JOURNAL OF AEROSPACE COMPUTING, INFORMATION, AND COMMUNICATION
Vol. 7, April 2010

Tutorial on Hardware and Software Reliability, Maintainability, and Availability

Norman Schneidewind*
Naval Postgraduate School, Pebble Beach, California 93953

DOI: 10.2514/1.36017

Software-based systems have become the dominant player in the computer systems world. Since it is imperative that computer systems operate reliably, considering the criticality of software, reliability is a critical property, particularly in safety critical systems. Software and hardware do not operate in a vacuum. Therefore, both software and hardware are addressed in this tutorial in an integrated fashion. The narrative of the tutorial is augmented with illustrative solved problems.

It is important for an organization to have a disciplined process if it is to produce high reliability software. This process uses a life cycle approach to software reliability that takes into account the risk to reliability due to requirements changes. A requirements change may induce ambiguity and uncertainty in the development process that cause errors in implementing the changes. Subsequently, these errors may propagate through later phases of development and maintenance. In view of the life cycle ramifications of the software reliability process, maintenance is included in this tutorial. Furthermore, because reliability and maintainability determine availability, the latter is also included.

I. Introduction

COMPUTER systems, whether hardware or software, are subject to failure. Precisely, what is a failure? It is defined as: The inability of a system or system component to perform a required function within specified limits. A failure may be produced when a fault is encountered and a loss of the expected service to the user results [1]. This brings us to the question of what is a fault? A fault is defect in the hardware or computer code that can be the cause of one or more failures [1]. Software-based systems have become the dominant player in the computer systems world. Since it is imperative that computer systems operate reliably, considering the criticality of software, particularly in safety critical systems. Software and hardware do not operate in a vacuum. Therefore, both software and hardware are addressed in this tutorial in an integrated fashion. The narrative of the tutorial is augmented with illustrative solved problems.

It is important for an organization to have a disciplined process if it is to produce high reliability software. This process uses a life-cycle approach to software reliability that takes into account the risk to reliability due to requirements changes. A requirements change may induce ambiguity and uncertainty in the development process that cause errors in implementing the changes. Subsequently, these errors may propagate through later phases of development and maintenance [2]. In view of the life-cycle ramifications of the software reliability process, maintenance is included in this tutorial. Furthermore, because reliability and maintainability determine availability, the latter is also included.

Received 3 December 2007; accepted for publication 20 October 2009. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States. Copies of this paper may be made for personal or internal use, on condition that the copier pay the $10.00 per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923; include the code 1542-9423/09 $10.00 in correspondence with the CCC. This material is a work of the U.S. Government and is not subject to copyright protection in the United States.
* Fellow of the IEEE, IEEE Congressional Fellow, US Senate 2005, ieeelife@yahoo.com

II. Reliability Basics

To set the stage for discussing software and hardware model, the following definitions and concepts are provided:

1) *Component*: any hardware or software entity, such as a module, subsystem, or system
2) t: operating time
3) $P(T \leqslant t)$: probability that operating time T of a component is at most t (also known as cumulative distribution function)
4) λ : failure rate (software or hardware failure rate)
5) *Reliability* $R(t) : P(T > t)$: probability of software or hardware surviving for $T > t = 1 - P(T \leqslant t)$ [3]
6) *Hazard function*: Letting operating time t have the probability density function $p(t)$, the *instantaneous failure rate* at time t, is defined as follows

$$h(t) = p(t)/R(t) \quad [3] \tag{1}$$

where $p(t)$ is defined as the probability that a failure will occur in the interval $(t, t+1)$.

The hazard function is frequently described in reliability literature, but a reliability metric that is more practical for calculations with empirical data is the failure rate $f(t)$. This is defined as the number of failures $n(t)$ in the interval t divided by $t : f(t) = n(t)/t$. The reason the hazard function may be impractical, when dealing with empirical data, is that the probability density function $p(t)$ may not be known.

III. Hardware Reliability

The exponential failure distribution with constant failure rate is particularly applicable to hardware reliability because it is assumed that the failure rate remains constant after the initial burn in period and before wear out occurs.

Exponential Failure Distribution ($\lambda e^{-\lambda t}$)*:*

This distribution has a constant failure rate λ. The exponential distribution is the only failure distribution that has a constant failure rate λ and a constant hazard function $h(t)$ in the operations phase of the life cycle. This failure rate is equal to $1/\bar{t}$, where $\bar{t}$ is the mean time to failure (MTTF).

Then, the reliability is given by

$$R(t) = e^{-\lambda t} \tag{2}$$

Then using Eq. (1), the hazard function for exponentially distributed failures is given by

$$h(t) = p(t)/R(t) = \lambda e^{-\lambda t}/e^{-\lambda t} = \lambda. \tag{3}$$

Then adapting Eq. (2) to use MTTF, Eq. (4) is produced as follows:

$$R(t) = e^{-(t/\bar{t})} \tag{4}$$

If we wish to solve for t for a given value of $R(t)$, Eq. (4) is solved for t as follows:

$$t = -\ln(R(t))\bar{t} \tag{5}$$

Problem (*Specifications*)*:*

1) Hardware in a computer system should have an expected (*mean*) life $\bar{t} > 100000$ h (MTTF) at a reliability of $R(t) = 0.85$. What is the minimum number of hours t the computer system would have to survive to meet these specifications?
2) If the hardware should have a 0.85 probability of surviving (i.e., reliability) for $t > 50000$ h, what is the MTTF required to meet these specifications?

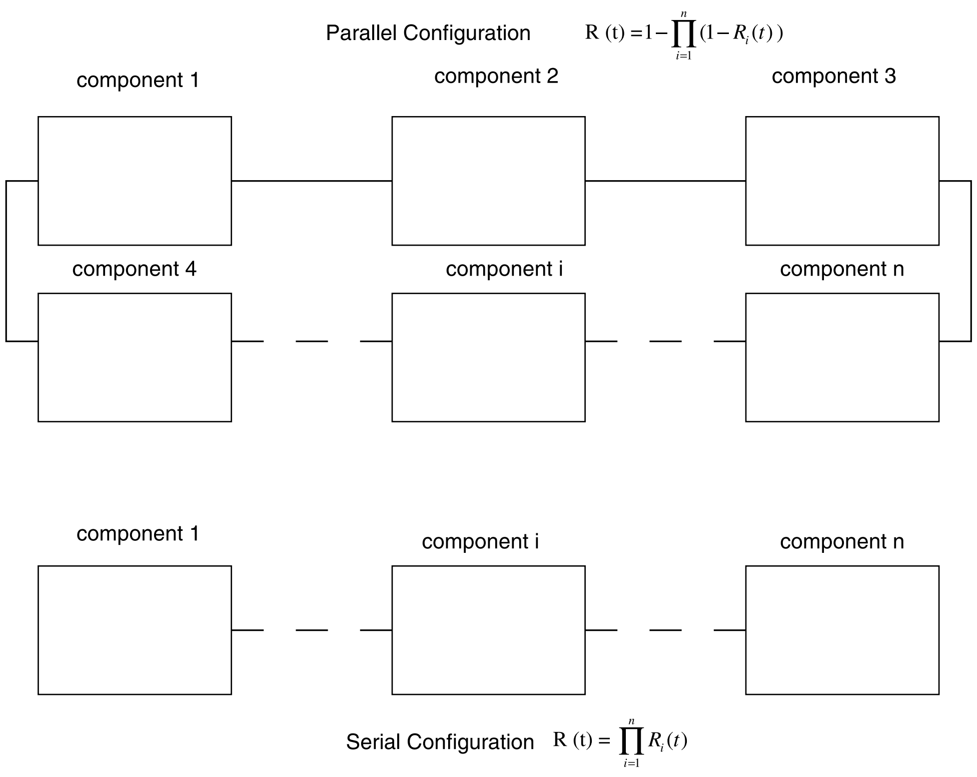
Solution:

