SOFTWARE RELIABILITY

SOFTWARE RELIABILITY CONCEPTS

Computers are being used in diverse areas for various applications, for example, air traffic control, nuclear reactors, aircraft, real-time military, industrial process control, automotive mechanical and safety control, and hospital patient monitoring systems. As the functionality of computer operations becomes more essential and complicated in our modern society and critical software applications increase in size and complexity, the reliability of computer software becomes more important, and faults in software design become more subtle. A computer system comprises two major components, hardware and software. Although extensive research has been done in the area of hardware reliability, the growing importance of software dictates that the focus shift to software reliability. Software reliability is different from hardware reliability in the sense that software does not wear out or burn out. The software itself does not fail. Rather, flaws within the software can possibly cause a failure in its dependent system.

In recent years, the costs of developing software and the penalty costs of software failures are the major expenses in a system (1). A research study has shown that professional programmers average six software defects for every 1000 lines of code (LOC) written. At that rate, a typical commercial software application of 350,000 LOC can easily contain over 2,000 programming errors including memory-related errors, memory leaks, language-specific errors, errors calling third-party libraries, extra compilation errors, standard library errors, and so on. As software projects become larger, the rate of software defects indeed increases geometrically. Finding software faults is extremely difficult and also very expensive. A Microsoft study shows that it takes an average of 12 programming hours to find and fix a software defect. At this rate, it can take over 24,000 h (or 11.4 work years) to debug a program of 350,000 LOC at a cost of over one million dollars.

Software errors have caused spectacular failures and led to serious consequences in our daily lives. Several examples are as follows. On March 31, 1986 a Mexicana Airlines Boeing 727 airliner crashed into a mountain because the software system did not correctly negotiate the mountain position. From March through June of 1986, the massive Therac-25 radiation therapy machines in Marietta, Georgia; Boston, Massachusetts; and Tyler, Texas overdosed cancer patients, apparently because the computer program controlling the highly automated devices was flawed. On September 17, 1991 a power outage at the AT&T switching facility in New York City interrupted service to 10 million telephone customers for nine hours. The problem was the deletion of three bits of code in a software upgrade and failure to test the software before its installation in the public network. On October 26, 1992 the computer-aided dispatch system of the Ambulance Service in London, which handles more than 5000 requests each day to transport patients in emergency situations, broke down right after installation. This led to serious consequences for many critical patients.

Recently, an inquiry revealed that a software design error and insufficient software testing caused an explosion that ended the maiden flight of the European Space Agency's (ESA) Ariane 5 rocket less than 40 s after liftoff on June 4, 1996. The problems occurred in the Ariane 5's flight control system and were caused by a few lines of Ada code containing three unprotected variables. One of these variables pertained to the rocket launcher's horizontal velocity. A problem occurred when the ESA used the same software for the inertial-reference flight-control system in the Ariane 5 that it used in the Ariane 4. The Ariane 5 has a high initial acceleration and a trajectory that leads to a horizontal velocity acceleration rate five times that found in Ariane 4. Upon liftoff, the Ariane 5's horizontal velocity exceeded a limit that was set by the old software in the backup inertial-reference system's computer. This stopped the primary and backup inertial-reference system computers, which caused the rocket to veer off course and ultimately explode.

Generally, software faults are more insidious and much more difficult to handle than are physical defects. In theory, software can be made that is error free, and unlike hardware components, software does not degrade or wear out but it does deteriorate. The deterioration here, however, is not a function of time. Rather, it is a function of the side effects of changes made to the software in the maintenance phase by correcting latent defects, modifying the code to changing requirements and specifications, environments, and applications, or improving software performance. All design faults are present from the time the software is installed in the computer. In principle, these faults could be removed completely. Yet the goal of perfect software remains evasive. Computer programs, which vary for fairly critical applications between hundreds and millions of lines of code, can make the wrong decision because the particular inputs that triggered the problem had not been tested during the testing phase when faults could have been corrected. Such inputs may even have been misunderstood or unanticipated by the designer who either correctly programmed the wrong interpretation or failed to take the problem into account altogether. These situations and other such events have made it apparent that we must estimate the reliability of the software systems before putting them into operation.

This article is divided into three sections. The first section provides the basic concepts of software reliability and testing, the general characteristics of software reliability including software definitions, software life cycle, software vesus hardware reliability, software verification and validation, and data collection and analysis. The second section presents several existing software reliability models based on a nonhomogeneous Poisson process. The last section presents a new software reliability model, which considers environmental factors, and its application illustrating the model.

Software Reliability Engineering Concepts

Research activities in software reliability engineering have been conducted during the past 25 years, and more than 50 statistical models have been proposed for estimating software reliability (2). Most existing models for predicting software reliability are based purely on observation of failures of the software product. These models also require considerable numbers of failure data to obtain an accurate reliability prediction. Information concerning the development of the software product, the method of failure detection, environmental factors, etc., however, are ignored.

Nevertheless, not many practitioners, developers, or software users use these models to evaluate the reliability of computer software because they do not know how to select and apply them. A survey conducted by the American Society for Quality Control (ASQC) reported in the late 1990s that only 4% of the survey participants responded positively when asked if they use a software reliability model.

Many researchers are currently pursuing the development of statistical models that can be used to evaluate the reliability of real-world software systems. To develop a useful software reliability model and to make sound judgments when using the model, one needs an in-depth understanding of how software is produced; how errors are introduced; how software is tested; how errors occur; and types of errors. Environmental factors can help us in justifying the reasonableness of the assumptions, the usefulness of the model, and the applicability of the model under a given user environment. In other words, these models would be valuable to software developers, users, and practitioners if they can use information about the software development process, incorporating the environmental factors, and can give greater confidence in estimates based on small numbers of failure data.

Why It Costs Too Much. Back in early 1970s when computers were first used in the business world, storage space was at a premium, and the use of a two-digit convention to represent the year seemed appropriate. For example, a date such as April 20, 1998 is typically represented in software as YY/MM/DD, or 98/04/20. Thus January 1, 2000 will look like 00/01/01, which many computers will interpret as January 1, 1900. The Year 2000 Problem is a major software problem of the twentieth century and is very widespread. It affects hardware, embedded firmware, languages and compilers, operating systems, nuclear power plants, air traffic control, security services, database-management systems, communications systems, trasaction processing systems, banking systems, and medical systems. Because the government will have to change an estimated 15 billion lines of code to cope with the Year 2000 Problem, the work may cost up to $30 billion and worldwide cost could approach $600 billion. This estimate, however, reflects only conversion costs and may not include the cost of replacing hardware and testing and upgrading systems.

Basic Definitions and Terminologies

Let us define the following definitions, terms, and other related software reliability engineering terminologies.

Operational Profile. The set of operations that the software can execute given the probabilities of their occurrence.

