Software Reliability Theory

Michael Rung-Tsong Lyu, The Chinese University of Hong Kong

Encyclopedia of Software Engineering
Copyright © 2002 by John Wiley & Sons, Inc. All rights reserved.
DOI: 10.1002/0471028959.sof329
Article Online Posting Date: January 15, 2002

Keywords: history; theory; random point process; exponential order statistics; modelingm markov software reliability models; comparisons; emerging techniques

Abstract

Software reliability modeling has, surprisingly to many, been around since the early 1970s with the pioneering works of Jelinski and Moranda, Shooman, and Coutinho. The theory behind software reliability is presented, and some of the major models that have appeared in the literature from both historical and applications perspectives are described. Emerging techniques for software reliability research field are also included. The following four key components in software reliability theory and modeling: *historical background, theory, modeling,* and *emerging techniques* are addressed. These items are discussed in a general way, rather than attempting to discuss a long list of details.

Software reliability modeling has, surprisingly to many, been around since the early 1970s with the pioneering works of Jelinski and Moranda (1972), Shooman (1972, 1973, 1976, 1977), and Coutinho (1973). We present the theory behind software reliability, and describe some of the major models that have appeared in the literature from both historical and applications perspectives. Emerging techniques for software reliability research field are also included. We address the following four key components in software reliability theory and modeling: *historical background, theory, modeling*, and *emerging techniques*. We describe these items in a general way, rather than attempting to discuss a long list of details. For a comprehensive treatment of this subject, see Lyu (1996).

1. Historical Background

1.1. Basic Definitions

Software reliability is centered on a very important software attribute: *reliability*. *Software reliability* is defined as the probability of failure-free software operation for a specified period of time in a specified environment (ANSI, 1991). We notice the three major ingredients in the definition of software reliability: *failure*, *time*, and *operational environment*. We now define these terms and other related software reliability terminology.

1.1.1. Failures
A failure occurs when the user perceives that a software program ceases to deliver the expected service.

The user may choose to identify several severity levels of failures, such as catastrophic, major, and minor, depending on their impacts to the system service and the consequences that the loss of a particular service can cause, such as dollar cost, human life, and property damage. The definitions of these severity levels vary from system to system.

1.1.2. Faults
A fault is uncovered when either a failure of the program occurs, or an internal error (e.g., an incorrect state) is detected within the program. The cause of the failure or the internal error is said to be a fault. It is also referred as a “bug.”

In most cases the fault can be identified and removed; in other cases it remains a hypothesis that cannot be adequately verified (e.g., timing faults in distributed systems).

In summary, a software failure is an incorrect result with to the specification or unexpected software behavior perceived by the user at the boundary of the software system, while a software fault is the identified or hypothesized cause of the software failure.

1.1.3. Defects
When the distinction between fault and failure is not critical, “defect” can be used as a generic term to refer to either a fault (cause) or a failure (effect). Chillarege and co-workers (1992) provided a complete and practical classification of software defects from various perspectives.

1.1.4. Errors
The term “error” has two different meanings:

1. A discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. Errors occur when some part of the computer software produces an undesired state. Examples include exceptional conditions raised by the activation of existing software faults, and incorrect computer status due to an unexpected external interference. This term is especially useful in fault-tolerant computing to describe an intermediate stage in between faults and failures.
2. A human action that results in software containing a fault. Examples include omission or misinterpretation of user requirements in a software specification, and incorrect translation or omission of a requirement in the design specification. However, this is not a preferred usage, and the term “mistake” is used instead to avoid the confusion.

1.1.5. Time
Reliability quantities are defined with respect to time, although it is possible to define them with respect to other bases such as program runs of number of transactions. We discuss the notation of time in the next session.

1.1.6. Failure Functions
When a time basis is determined, failures can be expressed in several ways: the cumulative failure function, the failure intensity function, the failure rate function, and the mean-time-to-failure function. The *cumulative failure function* (also called the *mean-value function*) denotes the expected cumulative failures associated with each point of time. The *failure intensity function* represents the rate of change of the cumulative failure function. The *failure rate function* (or called the *rate of occurrence of failures*) is defined as the probability that a failure per unit time occurs in the interval [*t* , *t* + D*t*], given that a failure has not occurred before *t*. The *mean time to failure* (MTTF) function represents the expected time that the next failure will be observed. (MTTF is also known as MTBF, *mean time between failures*.)

1.1.7. Mean Time to Repair and Availability
Another quantity related to time is mean time to repair (MTTR), which represents the expected time until a system will be repaired after a failure is observed. When the MTTF and MTTR for a system are measured, its availability can be obtained. Availability is the probability that a system is available when needed. Typically, it is measured by

$$Availability = \frac{MTTF}{MTTF + MTTR}$$

1.1.8. Operational Profile
The operational profile of a system is defined as the set of operations that the software can execute along with the probability with which they will occur. An operation is a group of runs that typically involve similar processing.

SOFTWARE RELIABILITY ENGINEERING in this encyclopedia provides a detailed description on the structure, development, illustration, and project application of the operational profile. In general, the number of possible software operations is quite large. When it is not practical to determine all the operations and their probabilities in complete detail, operations based on grouping or partitioning of input states (or system states) into domains are determined.

1.2. The Advantages of Using Execution Time
Three kinds of time are relevant to software reliability: execution time, calendar time, and clock time. The *execution time* for a software system is the CPU time that is actually spent by the computer in executing the software, the *calendar time* is the time

people normally experience in terms of years, months, weeks, days, etc.; and the *clock time* is the elapsed time from start to end of computer execution in running the software. In measuring clock time, the periods during which the computer is shut down are not counted. If computer utilization, the fraction of time the processor is executing a program, is constant, clock time will be proportional to execution time.

In 1972 Shooman published a study where the behavior of a key parameter in his model was related to how the project personnel profile varied over time. Several different mathematical forms were proposed for the project personnel profile, and the choice of the best one depended on the particular project. Similarly, Schneidewind (1972) approached software reliability modeling from an empirical viewpoint. He recommended the investigation of different reliability functions and selection of the one that best fit the particular project in question. He found that the best distribution varied from project to project.

Up to 1975, the time domain used in software reliability modeling was exclusively calendar time. The lack of modeling universality resulting from using calendar time caused Musa (1975) to question the suitability of this time domain. He postulated that execution time was the best practical measure for characterizing failure-inducing stress being placed on a software program. Calendar time, used by Shooman and Schneidewind, did not account for varying usage of a program in a straightforward way. It turned out that the removal of this confounding factor greatly simplified modeling and yielded better model prediction results.

There is substantial evidence showing the superiority of execution time over calendar time for software reliability growth models (Trachtenberg, 1985; Musa and Okumoto, 1984a; Hecht, 1981). Nevertheless, many published models continue to use calendar time or do not explicitly specify the type of time being used. Some models, originally developed as calendar-time models or without regard to the type of time, are now being interpreted as execution-time models. If execution time is not readily available, approximations such as clock time, weighted clock time, staff working time, or units that are natural to the application, such as transactions or test cases executed, may be used. Musa *et al*., 1987 developed a modeling component that converts modeling results between execution time and calendar time. Huang *et al*., (2001) proposed the incorporation of testing effort into the modeling process, which can be measured by the human power, the number of test cases, or the execution-time information.

1.3. Two Data Types in Time-Domain Modeling

Software reliability models generally fall into two categories depending on the domain they operate in. By far the largest and most popular category of models is based on time. Their central feature is that reliability measures, such as failure intensity, are derived as a function of time. The second category of software reliability models provides a contrasting approach by using operational inputs as their central feature. These models measure reliability as the ratio of successful runs to total number of runs. However, this approach has some problems, including the fact that many systems have runs of widely varying lengths (so that the proportion may give inaccurate estimates) and that the resulting measures are incompatible with the time-based measures used for hardware.

Because of this and the amount of research currently being devoted to time-based models, the second model category will not be considered further here. (See Software Reliability Engineering.)

The time-domain modeling approach employs either the observed number of failures discovered per time period or the observed time between failures of the software. The models therefore fall into two basic classes depending on the type of data the model uses: (**1**) failures per time period or (**2**) time between failures.

These classes are, however, not mutually disjoint. There are models that can handle either data type. In fact many of the models for one data type can still be applied even if the user has data of the other type. For instance, if the user has only time between failures data and wants to apply a particular model that uses only failures-per time period, the user needs only to set a specified time length (e.g., a week) and then superimpose the failure history over these time intervals and observe how many fell within each interval. Conversely, if one had only failures-per-time period data, one could randomly assign the fault occurrences within each period and then observe the time-between-failures data created when the periods are linked. Either of these procedures from “transforming” from one data type into another requires that the user test the applied model to determine the adequacy of the resulting fit.

These classes can themselves be considered part of the larger “time domain” approach to software reliability modeling in contrast to the “error seeding and tagging” approach and the “data domain” approach. Space limitations do not allow us to address these other important approaches as well as time-series models in this chapter. The reader is referred to Farr (1983) and Xie (1991), among others, where these alternative approaches are described.

For the failures-per-time-period data, the unit of time could represent a day, week, etc., over which the software is being tested or observed. For the class based upon time between failures, we have recorded either the elapsed calendar time or execution time between each software failure. Typical failures-per-time period data are shown in Table 1, and typical time-between-failures data are shown in Table 2.

Table 1. Failures-per-Time-Period Data

Time (hours)	Failures per Time Period	Cummulative Failures
8	4	4
16	4	8
24	3	11

32	5	16
40	3	19
48	2	21
56	1	22
64	1	23
72	1	24

Table 2. Time-between-Failures Data

Failure Number	Failure Interval (hours)	Failure Times (hours)
1	0.5	0.5
2	1.2	1.7
3	2.8	4.5
4	2.7	7.2
5	2.8	10.0
6	3.0	13.0
7	1.8	14.8
8	0.9	15.7
9	1.4	17.1
10	3.5	20.6
11	3.4	24.0
12	1.2	25.2
13	0.9	26.1
14	1.7	27.8
15	1.4	29.2
16	2.7	31.9
17	3.2	35.1
18	2.5	37.6
19	2.0	39.6
20	4.5	44.1
21	3.5	47.6
22	5.2	52.8

23	7.2	60.0
24	10.7	70.7

1.4. Software Reliability Modeling and Measurement

Software reliability measurement includes two types of activity, reliability estimation and reliability prediction:

Estimation. This activity determines current software reliability by applying statistical inference techniques to failure data obtained during system test or during system operation. This is a measure regarding the achieved reliability from the past until the current point. Its main purpose is to assess the current reliability, and determine whether a reliability model is a good fit in retrospect.

Prediction. This activity determines future software reliability based on available software metrics and measures. Depending on the software development stage, prediction involves different techniques:

When failure data are available (e.g., software is in system test or operation stage), the estimation techniques can be used to parameterize and verify software reliability models, which can perform future reliability prediction.

When failure data are not available (e.g., software is in the design or coding stage), the metrics obtained from the software development process and the characteristics of the resulting product can be used to determine reliability of the software upon testing or delivery.

The first definition is also referred to as "reliability prediction" and the second definition, as "early prediction." When there is no ambiguity in the text, only the word "prediction" will be used.

Most current software reliability models fall in the estimation category to do reliability prediction. A software reliability model specifies the general form of the dependence of the failure process on the principal factors that affect it: fault introduction, fault removal, and the operational environment. Figure 1 shows the basic ideas of software reliability modeling.

