected for mathematical reasons, in order to obtain a closed form solution for the corresponding posterior.

Reliability Growth Continues with Additional Testing Time: It is implicitly assumed that, as test time proceeds, new faults will be uncovered and removed. Hence, the software reliability will increase. This precludes the introduction of new faults as well as the increase in program complexity by the maintenance operation.

Seeded Faults Have the Same Failure Rate as Indigenous: In this approach to software reliability, the developer intentionally introduced a number of faults in the program (e. g., fault "seeding"). Then the testing team "uncovers" some of them during testing, along with other "indigenous" faults (not seeded, but actual programming ones). Based on the number of indigenous and seeded faults uncovered, and estimate of total number of program faults is obtained. This estimation approach assumes that the complexity and location of seeded faults are the same as the complexity and program location of the indigenous faults. This is not necessarily so.

Software Input and User Profiles are Known and Representative: Some software reliability models are input domain profiles, which are difficult to establish. Some users exercise some parts of the code more than others do, establishing a particular user profile. Estimating such profiles constitutes a complex statistical problem, involving multivariate parameter estimation and goodness of fit tests. In addition, these profiles are user dependent, requiring one for each different user.

(Continued on page 8)

(Continued from page 6)

Failure Data Collection is Accurate: In software development, the basic activity is to develop good code. Data collection is usually a peripheral activity that programmers are assigned, in addition to their work. Data collection forms (problem reports, etc.) are complicated and data elements such as exact times, etc. may not be recorded accurately.

In addition, other issues associated with software development and model use, impact software reliability model assessment. Some of them are:

Software Doesn't Wear Out with Time: This has always been a key difference between software and hardware reliability. Hardware devices "age". Software also "ages" – just in a different way! As time proceeds and software maintenance occurs, new functions, modules, hardware, capabilities, are added/ modified. Its complexity increases to the point, that it becomes more economic to "retire" it than to continue maintaining it. This process is, conceptually, akin to hardware "aging" processes.

Development Phases and Fault Exposure: In different software development phases, different types of faults are uncovered. Hence, combining failure data from different phases for model input may be detrimental to the overall software reliability estimation.

Experimentation: Many software development experiments undertaken to assess software reliability models and methods have been implemented using very specific problems and subjects. This situation poses restrictions on the extrapolation of results. The experimental subjects are usually students or preselected professional teams. The problems are usually theoretical, or replications of available real life ones. Neither has been randomly selected and may be far from representative. Experimental results are still useful and provide valuable insight, but care should be exercised in their interpretation and especially in extrapolation.

Additional Factors Not Accounted For: Most software models are affected by project requirements, software environment and documentation, user profile, programmer and management experience, and other factors not accounted for in the model. Their contribution to the unreliability of the software program is therefore not reflected in the model results.

Initial Reliability Estimations: In some software models, initial estimates are a function of (1) the processor speed, (2) programming language used (via expansion rates) and (3) error exposure rate. Program size cancels out and does not constitute a factor at this early time. Other previously mentioned factors that also affect software complexity and reliability are not included in this initial estimation.

Fault Exposure Ratios: These ratios, used for initial estimation, were obtained several years ago. In a rapidly advancing area such as software programming, where new languages, new environments (e.g., visual) and technologies are coming out ever day, such fault exposure ratios ma no longer be representative. In addi tion, they were obtained from spe cific environments and projects o the past and may not represent th projects and new application areas o today.

Language Exposure Ratios: Thes ratios are subject to the criticisn making of fault exposure ratios. I addition, new programming lan guages (e.g., Java) have appeare recently for which accurate languag exposure ratios may not yet be avail able.

Having a Large Pool of Softwar Reliability Models from Which t Choose: This constitutes an addi tional and serious problem. Since n single model has been complete established, software develope must choose one. For example, th developer may try fitting severa models and then choose the mos accurate among them, based on pa behavior. However, can one be sur that past behavior always guarantee a model's correct future behavior Model selection is not an easy task.

Some Suggestions to Improve Software Reliability Modeling

Software reliability models are use to assess the end result of the sof ware development process. This f nal result (program) is a function c at least three broad factors: peopl project, and environment. The pec ple include programmers, manage ment, testers, etc. The project is re resented by its characteristics: siz complexity, requirements, function interfaces, etc. The environment ir cludes all the characteristics of the software development shop: management style, software tools, methods, etc. This author believes that, in addition to using and improving the software reliability models, resources should also be dedicated to obtaining a better understanding of one's own software organization (strengths and weaknesses) and to improve it by training its people and readjusting its methods.

In-depth forensic analysis of an organization's past work will reveal its strong and weak points. It will also provide for assessing key components of the three factors mentioned in the previous paragraph. This author also proposed that error prevention, rather than correction, be emphasized. Organizational improvements based on the results of forensic analysis may provide substantial mid and long-range software reliability gains.

It is known that about 70% of software faults result from problems introduced during the requirements definition and design phases (including software reuse problems). Dedicating more time and staff to better understanding, stating and conveying to programmers the special requirements and design issues of each project will pay off at the end. Better training, software tools, programmer and resource time management may also contribute to reducing programming stress and thus software errors.

Finally, forensic analysis may also show that improvements are needed for important concepts such as faults avoidance, prevention techniques (tools, training), fault tolerance (recovery blocks, n-version programming), fault detection/correction (walkthroughs).

Jorge Luis Romeu is a Senior Engineer with IIT Research Institute's (IITRI) Assurance Technology Center in Rome, NY, where he has supported RAC, DACS and AMPTIAC. He has 30 years of experience applying statistical and operations research methods in teaching, research and consulting. Romeu has worked in petrochemical, construction, agricultural, reliability, software engineering and pattern recognition applications of statistics and O.R. He was a mathematics faculty at SUNY, where he retired Emeritus after fourteen years of teaching statistic and computer science. Romeu, a Fullbright Scholar to Mexico, has taught statistics, simulation and operations research at several universities abroad. He is an Adjunct for the Engineering Program at Syracuse University, where he teaches statistics and operations research. In addition, Romeu has developed and taught workshops and training courses for practicing engineers and statistics faculty and has published several articles on applied statistics and statistical education. In 1997 he obtained the Saaty Award for Best Applied Statistics Paper published in the American Journal of Mathematics and Management Sciences (AJMMS). He is the lead author of the book A Practical Guide to Statistical Analysis of Materials Property Data. Romeu hold a Ph.D. in Operations Research, is a Chartered Statistician Fellow of the Royal Statistical Society, a member of the American Statistical Association and the Institute for Operations Research and Management Sciences.