Software Availability. The probability that a system is not down due to a software fault.

Software Defect. A generic term referring to a fault or a failure.

Software Error. An error made by a programmer or designer, such as a typographical error, an incorrect numerical value, an omission, etc.

Software Failure. A failure that occurs when the user perceives that the software ceases to deliver the expected result with respect to the specification input values. The user may need to identify the severity levels of failures, such as catastrophic, critical, major, and minor, depending on their impacts to the systems. Severity levels may vary from one system to another and from application to application. Typically the severity of a software system effect is classified into the four following categories:

Catastrophic: This category is for disastrous effects, such as loss of human life or permanent loss of property, for example, the effect of an erroneous prescription of medication or an air-traffic controller error.

Critical: This category is for disastrous but restorable damage. It includes damage to equipment where no human life is hurt or where there is major but curable illness or injury.

Major: This category is for serious failures of the software system where there is no physical injury to people or other systems. Included in this category might be erroneous purchase orders or the breakdown of a road vehicle.

Minor: This category is reserved for faults that lead to marginal inconveniences to a software system or its users. Examples might be a vending machine that momentarily cannot provide change or a bank's computer system that is down when a consumer requests a balance.

Software Fault. An error that leads to a fault in the software. Software faults can remain undetected for extended periods of time. Often they are not detected until they cause software failure.

Software Reliability. The probability that software will not fail during a mission.

Software MTTF. The expected time when the next failure is observed due to software faults.

Software MTTR. The expected time to restore a system to operation upon a failure due to software faults.

Software Testing. A verification process for software quality evaluation and improvement.

Software Validation. The process of ensuring that the software is performing the right process.

Software Verification. The process of ensuring that the software is performing the process right.

System Availability. The probability that a system is available when needed.

SOFTWARE DEVELOPMENT LIFE CYCLE

As software becomes an increasingly important part of many different types of systems that perform complex and critical functions in many applications, such as military defense, nuclear reactors, etc., the risk and impacts of software-caused failures have increased dramatically. There is now general agreement on the need to increase software reliability by eliminating errors made during software development. Industry and academic institutions have responded to this need by improving developmental methods in the technology known as software engineering and by employing systematic checks to detect errors in software during and in parallel with the developmental process. Many organizations today make reducing defects their first quality goal. The consumer electronics business, however, pursues a different goal: keeping the number of defects in the field at zero. When electronics products leave the showroom floor, the final destination of these products is unknown. Therefore, detecting and correcting a serious software defect would entail recalling hundreds of thousands of products.

Software Versus Hardware Reliability

The development of hardware reliability theory has a long history and hardware reliability has improved greatly while the size and complexity of software applications have increased. In hardware reliability, the mechanism of failure occurrence is often treated as a black box. Emphasis is on the analysis of failure data. In software reliability, one is interested in the failure mechanism. The emphasis is on the model's assumptions and the interpretation of parameters. Hardware reliability encompasses a wide spectrum of analyses that strive systematically to reduce or eliminate system failures which adversely affect product performance. Reliability also provides the basic approach for assessing safety and risk analysis. The consequence of these considerations is that software quality and reliability must be built into software during the developmental process.

Software reliability strives systematically to reduce or eliminate system failures which adversely affect performance of a software program. Software systems do not degrade over time unless they are modified. Although many of the reliability and testing concepts and techniques of hardware are applicable to software, there are many differences. Therefore, a comparison of software reliability and hardware reliability would be useful in developing software reliability modeling. Table 1 shows the differences and similarities between the two. In hardware, materials deteriorate over time. Hence, calendar time is a widely accepted index for a reliability function. In software, failures never happen if the program is not used. In the context of software reliability, *time* is more appropriately interpreted as the *stress* placed on or *amount of work* performed by the software. The following time units are generally accepted as indices of the software reliability function:

Execution time: CPU time; time during which the CPU is busy

Operating time: Time the software is in use

Calendar time: Index used for software running 24 h a day

Run: A job submitted to the CPU

Instruction: Number of instructions executed

Path: The execution sequence of an input

Software Testing Concepts

Software is a collection of instructions or statements in a computer language. It is also called a computer program, or simply a program. Upon the execution of a program, an input state is translated into an output state. Hence, a program can be regarded as a function f, mapping the input space to the output space (f: input → output), where the input space is the set of all input states and the output space is the set of all output states. An input state can be defined as a combination of input variables or a typical transaction to the program.

A software program is designed to perform specified functions. When the actual output deviates from the expected output, a failure occurs. However, the definition of failure differs from application to application and should be clearly defined

Table 1. Software Reliability versus Hardware Reliability

Software Reliability	Hardware Reliability
Without considering program evolution, failure rate is statistically nonincreasing.	Failure rate has a bathtub curve. The burn-in state is similar to the software debugging state.
Failures never occur if the software is not used.	Material deterioration can cause failures even though the system is not used.
Failure mechanism is studied.	Failure mechanism is also studied, similar to the software.
CPU time and run are two popular indices for the reliability measure.	Calendar time is a generally accepted index for the reliability measure.
Most models are analytically derived from assumptions. Emphasis is on developing the model, the interpretation of the model assumptions, and the physical meaning of the parameters.	Failure data are fitted to some distributions. The selection of the underlying distribution is based on the analysis of failure data and experiences. Emphasis is placed on analyzing failure data.
Failures are caused by incorrect logic, incorrect statements, or incorrect input data. This is similar to the design errors of a complex hardware system.	Failures are caused by material deterioration, random failures, design errors, misuse, and environment.
Software reliability can be improved by increasing the testing effort and by correcting detected faults. Reliability tends to change continuously during testing periods due to the addition of problems in new code or to the removal of problems by debugging errors.	Hardware reliability can be improved by improving design, better material, applying redundancy, and accelerated life testing.
Software repairs establish a new piece of software.	Hardware repairs restore the original condition.
Software failures are rarely preceded by warnings.	Hardware failures are usually preceded by warnings.
Software components have rarely been standardized.	Hardware components can be standardized.
Software essentially requires infinite testing.	Hardware can usually be tested exhaustively.

in specifications. For instance, a response time of 30 s could be a serious failure for an air traffic control system but is acceptable for an airline reservation system. A fault is incorrect logic, an incorrect instruction, or an inadequate instruction that, by execution, causes a failure. In other words, faults are the sources of failures, and failures are the realization of faults. When a failure occurs, there must be a corresponding fault in the program, but the existence of faults may not cause the program to fail because a program never fails as long as the faulty statements are not executed.

Software reliability is the probability that a given software functions without failure in a given environmental condition during a specified time. Another deterministic model defines software reliability as the probability of successful execution(s) of an input state randomly selected from the input space under specified operating conditions. Another definition is the probability of failure-free execution of the software for a specified time in a specified environment. For example, an operating system with a reliability of 95% for 8 h for an average user should work 95 out of 100 periods of 8 h without any problems. A software failure here means the inability to perform an intended task specified by a requirement. A software fault is an error in the program source-text, which causes a software failure when the program is executed under certain conditions. Hence a software fault is generated at the moment a programmer, designer, or system analyst makes a mistake.