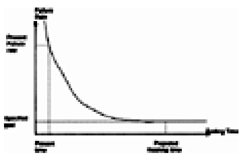

Figure 1. Basic ideas on software reliability modeling. [Full View]

If the reliability improves over time, as faults are discovered and corrected, one would expect that the number of failures detected per unit of time would be decreasing and the time between failures would be increasing. It is this behavior that the software reliability models attempt to capture. In Figure 1, the failure rate of a software system is generally decreasing due to the discovery and removal of software faults. At any particular time (say, the point marked “present time”), it is possible to observe a history of the failure rate of the software. Software reliability modeling forecasts the curve of the failure rate by statistical evidences. The purpose of this measure is twofold: (**1**) to predict the extra time needed to test the software to achieve a specified objective and (**2**) to predict the expected reliability of the software when the testing is finished. If an adequate fit to the past data is achieved and the testing or operation of the software doesn't radically change in the future, the fitted reliability curves can be used to make various reliability predictions.

1.5. Relationship between Hardware and Software Reliability

Hardware reliability is a generally understood and accepted concept with a long history. Early during the much shorter history of software reliability it became apparent to researchers that a division (often perceived to be large) exists between hardware reliability and software reliability. Software reliability is similar to hardware reliability in that both are stochastic processes and can be described by probability distributions. However, software reliability is different from hardware reliability in the sense that software does not wear out, burn out, or deteriorate; i.e., its reliability does not decrease with time. Moreover, software generally enjoys reliability growth during testing and operation since software faults can be detected and removed when software failures occur. On the other hand, software may experience reliability decrease because of abrupt changes of its operational usage or incorrect modifications to the software. Software is also continuously modified throughout its life cycle. The malleability of software makes it inevitable for us to consider variable failure rates.

At first these differences raised the question of whether reliability theory can be applied to software at all. It was discovered that the distinction between hardware and software is somewhat artificial. Both may be defined in the same way, so that hardware and software component reliabilities can be combined to get system reliability. Traditionally, hardware reliability focused on physical phenomena because failures resulting from these factors are much more likely to occur than design-related failures. It was possible to keep hardware design failures low because hardware was generally less complex logically than software. Besides, hardware design failures had to be kept low because of the large expense involved in retrofitting of manufactured items in the field. However, when hardware tests show that reliability is not within specified design limits because of problems or faults in the original design, a sequence of engineering changes may be necessary to improve reliability. Thus hardware reliability can and has been modeled like software reliability when the failures are the result of design faults, such as the highly visible Pentium floating-point division fault that resulted in massive callbacks in 1994 (Wolfe, 1994).

Perhaps the first hardware reliability model that can also be used as a model of reliability for software was developed in 1956 by Northrop Aircraft (Weiss, 1956). This model considers complex systems where engineering changes are made to improve system reliability. It was used to determine the level of system reliability, how rapidly reliability was improving, and the expected reliability at the end of the projected development program. Two other early hardware reliability models along similar lines consider the problem of estimating reliability of a system undergoing development testing and changes to correct design deficiencies (Corcoran, *et al*., 1964; Barlow and Scheuer, 1966).

It is now also generally accepted that the software failure process is random. This randomness is introduced in many ways. The location of design faults within the software is random because the overall system design is extremely complex. The programmers who introduce the design faults are human, and human failure behavior is so complex that it can best be modeled using a random process. Also, the occurrence of failures is dependent on the operational profile, which is defined by input states. It is usually not known which input state will occur next, and sometimes an input state may occur unexpectedly. These events make it impossible to predict where a fault is located or when it will be evoked to cause a failure in a large software system.

In an attempt to unify hardware and software for an overall system reliability, software reliability theory has generally been developed in a way that is compatible with hardware reliability theory, so that system reliability figures may be computed using standard hardware combinatorial techniques (Shooman, 1990; Lloyd and Lipow, 1984). However, unlike hardware faults that are mostly physical faults, software faults are design faults that are harder to visualize, classify, detect, and correct. As a result, software reliability is a much more difficult measure to obtain and analyze than hardware reliability. Usually hardware reliability theory relies on the analysis of stationary processes, because only physical faults are considered. However, with the increase of systems complexity and the introduction of design faults in software, reliability theory based on stationary process becomes unsuitable to address nonstationary phenomena such as reliability growth or reliability decrease experienced in software. This makes software reliability a challenging problem that requires an employment of several methods to attack. We now describe the theory behind software reliability.

2. Theory

2.1. Reliability Theory

Since the development of software reliability models was based on concepts adapted from hardware reliability theory, we need to define some reliability functions and concepts that show the relationships among these different functions.

Because the failure time of a software system is a random variable, which we'll denote as *T*, this variable has an associated probability density function, $f_T(t)$, and cumulative distribution function, $F_T(t)$, where

$$R_T(t_0) = P(T \leq t_0) = \int_0^{t_0} f_T(t)\,dt \qquad (1)$$

The reliability function, $R_T(t)$, which is the probability the software has not failed by time *t*, is then calculated as

$$
\begin{aligned}
R_T(t) &= P(T \geq t) = 1 - F_T(t) \\
&= 1 - \int_0^t f_T(x)\,dx \qquad (2) \\
&= \int_t^\infty f_T(x)\,dx
\end{aligned}
$$

A third function that is important in reliability modeling is the hazard rate function, $Z_T(t)$. It is the conditional probability that the software will fail in an interval (t , t + Dt), given that it has not failed before time t. If T is the time at which the failure occurs, then

$$Z_T(t)\Delta t = P(t < T < t + \Delta t \,|\, T > t)$$

Dividing both sides by Dt, expressing the probability in its conditional form, and letting Dt approach zero, we have

$$
\begin{aligned}
Z_T(t) &= \lim_{\Delta t \to 0}\left(\frac{P(t < T < t + \Delta t)}{\Delta t P(T > t)}\right) \\
&= \frac{f_T(t)}{1 - F_T(t)}
\end{aligned}
\qquad (3)
$$

$$Z_T(t) = \frac{f_T(t)}{R_T(t)} \qquad (4)$$

From Equation (3) we see that the hazard rate function is simply the conditional probability density function for the failure of the system given no failures have occurred up to time t. From Equation (4) and the fact that $R_T(t) = 1 – F_T(t)$, we have the following:

$$Z_T(x)dx = \frac{dF_T(x)}{1 - F_T(x)}$$

or

$$\int_0^t Z_T(x)\,dx = -\log\left(\frac{1 - F_T(t)}{1 - F_T(0)}\right)$$

or

$$1 - F_T(t) = \exp(-\int_0^t Z_T(x)\,dx)$$

thus

$$R_T(t) = \exp(- \int_0^t Z_T(x)dx) \tag{5}$$

The reliability function, in turn, can be related to the mean time to failure, (MTTF), of the software;

$$\begin{aligned} MTTF = E\{T\} &= \int_0^\infty t f_T(t)dt \\ &= \int_0^\infty R_T(t)dt \end{aligned} \tag{6}$$

All of these relationships hold for the corresponding conditional functions as well. One simply replaces the hazard, reliability, cumulative distribution, or probability density function for a “single” fault, *T*, by the associated conditional functions. For example, suppose the system has failed at time t_i, then the conditional hazard rate function is denoted as $Z_T(t|t_i)$, where $t \geq t_i$ and

$$Z_T(t \mid t_i) = \frac{f_T(t \mid t_i)}{1 - F_T(t \mid t_i)} = \frac{f_T(t \mid t_i)}{R_T(t \mid t_i)}$$

The last important functions that we will consider are the failure intensity function and the mean value function for the cumulative number of failures. We'll denote the failure intensity function as l(*t*). This is the instantaneous rate of change of the expected number of failures with respect to time. Suppose we let *M*(*t*) be the random process denoting the cumulative number of failures by time *t* and we denote m(*t*) as its mean value function:

$$\mu(t) = E\{M(t)\} \tag{7}$$

The failure intensity function is then obtained from m(*t*) as its derivative:

$$\lambda(t) = \frac{d}{dt}(E\{M(t)\}) \tag{8}$$

In order to have reliability growth, we should have (*d* l(*t*)/*dt*) < 0 for all $t \geq t_0$ for some t_0. The failure rate function may also exhibit a zigzag type behavior but it must still be decreasing to achieve reliability growth.

2.2. Random Point Processes

The reliability of software is influenced or determined mainly by three factors: fault introduction, fault removal, and the operational profile. Fault introduction depends primarily on the characteristics of the developed code (code written or modified for the program) and the development process. The code characteristic with the greatest effect is size. Development process characteristics include the software engineering technologies and tools employed and the average level of experience of programmers. Note that code is developed when adding features or removing faults. Fault removal is affected by time, the operational profile, and the quality of the repair activity. Since most of these factors are probabilistic in nature and operate over time, the behavior of software failures with execution time is usually modeled using some kind of random point process. The upper portion of Figure 2 shows an example of such a process where each failure epoch is represented by an X. A counting process giving the number of failures experienced by execution time *t* is associated with every point process. Figure 2 also shows a typical counting process, denoted by $M(t)$, having a unit jump at each failure epoch.

If we let $M(t)$ be the random number of failures that are experienced by time *t* with mean value function $m(t)$, i.e., $m(t)$ We remark that the failure counting process being considered here is strictly nondecreasing. Furthermore, if $\lim_{t \to \infty} m(t) < \infty$ i.e., is finite, we have a finite failure model category; otherwise we have a model of the infinite failures category.

We will now relate some important properties of the Poisson and binomial processes. These processes play a key role in the unification and classification of many published models, and there are some central theoretical results that come into consideration. We will make use of these properties for specific classes of models in subsequent sections.

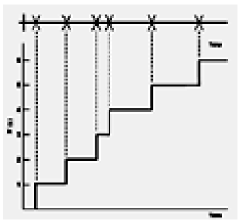

Figure 2. A random point process and its associated counting process. [Full View]

First for the Poisson type models, we consider that we have a Poisson process over time. By this we mean that if let $t_0 = 0$, t_1 , ¼ , t_{i-1} , t_i , ¼ , $t_n = t$ be a partition of our time interval 0 to *t* and let $m(t)$ be as defined as above, then we have a Poisson process if each $t_i', i = 1, \ldots, n$ the number of failures detected in the *i*th interval, t_{i-1} to t_i), are independent Poisson random variables with means, $E\{t_i'\} = \mu(t_i) - \mu(t_{i-1})$. We make the following assumptions:

1. The probability of failure (failure intensity) depends on time t and the number of past failures $M(t)$.
2. Failures do not occur simultaneously.
3. Failures do not occur at preassigned times.
4. There is no finite interval in $[t, \infty]$ where failures occur with certainty.

5. There is no failure in the beginning of the process.
Then for each of the random variables t'_i values , $i = 1$, ¼ , n, the probability mass function is:

$$P\{t_i' = x\} = e^{-(\mu(t_i)-\mu(t_{i-1}))}[(\mu(t_i) - \mu(t_{i-1}))]^x/x! \quad \text{for} \quad x = 0, 1 \ldots$$

[*Note*: If m(t) is a linear function of time, i.e., m(t) = at for some constant, a > 0, we say that the Poisson process, $M(t)$, is a *homogeneous Poisson process* (HPP). If, however, it is nonlinear, we refer to the process as being a *nonhomogeneous Poisson process* (NHPP).]

If we have a Poisson process model, we can show a relationship between the failure intensity function and the reliability function (hence the hazard rate and the probability density function using the relationships established in the previous section). Suppose that we denote $R($ t + Dt|t) as the conditional reliability function that the software will still operate after t + D given that it has not failed after time t. Then

$$\begin{aligned} R(t+\Delta t \mid t) &= P(t' = 0 \mid t) \text{ where } t' \text{ is a Poisson random variable over the inverval } t \text{ to } t+\Delta t \\ &= \exp(-(\mu(t+\Delta t) - \mu(t))) \\ &= \exp(-\lambda(\bar{t}\Delta t), \ \bar{t} \text{ is a point i} \text{ the interval } t \text{ to } t\ \Delta t \text{ and using the definition of } \lambda(t) \\ &= \exp\left(-\int_t^{t+\Delta t} \lambda(x)\,dx\right) \text{using the mean value theorem of integra} \end{aligned}$$

The relationship between the failure intensity function and the hazard rate for a Poisson process can also be derived. It can be shown that

$$Z_T = (\Delta t \mid t_{i-1}) = \lambda(t_{i-1} + \Delta t) \tag{9}$$

where t_{i-1} is the time of the (i – 1)st failure and Dt is any point such that $t_{i-1}t_i$ + Dt < t_i. This shows that the conditional hazard rate and the failure intensity function are the same if the failure intensity function is evaluated at the current time t_{i-1} + Dt.