1) Use Eq. (5) to compute t as follows:

$$t = -\ln(0.85)(100000) = -(-0.1625)(100000) = 16250\,\text{h}$$

2) Solve Eq. (5) for $\bar{t}$ as follows:

$$t = t/[-\ln(R(t))] = 50000/[-\ln(0.85)] = 307692\,\text{h}$$



Parallel Configuration $R(t) = 1 - \prod_{i=1}^{n} (1 - R_i(t))$
component 1
component 2
component 3
component 4
component i
component n
component 1
component i
component n
Serial Configuration $R(t) = \prod_{i=1}^{n} R_i(t)$

Parallel Hardware: Components Physically Connected in Parallel
Parallel Software: Components Execute Concurrently in Time
Serial Hardware: Components Physically Connected in Series
Serial Software: Components Execute Serially in Time

Fig. 1 Parallel and serial reliability configurations.

IV. Multiple Component Reliability Analysis

Since the majority of computer systems in industry employ multiple components, the reliability analysis must be focused on predicting reliability for these systems. Hardware (and software) components can be operated in serial or hardware configurations. In hardware, the differences are more obvious because of the physical connection between components. In software, the difference is not obvious because there is no physical connection. The difference is based on how the components execute, as indicated in Fig. 1.

A. Parallel System

As Fig. 1 shows, the purpose of a parallel system is to provide a redundant configuration so that if one component fails, another component can take its place, thus increasing reliability. The reliability of a single component i, operating for a time t, is designated by $R_i(t)$. The unreliability is then $(1 - R_i(t))$.

Referring to Fig. 1, the reliability of n components operating in parallel is given by

$$R(t) = 1 - \prod_{i=1}^{n} (1 - R_i(t)) \quad [4] \tag{6}$$

This equation is obtained by observing that the unreliability of n components in parallel is computed by the product of the individual component unreliabilities. Then, the reliability of n components is obtained by subtracting this product from "1".

The most common parallel configuration involves using two components, so using Eq. (6) and some algebraic manipulation, the reliability of two components operating in parallel is given by

$$R(t) = [R_1(t) + R_2(t)] - [R_1(t)R_2(t)] = 1 - [(1 - R_1(t))(1 - R_2(t))] \tag{7}$$

If both components have the same reliability, then

$$R(t) = 2R(t) - R^2(t) \tag{8}$$

A traditional assumption in reliability is that the time between failures is exponentially distributed [3]. This is based on the idea that there is a higher probability of small times between failures and a low probability of large times between failures. Therefore, when failures are exponentially distributed with failure rate λ, then the reliability in Eq. (8) becomes as follows:

$$R(t) = 2e^{-\lambda t} - e^{-2\lambda t} \tag{9}$$

MTTF refers to the average time t_o the next failure [4]. It is a common metric for hardware reliability because the physics of failures is well understood. However, it can be misleading because equipment will fail at specific times and not according to a mean value! MTTF is even less applicable for software because the distribution of time when software fails can be erratic. Before proceeding further, it is important to note that just because the *distribution* of failure times for both hardware and software is a better metric of reliability, does not mean that MTTF and mean time between failure (MTBF) (see below) are not used! These metrics have become so embedded in the lore of reliability that it is imperative to describe their usage.

In the case of hardware, MTTF is used when components are not repaired (i.e., replaced). In other words, with no repair, the time to next failure is *direct*, with no intervening repair time. In nonredundant software systems, the software must be repaired to continue operation, unless the fault causing the failure is trivial. Therefore, MTTF is not completely applicable for this type of software. On the other hand, for redundant software systems (e.g., fault tolerant), MTTF is applicable, with the caveat noted above.

MTBF, defined as the average time *between* failures, is used when components are repaired [4]. Thus, it is the time between failures, with an intervening repair time.

The general form for MTTF, whether hardware or software, is derived from the reliability function $R(t)$, as follows:

$$\int_0^\infty R(t)\,\mathrm{d}t \quad [3].$$

Therefore, the MTTF for the two component parallel arrangement, from Eq. (9), is given by

$$\bar{t} = \int_0^\infty R(t)\,\mathrm{d}t = \int_0^\infty (2e^{-\lambda t} - e^{-2\lambda t})\,\mathrm{d}t = \left[\frac{-2e^{-\lambda t}}{\lambda}\right]_0^\infty - \left[\frac{-e^{-2\lambda t}}{2\lambda}\right]_0^\infty = \frac{1.5}{\lambda} \tag{10}$$

B. Series System

Often, particularly for software systems, in order to produce a conservative prediction of reliability, components are assumed to operate in series for the *purpose* of reliability prediction [5]. This represents the weakest link in the chain concept (i.e., the system would fail if *any* component fails).

Then, this conservative reliability approach of n components operating in series is given by

$$R(t) = \prod_{i=1}^{n} R_i(t) \quad [4] \tag{11}$$

Using Eq. (11), the reliability of two components operating in series, with equal reliabilities, is given by Eq. (12), if the failures are exponentially distributed as follows:

$$R(t) = R^2(t) = e^{-2\lambda t} \tag{12}$$

Then, the MTTF for the series arrangement is given by

$$\bar{t} = \int_0^{\infty} R(t)\,\mathrm{d}t = \int_0^{\infty} e^{-2\lambda t} = \frac{-[e^{2\lambda t}]_0^{\infty}}{2\lambda} = \frac{1}{2\lambda} \tag{13}$$

It is often of interest to predict the improvement that can be achieved by using a parallel rater that a series configuration. Then, using Eqs. (9) and (12), the improvement of the parallel system reliability over a series system, for two components, can be shown as follows:

$$RI = (2e^{-\lambda t} - e^{-2\lambda t}) - e^{-2\lambda t} = 2(e^{-\lambda t} - e^{-2\lambda t}) \tag{14}$$

In addition, using Eqs. (10) and (13), the increase in *mean time to failure* can be shown to be

$$\frac{1.5}{\lambda} - \frac{1}{2\lambda} = \frac{1}{\lambda} \tag{15}$$

It is not only the improvement RI that is of interest. In addition, the rate of change of RI will reveal the rate of change of RI that will indicate how fast the improvement will occur. Then, differentiating RI (Eq. (14)) with respect to t, and setting it equal to 0, gives us Eq. (16) as follows:

$$\frac{\mathrm{d}(RI)}{\mathrm{d}(t)} = 2(-\lambda)e^{-\lambda t} - 2(-2\lambda)e^{-2\lambda t} = 0 \tag{16}$$

Noting that the derivative of Eq. (16) is negative, because the first negative term decreases less rapidly that the second positive term, we know that Eq. (16) will provide a value of t that will maximize RI.