Software testing is the process of executing a program to find an error. A good test case is one that has a high probability of finding undiscovered error(s). In software testing, it is not possible, if not unrealistic, to continue testing the software until all faults are detected and removed, because, for most computer programs, testing of all possible inputs would require millions of years. Therefore, failure probabilities must be inferred from testing a sample of all possible input states, called the input space. In other words, input space is the set of all possible input states. Similarly, output space is the set of all possible output states for a given software and input space. As we know, different inputs have different chances of being selected, and we can never be sure which inputs are selected in the operational phase of real-world applications. During the operational phase, some input states are executed more frequently than others. A probability can be assigned to each input state to form the operational profile of the program. This operational profile can be used to construct the software reliability model. This type of model is also called the input-domain model.

Software Life Cycle

A software life cycle provides a systematic approach to developing, using, operating, and maintaining any software system. The standard definition of the software life cycle as follows: "That period of time in which the software is conceived, developed and used." A software life cycle consists of the following five successive phases which are shown in Fig. 1:

1. Analysis
2. Design
3. Coding
4. Testing
5. Operation

In the early phases of the software life cycle, a predictive model is needed because no failure data are available. This type of model predicts the number of initial faults in the software before testing.

Analysis Phase. The analysis phase is the first step in the software development process. It is also the most important phase in the whole process and the foundation of building a successful software product. A survey at the North Jersey Software Process Improvement Network workshop in 1995 showed that, on average, about 35% of the effort in software

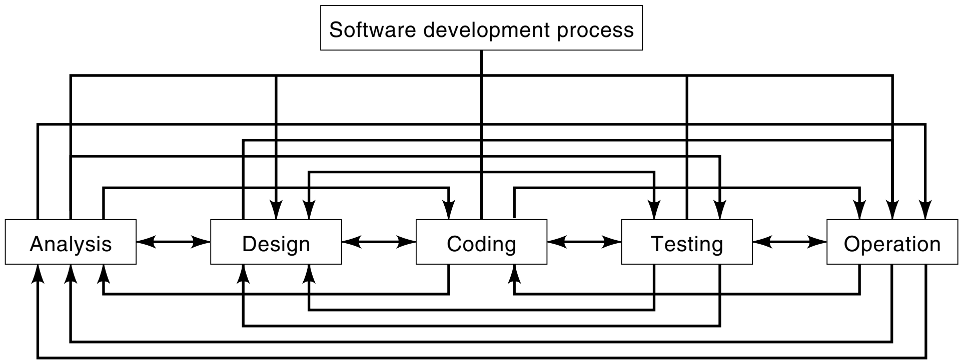

Figure 1. A software development life cycle.

development projects should be concentrated in the analysis phase. The purpose of the analysis phase is to define the requirements and provide specifications for the subsequent phases and activities. The analysis phase is the foundation of building a successful reliable software product and is composed of three major activities: problem definition, requirements, and specifications.

Problem definition develops the problem statement and the scope of the project. It is important to understand what the user's problem is and why the user needs a software product to solve the problem. The requirement activity consists of collecting and analyzing requirements. Requirement collection includes product capabilities and constraints. Requirement analysis includes a feasibility study and documentation. Based on the collected user requirements, further analysis is needed to determine if the requirements are feasible. After requirements, the next activity in the analysis phase is specifications, which is transforming the user-oriented requirements into a precise form oriented to the needs of software engineers.

Design Phase. The design phase is concerned with how to build the system to behave as described. There are two parts of designs: system architecture design and detailed design. The system architecture design includes system structure and the system architecture document. System structure design is the process of partitioning a software system into smaller parts. Before subdividing the system, we need to do further specification analysis, examining the details of performance requirements, security requirements, assumptions and constraints, and the needs for hardware and software. Detailed design is about designing the program and algorithmic details. The activities within detailed design are program structure, program language and tools, validation and verification, test planning, and design documentation.

Coding Phase. Coding involves translating the design into the code of a programming language. It starts when the design document is base lined. Coding is composed of the following activities: identifying reusable modules, code editing, code inspection, and final test planning. The final test plan should be ready at the coding phase. Based on the test plan initiated at the design phase, with the feedback of coding activities, the final test plan should provide details of what needs to be tested, testing strategies and methods, testing schedules, and all necessary resources.

Testing Phase. Testing is the verification and validation activity for the software product. The goals of the testing phase are (1) to affirm the quality of the product by finding and eliminating faults in the program; (2) to demonstrate the presence of all specified functionality in the product; and (3) to estimate the operational reliability of the software. During the testing phase, program components are combined into the overall software code and testing is performed according to a developed test (software verification and validation) plan. During this phase, system integration of the software components and system acceptance tests are performed against the requirements.

Operating Phase. The final phase in the software life cycle is operation. The operating phase usually contains activities such as installation, training, support, and maintenance. After completion of the testing phase, the turnover of the software product is a very small part of the life cycle, but it is quite important. It involves transferring responsibility for maintaining the software from the developer to the user by installing the software product. Then the user is responsible for establishing a program to control and manage the software.

Software Verification and Validation

Verification and validation (V&V) are the two ways to check whether the design satisfies the user's requirements. According to the IEEE *Standard Glossary of Software Engineering Terminology,*

> Software verification is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.
> Software validation is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.

In short, Boehm (3) expressed the difference between the software verification and software validation as follows:

> Verification: "Are we building the product right?"
> Validation: "Are we building the right product?".

In other words, verification checks whether the product under construction meets the requirements definition. Validation checks whether the product's functions are what the customer wants.

Table 2. Data Recording for Time-Domain Approach

Failure Records	Actual Failure Time (minutes)	Time Between Failures
1	25	25
2	55	30
3	70	15
4	95	25

More often, programming is done primarily by scientists or engineers, who have little training in the aspects of software development or programming skills. These people, of course, are highly motivated to get a program running in the shortest time possible. The consequence of expedited results is that the users find the bugs in the software program after the software product is put into operation. Although it costs the developer very little to fix faults during the development phase, that is, testing phase, it would definitely cost orders of magnitude more to fix faults during the operating and maintenance phases. The cost of fixing an error, both in time and money, increases dramatically as the software life cycle progresses.

Verification should be integrated not only in the testing phase alone but in all phases of the software development life cycle. In fact, verification is most effective and efficient when applied from the beginning of the development process and should be performed independently by a group different from the development group.

In general, validation determines the correctness of the end product, for example, code, with respect to the software requirements, that is, Does the output conform with what was required? Verification is performed at each phase and between each phase of the development life cycle. It determines that each phase and subphase product is correct, complete, and consistent with itself and with its predecessor product.