Another relationship that one can establish for the Poisson type of models is that

$$\mu(t) = aF_a(t) \tag{10}$$

where a is some constant and $F_a(t)$ is the cumulative distribution function of the time to failure of an individual failure a. From this, if we consider also distributions that belong to the finite failure category, i.e., $\lim_{t\to\infty} m(t) < \infty$ we have that $\lim_{t\to\infty} m(t) = a$ since $\lim_{t\to\infty} F_a(t) = 1$. Thus a represents the eventual number of failures detected in the system if it could have been observed over an infinite amount of time. Using Equation (10) and the relationship between the mean value function and the failure intensity function, we have also for the Poisson type of model

$$\lambda(t) = \mu(t) = \alpha f_a(t) \tag{11}$$

where $f_a(t)$ is the probability density function of the time to failure of the individual failure a.

For the binomial type of model, we have the following assumptions:

1. There is a fixed number of faults (N) in the software at the beginning of the time in which the software is observed.
2. When a fault is detected, it is removed immediately.
3. If T_a is the random variable denoting the time to failure of fault a, then the T_a values, $a = 1$, ¼ , n, are independently and identically distributed random variables as $F_a(t)$ for all remaining faults.

The cumulative distribution function, $F_a(t)$, density function, $f_a(t)$, and hazard rate function, $Z_a(t)$, are the same for all faults for this class. Moreover, we notice for this class that a failure and a fault are synonymous and no new faults are introduced into the software in the fault detection/correction process. We can show for this class, the failure intensity function is obtained from the probability density function for a single fault as

$$\lambda(t) = Nf_a(t) \tag{12}$$

The mean value function is, in turn, related to the cumulative distribution function, $F_a(t)$, as

$$\mu(t) = NF_a(t) \tag{13}$$

Notice the similarity between equations (12) and (13) and (10) and (11) for the Poisson type. For the binomial we have a fixed number of faults (one-to-one correspondence to failures) at start, N, while for the Poisson type, a, is the eventual number of failures that could be discovered over an infinite amount of time.

Kremer (1983) unified many software reliability models using a nonhomogeneous birth–death Markov process. He showed that many of the Poisson-type and binomial-type models discussed extensively in the literature are special cases of a Markov birth process with specific forms for l(t) and m(t).

It is well worth discussing the preceding conditions for a Markov process here because this will help determine the plausibility of the entire modeling approach. Some of the affects of altering the first condition were already investigated when the origin of the Poisson-type models and binomial-type models was clarified. If this condition is relaxed so that the failure intensity may depend on t, $M(t)$, and the occurrence times for all failures before t, a self-exciting random point process is obtained. In this type of process, since $M(t)$ is a random variable, the failure intensity itself is a random process. Other types of point process can be obtained if an "outside" process affects the failure intensity. These processes are beyond the scope of this discussion. A general self-exciting point process can be conceived of as a modified nonhomogeneous Poisson process in which the future evolution of failures not only is a function of time but also can be influenced by all past occurrences of failures. Of course, when the evolution depends only on time and the total number of past failures, a Markov birth process results.

Note that the assumptions made for a Poisson process in modeling software failures are generally well accepted by researchers in the field, but they can also be easily relaxed, especially for a nonhomogeneous Poisson process. For example, the condition requiring that the process start out with no failures can be easily changed to one that assumes that the process starts out with a known or random number of failures simply by treating this as a separate term in the model formulations. As another example, the condition that no failures occur simultaneously can be relaxed, leading to what is called a *compound Poisson process* (Sahinoglu, 1992). Many other powerful and useful generalizations are possible and can be found in Snyder (1975).

Reliability estimates for nonhomogeneous Poisson process models come directly from Equation (3) and by noting that the event $M(T) > i$ is equivalent to the event T_{int}, where T is a random variable for the ith failure time. In addition, unknown model parameters are usually determined using the maximum likelihood principle, least squares, or Bayesian techniques. Again, specific derivations are omitted; however, Table 3 provides a summary of some important relationships for these models (Musa, *et al*., 1987). These can be used for a particular model, that is, for a particular mean value function.

Table 3. Some Derived Relationships for a General Poisson Process

Quality	**Formula**[a]
Failures experienced	$\Pr\left[M(t) = m\right] = \frac{(\mu t)^m e^{-\mu(t)}}{m!}$

	Expected Value = m(t)
	Variance = m(t)
Failure time	$\Pr\left[T_i \le t\right] = F(t_i) = \sum_{j=i}^{x} \frac{(\mu t)^j e^{-\mu(t)}}{j!}$
Reliability	$R(t'_i \mid t_{i-1}) = \Pr\left[T'_i > t'_i \mid (T_{i-1} = t_{i-1})\right] = e^{-[\mu(t_{i-1}+t'_i)-\mu(t_{i-1})]}$
Conditional failure time	$f\left(t'_i \mid t_{i-i}\right) = \lambda\left(t_{i-1} + t'_i\right)e^{-[\mu(t_{i-1}+t'_i)-\mu(t_{i-1})]}$
Failure intensity	$\lambda(t) = \frac{d\mu(t)}{dt}$
Unconditional failure time	$f(t_i) = \frac{dF(t_i)}{dt_i} = \frac{\lambda(t_i)\left[\mu(t_i)^{i-1}\right]e^{-\mu(t_i)}}{(i-1)!}$
Maximum-likelihood equations	$-\frac{\partial\mu(t_e)}{\partial^{\cdot}\beta_k} = \sum_{i=1}^{me} \frac{1}{\lambda(t_i)}\frac{\partial\lambda(t_i)}{\partial^{\cdot}\beta_K}; k = 1, 2, \ldots n$
	$\hat{\beta}_0 = \frac{m_e}{\mu l\left(t_e; \hat{\beta}_1, \hat{\beta}_2, \ldots, \hat{\beta}_n\right)}$

[a] Where

t_i	=	ith failure time
t'_i	=	time interval between$(i-1)$th and i failures
$\mu(t)$	=	$\beta_0\mu_1(t; \beta_1, \beta_2, \ldots, \beta_n)$
$\beta_j(j = 0, \ldots, n)$	=	unknown model parameters
$\hat{\beta}_j$	=	estimated value of β_j
m_e	=	observed failures at time t_e
t_e	=	total time of observation or testing

2.3. Exponential Order Statistics

The exponential order statistics approach to modeling software reliability, studied by Downs (1985; 1986), Miller (1986), and Ross (1985a,1985b) is essentially equivalent to the preceding approach except that it provides a more intuitive feeling for the actual failure process. Figure 3 shows the modeling environment. A piece of software initially containing an unknown number of faults is subjected to testing where input states *A*, *B*, and *C*, are chosen at random according to some operational profile. Most input states result in successful execution (correct program results). Some input states exercise a collection of instructions containing a fault and cause a failure to occur. Still others exercise a collection of instructions containing a fault but do not cause a failure to occur because the data state or specific conditions are not right. For example, input state *C* in Figure 3 does not encounter a fault and causes no failure, whereas input states *A* and *B* both encounter fault ***a*** with only input state *A* causing a failure. Therefore, the only input states of consequence as far as fault a causing a failure are the ones like input state *A* in the example. The collection of these input states is called fault a's ***fail set***.

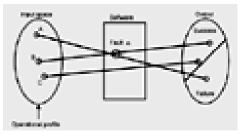

Figure 3. Software reliability modeling environment. [Full View]

Two factors determine the failure-causing potential of a fault. They are the size or number of input states in the fault's fail set and the frequency with which these states are chosen for execution. Clearly, if the operational profile were to change, so also would the importance or contribution of the input states in the fail set to the failure-causing potential of the fault.

Let the failure intensity for fault ***a*** denoted by l_a; then the failure intensity for the program depends on whether faults are being removed. Assume for the moment that they are not being removed. This is typical of a program in production use. The program failure intensity l is determined by summing the contributions of the individual faults:

$$\lambda = \sum_{a=1}^{\omega_0} \lambda_a \tag{14}$$

Equation (14) implicitly assumes that the failures from different faults are independent. This assumption may raise several issues. The first issue that may come up is the matter of input state dependencies; for example, a specific input state may always be executed after a given input state. This can be incorporated into the model, but the increase in accuracy, if any, is probably outweighed by the added complexity. The second issue is that a fault may prevent access to code containing one or more faults. This issue is less common during system testing or operational use than it is during unit testing and is considered to be a secondary effect and is therefore ignored. Finally, another issue concerns multiple failures resulting from an execution of a given input state. This is usually considered to be secondary or not worth the effort to explicitly model. In most cases, failures are independent because they are the result of two processes: the introduction of faults and the activation of faults through the selection of input states. Both of these processes are random, and hence the chance that one failure would affect another is small. The independence conclusion is supported by a study of correlations on failure data from 15 projects (Musa, 1979) that found no significant correlation.

Now suppose that faults are being repaired. The program is exercised for a period of time *t*, and each time a failure occurs, the software is debugged and the fault responsible for the failure is removed. Let $I_a(t)$ be a function that takes on a value of 1 if fault ***a*** has not caused a failure by *t* and 0 otherwise. As before, the program failure intensity at time *t*, denoted by L(*t*) since this time it is a random variable, is determined by summing the contributions of the individual faults still remaining in the program. Thus

$$\Lambda(t) = \sum_{a=0}^{\omega_0} \lambda_a I_a(t) \tag{15}$$

Note that I_n contributes to the sum only if fault ***a*** has not been removed yet. Implicit assumptions being made here are that failures from different faults are independent and that a fault causing an observed failure is immediately resolved. The latter assumption, however, need not be the case. Subsequent occurrences of the same failure can be ignored in the analysis if necessary.

An assumption concerning the failure process itself must be made. The usual assumption is that faults cause failures to occur in accordance with independent homogeneous Poisson processes with unknown failure intensities. A natural result of this is that failure times for a fault are exponentially distributed. Miller (1986) defines an exponential order statistic model based on a failure counting process and an associated failure occurrence-time process, which is characterized by the parameter set of failure intensities set of l_a , $1 \le a \le w_0$, with the only restrictions that $l_a > 0$ and $\sum_{i=1}^{\omega_0} \lambda_i < \infty$. Miller shows many possibilities exist for the form of l_a, values which can be treated in several ways. One such way is deterministically; that is, following a known pattern. Two examples include constant failure intensities ($l_a = l_0$ for all *a*) and geometric failure intensities ($l_a = a\ b^a$, $0 < b < 1$ for all *a*).The failure intensities can also be treated as a finite collection of independent and identically distributed random variables drawn from some distribution; for example, the gamma distribution. Finally, the failure intensities can be treated as a realization of a random point process such as the nonhomogeneous Poisson process. It is also mathematically possible to permit an infinite value for w_0 and treat finite w_0 as a special subcase. Doing this, it is possible to simultaneously deal with infinite failures and finite failures models and to unify a great many models. Miller (1986) gives a complete discussion on the types of models resulting from these and other patterns of failure intensities with some rather significant results. One result is the inability to distinguish between the different types of models for the l_a, values based on a single observation of the failure process. Another closely linked idea is that the mean value function is the primary characteristic of a model and the particular type of model is a secondary characteristic. For example, a model based on deterministic I_n values and a model based on I_n values drawn from a probability distribution are close to each other if their mean value functions are close. An example of an application of this idea is in Musa and Okumoto (1984b) where the Littlewood and Verrall (1973) model was analyzed using a nonhomogeneous Poisson process with appropriate mean value function.