Then solving Eq. (16) for t, yields t^* as the value of t that maximizes RI as follows:

$$t^* = -(1/\lambda)(\log(0.5)) \tag{17}$$

Problem:

For a computer system with failure rate of $\lambda = 0.001$ failures per hour and *time to failure* listed below, plot Eqs. (9), (12), and (14) on the same graph, vs t, and indicate the value of $t = t^*$ that maximizes RI, assuming an exponential distribution of time to failure t.

t (h)
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1400
1500
1600
1700
1800
1900
2000

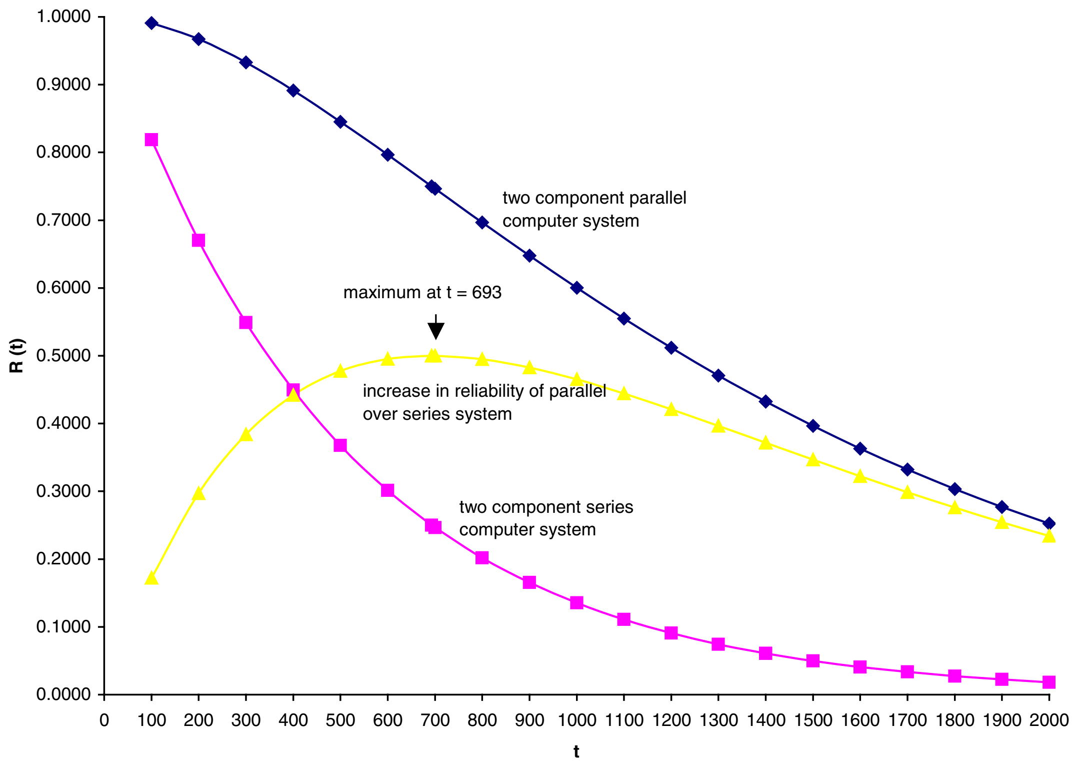


Fig. 2 Reliability $R(t)$ vs operating time t.

Solution:

Figure 2 contrasts parallel reliability, serial reliability, and the improvement of parallel over serial reliability. The figure also delineates the operating time where the greatest improvement is achieved. A reliability analyst, using this plot, would understand that at $t = 683$ h the greatest gain in reliability would occur and that at operating times either below or above this value, the gain falls off rapidly.

C. Number of Components that are Needed to Achieve Reliability Goals

When the reliability of a system is required to be $R_n(t)$ in a parallel configuration, the required number n components, each with a reliability of $R(t)$ equal to

$$R_n(t) = 1 - (1 - R(t))^n \tag{18}$$

Solving Eq. (18) for n yields

$$n = \frac{\ln(1 - R_n(t))}{\ln(1 - R(t))} \tag{19}$$

Problem:

How many components are needed to operate in parallel, if each component has a reliability of $R(t) = 0.80$, and it is desired to achieve a system reliability of $R_n(t) = 0.98$.

Solution:

Solving Eq. (18) for n yields

$$n = \frac{\ln(1 - R_n(t))}{\ln(1 - R(t))} = \frac{\ln(0.02)}{\ln(0.20)} = \frac{-3.912}{-1.609} = 2.43 \text{ components} = 3$$

V. Computer System Maintenance and Availability

Preventive maintenance strategy: Routine inspection and service activities designed to detect potential failure conditions and make adjustments and repairs that will help prevent major operating problems [6].

Two fundamental preventive strategies are differentiated, *time- and condition-based preventive maintenance*. In time-based preventive maintenance, after a fixed period of time, a component is serviced or overhauled, independent of the wear of the component at that moment. In condition-based preventive maintenance, one inspects a condition of a component, according to some schedule. If the condition exceeds a specified critical value, preventive maintenance is performed. With regard to the timing of the inspections, there are two variants, *constant*- and *condition-based inspection interval*. If one applies a constant inspection interval, an inspection is performed after a fixed period of time, analogous to time-based preventive maintenance. When deciding to perform a condition-based inspection interval, the time until the next inspection depends on the condition in the previous inspection. If the condition in the previous inspection was good, the time until the next inspection will be quite long. If the condition in the previous inspection was bad, the time until the next inspection will be quite short.

Predictive maintenance strategy: Predictive maintenance is a condition-based approach to maintenance. The approach is based on predicting component condition in order to assess whether components will fail during some future period, and then taking action to avoid the consequences of the failures.

VI. Component Availability

Now, in order to compute component availability, a number of quantities must be defined as follows:

1) t_p is the duration of component preventive maintenance

2) t_o is the duration of component operation

3) t_f is the duration of component failure

4) t_r is the duration of component repair

5) f_p is the frequency of component preventive maintenance

6) f_o is the frequency of component operation

7) f_f is the frequency of component failures

8) f_r is the frequency of component repair

9) $\bar{t}$ is the mean time to component failure

With the definitions in hand, availability A, can be computed as follows:

$$A = \frac{f_0 t_0}{f_0 t_0 + f_p t_p + f_f t_f + f_r t_r} \tag{20}$$

Availability is also expressed by

$$A = \frac{\bar{t}}{\bar{t} + t_r} \tag{21}$$

These quantities are portrayed graphically in Fig. 3.

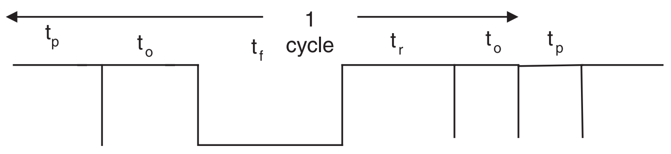
Problem:

Given the data below for a system, compute the availability A.