Data Analysis

Traditionally, there are two commonly types of failure data, time-domain data and interval-domain data. These types of data are usually used by practitioners when analyzing and predicting reliability applications. Some software reliability models can handle both types of data. The time domain approach involves recording the individual times at which failure occurred. This approach is illustrated in Table 2. The first failure occurred 25 min into the test, the second 55 min into the test, the third at 70 min, and the fourth at 95 min. Some models may require obtaining the time between failures in lieu of the actual failure time. From this example, the values 25, 30, 15, and 25 should be used as the time-domain data set.

The interval-domain approach is characterized by counting the number of failures occurring during a fixed period (e.g., test session, hour, week, day). Using this method, the collected data are a count of the number of failures in the interval. This approach is illustrated in Table 3. Using the same failures as in the time-domain example, we would record two failures in the first 1 h interval, four failures in the second interval, one failure in the third interval, and one in the fourth. Intervals, however, do not need to be equally spaced for data collection. For example, if the interval for data collection is a test session, one session may last 4 h, and the next may be 8 h. Models with assumptions that handle this situation should be considered for higher fidelity forecasts for systems with interval-domain data.

Table 3. Data Recording for Interval-Domain Approach

Time (h)	Observed Number of Failures	Cumulative Number of Failures
1	2	2
2	4	6
3	1	7
4	1	8

The time-domain approach always provides better accuracy in the parameter estimates with current existing software reliability models but involves more data collection efforts than the interval-domain approach. The practitioners must trade off the cost of data collection with the accuracy reliability level required by the model predictions.

NHPP SOFTWARE RELIABILITY MODELS

Software reliability assessment is increasingly important in developing and testing new software products. Before newly developed software is released to the market, it is extensively tested for errors that may have been introduced during the development process. Detected errors are removed immediately. In the process of debugging, however, new errors may be introduced. Erroneous software released to the market incurs high failure costs. Debugging and testing, on the other hand, reduce the error content but increase the development costs.

Thus, one wishes to determine when to stop testing the software. During system testing, reliability measure is an important criterion in deciding when to release the software. Several other criteria, such as number of remaining errors, failure rate, or total system cost may also be used to determine optimal testing time (4).

In this section, we define a nonhomogeneous Poisson process (NHPP). Allowing both the error content function and the error detection rate to be time-dependent, a generalized software reliability model based on NHPP and an analytical expression for the mean value function are presented. Several existing NHPP models are also summarized. An NHPP is a realistic model for predicting software reliability and has a very interesting and useful interpretation in debugging and testing the software.

Notation

- $m(t)$ — expected number of error detected by time t (“mean value function”)
- $N(t)$ — random variable representing the cumulative number of software errors detected by time t
- $\lambda(t)$ — the intensity function
- $y(t)$ — actual values of $N(t)$ $[y_i := y(t_i)]$
- S_j — actual time at which the jth error is detected
- $R(s/t)$ — reliability during $(t, t + s]$ given that the last error occurred at time t

Nonhomogeneous Poisson Processes

The counting process $\{N(t), t \geq 0\}$ that represents the cumulative number of software errors detected by time t is an NHPP process. Basic assumptions about that counting process lead to the commonly accepted conclusion that, for any fixed $t \geq 0$, $N(t)$ is Poisson-distributed with a time-dependent Poisson parameter $m(t)$, the so-called mean value function. The main issue in the NHPP model is to determine an appropriate mean value function to denote the expected number of failures experienced up to a certain time. The NHPP model is based the following assumptions:

- The failure process has an independent increment, that is, the number of failures during the time interval $(t, t + s]$ depends on the current time t and the length of time interval s and does not depend on the past history of the process.
- The failure rate of the process is given by

$$\begin{aligned} P\{\text{exactly 1 failure in } (t, t+\Delta t)\} &= P[N(t+\Delta t) - N(t) = 1] \\ &= \lambda(t)\Delta t + o(\Delta t) \end{aligned}$$

  where $\lambda(t)$ is the intensity function.
- During a small interval Δt, the probability of more than one failure is negligible, that is,

$$P\{2 \text{ or more failures in } (t, t+\Delta t)\} = o(\Delta t)$$

- The initial condition is $N(0) = 0$.

On the basis of these assumptions, the probability that exactly n failures occurring during the time interval $(0, t)$ for the NHPP is given by

$$Pr\{N(t) = n\} = \frac{[m(t)]^n}{n!} e^{-m(t)}, \quad n = 0, 1, 2, \ldots \tag{1}$$

where

$$m(t) = E[N(t)] = \int_0^t \lambda(s)\, ds \tag{2}$$

It can be shown that the mean value function $m(t)$ is nondecreasing.

Reliability Function. The reliability $R(t)$, defined as the probability that there are no failures in the time interval $(0, t)$, is given by

$$\begin{aligned} R(t) &= P\{N(t) = 0\} \\ &= e^{-m(t)} \end{aligned}$$

In general, the reliability $R(x/t)$, the probability that there are no failures in the interval $(t, t + x)$, is given by

$$\begin{aligned} R(x|t) &= P\{N(t+x) - N(t) = 0\} \\ &= e^{-[m(t+x) - m(t)]} \end{aligned} \tag{3}$$

and its density is given by

$$f(x) = \lambda(t+x) e^{-[m(t+x) - m(t)]}$$

A Generalized NHPP Software Reliability Model

The mean value function represents the expected number of software errors that have accumulated up to time t. In mathematical functions, then $m(t) = E[N(t)]$. Therefore, with different assumptions, the model ends up with different functional forms of the mean value function. The mean value function must be defined analytically. This is usually done by expressing the mean value function as a function of two other functions, the error content $a(t)$ and the error detection rate $b(t)$. By making assumptions about the analytical behavior of these two functions, $a(t)$ and $b(t)$ are then defined as functions of time with one or more free parameters. Some of these parameters might be determined through mathematical or physical inferences. In most cases, however, these parameters have to be inferred statistically. The derivation of the generalized mean value function is presented next. Most of the existing NHPP models for the mean value function build upon the assumption that the error detection rate is proportional to the residual error content (5–9). Pham and Nordmann (10) recently formulated a generalized NHPP software reliability model and provide an analytical expression for the mean value function. The generalized form for the mean value function can be obtained by solving the following equations (10):

$$\frac{\partial m(t)}{\partial t} = b(t)[a(t) - m(t)] \tag{4}$$

with the initial condition

$$m(t_0) = m_0$$

where

$a(t)$ = total error content at time t
$b(t)$ = error detection rate per error at time t.