Given the preceding assumption about the failure process, the probability of fault a not causing a failure by time *t* is e^{-l} *at*. Therefore, the expected failure intensity for the program, l(*t*), is given by

$$E[\Lambda(t)] = \lambda(t) = \sum_{a=1}^{\omega_0} \lambda_a E[I_a(t)] = \sum_{a=1}^{\omega_0} \lambda_a e^{-\lambda_a t} \tag{16}$$

This equation is sufficient to completely describe the overall failure process for the program. The integral of this equation gives the mean value function m(*t*) or the expected number of failures at time *t*:

$$\mu(t) = \sum_{a=1}^{\omega_0} \left(1 - e^{-\lambda_a t}\right) \tag{17}$$

Equations (16) and (17) show how l(*t*) and m(*t*), respectively, depend on the failure intensity of the individual faults.

Now we determine what Equation (17) reveals about the kinds of mean value functions likely to be good candidates for models. Let $\overline{\lambda}$represent the average per-fault failure intensity of the inherent faults:

$$\overline{\lambda} = \frac{1}{\omega_0} \sum_{a=1}^{\omega_0} \lambda_a$$

Also, let s^2 be a measure of the variation in the inherent per-fault failure intensities:

$$\sigma^2 = \sum_{a=1}^{\omega_0} \left(\lambda_a - \overline{\lambda}\right)^2$$

Then, expanding Equation (17) in a Taylor series about l keeping *t* fixed yields

$$\begin{aligned} \mu(t) &= \omega_0\left(1 - e^{-\overline{\lambda}t}\right) + \sum_{a=1}^{\omega_0} te^{-\overline{\lambda}t}\left(\lambda_a - \overline{\lambda}\right) - \frac{\tau^2 e^{-\overline{\lambda}t}}{2} \\ \sum_{a=1}^{\omega_0}\left(\lambda_a - \overline{\lambda}\right)^2 \cdots &= \omega_0\left(1 - e^{-\overline{\lambda}\tau}\right) - \frac{t^2 e^{-\overline{\lambda}t}\sigma^2}{2} \cdots \end{aligned} \tag{18}$$

where higher order terms are not explicitly shown. The significant point about Equation (18) is the first term, which represents an exponential mean value function characteristic of many popular and useful models (Musa, 1975; Goel and Okumoto, 1979). The conclusion to be drawn is that the exponential mean value function is a first-order approximation of all possible mean value functions based on a finite number of initial faults. This explains, in part, why the exponential nonhomogeneous Poisson process model enjoys such success in many applications (Zinnel, 1990; Ehrlich *et al*., 1991; Iannino and Musa, 1991).

As can be seen, the exponential nonhomogeneous Poisson process model is exact if all per-fault failure intensities are the same. Note that this does not imply a uniform operational profile, as is sometimes claimed in the literature. It does imply that the joint effect of the operational profile and of the fail set size, the fault exposure, is the same for all faults. Thus a fault with a large fail set size and small individual operation usage probabilities may be the same, in terms of failure intensity, as a fault with small fail set size and large individual operation usage probabilities.

Relaxing the homogeneous Poisson process assumption underlying each fault's failure process can further strengthen the case for the exponential mean value function. It turns out that any per-fault failure process whose first time to failure is exponentially distributed will lead to the same conclusions, provided subsequent occurrences of the same failure are ignored. This is an important observation especially when dealing with failures that tend to cluster as described in Ehrlich and co-workers (1991).

In summary, the nonhomogeneous Poisson process is emerging as the most practical and useful choice for modeling software reliability. This is based on many empirical results. The process is fully specified by its mean value function, and an emerging choice here, both in theory and in practice, is the exponential mean value function. This function was shown to play a central role in modeling and to be quite robust from departures in its assumptions.

3. Modelling

3.1. Model Classification

A model classification scheme proposed in Musa and Okumoto (1983) allows relationships to be established for models within the same classification groups and shows where model development has occurred. It classified models in terms of five different attributes:

Time domain—calendar versus execution time.

Category—the total number of failures that can be experienced in infinite time. This is either *finite* or *infinite*, representing two subgroups.

Type—The distribution of the number of the failures experienced by time *t*. Two important types that we will consider are the Poisson and binomial.

Class—(finite failure category only) functional form of the failure intensity expressed in terms of time.

Family—(infinite failure category only) Functional form of the failure intensity function expressed in terms of the expected number of failures experienced.

In the “Random Point Processes” section we described the Markov processes that are characterized by the distribution of the number of failures over time, and the two most important distributions, the Poisson and binomial. Models based on the binomial distribution are finite failure models, that is, they postulate that a finite number of failures will be experienced in infinite time. Models based on the Poisson distribution can be either finite failure or infinite failure models, depending on how they are specified. Table 4 shows how some of the published models are classified using this approach. The Bayesian model of Littlewood and Verrall (1973) and the geometric deeutrophication model of Moranda (1975, 1979) are among the few published models that are not Markovian.

Table 4. Markov Software Reliability Models

Poisson Type	**Binomial Type**	**Other Types**
Crow (1974)	Jelinski and Moranda (1972)	Shooman and Trivedi (1975)
Musa (1975)	Shooman (1972)	Kim and co-workers (1982)
Moranda (1975, 1979)	Schick and Wolverton (1973)	Kremer (1983)
Schneidewind (1975)	Wagoner (1973)	Laprie (1984)
Goel and Okumoto (1979)	Goel (1988a)	Shanthikumar and Sumita (1986)
Brooks and Motley (1980)	Schick and Wolverton (1978)	
Angus and co-workers (1980)	Shanthikumar (1981)	
Yamada and co-workers (1983)	Littlewood (1981)	
Yamada and Osaki (1984)		
Ohba (1984)		
Yamada and co-workers (1984)		

These unifications highlight relationships among the models and suggest new models where gaps occur in the classification scheme. Furthermore, they greatly reduce the task of model comparison.

3.2. The Exponential Models

In the literature on software reliability, this class has the most articles written on it. Using Musa and Okumoto's classification scheme, this group contains all finite failure models with the functional form of the failure intensity function being exponential. The binomial type in this class are all characterized by: a per-fault constant hazard rate, i.e., $z_T(t)$ = F the hazard rate function after the ith fault that has been detected is a function of the remaining number of faults, i.e., $N – (i – 1)$; and the failure intensity function is exponential in form, l(t) = N fexp(–ft). The Poisson types in this class are all characterized by a per-fault constant hazard rate, $z_T(t)$ = f, and an exponential time to failure of an individual fault, $f_X(x)$ = N fexp(–fx). Since we have either a homogeneous or nonhomogeneous Poisson process, the number of failures that occur over any fixed period of time is a Poisson random variable. For the time between failures models, the distribution is exponential.

The first software reliability model was independently developed by researchers in 1972 (Jelinski and Moranda, 1972; Shooman, 1972). In this model, the elapsed time between failures is taken to follow an exponential distribution with a parameter that is proportional to the number of remaining faults in the software; i.e., the mean time between failures at time t is 1/ f($N – (i – 1)$). Here t is any point in time between the occurrence of the $(i – 1)$th and the ith fault occurrence. The quantity f, is the proportionality constant and N is the total number of faults in the software from the initial point in time at which the software is observed. Figure 4 shows the characteristic step curve for the variation of program failure intensity with execution time for this model. One can see as each fault is discovered the hazard rate is reduced by the proportionality constant f. This indicates that the impact of each fault removal is the same. In Musa and Okumoto's classification scheme, this is a binomial type model. This model, henceforth referred to as the exponential model, makes the following basic assumptions.

1. All faults in the program contribute the same amount to the overall failure intensity of the program. Thus at given any time the program failure intensity is proportional to the number of remaining faults.
2. The failure detection rate remains constant over the intervals between failure occurrences.
3. A fault is corrected instantaneously without introducing new faults into the software.
4. The software is operated in a similar manner as that in which reliability predictions are to be made.
5. Every fault has the same chance of being encountered and is of the same severity as any other faults.

6. The failures, when the faults are detected, are independent.

The numbered assumptions 4–6 are fairly standard as we consider other models. Assumption 4 is provided to ensure that the model estimates that are derived using data collected in one particular environment are applicable to the environment in which the reliability projections are to be made. The fifth assumption is provided to ensure that the various failures all have the same distributional properties. One severity class might have a failure rate different from that of the others requiring a separate reliability analysis be done. The last assumption allows simplicity in deriving the maximum likelihood estimates. Since assumptions 4–6 appear so often in the software reliability models, we usually refer to them as the *standard assumptions* for reliability modeling.

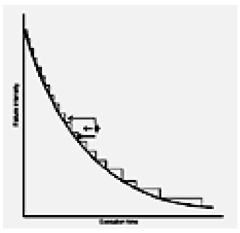

Figure 4. Characteristic step curve and continuous approximation for program failure intensity reduced by with execution time for the exponential model (with proportionality constant f as per-fault failure rate, this becomes a Jelinski–Moranda model). [Full View]

Several other models that are either identical to the exponential model except for notational differences or are very close approximations were developed by Musa (1975), Schneidewind (1975), and Goel and Okumoto (1979). The latter is a continuous approximation to the original exponential model and is described in terms of a nonhomogeneous Poisson process with a failure intensity function that is exponentially decaying. Figure 4 also shows this curve and how closely it approximates the exponential model curve. For all practical purposes the Goel–Okumoto and the other models are indistinguishable from the exponential model.

The parameter estimation of the exponential model (and, therefore, the other closely related or equivalent models) has been often criticized. Forman and Singpurwalla (1977, 1979), Littlewood and Verrall (1973), and Joe and Reid (1984), to name a few, have shown that parameter estimation for the model suffers from two unfortunate tendencies. Sometimes there is a tendency for estimates of the total failures expected to come out nearly equal to the number of failures experienced. This leads to overly optimistic conclusions. Sometimes there is a tendency for these same estimates to be nonfinite. Meinhold and Singpurwalla (1983) suggested that when nonfinite parameter estimates are obtained it is the method of inference that needs to be questioned not the model.

Nevertheless, the exponential model plays a key role in software reliability theory. Shock models and renewal theoretic arguments were used by Stefanski (1982) and Langberg and Singpurwalla (1985), respectively, to provide alternative motivations for and alluded to the centrality of the exponential model. In the former reference, it was also illustrated that many other well-known models, including that of Littlewood and Verrall (1973), can be obtained by specifying specific distributions for the parameters of the exponential model. The following sections on execution time modeling provide further justification for the equivalent exponential nonhomogeneous Poisson process model.

The exponential model can be further generalized (AIAA, 1993) to simplify the modeling process by having a single set of equations to represent a number of important models having the exponential hazard rate function. The overall idea is that the failure occurrence rate is proportional to the number of faults remaining and the failure rate remains constant between failures while it is reduced by the same amount when a fault is removed. Besides the *standard assumptions*, the other assumptions of the model are

1. The failure rate is proportional to the current fault content of the software.
2. The faults that caused a failure are corrected instantaneously, and no additional faults are introduced by the correction process.
3. The data required are the usual time between failures, x_i values, or the time of the failures, the t_i values.

The model form is expressed as

$$Z(t) = K[E_0 - E_c(t)]$$

where $Z(\,.\,)$ is the software hazard rate function; t is a time or resource variable for measuring the progress of the project; K is a constant of proportionality denoting the failures per unit of t ; E_0 is the initial number of faults in the software; and E_c is the number of faults in the software that have been found and corrected after t units have been expended. Table 5 reflects how this model is related to some of the models in the literature.