Duration of operation: $t_o = 10$

Duration of preventive maintenance: $t_p = 1$

Duration of failure: $t_f = 0.5$



t_p: duration of preventive maintenance
t_o: duration of operation
t_f: duration of failure
t_r: duration of repair

Fig. 3 Computer maintenance process.

Duration of repair: $t_r = 2$
Frequency of operation: $f_o = 20$
Frequency of preventive maintenance: $f_p = 20$ (for every operation there is preventive maintenance)
Frequency of failure: $f_f = 4$
Frequency of repair: $f_r = 4$ (for every failure there is a repair)
Then, using Eq. (20) we have

$$A = \frac{f_0 t_0}{f_0 t_0 + f_p t_p + f_f t_f + f_r t_r} = \frac{(20)(10)}{(20)(10) + (20)(1) + (4)(0.5) + (4)(2)} = 0.870 \quad \text{(system availability)}$$

VII. Software Reliability Engineering Risk Analysis

Software reliability engineering (SRE) is an established discipline that can help organizations improve the reliability of their products and processes. The IEEE/AIAA defines SRE as “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 IEEE/AIAA recommended practice is a composite of models and tools and describes the “what and how” of SRE [1]. It is important for an organization to have a disciplined process if it is to produce high reliability software. The process includes a life-cycle approach to SRE that takes into account the risk to reliability due to requirements changes. A requirements change may induce ambiguity and uncertainty in the development process that cause errors in implementing the changes. Subsequently, these errors may propagate through later phases of development and maintenance. These errors may result in significant risks associated with implementing the requirements. For example, reliability risk (i.e., risk of faults and failures induced by changes in requirements) may be incurred by deficiencies in the process (e.g., lack of precision in requirements). Figure 4 shows the overall SRE closed-loop holistic process.

In the figure, risk factors are metrics that indicate the degree of risk in introducing a new requirement or making a requirements change. For example, in the NASA Space Shuttle, program size and complexity, number of conflicting requirements, and memory requirements have been shown to be significantly related to reliability (i.e., increases in these risk factors are associated with decreases in reliability) [7]. Organizations should conduct studies to determine what factors are contributing to reliability degradation. Then, as in Fig. 4, organizations could use feedback from operations, testing, design, and programming, to determine which risk factors are associated with reliability, and revise requirements, if necessary. For example, if requirements risk assessment finds that through risk factor analysis, that defects are occurring because of excessive program size, design, and programming would receive revised requirements to modularize the software.

A reliability risk assessment should be based on the risk to reliability due to software defects or errors caused by requirements and requirements changes. The method to ascertain risk based on the number of requirements and the impact of changes to requirements is inexact, but nevertheless, it necessary for early requirements assessments of large-scale systems.

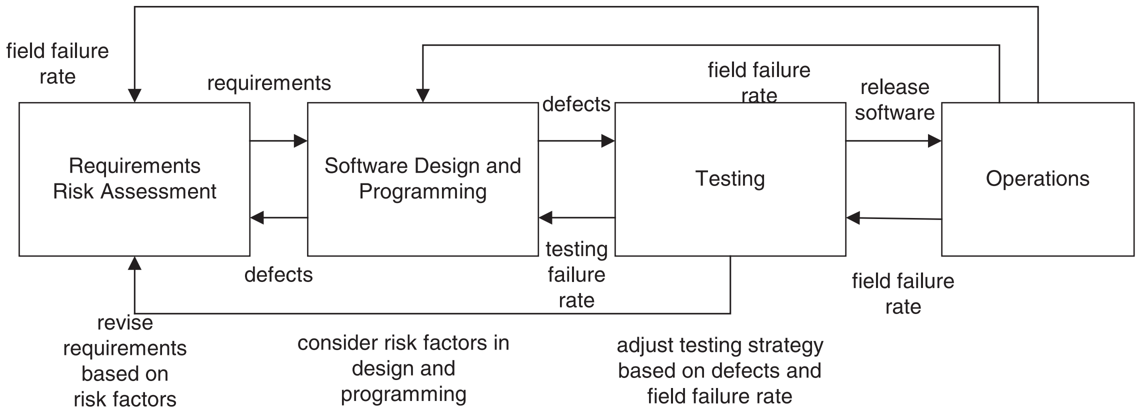

field failure rate
requirements
Requirements Risk Assessment
Software Design and Programming
defects
field failure rate
release software
Testing
Operations
defects
testing failure rate
field failure rate
revise requirements based on risk factors
consider risk factors in design and programming
adjust testing strategy based on defects and field failure rate

Fig. 4 Software reliability engineering risk analysis.

A. Criteria for Safety

In safety critical systems, in particular, safety criteria are used, in conjunction with risk factors, to assess whether a system is safe to operate. Two criteria are used. One is based on predicted remaining failures in relation to a threshold and the second is based on predicted time to next failure in relation to mission duration [8]. These criteria are computed as follows:

compute predicted *remaining failures* $r(t_t) < r_c$, where r_c is a specified remaining failures critical value, and compute predicted *time to next failure* $T_F(t_t) > t_m$, where t_m is mission duration.

Once $r(t_t)$ has been predicted, the risk criterion metric (RCM) for *remaining failures* at total test time t_t is computed in Eq. (22) as follows:

$$\text{RCM } r(t_t) = \frac{r(t_t) - r_c}{r_c} = \frac{r(t_t)}{r_c} - 1 \tag{22}$$

In order to illustrate the remaining failure risk criterion in relation to the predicted maximum number of failures in the software $F(\infty)$, the following parameter is needed:
$p(t)$: Fraction of remaining failures predicted at time t_t in Eq. (23) as follows:

$$p(t_t) = \frac{r(t_t)}{F(\infty)} \tag{23}$$

The RCM for *time to next failure* at total test time t_t is computed in Eq. (24) based on the predicted time to next failure in Eq. (25) [7] as follows:

$$\text{RCM } T_F(t_t) = \frac{t_m - T_F(t_t)}{t_m} = 1 - \frac{T_F(t_t)}{t_m} \tag{24}$$

$$T_F(t_t) = -\frac{1}{\beta}\log\left[1 - ((F(t_t) + X_{s-1}))\left(\frac{\beta}{\alpha}\right)\right] + (s-1) \quad \text{for } (F(t_t) + X_{s-1})\left(\frac{\beta}{\alpha}\right) < 1 \tag{25}$$

where β and α are parameters estimated from the failure data. Parameter β is the rate of change of the failure rate and α is the initial failure rate. The parameter s is the starting failure interval count that produces the most accurate reliability predictions, and X_{s-1} is the observed failure count in the range of the test data from s to t_t. Finally, $F(t_t)$ refers to the specified number of failures—usually one—that is used in the prediction.