In the simplest model, the two functions $a(t)$ and $b(t)$ are constants. This model is known as the Goel–Okumoto NHPP model (7). A constant $a(t)$ stands for the assumption that no new errors are introduced during the debugging process (perfect debugging). A constant $b(t)$ implies that the proportional factor relating the error detection rate $\lambda(t)$ to the total number of remaining errors is constant. Many existing models describe perfect debugging, that is, $a(t) = a$, with a time-dependent error detection rate $b(t)$. Other studies deal with an imperfect debugging process and a constant error-detection rate $b(t) = b$.

In the generalized model, the functions $a(t)$ and $b(t)$ are both functions of time, and for practical purposes both of them are increasing with time. An increasing $a(t)$ shows that the total number of errors (including those already detected) increases with time because new errors are introduced during the debugging process. An increasing proportional factor $b(t)$ indicates that the error detection rate usually increases as debuggers establish more and more familiarity with the software.

The general solution for the mean value function $m(t)$ of Eq. (4) can be obtained using the techniques of differential equations and is given as follows (10):

$$m(t) = e^{-B(t)} \left[m_0 + \int_{t_0}^t a(s) b(s) e^{B(s)}\, ds \right] \tag{5}$$

where

$$B(t) = \int_{t_0}^{t} b(s)\, ds$$

and t_0 is the time to begin the debugging process.

In the following, we assume that $m(0) = 0$, which means that no errors are yet detected at time $t = 0$, the starting point of the debugging process. Unfortunately, the cumbersome integration in the previous equation cannot be eliminated by an algebraic exercise, unless particular function types are specified for $a(t)$ and $b(t)$. Simple functional relationships yield solutions that are not complex but less realistic. More elaborate functions yield more complex but more realistic results. Depending on how elaborate a model one wishes to obtain, imposing more or less restrictions on the functions $a(t)$ and $b(t)$ will yield more or less complex analytical solutions for the function $m(t)$. For given the two functions $a(t)$ and $b(t)$, the mean value function $m(t)$ can be easily obtained by using Eq. (5). Following is a summary of the NHPP models for the mean value functions appearing in the current literature (Table 4) (11).

PARAMETER ESTIMATION

Parameter estimation is of primary importance in software reliability prediction. Once the analytical solution for $m(t)$ is known for a given model, the parameters in this solution have to be determined. Parameter estimation is achieved by applying a technique of the maximum likelihood estimate (MLE), the most important and widely used estimation technique. Depending on the format in which test data are available, two different approaches are frequently used. A set of

Table 4. NHPP Software Reliability Models and Their Mean Value Functions (MVFs) (11)

Model Name	Model Type	$MVF[m(t)]$	Comments
Goel–Okumoto (G–O) (7)	Concave	$m(t) = a(1 - e^{-bt})$ $a(t) = a$ $b(t) = b$	Also called exponential model
Delayed S-shaped SRGM (5)	S-shaped	$m(t) = a[\mathrm{i} - (1 + bt)e^{-bt}]$	Modification of G–O model to make it S-shaped
Inflection S-shaped SRGM (5)	Concave	$m(t) = \dfrac{a(1 - e^{-bt})}{1 + \beta e^{-bt}}$ $a(t) = a$ $b(t) = \dfrac{b}{1 + \beta e^{-bt}}$	Solves a technical condition with the G–O model; becomes the same as G–O if $\beta = 0$
Gompertz	S-shaped	$m(t) = a(b^{c^{t}})$	Used by Fujitsu, Numazu Works
Pareto	Concave	$m(t) = a\left[1 - \left(1 + \dfrac{t}{\beta}\right)^{1-\alpha}\right]$	Assumes failures have different failure rates, and failure with highest rate is removed first
Weibull	Concave	$m(t) = a(1 - e^{-bt^{c}})$	Same as G–O when $c = 1$
Yamada exponential	Concave	$m(t) = a\{1 - e^{-r\alpha[1-e^{(-\beta t)}]}\}$ $a(t) = a$ $b(t) r\alpha\beta t e^{-\beta t}$	Attempt to account for testing effort
Yamada Rayleigh	S-shaped	$m(t) = a\{1 - e^{-r\alpha[1-e^{(-\beta t^{2}/2)}]}\}$ $a(t) = a$ $b(t) = r\alpha\beta t e^{-\beta t^{2}/2}$	Attempt to account for testing effort
Yamada imperfect debugging model 1	S-shaped	$m(t) = \dfrac{ab}{\alpha + b}(e^{\alpha t} - e^{-bt})$ $a(t) = a e^{\alpha t}$ $b(t) = b$	Assumes exponential fault content function and constant error detection rate
Yamada imperfect debugging model 2	S-shaped	$m(t) = a(1 - e^{-bt})\left(1 - \dfrac{\alpha}{b}\right) + \alpha a t$ $a(t) = a(1 + \alpha t)$ $b(t) = b$	Assumes constant introduction rate α and error detection rate
Pham–Nordmann (10)	S-shaped and concave	$m(t) = \dfrac{a(1 - e^{-bt})\left(1 - \dfrac{\alpha}{b}\right) + \alpha a t}{1 + \beta e^{-bt}}$ $a(t) = a(1 + \alpha t)$ $b(t) = \dfrac{b}{1 + \beta e^{-bt}}$	Assumes introduction rate is a linear function of testing time, and the error detection rate function is nondecreasing with an inflectional S-shaped model
Pham–Zhang (11)	S-shaped and concave	$m(t) = \dfrac{1}{(1 + \beta e^{-bt})}\left[(c + a)(1 - e^{-bt}) - \dfrac{ab}{b - \alpha}(e^{-\alpha t} - e^{-bt})\right]$ $a(t) = c + a(1 - e^{-1\alpha t})$ $b(t) = \dfrac{b}{1 + \beta e^{-bt}}$	Assumes introduction rate is an exponential function of the testing time, and the error detection rate is nondecreasing with an inflectional S-shaped model

failure data is usually collected in one of two common ways and is discussed next.

Type 1 Data: Interval-Domain Data. Assume that the data are given for the cumulative number of detected errors y_i in a given time-interval $(0, t_i)$ where $i = 1, 2, \ldots, n$ and $0 < t_1 < t_2 < \ldots < t_n$. Then the log likelihood function (LLF) takes on the following form:

$$\text{LLF} = \sum_{i=1}^{n} (y_i - y_{i-1}) \cdot \log[m(t_i) - m(t_{i-1})] - m(t_n)$$

Thus the maximum of the LLF is determined by the following system of equations:

$$0 = \sum_{i=1}^{n} \frac{\frac{\partial}{\partial\theta} m(t_i) - \frac{\partial}{\partial\theta} m(t_{i-1})}{m(t_i) - m(t_{i-1})} (y_i - y_{i-1}) - \frac{\partial}{\partial\theta} m(t_n)$$

where θ is one of the unknown parameters, is to be substituted.