Table 5. Generalized Exponential Model Relationships

Model	**Original Hazard Rate Function**	**Parameter Equivalences**
Generalized form	$k[E_0 - E_c(t)]$	
Shooman model (Shooman 1972)	$K' \, [E_0/I_T - \P_c(t)]$	$\P_c = E_c/I_T$ where I_T is the number of instructions $K' = KI_T$
Jelinski–Moranda, model (Jelinski and Moranda, 1972)	$f(N - (i - 1))$	$f = K$, $N = E_0$, $(i - 1) = E_c(t)$
Basic execution model (Musa, 1975)	$b_1 \, b_0[1 - m(t)/b_0]$ where $m(t) = b_0[1 - \exp(-b_1 t)]$	$b_0 = E_0$, $b_1 = K$, $m(t) = E_c(t)$

Logarithmic Poisson model (Musa and Okumoto, 1984b)	$b_1 b_0 \exp(-m(t)/b_0)$, where $m(t) = b_0 \ln(b_1 t + 1)$	$b_1 b_0 = KE_0$, $E_0 - E_c(t) = E_0 \exp(-m(t)/b_0)$

3.3. The Bayesian Models

The Bayesian approach essentially challenges some of the deterministic assumptions made in the classical Markovian and exponential approaches. For example, the exponential model assumes that each fault contributes equally to the overall program failure intensity. The Bayesian approach argues that each fault's contribution to the overall failure intensity is unknown and can be modeled as originating from a given random distribution (with unknown parameters) of values (Littlewood, 1981). The analysis then proceeds along traditional Bayesian techniques.

This group of models views reliability growth and prediction in a Bayesian framework rather than the “traditional” one considered in the previous sections. The previous models only allow change in the reliability whenever an error occurs. Most of them also look at the impact of each fault as being of the same magnitude. A Bayesian model takes a subjective viewpoint in that if no failures occur while the software is observed, then the reliability should increase, reflecting the growing confidence in the software by the user. The reliability is therefore a reflection of both the number of faults that have been detected and the amount of failure-free operation. This reflection is expressed in terms of a prior distribution representing the view from past data and a posterior distribution that incorporates the past and the current data.

The Bayesian models also reflect the belief that different faults have different impacts on the reliability of the program. The number of faults is not as important as their impacts. If we have a program that has a number of faults in seldom used code, is that program less reliable than one that has only one in the part of the code that is used often? The Bayesian would say “No!” The Bayesian modeler says that it is more important to look at the behavior of the software than to estimate the number of faults in it. The mean time to failure would therefore be a very important statistic in this framework.

The prior distribution reflecting the view of the model parameters from past data is an essential part of this methodology. It reflects the viewpoint that one should incorporate past information, say, projects of similar nature etc., in estimating reliability statistics for the present and future. This distribution is simultaneously one of the Bayesian's framework strengths and weaknesses. One should incorporate the past, but *how* is the question.

The basic idea of the mathematics behind this theory is as follows. Suppose that we have a distribution for our reliability data that depends on some unknown parameters, x i.e., $f_T(\boldsymbol{t} | x)$, and a prior $g(x ; f)$ that reflects our views on those parameters, x from historical data. Once additional data have been gathered through the vector **t** [note that boldfacing of a component denote a possible vector of subcomponents to allow for multidimensionality i.e., x = (x_1 , x_2 , ¼ x_K)], our view of the parameter x changes. That change is reflected in the posterior distribution, which is calculated as

$$h(\xi \mid \boldsymbol{t}; \phi) = \frac{f_T(\boldsymbol{t} \mid \xi) g(\xi; \phi)}{\int \cdots f_T(\boldsymbol{t} \mid \xi) g(\xi; \phi) d\xi} = \frac{f_T(\boldsymbol{t} \mid \xi) g(\xi; \phi)}{f_T(\boldsymbol{t}; \phi)} \tag{19}$$

Using the posterior distribution, editor estimates of x can then be obtained, leading to reliability estimates involving x. A common Bayesian procedure is to define a loss function, x(***t***) is an estimate of x, and then choose the estimate of x that minimizes the expected loss using the posterior distribution. For a squared-error function or quadratic loss function, *l*(x′(***t***) , x) where x′(***t***) is an estimate of x, and then choose the estimate of x that minimizes the expected loss using the posterior distribution. For a squared-error function or quadratic loss function, *l*(x′(***t***) , x) = (x′(***t***) –x)^2, the estimate is the mean of the posterior distribution, E{ x|*t*}. The reader is referred to any mathematical statistics book for further details (e.g., Mood *et al*., 1974).

The Littlewood–Verrall model (1973, 1974) is probably the best example of this class of model. The model tries to account for fault generation in the fault correction process by allowing for the probability that the software program could become “less reliable” than before. With each fault correction, a sequence of software programs is generated. Each is obtained from its predecessor by attempting to fix the fault. Because of the uncertainty, new version could be “better” or “worse” than its predecessor. Thus another source of variation is introduced. This is reflected in the fact that the parameters that define the failure time distributions are taken to be random. The distribution of failure times is, as in the earlier models, assumed to be exponential with a certain failure rate, but it is that rate that is assumed to be random rather than constant as before. The distribution of this rate, i.e., the prior, is assumed to be a gamma distribution with shape a and scale parameter y(*i*) . y(*i*) function is further considered as either a linear form or a quadratic form with parameters b_0 and b_1.

A paper by Mazzuchi and Soyer (1988) considers a variation of this model by assuming all of the parameters a , b_0 and b_1 are random variables with appropriate priors. Employing some approximations because of computational difficulties, they then obtain some corresponding results. Musa (1984) considered the use of a rational function for y(*i*). He felt that this parameter should be inversely related to the number of failures remaining. The form of this function was expressed as

$$\psi(i) = \frac{N(\alpha + 1)}{\lambda_0 (N - i)}$$

Here *N* is the expected number of faults within the software as time becomes infinite, l_0 is the initial failure intensity function, and a is the parameter of the gamma distribution considered earlier. The index *i* is the failure index. One can see as the number of remaining failures decreases the scale parameter, y(*i*), increases. Another variation of this model is the one considered by Keiller and co-workers (1983). Again successive failures follow an exponential distribution with an associated gamma prior. However for this case, the reliability growth is induced by the shape parameter a rather than the scale parameter y(*i*).

Any of the classical models can be made Bayesian by specifying appropriate distributions for one or more of their parameters. Interestingly, most of the Bayesian models use the exponential model as a starting point (e.g., Littlewood and Verrall, 1974; Goel, 1977; Littlewood, 1980; Jewell, 1985; Langberg and Singpurwalla, 1985; Littlewood and Sofer, 1987; Becker and Camarinopoulos, 1990; Csenki, 1990) or are completely new models (Littlewood and Verrall, 1973; Thompson and Chelson, 1980; Kyparisi and Singpurwalla 1984; Liu 1987). It seems, however, that the Bayesian approach suffers from its complexity and from the difficulty in choosing appropriate distributions for the parameters. Added to this is the fact that most software engineers do not have the required statistical background to completely understand and appreciate Bayesian models. The latter is perhaps the main reason why these models have not enjoyed the same attention as the classical models (there are almost 5 times as many classical models as Bayesian models, and they are used in a great majority of the practical applications). Note that Bayesian models lead to the intuitive notion that earlier failure corrections have a greater effect than do later ones on the program failure intensity. However, many classical Markovian and exponential models also share this property.

3.4. Comparison of Different Software Reliability Models

Various models proposed in literature tend to give quite different predictions for the same set of failure data. It should be noted that this kind of behavior is not unique to software reliability modeling but is typical of models that are used to project values in time and not merely represent current values. Furthermore, a particular model may give reasonable predictions on one set of failure data and unreasonable predictions on another. Consequently, potential users may be confused and adrift with little guidance as to which models may be best for their applications.

The search for the best software reliability model(s) started in the late 1970s and early 1980s. Initial efforts at comparison by Schick and Wolverton (1978) and Sukert (1979) suffered from a lack of good failure data and a lack of agreement on the criteria to be used in making the comparisons. The former deficiency was remedied to some degree when 50 reasonably good-quality sets of failure data were published (Lyu, 1996). The data sets were collected under careful supervision and control and represent a wide variety of applications including real-time command and control, commercial, military, and space systems.

The latter deficiency was remedied when Iannino and co-workers (1984) worked out a consensus from many experts in the field on the comparison criteria to be employed. The proposed criteria include the following.

1. The capability of a model to predict future failure behavior from known or assumed characteristics of the software; for example, estimated lines of code, language planned to be used, and present and past failure behavior (i.e., failure data). This is significant principally when the failure behavior is changing, as occurs during system testing.
2. The ability of a model to estimate with satisfactory accuracy the quantities needed for planning and managing software development projects or for running operational software systems. These quantities include the present failure intensity, the expected date of reaching the failure intensity objective, and resource and cost requirements related to achieving the failure intensity objective.
3. The quality of modeling assumptions (e.g., support by data, plausibility, clarity, and explicitness).
4. The degree of model applicability across software products that vary in size, structure, and function, different development environments, different operational environments, and different life-cycle phases. Common situations encountered in practice that must be dealt with include programs being integrated in phases, system testing driven by a strategy to test one system feature at a time, the use of varying performance computers, and the need to handle different failure severity classes.
5. The degree of model simplicity (e.g., simple and inexpensive data collection, concepts, and computer implementation).

3.5. Model Selection Approach

Although model comparison criteria can be set as above, it is still very difficult to identify *a priori* those characteristics of a program that will ensure that a particular model can be trusted to produce accurate reliability predictions. In fact, this is not surprising, since the models involve rather crude assumptions about what may be a quite complex underlying failure process. There are many things that might impact on the properties of the failure process that are simply ignored by the models. Examples include the nature of the operational environment, the internal fault-handling procedure (e.g., whether the software is fault-tolerant), etc. Such factors represent a source of uncontrolled variability in the properties of the failure process that is not treated by any of the models. In the absence of specific ways of taking account of such factors, we can expect the models to vary in their performance as the factors vary from one data source to another.

Consequently, some researchers advocated the approach in selecting the best model by using a set of models at the same time on a given set of failure data and then picking the one that is working best (Brocklehurst *et al*., 1990; Lyu and Nikora, 1991). They suggested comparing a prediction with the actual observation (when this is later made), and recursively build up a sequence of such prediction/observation comparisons. From this sequence we should be able to gain information about the accuracy of past predictions, and so make decisions about the current prediction (i.e., which model to trust, if any). This approach, originally seen as impractical, becomes attractive with the help of software reliability modeling tools such as CASRE, SMERFS, and SRMP (Lyu, 1996).

The basic idea in analyzing model predictive accuracy is to perform recursive comparison of predictions with eventual outcomes. Dawid (1984) proposed a statistical measure that can be engaged: the prequential likelihood ratio (PLR), which can be extended to obtain a *u* plot and *y* plot.