Problem:
Part 1 (*remaining failures risk*):
Using one of the models in [1] recommended for initial use and either the software reliability tool statistical modeling and estimation of reliabiltiy functions for software (SMERFS) or CASRE, compute Eqs. (22) and (23)

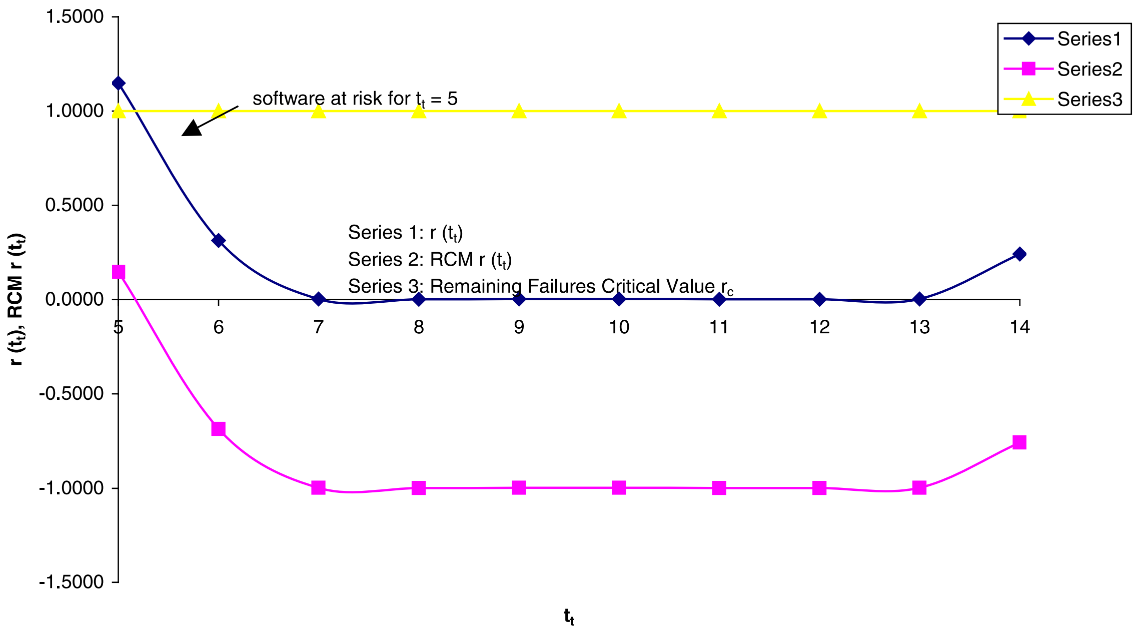


Fig. 5 Predicted remaining failures $r(t_t)$ and risk criterion metric RCM $r(t_t)$ vs test time t_t for NASA Space Shuttle Release OI6.

to produce Figs. 5 and 6 for the NASA Space Shuttle software release OI6. The failure counts for each value of test time t_t for OI6 is shown in Table 1. Once you have inputted a text file of these counts, one at a time, the software reliability tools will compute $r(t_t)$ and $F(\infty)$ for each of the ten cases. The tools can be downloaded at http://www.slingcode.com/smerfs/ for SMERFS and at http://www.openchannelfoundation.org/projects/CASRE 3.0 for CASRE.

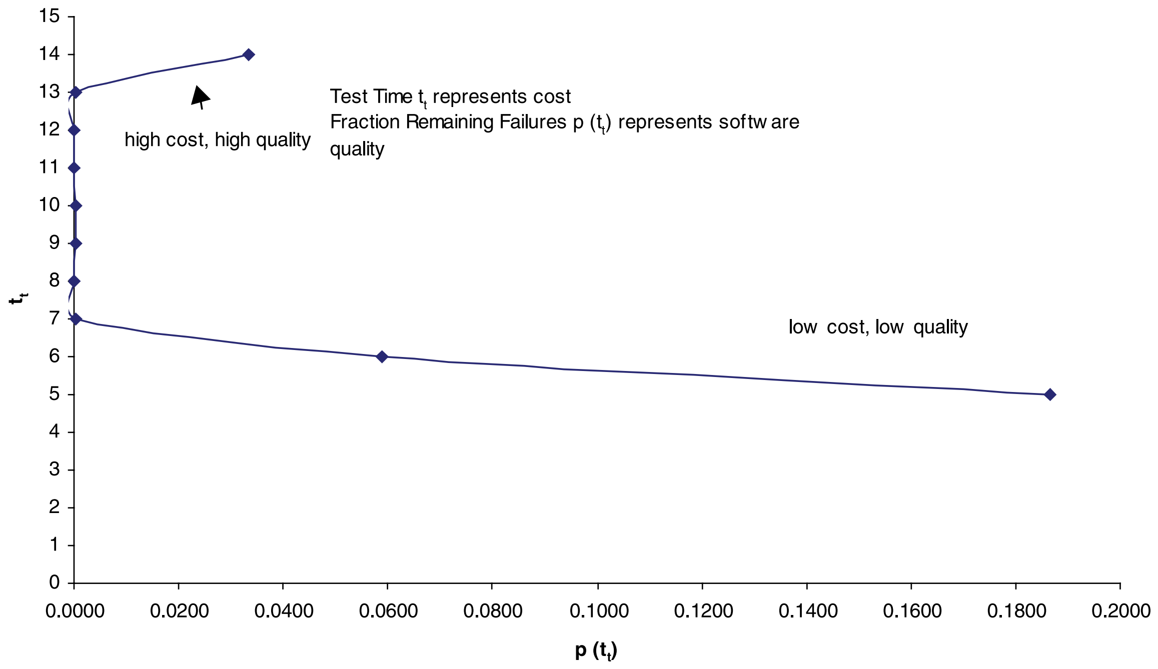


Fig. 6 Cost of testing t_t vs software quality $p(t_t)$ for NASA Space Shuttle Release OI6.

Table 1 Failure counts for NASA space shuttle software release OI6

t_t									
5	6	7	8	9	10	11	12	13	14
0	0	0	0	0	0	0	0	0	0
2	2	2	2	2	2	2	2	2	2
1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0
		0	0	0	0	0	0	0	0
			0	0	0	0	0	0	0
				0	0	0	0	0	0
					0	0	0	0	0
						0	0	0	0
							0	0	0
								1	1
									1

Part 2 *(time to next failure risk):* In this part, a specific recommended model in [1] is used [8] in order to illustrate the use of this model's predicted time to next failure and the application of the prediction to evaluating the risk of not satisfying the mission duration requirement, as formulated in Eq. (24). Other recommended models could be used to perform the analysis.

After using one of the tools to estimate the parameters in Eq. (24), predict $T_F(t_t)$ for one more failure and plot it and the RCM, in Fig. 7, as a function of the test time t_t in Table 1.