Using the observed failure data (t_i, y_i) for $i = 1, 2, \ldots, n$, we can use the mean value function $m(t_i)$ to determine the expected number of errors to be detected by a future time t_i where $i = n + 1, n + 2$, etc.

Type 2 Data: Time-Domain Data. Assume that the data are given for the occurrence times of the failures or the times of successive failures, that is the realization of random variables S_j for $j = 1, 2, \ldots, n$. Given that the data provide n successive times of observed failures s_j for $0 \le s_1 \le s_2 \le \cdots \le s_n$, we can convert these data into the time between failures x_i where $x_i = s_i - s_{i-1}$ for $i = 1, 2, \ldots, n$. Given the recorded data on the time of failures, the log likelihood function takes on the following form:

$$\text{LLF} = \sum_{i=1}^{n} \log[\lambda(s_i)] - m(s_n)$$

The MLE of unknown parameters $\theta = (\theta_1, \theta_2, \ldots, \theta_n)$ can be obtained by solving the following equations:

$$0 = \sum_{i=1}^{n} \frac{\frac{\partial}{\partial\theta} \lambda(S_i)}{\lambda(S_i)} - \frac{\partial}{\partial\theta} m(S_n)$$

where

$$\lambda(t) = \frac{\partial}{\partial t} m(t)$$

and for θ every of the unknown parameters is to be substituted.

The equations to be solved for the MLE of the system parameters are nonlinear. To make use of the iterative Newton method, the author developed an Excel-macro, called FREE-ME, that computes the maximum likelihood estimates of free parameters for an arbitrary mean value function of a given set of test data.

ADVANCES IN SOFTWARE RELIABILITY MODELS WITH ENVIRONMENTAL FACTORS

The software reliability models which use testing time only as an influencing factor may not be appropriate for a software reliability assessment. It is necessary to develop a software reliability model which incorporates the environmental factors during the development of the software systems. Several researchers (12–14) have indicated that many environmental factors, such as programmer's skill, programming language, programming techniques, reuse of existing code, mental stress, and human nature, have some influence on error characteristics. They need to be incorporated into the software reliability model to predict an accurate reliability measurement. In this section, a generalized software reliability model that incorporates environmental factors is presented (15).

A Generalized Model With Environmental Factors

A newly developed software reliability model that considers environmental factors by combining the proportional hazard model (16) and an existing software reliability model is discussed in this section. Such factors are, for example, the complexity metrics of the software, the development and environmental conditions, the effect of mental stress and human nature, the level of the test-team members, and the facility level during testing. The proportional hazard model has been widely used in medical applications to estimate the survival times of patients.

Based on the proportional hazard model, let us consider the failure intensity function of a software system as the product of an unspecified baseline failure intensity $\lambda_0(t)$, a function that only depends on time, and an exponential function term incorporating the effects of a number of environmental factors. The basic assumption of this model is that the ratio of the failure intensity functions of any two errors observed at any time t associated with any environmental factor sets z_{1i} and z_{2i} is a constant with respect to time and they are proportional to each other. In other words, $\lambda(t_i; z_{1i})$ is directly proportional to $\lambda(t_i; z_{2i})$.

A generalized failure intensity function of the software reliability model that considers environmental factors can be written as

$$\lambda(t_i; z_i) = \lambda_0(t_i) e^{(\sum_{j=1}^{m} \beta_j z_{ji})} \tag{6}$$

where

z_{ji} is an environmental factor j of the ith error
β_j is the regression coefficient of the jth factor
t_i is the failure time between the $(i - 1)$th error and ith error, $i = 1, 2, \ldots, n$
z_i is the environmental factor of the ith error
m is the number of environmental factors

It is easy to see that $\lambda_0(t)$ is a baseline failure intensity function that represents the failure intensity when all environmental factor variables are set to zero.

Let Z be a column vector consisting of the environmental factors and B be a row vector consisting of the corresponding regression parameters. Then the above failure intensity model can be rewritten as

$$\lambda(t; Z) = \lambda_0(t)e^{(BZ)} \tag{7}$$

Therefore, the reliability of the software systems can be written, in a general form, as follows:

$$\begin{aligned} R(t; Z) &= e^{-\int_0^t \lambda_0(s)e^{BZ}\,ds} \\ &= \left[e^{-\int_0^t \lambda_0(s)\,ds}\right]^{e^{(BZ)}} \\ &= [R_0(t)]^{e^{BZ}} \end{aligned} \tag{8}$$

where $R_0(t)$ is the time-dependent software reliability.

The pdf of the software system is given by

$$\begin{aligned} f(t; Z) &= \lambda(t; Z)\cdot R(t; Z) \\ &= \lambda_0(t)e^{BZ}[R_0(t)]^{e^{BZ}} \end{aligned} \tag{9}$$

The regression coefficient B can be estimated, using either the MLE method or the maximum partial likelihood approach, which is discussed later, without assuming any specific distributions about the failure data and estimating the baseline failure intensity function.

Environmental Factors Estimation Using MLE

Assume that there are p unknown parameters in the baseline failure intensity function $\lambda_0(t)$, say, $\alpha_1, \alpha_2, \ldots, \alpha_p$ and there are m environmental factors $\beta_1, \beta_2, \ldots, \beta_m$. Let $A = (\alpha_1, \alpha_2, \ldots, \alpha_p)$ be a set of unknown parameters $\alpha_1, \alpha_2, \ldots, \alpha_p$ and B be a set of $\beta_1, \beta_2, \ldots, \beta_m$. Then the likelihood function is given by

$$\begin{aligned} L(A, B) &= \prod_{i=1}^{n} f(t_i; z_i) \\ &= \prod_{i=1}^{n} \left\{\lambda_0(t_i)e^{(\sum_{j=1}^m \beta_j z_{ji})}[R_0(t_i)]^{e^{(\sum_{j=1}^m \beta_j z_{ji})}}\right\} \end{aligned} \tag{10}$$

The log likelihood function is given by

$$\begin{aligned} \ln L(A, B) &= \sum_{i=1}^{n} \ln[\lambda_0(t_i)] + \sum_{i=1}^{n}\sum_{j=1}^{n} \beta_j z_{ji} \\ &\quad + \sum_{i=1}^{n} e^{(\sum_{j=1}^m \beta_j z_{ji})} \ln[R_0(t_i)] \end{aligned}$$