First we define the PLR. Let us assume that we have observed the successive times between failures t_1 , t_2 , ¼ , t_{j-1}, and we want to predict the next time to failure T_j. We shall do this by using one of the models to obtain an estimate, $\hat{F}_j(t)$ of the true (but unknown) distribution function $F_j(t) \equiv P(T_j < t)$, the probability that time to the *j*th (the next) failure is less than *t*. True predictive pdf is noted as $f_j(t)$, with estimates of this PDF, $\hat{f}_j^{\,A}(t_j)$ and, $\hat{f}_j^{\,B}(t_j)$, coming from two different models, *A* and *B*. After making these two predictions, which are based only on the data we have seen prior to stage *j*, we wait and eventually see the next failure occur after a time t_j. Since this is a realization of a random variable whose distribution is the true one, we would expect t_j to lie in the main body of this true distribution; that is, it is more likely to occur where $f_j(t_j)$ is larger. If we evaluate the two predictive PDFs at this value of *t*, there will be a tendency for $\hat{f}_j^{\,A}(t_j)$ to be larger than $\hat{f}_j^{\,B}(t_j)$ if the predictions from model *A* are more accurate than those from model *B*; i.e., $\hat{f}_j^{\,A}(t_j) / \hat{F}_j^{\,B}(t_j)$ will tend to be larger than 1. This is because the *A* PDF tends to have more large values close to the large values of the true distribution than does the *B* PDF. In fact, this is what we mean when we say informally that “the *A* predictions are closer to the truth than the *B* predictions”—that the value of the *A* PDF tends to be everywhere closer to that of the true PDF than is the value of the *B* PDF. The PLR then is merely a running product of such terms over many successive predictions:

$$PLR_i^{AB} = \prod_{j=8}^{j=1} \frac{\hat{f}_j^{\,A}(t_j)}{\hat{f}_j^{\,B}(t_j)} \tag{20}$$

and this should tend to increase with *i* if the *A* predictions are better than the *B* predictions. Conversely, superiority of *B* over A will be indicated if this product shows a decreasing trend. Namely, given that no ***a priori*** preference is given to two models, *A* and *B*, then PLR_i^{AB} indicates the likelihood that model *A* will provide more accurate predictions than model *B*, or vice versa.

The purpose of the *u* plot is to determine whether the predictions, $\hat{F}_j(t)$, are on average close to the true distributions, $F_j(t)$. It can be shown that, if the random variable T_j truly had the distribution $\hat{F}_j(t)$—in other words, if the prediction and the truth were identical—then the random variable $U_j = \hat{F}_j(t)$ would be uniformly distributed on (0,1). This is called the ***probability integral transform*** in statistics. If we were to observe the realization t_j of T_j, and calculated, $u_j = \hat{F}_j(t_j)$, the number u_j will be a realization of a uniform random variable. When we do this for a sequence of predictions, we get a sequence $\{u_j\}$, which should look like a random sample from a uniform distribution. Any departure from such uniformity will indicate some kind of deviation between the sequence of prediction, $\{\tilde{F}_j(t)\}$, and the truth $\{F_j(t)\}$.

One way of looking for departure from uniformity is by plotting the sample distribution function of the { m$_j$} sequence. This is a step function constructed as follows. For a sequence of predictions $\tilde{F}_j(t), j = s, \ \dots, \ i$ on the interval (0,1), place the points u_s , u_{s+1} , ¼ , u_i (each of these is a number between 0 and 1); then from left to right plot an increasing step function. With each step of height $1/(i - s + 2)$ at each m on the abscissa, as shown in Figure 5. The range of the resulting monotonically increasing function is (0,1). And we call it the ***u*** plot. A common way of testing whether the departure is significant is via the Kolmogorov–Smirnov distance, which is the maximum vertical deviation between the plot and the line of unit slope.

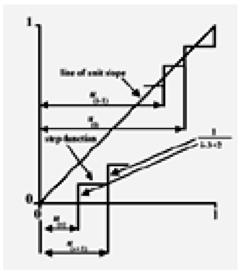

Figure 5. How to draw the *u* plot for predictions of T_s , ¼ , T_i. Here, $\{u_{(s)}$, $u_{(s+1)}$, ¼ , $u_{(i)}\}$ are the original set of values $\{u_s$, u_{s+1} , ¼ , $u_i\}$ reordered in ascending order of magnitude. [Full View]

Note that the number u_j is the estimate we would have made, before the event, of the probability that the next failure will occur before t_j, the time when it *actually does* eventually occur. In the case of consistently over optimistic predictions, this number would therefore tend to be smaller than it would be if the predictions were accurate. That means that the u_j values will tend to bunch too far to the left in the (0,1) interval, and the resulting ***u*** plot will tend to be above the line of unit slope. A similar argument shows that, if a ***u*** plot is entirely below the line of unit slope, it indicates that the predictions are too pessimistic.

Remember that the u_j sequence should look like a sequence of independent, identically distributed uniform random variables on (0,1). Since the range, (0,1), remains constant, any trend will be difficult to detect in the u_j sequence, which will look very regular. If, however, we make the transformation $x_j = -\ln(1 - \text{m}_j)$, we produce a sequence of numbers that should look like realizations of independent, identically distributed unit *exponential* random variables. That is, the sequence should look like the realization of the successive interevent times of a homogeneous Poisson process; any trend in the u_i values will show itself as a nonconstant rate for this process. There are many tests for trend in a Poisson process. We begin by normalizing the whole transformed sequence onto (0,1). That is, for a sequence of predictions from stage *s* through stage *i*, we define

$$y_k = \frac{\sum_{j=s}^{k} x_j}{\sum_{j=s}^{i} x_j}, \quad \text{where} \quad k = s, \ \dots, \ i-1 \tag{21}$$

A step function with steps of size $1/(i - s + 1)$ at the points y_s , y_{s+1} , ¼ y_{i-1} is drawn from the left on the interval (0,1), exactly as in the case of the ***u*** plot. We refer this sequence as a *y* plot. The *y* plot that can be used to detect whether there is trend in model bias represented by the ***u*** plot.

In summary, PLR will only tell us about *relative* performance among competing models, but it will do this in the most *general* way possible, with the underlying theory (Dawid, 1984), providing an assurance that all deficiencies have been taken into account. The ***u*** plot and *y* plot, on the other hand, give us some *absolute* information, but only about certain *specific* ways in which predictions can differ from the truth.

4. Emerging Techniques

4.1. Recalibration Approach

One promising development for improving model predictions is adaptive prediction (Abdel-Ghaly *et al*., 1986). Adaptive prediction is a statistical procedure that allows a model to "learn" from its past mistakes and produce improved predictions by removing prediction bias experienced with models. One way of expressing the notion of prediction biasmore formally is to say that at stage ***i*** there is some function G, which relates the predicted to the true distribution of the time-to-next-failure random variable $F_i(t) = G_i[F_s(t)$. Such a function, if we knew it, would tell us everything there is to know about the bias in the predictions being made at a particular stage. In particular, if we knew *G*, we could recover the true distribution, $F_i(t)$, from the inaccurate prediction, $F_s(t)$. If there is only a single *G* function for the whole sequence of predictions, we might try to estimate it and thus provide a means of recalibrating future inaccurate predictions to produce better ones. When this occurs, we have the opportunity of *estimating* this bias function from the earlier predictions we have made by comparing these with the observed outcomes. In fact, it can be shown that the ***u*** plot based on these earlier predictions is a suitable estimator of *G*.

Recalibration techniques for assessing the quality of adapted predictions relative to the raw predictions were thus developed (Brocklehurst *et al*., 1990). The steps of the recalibration procedure are as follows:

1. Obtain the ***u*** plot, say, G_i^* based on the raw predictions, $F_s(t), \ \dots, \ F_{i-1}(t)$, that have been made before stage *i*. This can be thought of as an estimate of the function *G* that is assumed to represent the (approximately) constant relationship between prediction and truth.
2. Obtain $F_i(t)$, the raw prediction at stage *i*.
3. Calculate the recalibrated prediction, $F_i^*(t) = G_i^*[F_i(t)]$.
4. Repeat this at each stage *i*. In this way a sequence of recalibrated predictions will

result.

The most important point to note about this procedure is that it is truly predictive, inasmuch as only the past is used to predict the future. The recalibration techniques have shown great improvement in the software reliability modeling results.

4.2. Linear Combination Models

Lyu and Nikora (1991, 1992) observed that linear combinations of model results, even in its simplest format, appear to provide more accurate predictions than the individual models themselves. They proposed the following strategy in forming combination models:

1. Identify a basic set of models (the component models). If the testing environment for the development effort can be characterized, select models whose assumptions are closest to the actual testing practices.
2. Select models whose predictive biases tend to cancel each other. We have seen that models can have optimistic or pessimistic biases.
3. Separately apply each component model to the data.
4. Apply certain selected criteria to weight the selected component models (e.g., changes in the prequential likelihood) and form the combination model for the final predictions. Weights can be either static or dynamically determined.

In general, this approach is expressed as a mixed distribution

$$\hat{F}_i(t) = \sum_{j=1}^{n} \omega j(t) \hat{F}_i^{\,j}(t) \tag{22}$$

where *n* represents the number of models and $\hat{f}_i^j(t)$ is the predictive probability density function of the *j*th component model, given that *i* – 1 observations of failure data have been made. Note that

$$\sum_{j=1}^{n} \omega j(t) = 1 \quad \text{for all} \quad t$$

The linear combination model tends to preserve the features inherited from its component models. Also, because each component model performs reliability calculations independently, the combination model remains fairly simple. The component models are plugged into the combination model only at the last stage for final predictions.

4.3. Phase-Based Model for Early Prediction

Gaffney and Davis (1988a, 1988b) developed the phase-based model. It makes use of error statistics obtained during the technical review of requirements, design, and the implementation to predict the reliability during test and operation.

The assumptions for this model are

1. The development effort's current staffing level is directly related to the number of faults discovered during the development phase, which is assumed to follow a Rayleigh curve.
2. The fault discovery curve is monomodal.
3. Code size estimates are available during the early phases of a development effort. The model expects that fault densities will be expressed in terms of the number of faults per thousand lines of source code (KSLOC), which means that faults found during the requirements analysis and software design will have to be normalized by the code size estimates.

Their model is then expressed as

$$\Delta V_t \quad = \text{number of discovered errors per KSLOC fro} \\ \text{time } t-1 \text{ to } t \\ = E\left[\exp\left(-B(t-1)^2\right) - \exp\left(-Bt^2\right)\right]$$

where E = total lifetime fault rate expressed in faults per thousand source lines of code (KSLOC)

t=fault discover index with (t=1—requirements analysis; t =2—software design;

t=3—implementation; t=4—unit test, t=5—software integration;

t=6—system test, t =7—acceptance test; note that t is not treated in the traditional sense of time)

$B=\frac{1}{2\tau_P^2}$, where t_p is the fault discovery phase constant, the peak of a continuous curve fit to the failure data (the point at which 39% of the faults have been discovered). The cumulative form of the model is $V_t = E[1 - \exp(-Bt^2)]$, where V_t is the number of faults per KSLOC that have been discovered through phase t. As data become available, B and E can be estimated. This quantity can also be used to estimate the number of remaining faults at stage t by multiplying $E\exp(-Bt^2)$ by the number of source line statements at that point.

4.4. Rome Laboratories Work

One of the earliest and most well known efforts to predict software reliability in the earlier phases of the lifecycle was the work initiated by the RADC (1987). For their model, they developed predictions of fault density, which could then be transformed into other reliability measures such as failure rates. To do this, the researchers selected a number of factors that they felt could be related to error density at the earlier phases. Included in the list were

A— application type (e.g., real-time control systems, scientific, information management)

D— development environment (characterized by development methodology and available tools)

The types of development environments considered are organic, semidetached, and embedded modes.

Additional codes are as follows:

Requirements and design representation metrics

SA— anomaly management

ST— traceability

SQ— incorporation of Quality Review results into the software

Software implementation metrics

SL— language type (e.g., assembly, high order.)