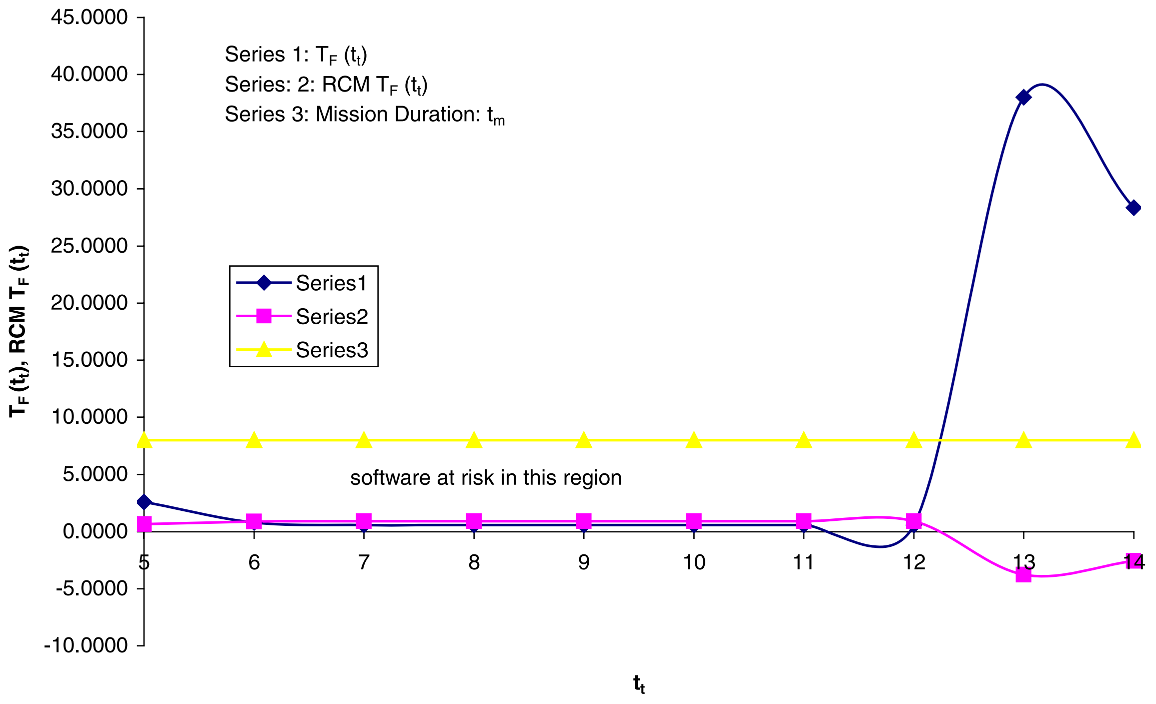


Fig. 7 Predicted time to next failure $T_F(t_t)$ and risk criterion metric RCM $T_F(t_t)$ vs test time.

Solution to Part 1 :

Figure 5 delineates the test time equal to 5, where the risk of exceeding the critical value of remaining failures is unacceptable. Therefore, a test time of at least 6 is required. Figure 6 shows how the software reliability analyst can do a tradeoff of the cost of testing version the quality of software produced by testing. Since test time is usually directly related to cost, the figure indicates that a very high cost would be incurred for attempting to achieve almost fault-free software. Therefore, tolerating a fraction remaining failures of about 0.0600 would be practical.

Solution to Part 2 :

Switching now to the evaluation of risk with respect to time to next failure, Fig. 7 demonstrates that unless the test time is greater than 12, the time to failure will not exceed the mission duration. The engineer using such a plot would use a mission duration appropriate for the software being tested. The concept behind Fig. 7 is that the software should be tested sufficiently long such that the RCM goes negative.

VIII. Parameter Analysis

It is possible to assess risk after the parameters α and β have been estimated by a tool, such as SMERFS and CASRE [1], but *before* predictions are made. An example is provided in Fig. 8 where remaining failures and its risk criterion, are plotted against the parameter ratio β/α [7]. The reason for this result is that a high value of β means that the failure rate decreases rapidly, coupled with a low value of α, leads to high reliability. High reliability, in turn means low risk of unsafe software. Furthermore, increasing values of PR are associated with increasing values of test time, thus decreasing risk. Thus, even *before* predictions are made, it is possible to know how much test time is required to yield predictions that the software is safe to deploy. In Fig. 8, this time is 6 corresponding to the same result in Fig. 7. A cautionary note is that the foregoing analysis is an a priori assessment of likely risk results and does not mean, necessarily, that high values of β/α will lead to low risk.

Problem:

After obtaining estimates of β and α using one of the reliability tools, for each value of test time in Table 1, plot Fig. 8 to show that risk decreases with the parameter ratio.

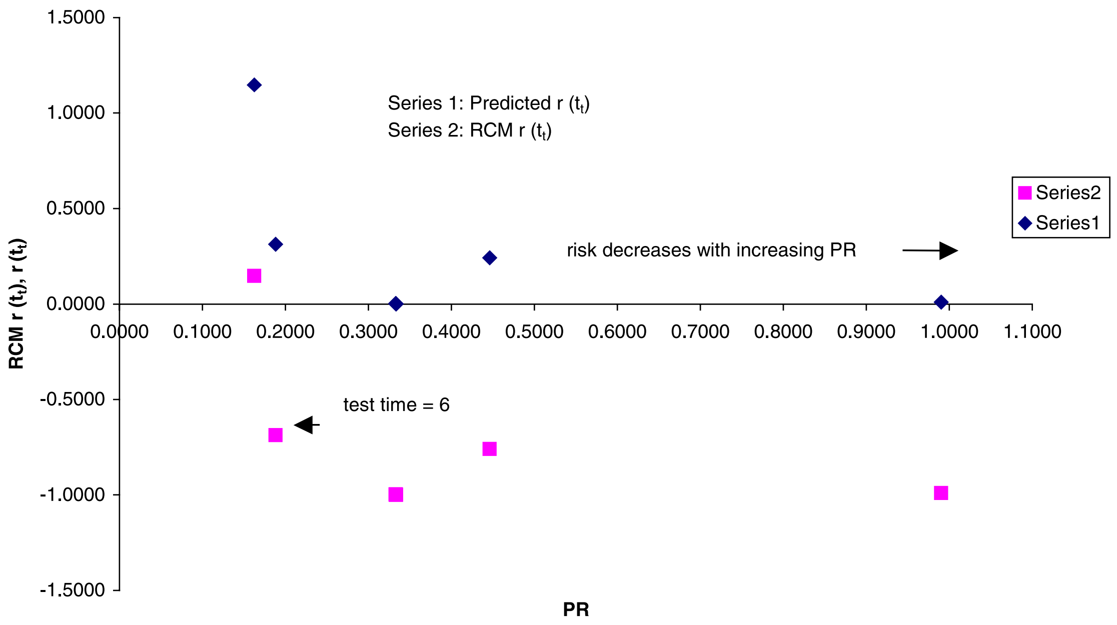


Fig. 8 Risk criterion metric RCM $r(t_t)$ and remaining failures $r(t_t)$ vs parameter ratio PR (β/α) for NASA Space Shuttle software release OI6.

IX. Overview of Recommended Software Reliability Models

In [1] it is stated that there are “initial models” recommended for using on an application, but if these models do not satisfy the organization’s need, other models that are described in the document could be used. Since this tutorial has included several practice problems, based in part on models, an overview is presented of two of the initially recommended models: Musa-Okumoto and Schneidewind. The third model — Generalized exponential — involves a great amount of detail that cannot be presented here. For readers interested in more detail on these models or to learn about the other models, the recommended practice can be consulted.

A. Musa-Okumoto Logarithmic Poisson Execution Time Model

Objectives

The logarithmic Poisson model is applicable when the testing is done according to an operational profile that has variations in frequency of application functions and when early fault corrections have a greater effect on the failure rate than later ones. Thus, the failure rate has a decreasing slope. The operational profile is a set of functions and their probabilities of use [9].