Taking the first partial derivatives of the log likelihood function with respect to $(m + p)$ parameters, we obtain

$$\frac{\partial}{\partial \alpha_k}[\ln L(A, B)] = \sum_{i=1}^{n} \frac{\frac{\partial}{\partial \alpha_k}[\lambda_0(t_i)]}{\lambda_0(t_i)} + \sum_{i=1}^{n} e^{(\sum_{j=1}^m \beta_j z_{ji})} \frac{\frac{\partial}{\partial \alpha_k}[R_0(t_i)]}{R_0(t_i)}$$

$$\frac{\partial}{\partial \beta_s} \ln L(A, B) = \sum_{i=1}^{n} z_{si} + \sum_{i=1}^{n} z_{si} e^{(\sum_{j=1}^m \beta_j z_{ji})} \ln[R_0(t_i)]$$

where $k = 1, 2, \ldots, p$ and $s = 1, 2, \ldots, m$. Setting the previous equations equal to zero, we can obtain all the $(m + p)$ parameters by solving the following system of $(m + p)$ equations simultaneously:

$$\sum_{i=1}^{n} \left\{ \frac{\frac{\partial}{\partial \alpha_k}[\lambda_0(t_i)]}{\lambda_0(t_i)} + e^{(\sum_{j=1}^m \beta_j z_{ji})} \frac{\frac{\partial}{\partial \alpha_k}[R_0(t_i)]}{R_0(t_i)} \right\} = 0 \quad \text{for } k = 1, 2, \ldots, p$$

$$\sum_{i=1}^{n} z_{si}\{1 + e^{(\sum_{j=1}^m \beta_j z_{ji})} \ln[R_0(t_i)]\} = 0 \quad \text{for } s = 1, 2, \ldots, m$$

Environmental Factor Estimation Using Maximum Partial Likelihood Approach

According to the idea of Cox's proportional hazard model, we can use the maximum partial likelihood method to estimate environmental factors without assuming any specific distributions about the failure data and estimating the baseline failure intensity function. The only basic assumption of this model is that the ratio of the failure intensity functions of any two errors observed at any time t associated with any environmental factor sets z_{1i} and z_{2i} is constant with respect to time and they are proportional to each other.

First we estimate the environmental factor parameters based on the partial likelihood function. The partial likelihood function of this model is given by

$$L(B) = \prod_{i=1}^{n} \frac{e^{(\beta_1 z_{1i} + \beta_2 z_{2i} + \cdots + \beta_m z_{mi})}}{\sum_{k \in R_i} e^{(\beta_1 z_{1k} + \beta_2 z_{2k} + \cdots + \beta_1 z_{1k})}} \tag{11}$$

where R_i is the risk set at t_i. Take the derivatives of the log partial likelihood function with respect to $\beta_1, \beta_2, \ldots, \beta_m$, and let them equal to zero. Therefore, we can obtain all of the estimated βs by solving these equations simultaneously using numerical methods. After estimating the factor parameters $\beta_1, \beta_2, \ldots, \beta_m$, the remaining task is to estimate the unknown parameters of the baseline failure intensity function $\lambda_0(t)$.

Enhanced Proportional Hazard Jelinski–Moranda Model

The Jelinski–Moranda (JM) (17) model is one of the earliest models developed for predicting software reliability. The failure intensity of the software at the ith failure interval of this model is given by

$$\lambda(t_i) = \phi[N - (i - 1)] \qquad i = 1, 2, \ldots, N$$

and the probability density function is given by

$$f(t_i) = \phi[N - (i - 1)]e^{-\phi[N-(i-1)]t_i}$$

From Eq. (6), the enhanced proportional hazard JM model (15), called the EPJM model, which is based on the proportional hazard and JM model, is expressed as

$$\lambda(t_i; z_i) = \phi[N - (i - 1)]e^{(\sum_{j=1}^m \beta_i z_{ji})}$$

and the pdf corresponding of $\lambda(t_i, z_i)$ is given by

$$f(t_i; z_i) = \phi[N - (i - 1)]e^{(\sum_{j=1}^m \beta_i z_{ji})}e^{\{-\phi[N-(i-1)]t_i e^{(\sum_{j=1}^m \beta_i z_{ji})}\}} \tag{12}$$

Next, we discuss how to estimate the $(m + 2)$ unknown parameters of the EPJM model using the MLE method and the maximum partial likelihood approach.

The Maximum Likelihood Method. From Eq. (12), the likelihood function of the model is given by

$$L(B,N,\phi) = \prod_{i=1}^{n} f(t_i; z_i) = \prod_{i=1}^{n}(\phi[N-(i-1)]e^{(\sum_{j=1}^{m}\beta_i z_{ji})} e^{\{-\phi[N-(i-1)]t_i e^{(\sum_{j=1}^{m}\beta_i z_{ji})}\}})$$

The log likelihood function is given by

$$\ln L(B,N,\phi) = n\ln\phi + \sum_{i=1}^{n}\ln[N-(i-1)] + \sum_{i=1}^{n}\left(\sum_{j=1}^{n}\beta_j z_{ji}\right) - \sum_{i=1}^{n}\phi[N-(i-1)]t_i e^{\sum_{j=1}^{m}(\beta_j z_{ji})}$$

Taking the first partial derivatives of the log likelihood function with respect to $(m + 2)$ parameters $\beta_1, \beta_2, \ldots, \beta_m, N$, and Φ, we obtain the following:

$$\frac{\partial \log L}{\partial\phi} = \frac{n}{\phi} - \sum_{i=1}^{n}[N-(i-1)]t_i e^{\sum_{j=1}^{m}(\beta_j z_{ji})}$$

$$\frac{\partial \log L}{\partial N} = \sum_{i=1}^{n}\frac{1}{[N-(i-1)]} - \phi\sum_{i=1}^{n} t_i e^{\sum_{j=1}^{m}(\beta_j z_{ji})}$$

and

$$\frac{\partial \ln L}{\partial \beta_j} = \sum_{i=1}^{n} z_{ji} - \sum_{i=1}^{n}\phi[N-(i-1)]t_i z_{ji} e^{\sum_{j=1}^{m}(\beta_j z_{ji})}$$

Setting all of these equations equal to zero, we can obtain the estimated $(m + 2)$ parameters by solving the following system equations simultaneously using a numerical method:

$$\sum_{i=1}^{n}[N-(i-1)]t_i e^{\sum_{j=1}^{m}(\beta_j z_{ji})} = \frac{n}{\phi}$$

$$\sum_{i=1}^{n}\frac{1}{[N-(i-1)]} = \phi\sum_{i=1}^{n} t_i e^{\sum_{j=1}^{m}(\beta_j z_{ji})}$$

$$\sum_{i=1}^{n}\phi[N-(i-1)]t_i z_{ji} e^{\sum_{j=1}^{m}(\beta_j z_{ji})} = \sum_{i=1}^{n} z_{ji} \quad for\ j-1,2,\ldots,m \tag{13}$$

The Maximum Partial Likelihood Method. Assume that the baseline failure intensity has the form of the JM model. That means that the basic assumption of this model is satisfied and that the ratio of the failure intensity functions of any two errors observed at any time t associated with any environmental factor sets z_{1i} and z_{2i} is a constant with respect to time and they are proportional to each other.