SS— program size

SM— modularity

SU— extent of reuse

SX— complexity

SR— incorporation of standards review results into the software

The initial fault density prediction is then

$$\delta_0 = A * D(SA * ST * SQ *)(SL * SS * SM * SU * SX * SR) \tag{24}$$

Once the initial fault density has been found, a prediction of the initial failure rate is made as follows:

$$\begin{aligned} \lambda_0 &= F * K * (\delta_0 * \text{number of line of surce code}) \\ &= F * K * W_0 \end{aligned} \tag{25}$$

The number of inherent faults = W_0 = (d_0 * number of lines of source code); F is the linear execution frequency of the program, and K is the fault expose ratio, ($1.4 \times 10^{-7} \leq K \leq 10.6 \times 10^{-7}$). By letting $F = R/I$, where R is the average instruction rate, i is the number of object instructions in the program and then further rewriting I as I_s * Q_X where I_s is the number of source instructions and Q_X is the code expansion ratio [the ratio of machine instructions to source instructions (RADC indicates an average value of 4)], the initial failure rate can be expressed as

$$\lambda_0 = \left(R\frac{K}{Q_X}\right)\left(\frac{W_0}{I_s}\right) \tag{26}$$

4.5. Other Approaches

The reliability theory established in this article is considered as a black-box approach; i.e., only the failure data from the software systems under measurement are included in the modeling process, while the system structures are ignored. As component-based software development (Kozaczynski and Booch, 1998) became popular more recently, white-box approaches to software reliability have gained considerable attention. Smidts and Sova (1999) considered an architecture-oriented modeling approach for software reliability estimation based on decomposition of requirements into software functions and attributes. Kuball and co-workers (1999) introduced a hierarchical model to estimate the probability of failure on demand of a component-based software system, under a Bayesian framework. Lyu and co-workers (2002) formulated a testing resource allocation requirement for component-based software development as a combinatorial optimization problem with known cost, reliability, effort, and other attributes of the components.

Generally speaking, the white-box approach to software reliability extends the black-box approach by including structural parameters into the reliability engineering process. Parameterization of software reliability models can also be based on alternate sources of information such as the metrics explored in the early prediction models. Other metrics include test coverage and system workload. Piwowarski and co-workers (1993) proposed a simple coverage-based reliability growth model. Malaiya and co-workers (1994) presented a logarithmic model that relates testing effort to test coverage, which can be directly linked to defect coverage and growth of reliability. Chen and co-workers (2001) included testing coverage into time-basis adjustment for more accurate software reliability measurement from the growth modeling. Gokhale and Trivedi (1999) worked out a software reliability modeling approach to include system structure as well as workload considerations.

Another class of approaches is to model various quality metrics (the dependent variables) on the basis of their relationships with other independent variables (size, number of data items, complexity, operators, operands, etc.) to establish reliability and quality predictions. Agresti and Evanco (1992) attempted to develop a model for predicting defect density the basis of based on product and process characteristics for Ada development efforts. They employed a multivariate linear regression model based on a log–log relationship. The modeling relationship, on the other hand, was assumed as linear models in Munson and Khoshgoftaar (1990), and Khoshgoftaar and co-workers (1992a), where various estimation techniques were evaluated for the creation of those linear models using regression techniques. Khoshgoftaar and Munson (1990) specifically considered an approach that used only complexity metrics to help predict indicators of quality. Gokhale and Lyu (1997) applied a regression tree analysis technique to establish the modeling relation between the dependent (predicted) variables and the independent (known) variables. Schneidewind (2000) developed the Boolean discriminant function and relative critical value deviation to discriminate between fault-prone and non-fault-prone modules for software quality control and maintenance purposes. Neural network models were also proposed to classify quality attributes such as reliability or code that appears to have problems. Khoshgoftaar and co-workers (1992b; 1993) and Karunanithi and co-workers (1991) investigated the problem by a supervised learning model, while Guo and Lyu (2000) approached the problem by an unsupervised learning model.

Simulation techniques for software reliability predication also attracted research investigations. Von Mayrhauser and co-workers (1993) performed experiments to investigate the nature of relationships between software failures and program structures, and established artifact-based simulation techniques for reliability prediction purpose. Tausworthe and Lyu (1996) demonstrated most of the traditional software reliability can be easily simulated by simple Monte Carlo method, and they established a rate-based simulation technique to simulate a complete software development life cycle, including the life cycle of faults and failures. Gokhale and co-workers (1998) further extended this simulation technique to analyze reliability of component-based systems under various software architectures and configurations.

In the area of time series modeling, we refer the reader to a paper by Singpurwalla and Soyer (1985) as a starting point, where the time between failures was formulated as an autoregressive process with random coefficients to reflect uncertainty in the specification of the power law as the autoregressive model.

5. Acknowledgment

The work described in this article by M. R. Lyu was fully supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region (Project No. CUHK4432/99E).

Bibliography

A. A. Abdel-Ghaly, P. Y. Chan, and B. Littlewood, Evaluation of Competing Software Reliability Predictions, IEEE Transactions on Software Engineering **SE-12**(9), 95–967 (September 1986).

W. W. Agresti and W. M. Evanco, Projecting Software Defects From Analyzing Ada Designs, IEEE Transactions on Software Engineering **SE-18**(11), 98–997 (November 1992).

AIAA (American Institute of Aeronautic and Astronautics), *Recommended Practice for Software Reliability*, ANSI/AIAA R-013–1992, AIAA, Washington, DC, 1993.

J. E. Angus, R. E. Schafer, and A. Sukert, Software Reliability Model Validation, *Proceedings of the 1980 Annual Reliability and Maintenance Symposium*, 1980.

ANSI/IEEE, Standard Glossary of Software Engineering Terminology, STD-729–1991, ANSI/IEEE, Washington, DC, 1991.

R. E. Barlow and E. M. Scheuer, Reliability Growth During a Development Testing Program, Technometrics **8**(1) (1966).

G. Becker and L. Camarinopoulos, A Bayesian Estimation Method for the Failure Rate of a Possibly Correct Program, IEEE Transactions on Software Engineering **SE-16**(11), 130–1310 (November 1990).

S. Brocklehurst, P. Y. Chan, B. Littlewood, and J. Snell, Recalibrating Software Reliability Models, IEEE Transactions on Software Engineering **SE-16**(4) (1990).

W. D. Brooks and R. W. Motley, *Analysis of Discrete Software Reliability Models,* RADC-TR-80–84, Rome Air Development Center, Rome, NY, 1980.

M. H. Chen, M. R. Lyu, and E. Wong, Effect of Code Coverage on Software Reliability Measurement, *IEEE Transactions on Reliability* (2001).

R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, Orthogonal Defect Classification—A Concept for In-Process Measurements, IEEE Transactions on Software Engineering **SE-18**(11), 94–956 (November 1992).

W. J. Corcoran, H. Weingarten, and P. W. Zehna, Reliability After Corrective Action, Management Science **10** (1964).

J. de. S. Coutinho, Software Reliability Growth, *IEEE Symposium on Computer Software Reliability*, 1973.

L. H. Crow, Reliability Analysis for Complex Repairable Systems, in F. Proshan and R. J. Serfling, eds., *Reliability and Biometry*, SIAM, Philadelphia, PA, 1974.

A. Csenki, Bayes Predictive Analysis of a Fundamental Software Reliability Model, IEEE Transactions on Reliability **R-39**, 177–183 (1990).

A. P. Dawid, Statistical Theory: The Prequential Approach, Journal of the Royal Statistical Society, series A, **147**, 278–292 (1984).

T. Downs, An Approach to the Modeling of Software Testing with Some Applications, IEEE Transactions on Software Engineering **SE-11**(4) (1985).

T. Downs, Extensions to an Approach to the Modeling of Software Testing with Some Performance Comparisons, IEEE Transactions on Software Engineering **SE–12**(9) (1986).

W. K. Ehrlich, A. Iannino, B. S. Prasanna, J. P. Stampfel, and J. R. Wu, How Faults Cause Software Failures: Implications for Software Reliability Engineering, *International Symposium on Software Reliability Engineering*, Astin, TX., 1991.

W. H. Farr, *A Survey of Software Reliability Modeling and Estimation*, NSWC TR–171, Naval Surface Warfare Center, Dahlgreen, VA September 1983.

E. H. Forman and N. D. Singpurwalla, An Empirical Stopping Rule for Debugging and Testing Computer Software, Journal of the American Statistical Association **72** (1977).

E. H. Forman and N. D. Singpurwalla, Optimal Time Intervals for Testing Hypotheses on Computer Software Errors, IEEE Transactions on Reliability **28** (1979).

J. E. Gaffney Jr., and C. F. Davis, An Approach to Estimating Software Errors and Availability, SPC-TR-88–007, version 1.0, March 1988a. *Proceedings of the 11th Minnowbrook Workshop on Software Reliability,* July 1988b.

A. L. Goel, *Summary of Technical Progress on Bayesian Software Prediction Models*, RADC-TR-77–112, Rome Air Development Center, Rome, NY, 1977.

A. L. Goel and K. Okumoto, Time-Dependent Error-Detection Rate Model for Software Reliability and Other Performance Measures, IEEE Transactions on Reliability **R-28**(3) (1979).

S. S. Gokhale and M. R. Lyu, Regression Tree Modeling for the Prediction of Software Quality, *Proceedings of the 3rd ISSAT International Conference on Reliability and Quality in Design*, Anaheim, CA, March 1997, pp. 31–36.

S. S. Gokhale and K. S. Trivedi, A Time/Structure Based Software Reliability Model, Annals of Software Engineering **8** (1999).

S. S. Gokhale, M. R. Lyu, and K. S. Trivedi, Reliability Simulation of Component-Based Software Systems, *Proceedings of the 9th International Symposium on Software Reliability Engineering (ISSRE'98)*, Paderborn, Germany, November 1998, pp. 192–201.

P. Guo and M. R. Lyu, Software Quality Prediction Using Mixture Models with EM Algorithm, *Proceedings of the 1st Asia-Pacific Conference in Quality Software (APAQS2000)*, Hong Kong, October 2000, pp. 69–78.

H. Hecht, Allocation of Resources for Software Reliability, *Proceedings of COMPCON Fall 1981* Washington, DC, 1981.

C. -Y. Huang, S. -Y. Kuo, and M. R. Lyu, A Framework for Modeling Software Reliability Considering Various Testing Efforts and Fault Detection Rates, *IEEE Transactions on Reliability* (2001).

G. R. Hudson, *Program Errors as a Birth and Death Process*, Report SP–3011, System Development Corporation, Santa Monica, CA, 1967.

A. Iannino and J. D. Musa, Software Reliability Engineering at AT&T, *Proceedings of PSAM* Beverly Hills, CA, 1991.

A. Iannino, J. D. Musa, K. Okumoto, and B. Littlewood, Criteria for Software Reliability Model Comparisons, IEEE Transactions on Software Engineering **SE–10**(6) (1984).

Z. Jelinski and P. B. Moranda, Software Reliability Research, in W. Freiberger, ed., *Statistical Computer Performance Evaluation*, Academic Press, New York, 1972.

W. S. Jewell, Bayesian Extensions to a Basic Model of Software Reliability, IEEE Transactions on Software Engineering **SE-11**(12) (1985).

H. Joe and N. Reid, Estimating the Number of Faults in a System, Journal of the American Statistical Association (1984).

N. Karunanithi, Y. K. Malaiya, and D. Whitley, Prediction of Software Reliability Using Neural Networks, *Proceedings of the 2nd International Symposium on Software Reliability Engineering* May 1991, pp. 124–130.

P. A. Keiller, B. Littlewood, D. R. Miller, and A. Sofer, On the Quality of Software Reliability Prediction, NATO ASI Series **F3**, 441–460 (1983).

T. M. Khoshgoftaar and J. C. Munson, Predicting Software Development Errors Using Software Complexity Metrics, IEEE Journal on Selected Areas in Communications **8**(2), 25–264 (Febuary 1990).

T. M. Khoshgoftaar, J. Munson, B. B. Bhattacharya, and G. Richardon, Predictive Modeling Techniques of Software Quality from Software Measures, IEEE Transactions on Software Engineering **SE-18**(11), 97–987 (November 1992a).

T. M. Khoshgoftaar, A. S. Pandya, and H. B. More, A Neural Network Approach for Predicting Software Development Faults, *Proceedings of the 3rd International Symposium on Software Reliability Engineering*, October 1992b, pp.83–89.

T. M. Khoshgoftaar, D. L. Lanning, and A. S. Pandya, A Neural Network Modeling Methodology for the Detection of High-Risk Programs, *Proceedings of the 4th International Symposium on Software Reliability Engineering*, November 1993, pp.302–309.