Assumptions

The assumptions for this model are as follows:

1) The software is operated in a similar manner as the anticipated operational usage
2) Failures are independent of each other
3) The failure rate decreases exponentially with execution time

Structure

From the model assumptions, we have:

$\lambda(t)$ = failure rate after t amount of execution time has been expended $\lambda_0 e^{-\theta\mu(t)}$

The parameter λ_0 is the initial failure rate parameter and θ is the failure rate decay parameter with $\theta > 0$.

Using a reparameterization of $\beta_0 = \theta^{-1}$ and $\beta_1 = \lambda_0\theta$, then the estimates of β_0 and β_1 are made, as shown in, according to Eq. (26) and (27), respectively, as follows:

$$\hat{\beta}_0 = \frac{n}{\ln(1+\hat{\beta}_1)t_n} \tag{26}$$

$$\frac{1}{\hat{\beta}_1}\sum_{i=1}^{n}\frac{1}{1+\hat{\beta}_1 t_i} = \frac{n t_n}{(1+\hat{\beta}_1 t_i)\ln(1+\hat{\beta}_1 t_i)} \tag{27}$$

Here, t_n is the cumulative CPU time from the start of the program to the current time. During this period, n failures have been observed. Once estimates are made for β_0 and β_1, the estimates for θ and λ_0 are made in Eq. (28) and (29) as follows:

$$\hat{\theta} = \frac{1}{n}\ln(1+\hat{\beta}_1 t_n) \tag{28}$$

$$\hat{\lambda}_0 = \hat{\beta}_0\hat{\beta}_1. \tag{29}$$

Limitation

The failure rate may rise as modifications are made to the software violating the assumption of decreasing failure rate.

Data Requirements

The required data is either as follows:

1) The time between failures, represented by X_i’s.

The time of the failure nth occurrences, given by $t_n = \sum_{i=1}^{n} X_i$.

Applications

The major applications are described below. These are separate but related applications that, in total, comprise an integrated reliability program.

Prediction: Predicting future failure times and fault corrections

Control: Comparing prediction results with predefined goals and flagging software that fails to meet goals.

Assessment: Determining what action to take for software that fails to meet goals (e.g., intensify inspection, intensify testing, redesign software, and revise process). The formulation of test strategies is also a part of assessment. It involves the determination of priority, duration and completion date of testing, and allocation of personnel, and computer resources to testing.

Reliability predictions

In [4], it is shown that from the assumptions above and the fact that the derivative of the mean value function of failure count is the failure rate function, Eq. (30) is obtained as follows:

$$\hat{\mu}(\tau) = \text{mean number of failures experienced by time } \tau \text{ is expended} = \frac{1}{\hat{\theta}} \ln(\hat{\lambda}_0 \hat{\theta} \tau + 1) \tag{30}$$

Implementation and application status

The model has been implemented by the Naval Surface Warfare Center, Dahlgren, VA as part of SMERFS and in CASRE.

B. Schneidewind Model [8]

Objectives

The objectives of this model are to predict following software reliability metrics:

1) $F(t_1, t_2)$ is the predicted failure count in the range $[t_1, t_2]$
2) $F(\infty)$ is the predicted failure count in the range $[1, \infty]$; maximum failures over the life of the software
3) $F(t)$ is the predicted failure count in the range $[1, t]$
4) $p(t)$ is the fraction of remaining failures predicted at time t
5) $Q(t)$ is the operational quality predicted at time t; the complement of $p(t)$; the degree to which software is free of remaining faults (failures)
6) $r(t_t)$ is the remaining failures predicted at test time t_t
7) t_t is the test time predicted for given $r(t_t)$
8) $T_F(t_t)$ is the time to next failure predicted at test time t_t

Parameters used in the predictions

1) α is the initial failure rate
2) β is the rate of change of failure rate
3) r_c is the critical value of remaining failures used in computing the RCM for remaining failures RCM $r(t_t)$
4) t_m is the mission duration (end time-start time) used in computing the RCM for time to next failure RCM $T_F(t_t)$

The philosophy of this model is that as testing proceeds with time, the failure detection process changes. Furthermore, recent failure counts are usually of more use than earlier counts in predicting the future. Three approaches can be employed in utilizing the failure count data (i.e., number of failures detected per unit of time). Suppose there are t intervals of testing and f_i failures were detected in the ith interval, one of the following is done:

1) Use all of the failures for the t intervals
2) Ignore the failure counts completely from the first $s-1$ time intervals $(1 \leqslant s \leqslant t)$ and only use the data from intervals s through t
3) Use the cumulative failure count from intervals 1 through $s-1$: $F_{s-1} = \sum_{i=1}^{s-1} f_i$

The first approach should be used when it is determined that the failure counts from all of the intervals are useful in predicting future counts. This would be the case with new software where little is known about its failure count distribution. The second approach should be used when it is determined that a significant change in the failure detection process has occurred and thus only the last $t-s+1$ intervals are useful in future failure forecasts. The last approach is an intermediate one between the other two. Here, the combined failure counts from the first $s-1$ intervals and the individual counts from the remaining intervals are representative of the failure and detection behavior for future predictions. This approach is used when the first $s-1$ interval failure counts are not as significant as in the first approach, but are sufficiently important not t_p be discarded, as in the second approach.

Assumptions

1) The number of failures detected in one interval is independent of the failure count in another. *Note*: in practice, this assumption has not proved to be a factor in obtaining prediction accuracy
2) Only new failures are counted
3) The fault correction rate is proportional to the number of faults to be corrected
4) The software is tested in a manner similar to the anticipated operational usage
5) The mean number of detected failures decreases from one interval to the next
6) The rate of failure detection is proportional to the number of failures within the program at the time of test. The failure detection process is assumed to be a nonhomogeneous Poisson process with an exponentially decreasing failure detection rate [7]. The rate is of the form $f(t) = \alpha e^{-\beta(t-s+1)}$ for the tth interval where $\alpha > 0$ and $\beta > 0$ are the parameters of the model

Structure

The method of maximum likelihood (MLE) is used to estimate parameters. This method is based on the concept of maximizing the probability that the true values of the parameters are observed in the failure data [9]. Two parameters are used in the model that have been previously defined as follows: α and β. In these estimates, t is the last observed failure count interval; s is the starting interval for using observed failure data in parameter estimation; X_k is the number of observed failures in interval k; X_{s-1} is the number of failures observed from 1 through $s-1$ intervals; $X_{s,t}$ is the number of observed failures from interval s through t; and $X_t = X_{s-1} + X_{s,t}$. The likelihood function (based on MLE) is then developed as as follows:

$$\begin{aligned}\log L = X_t\left[\log X_t - 1 - \log(1-e^{-\beta t})\right] + X_{s-1}\left[\log\left(1-e^{-\beta(s-1)}\right)\right]\\ + X_{s,t}\left[\log\left(1-e^{-\beta}\right)\right] - \beta\sum_{k=0}^{t-s}(s+k-1)X_{s+k}\end{aligned} \tag{31}$$

Equation (31) is used to derive the equations for estimating α and β for each of the three approaches described earlier. The parameter estimates can be obtained by using the SMERFS or CASRE tools.

Approach 1:

Use all of the failure counts from interval 1 through t (i.e., $s = 1$). Equations (32) and (33) are used to estimate β and α, respectively.

$$\frac{1}{e^{\beta}-1} - \frac{t}{e^{\beta t}-1} = \sum_{k=0}^{t-1} k\frac{X_{k+1}}{X_t} \tag{32}$$

$$\alpha = \frac{\beta X_t}{1-e^{-\beta t}} \tag{33}$$

Approach 2:

Use failure counts only in intervals s through t (i.e., $1 \leqslant s \leqslant t$). Equations (34) and (35) are used to estimate β and α, respectively. (Note that Approach 2 is equivalent to Approach 1 for $s = 1$.)

$$\frac{1}{e^{\beta}-1} - \frac{t-s+1}{e^{\beta(t-s+1)}-1} = \sum_{k=0}^{t-s} k\frac{X_{k+s}}{X_{s,t}} \tag{34}$$

$$\alpha = \frac{\beta X_{s,t}}{1-e^{-\beta(t-s+1)}} \tag{35}$$

Approach 3:

Use cumulative failure counts in intervals 1 through $s - 1$ and individual failure counts in intervals s through t (i.e., $2 \leqslant s \leqslant t$). This approach is intermediate to approach 1 which uses all of the data and Approach 2 that discards "old" data. Equations (36) and (37) are used to estimate β and α, respectively. (Note that Approach 3 is equivalent to Approach 1 for $s = 2$.)

$$\frac{(S-1)X_{s-1}}{e^{\beta(s-1)}-1} + \frac{X_{s,t}}{e^{\beta}-1} - \frac{tX_t}{e^{\beta m}-1} = \sum_{k=0}^{t-s}(s+k-1)X_{s+k} \tag{36}$$

$$\alpha = \frac{\beta X_t}{1-e^{-\beta t}} \tag{37}$$

Limitations

1) Model does not account for the possibility that failures in different intervals may be related
2) Model does not account for repetition of failures
3) Model does not account for the possibility that failures can increase over time as the result of software modifications

These limitations should be ameliorated by configuring the software into versions that, starting with the second version, the next version represents the previous version plus modifications introduced by the next version. Each version represents a different module for reliability prediction purposes. The model is used to predict reliability for each module. Then, the software system reliability is predicted by considering the N modules to be connected in series (i.e., worst case situation), and computing the MTTF for N modules in series [10].

Data requirements

The only data requirements are the number of failures, f_i, where $i = 1, \ldots, t$, per testing interval. A reliability database should be created for several reasons: input data sets will be rerun, if necessary, to produce multiple predictions rather than relying on a single prediction; reliability predictions and assessments could be made for various projects; and predicted reliability could be compared with actual reliability for these projects. This database will allow the model user to perform several useful analyses: to see how well the model is performing; to compare reliability across projects to see whether there are development factors that contribute to reliability; and to see whether reliability is improving over time for a given project or across projects.

Applications

The major model applications are described below. These are separate but related uses of the model that, in total, comprise an integrated reliability program.

Prediction: Predicting future reliability metrics such as remaining failures and time to next failure

Control: Comparing prediction results with predefined reliability goals and flagging software that fails to meet those goals

Assessment: Determining what action to take for software that fails to meet goals (e.g., intensify inspection, intensify testing, redesign software, and revise process). The formulation of test strategies is also part of assessment. Test strategy formulation involves the determination of: priority, duration and completion date of testing, allocation of personnel, and allocation of computer resources to testing.

Risk analysis: Compute risk criterion metrics for remaining failures and time to next failure.

Predict *test time* required to achieve a specified *number of remaining failures* at t_t, $r(t_t)$ in Eq. (38) as follows:

$$t_t = [\log[\alpha/\beta[r(t_t)]]]/\beta \tag{38}$$

Implementation and application status

The model has been implemented in FORTRAN and C++ by the Naval Surface Warfare Center, Dahlgren, VA as part of the SMERFS. In addition, it has been implemented in CASRE. It can be run on an IBM PCs under all Windows operating systems.

Known applications of this model are as follows:

1) IBM, Houston, TX: Reliability prediction and assessment of the onboard NASA Space Shuttle software
2) Naval Surface Warfare Center, Dahlgren, VA: Research in reliability prediction and analysis of the TRIDENT I and II Fire Control Software
3) Marine Corps Tactical Systems Support Activity, Camp Pendleton, CA: Development of distributed system reliability models
4) NASA JPL, Pasadena, CA: Experiments with multimodel software reliability approach
5) NASA Goddard Space Flight Center, Greenbelt, MD: Development of fault correction prediction models
6) NASA Goddard Space Flight Center
7) Hughes Aircraft Co., Fullerton, CA: Integrated, multimodel approach to reliability prediction

X. Summary

The purpose of this tutorial has been two-fold: 1) serve as a companion to the IEEE/AIAA Recommended Practice on Software Reliability and 2) assist the engineer in understanding and applying the principles of hardware and software reliability, and the related subjects of maintainability and availability. Due to the prevalence of software-based systems, the focus has been on learning how to produce high reliability software. However, since hardware faults and failures can cause the highest quality software to fail to meet user expectations, considerable coverage of hardware reliability was provided. Practice problems with solutions were included to provide the reader with real-world applications of the principles that were discussed.

References

[1] IEEE/AIAA P1633™/Draft 13, Recommended Practice on Software Reliability, IEEE, Washington DC, Nov. 2000.

[2] Schneidewind, N. F., "Reliability and Maintainability of Requirements Changes", *Proceedings of the International Conference on Software Maintenance*, Florence, Italy, IEEE, Washington DC, 7–9 Nov. 2001, pp. 127–136.
doi: 10.1109/ICSM.2001.972723

[3] Lyu, M. R., (ed.), *Handbook of Software Reliability Engineering*, IEEE Computer Society Press/McGraw–Hill, Piscataway/New York, 1996.

[4] Musa, J. D., Iannino, A., and Okumoto, K., *Software Reliability: Measurement, Prediction, Application*, McGraw–Hill, New York, 1987.

[5] Keller, T., and Schneidewind, N. F., "A Successful Application of Software Reliability Engineering for the NASA Space Shuttle," *Proceedings of the Software Reliability Engineering Case Studies, International Symposium on Software Reliability Engineering*, Nov. 3, Albuquerque, NM, 4 Nov. 1997, pp. 71–82.

[6] Monks, J. G., *Operations Management*, 2nd ed. McGraw–Hill, New York, 1996.

[7] Schneidewind, N. F., "Risk-Driven Software Testing And Reliability," *International Journal of Reliability, Quality and Safety Engineering*, Vol. 14, No. 2, 2007, pp. 99–132.
doi: 10.1142/S0218539307002532

[8] Schneidewind, N. F., "Reliability Modeling for Safety Critical Software," *IEEE Transactions on Reliability*, Vol. 46, No. 1, March 1997, pp. 88–98.
doi: 10.1109/24.589933

[9] Musa, J. D., *Software Reliability Engineering: More Reliable Software, Faster and Cheaper*, 2nd ed., Authorhouse, Bloomington, IN, 1999.

[10] Schneidewind, N. F., and Keller, T. M., "Applying Reliability Models to the Space Shuttle," *IEEE Software*, July 1992, pp. 28–33.
doi: 10.1109/52.143099

Michael Hinchey
Associate Editor