J. H. Kim, Y. H. Kim, and C. J. Park, A Modified Markov Model for the Estimation of Computer Software Performance, *Operations Research Letters*, p. 1 (1982).

W. Kozaczynski and G. Booch, Component-Based Software Engineering, IEEE Software **155**, 34–36 (1998).

W. Kremer, Birth-Death and Bug Counting, IEEE Transactions on Reliability **R–32**(1) (1983).

S. Kuball, J. May, and G. Hughes, Building a System Failure Rate Estimator by Identifying Component Failure Rates, *Proceedings of the 10th International Symposium on Software Reliability Engineering*, November 1999, pp. 32–41.

J. Kyparisis and N. D. Singpurwalla, Bayesian Inference for the Weibull Process with Applications to Assessing Software Reliability Growth and Predicting Software Failure, *Computer Science and Statistics: The Interface* Elsevier North-Holland, Amsterdam, March 1984, pp. 57–64.

N. Langberg and N. D. Singpurwalla, Unification of Some Software Reliability Models Via the Bayesian Approach, SIAM Journal of Scientific and Statistical Computation **6**(3), 781–790 (1985).

J.-C. Laprie, Dependability Evaluation of Software Systems in Operation, IEEE Transactions on Software Engineering **SE–10**(6) (1984).

B. Littlewood, A Bayesian Differential Debugging Model for Software Reliability, *Proceedings of the IEEE Computer Society, International Computer Software Applications Conference*, 1980 pp. 511–519.

B. Littlewood, Stochastic Reliability-Growth: A Model for Fault Removal in Computer Programs and Hardware-Design, IEEE Transactions on Reliability **R–30**(4) (1981).

B. Littlewood and A. Sofer, A Bayesian Modification to the Jelinski-Moranda Software Reliability Growth Model, Journal of Software Engineering **2**, 30–41 (1987).

B. Littlewood and J. L. Verrall, A Bayesian Reliability Growth Model for Computer Software, Journal of the Royal Statistical Society, Series C, **22**(3), 332–346 (1973).

B. Littlewood and J. L. Verrall, A Bayesian Reliability Model with a Stochastically Monotone Failure Rate, IEEE Transactions on Reliability **R–22**(2) (1974).

B. Littlewood and J. L. Verrall, Likelihood Function of a Debugging Model for Computer Software Reliability, IEEE Transactions on Reliability **R–30** (1981).

G. Liu, A Bayesian Assessing Method of Software Reliability Growth, in S. Osaki and J. Cao, eds., *Reliability Theory and Applications*, World Scientific, Singapore, 1987, pp. 237–244 (1987).

D. K. Lloyd and M. Lipow, *Reliability: Management, Methods, and Mathematics*, 2nd ed., ASQC, Milwaukee, WI, 1984.

M. R. Lyu, *Handbook of Software Reliability Engineering*, McGraw-Hill IEEE Computer Society Press, New York, 1996.

M. R. Lyu and A. Nikora, A Heuristic Approach for Software Reliability Prediction: The Equally-Weighted Linear Combination Model, *Proceedings of the 2nd International Symposium on Software Reliability Engineering*, May 1991, pp.172–181.

M. R. Lyu and A. Nikora, Using Software Reliability Models More Effectively, IEEE Software, pp. 4–52 (July 1992).

M. R. Lyu, S. Rangarajan, and A. P. A. Van Moorsel, Optimal Allocation of Testing Resources for Software Reliability Growth Modeling in Component-Based Software Development, IEEE Transactions on Reliability (2002).

Y. K. Malaiya, N. Li, R. Karcich and B. Skbbe, The Relationship between test Coverage and Reliability *Proceedings of the 5th International Symposium on Software Reliability Engineering*, November 1984.

T. A. Mazzuchi and T. Soyer, A Bayes Empirical-Bayes Model for Software Reliability, IEEE Transactions on Reliability **R-37**, 248–254 (1988).

R. J. Meinhold and N. D. Singpurwalla, Bayesian Analysis of a Commonly Used Model for Describing Software Failures, American Statistician **32** (1983).

D. R. Miller, Exponential Order Statistic Models of Software Reliability Growth, IEEE Transactions on Software Engineering **SE–12**(1) (1986).

A. Mood, F. Graybill, and D. Boes, *Introduction to the Theory of Statistics*, 3rd ed., McGraw-Hill, New York, 1974.

P. B. Moranda, Predictions of Software Reliability During Debugging, *Proceedings of the Annual Reliability and Maintenance Symposium*, Washington, DC, 1975.

P. B. Moranda, Event-Altered Rate Models for General Reliability Analysis, IEEE Transactions on Reliability **R-28**(5), 376–381 (1979).

J. Munson and T. Khoshgoftaar, Regression Modeling of Software Quality: Empirical Investigation, Information of Software Technologies **32**, 10–114 (March 1990).

J. D. Musa, A Theory of Software Reliability and Its Application, IEEE Transactions on Software Engineering **SE-1**(3), 312–327 (1975).

J. D. Musa, *Software Reliability Data* report available from Data and Analysis Center for Software, Rome Air Development Center, Rome, NY, 1979.

J. D. Musa, Software Reliability, in C. R. Vick and C. V. Ramamoorthy, eds., *Handbook of Software Engineering* 1984, pp. 392–412.

J. D. Musa and K. Okumoto, Software Reliability Models: Concepts, Classification, Comparisons, and Practice, NATO ASI Series **F3**, 395–424 (1983).

J. D. Musa and K. Okumoto, Comparison of Time Domains for Software Reliability Models, Journal of Systems and Software **4**(4) (1984a).

J. D. Musa and K. Okumoto, A Logarithmic Poisson Execution Time Model for Software Reliability Measurement, *Proceedings of the 7th International Conference on Software Engineering*, Washington, DC, 1984b, pp. 230–238.

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

M. Ohba, Software Reliability Analysis Models, IBM Journal of Research and Development **28**(4) (1984).

P. Piwowarski, M. Ohba, and J. Caruso, Coverage Measurement Experience During Function Test, *Proceedings of the 15th International Conference on Software Engineering* May 1993, pp. 287–301.

RADC (Rome Air Development Center), *Methodology for Software Reliability Prediction and Assessment*, Technical Report RADC-TR-87–171, Vol. 1 and 2, RADC, Rome, NY, 1987 (revised in Technical Report RL-TR-92–52, 1992).

S. M. Ross, Statistical Estimation of Software Reliability, IEEE Transactions on Software Engineering **SE–11**(5) (1985a).

S. M. Ross, Software Reliability: The Stopping Rule Problem, IEEE Transactions on Software Engineering **SE-11**(12) (1985b).

M. Sahinoglu, Compound-Poisson Software Reliability Model, IEEE Transactions on Software Engineering **SE-18**(7), 624–630 (1992).

G. J. Schick and R. W. Wolverton, Assessment of Software Reliability, *Proceeding of Operation Research* hysica-Verlag, Wurzburg-Wein, 1973.

G. J. Schick and R. W. Wolverton, An Analysis of Competing Software Reliability Models, IEEE Transactions on Software Enginering **SE–4**(2) (1978).

N. F. Schneidewind, An Approach to Software Reliability Prediction and Quality Control, *1972 Fall Joint Computer Conference* Vol. 41, AFIP Press, Montvale, NJ, 1972.

N. F. Schneidewind, Analysis of Error Processes in Computer Software, *Proceedings of the 1975 International Conference on Reliable Software*, Los Angeles, CA, 1975, Vol. 10, No. 6, pp.337–346.

N. F. Schneidewind, Software Quality Control and Prediction Model for Maintenance, Annals of Software Engineering **9** (2000).

J. G. Shanthikumar, A State- and Time-Dependent Error Occurrence-Rate Software Reliability Model with Imperfect Debugging, *Proceedings of the National Computer Conference* 1981.

J. G. Shanthikumar and U. Sumita, A Software Reliability Model with Multiple-Error Introduction and Removal, IEEE Transactions on Reliability **R–36**(4) (1986).

M. L. Shooman, Probabilistic Models for Software Reliability Prediction, in W. Freidberger, ed., *Statistical Computer Performance Evaluation* cademic Press, New York, 1972.

M. L. Shooman, Operational Testing and Software Reliability Estimation During Program Developments, *Record of 1973 IEEE Symposium on Computer Software Reliability,* IEEE Computer Society, New York, 1973.

M. L. Shooman, Structural Models for Software Reliability Prediction, *Proceedings of the 2nd International Conference on Software Engineering* October 1976.

M. L. Shooman, Spectra of Software Reliability and Its Exorcism, *Proceedings of the Joint Automatic Control Conference,* 1977, pp. 225–231.

M. L. Shooman, *Probabilistic Reliability: An Engineering Approach*, 2nd ed., Krieger, New York, 1990.

M. L. Shooman and A. K. Trivedi, A Many-State Markov Model for the Estimation and Prediction of Computer Software Performance Parameters, *Proceedings of the 1975 International Conference on Reliable Software*, 1975.

N. D. Singpurwalla and R. Soyer, Assessing (Software) Reliability Growth Using a Random Coefficient Autoregressive Process and Its Ramifications, IEEE Transactions on Software Engineering **SE–11**(12), 145–1464 (December 1985).

C. Smidts and D. Sova, An Architectural Model for Software Reliability Quantification: Sources of Data, Reliability Engineering and System Safety **64**, 279–290 (1999). Links

D. L. Snyder, *Random Point Processes*, Wiley, New York, 1975.

L. A. Stefanski, An Application of Renewal Theory to Software Reliability, *Proceedings of the 27th Conference on the Design of Experiments in Army Research Development Testing*, 1982, ARO Report 82–2.

A. N. Sukert, Empirical Validation of Three Software Error Prediction Models, IEEE Transactions on Reliability **R-28**(3) (1979).

R. C. Tausworthe and M. R. Lyu, A Generalized Technique for Simulating Software Reliability, *IEEE Software*, pp. 77–88 (March 1996).

W. E. Thompson and P. O. Chelson, On the Specification and Testing of Software Reliability, *1980 Proceedings of the Annual Reliability and Maintainability Symposium*, 1980, pp.379–383.

M. Trachtenberg, The Linear Software Reliability Model and Uniform Testing, IEEE Transactions on Reliability **R–34**(1) (1985).

A. von Mayrhauser, Y. K. Malaiya, J. Keables, and P. K. Srimani, On the Need for Simulation for Better Characterization of Software Reliability, *Proceedings of the 4th International Symposium on Software Reliability Engineering*, Denver, CO, 1993.

W. L. Wagoner, *The Final on a Software Reliability Measurement Study*, Report TOR-0074 (4112)-1, Aerospace Corporation, 1973.

H. K. Weiss, Estimation of Reliability Growth in a Complex System with a Poisson-type Failure, Operations Research **4** (1956).

A. Wolfe, Intel Fixes a Pentium FPU Glitch, EE Times **822**, 1 (November 1994).

M. Xie, *Software Reliability Modeling*, World Scientific, NJ, 1991.

S. Yamada and S. Osaki, Non-homogeneous Error Detection Rate for Software Reliability Growth, in *Stochastic Models in Reliability Theory*, ringer-Verlag, New York, 1984.

S. Yamada, M. Ohba, and S. Osaki, S-Shaped Reliability Growth Modeling for Software Error Detection, IEEE Transactions on Reliability **R-32**(5), 47–484 (December 1983).

S. Yamada, M. Ohba, and S. Osaki, S-Shaped Software Reliability Growth Models and Their Applications, IEEE Transactions on Reliability **R-33**(4), (1984).

K. C. Zinnel, Using Software Reliability Growth Models to Guide Release Decisions,

Proceedings of the 1st International Symposium on Software Reliability Engineering
Washington, DC, 1990.