Having estimated the factor parameters $\beta_1, \beta_2, \ldots, \beta_m$, the remaining tasks are to estimate the unknown parameters of the baseline failure intensity function. Note that the failure intensity function model has the form

$$\lambda(t_i; z_i) = \phi[N-(i-1)]e^{(\beta_1 z_{1i}+\beta_2 z_{2i}+\cdots+\beta_m z_{mi})} = \phi[N-(i-1)]E_i$$

where

$$E_i = e^{(\beta_1 z_{1i}+\beta_{2i}+\cdots+\beta_m z_{mi})}$$

The pdf is given by

$$f(t_i; z_i) = \phi E_i[N-(i-1)]e^{-\{\phi E_i[N-(i-1)]t_i\}}$$

The likelihood function is given by

$$L(N,\phi) = \prod_{i=1}^{n}(\phi E_i[N-(i-1)]e^{-\{\phi E_i[N-(i-1)]t_i\}})$$

By taking the log of the likelihood function and its derivatives with respect to N and ϕ and setting them equal to zero, we obtain the following equations:

$$\frac{\partial \ln L}{\partial N} = \sum_{i=1}^{n}\frac{1}{N-(i-1)} - \sum_{i=1}^{n}\phi E_i t_i = 0$$

and

$$\frac{\partial \ln L}{\partial \phi} = \frac{n}{\phi} - \sum_{i=1}^{n}E_i[N-(i-1)]t_i = 0$$

The estimated N and ϕ can be obtained as follows. First, the parameter N can be obtained by solving the following equation:

$$\left\{\sum_{i=1}^{n}E_i[N-(i-1)]t_i\right\}\left\{\sum_{i=1}^{n}\frac{1}{[N-(i-1)]}\right\} = n\sum_{i=1}^{n}E_i t_i \tag{14}$$

After finding N, now the parameter ϕ can be easily obtained and is given by

$$\phi = \frac{\sum_{i=1}^{n}\frac{1}{[N-(i-1)]}}{\sum_{i=1}^{n}E_i t_i}$$

Applications

To illustrate the EPJM model, we use the existing software failure data reported by Musa (18), which is related to a real-time command and control system. To demonstrate the use of the EPJM model, we generate a failure-cluster factor and give its value, which is logically realistic based on the failure data and consultation with several local software firms by the author.

One of the assumptions of the JM model is that the time between failures is independent. However, in many real testing environments, the failure times indeed occur in a cluster, that is, the failure time within a cluster is relatively shorter than that between the clusters. The data shows that it is reasonable in that particular application. This may indicate that the assumption of independent failure time is not correct. We can enhance the JM model considering the failure-cluster factor by generating this factor based on the failure data.

We assume that if the present failure time compared with the previous failure time is relatively short, then some correlation may exist between them. Let us define a failure-cluster factor, such as

$$z_i = \begin{cases} 1 & \text{when } \dfrac{t_{i-1}}{t_i} \geq 7 \quad \text{or} \quad \dfrac{t_{i-2}}{t_i} \geq 5 \\ 0 & \text{otherwise} \end{cases}$$

for $i = 1, 2, . . .$ The data used in this model include both the failure time data and the explanatory environmental factor data and are given in Table 5. The explanatory variable data is dynamic, that is, it changes depending on the failure time. For example, in Table 5, the time between the fourth and fifth errors is 115 s, the time between the fifth and sixth errors is 9 s. Therefore, z_5 is assigned to 0 and z_6 is equal to 1.

Jelinski–Moranda Model. The Jelinski–Moranda model (14) is one of the earliest software reliability models. Many probabilistic software reliability models are variants or extensions of this basic model. The assumptions in this model include the following:

- The program contains N initial faults which is an unknown but fixed constant.
- Each fault in the program is independent and equally likely to cause a failure during test.
- Time intervals between occurrences of failure are independent of each other.
- Whenever a failure occurs, a corresponding fault is removed with certainty.
- The fault that causes a failure is assumed to be instantaneously removed, and no new faults are inserted during the removal of the detected fault.
- The software failure rate during a failure interval is constant and is proportional to the number of faults remaining in the program.

Based on the above assumptions, the program failure rate (without environmental factors) at the ith failure interval is given by

$$\lambda(t_i) = \phi[N - (i-1)], \qquad i = 1, 2, \ldots, N$$

where

ϕ = a proportional constant
N = the number of initial faults in the program
t_i = the time between the $(i-1)$th and the ith failures.

The software reliability function (without environmental factors) is given by

$$R(t_i) = e^{-\int_0^{t_i} \lambda(s)\,ds} = e^{-\phi[N-(i-1)]t_i}$$

Table 5. Software Failure Data with an Environmental Factor

Num	Time	z	Num	Time	z	Num	Time	z	Num	Time	z	Num	Time	z
2	3	0	31	36	1	61	0	1	91	724	0	121	75	1
2	30	0	32	4	1	62	232	0	92	2323	0	122	482	0
3	113	0	33	0	1	63	330	0	93	2930	0	123	5509	0
4	81	0	34	8	0	64	365	0	94	1461	0	124	100	1
5	115	0	35	227	0	65	1222	0	95	843	0	125	10	1
6	9	1	36	65	0	66	543	0	96	12	1	126	1071	0
7	2	1	37	176	0	67	10	1	97	261	0	127	371	0
8	91	0	38	58	0	68	16	1	98	1800	0	128	790	0
9	112	0	39	457	0	69	529	0	99	865	0	129	6150	0
10	15	1	40	300	0	70	379	0	100	1435	0	130	3321	0
11	138	0	41	97	0	71	44	1	101	30	1	131	1045	1
12	50	0	42	263	0	72	129	0	102	143	1	132	648	1
13	77	0	43	452	0	73	810	0	103	108	0	133	5485	0
14	24	0	44	255	0	74	290	0	104	0	1	134	1160	0
15	108	0	45	197	0	75	300	0	105	3110	0	135	1864	0
16	88	0	46	193	0	76	529	0	106	1247	0	136	4116	0
17	670	0	47	6	1	77	281	0	107	943	0			
18	120	0	48	79	0	78	160	0	108	700	0			
19	26	1	49	816	0	79	828	0	109	875	0			
20	114	0	50	1351	0	80	1011	0	110	245	0			
21	325	0	51	148	1	81	445	0	111	729	0			
22	55	0	52	21	1	82	296	0	112	1897	0			
23	242	0	53	233	0	83	1755	0	113	447	0			
24	68	0	54	134	0	84	1064	0	114	386	0			
25	422	0	55	357	0	85	1783	0	115	446	0			
26	180	0	56	193	0	86	860	0	116	122	0			
27	10	1	57	236	0	87	983	0	117	990	0			
28	1146	0	58	31	1	88	707	0	118	948	0			
29	600	0	59	369	0	89	33	1	119	1082	0			
30	15	1	60	748	0	90	868	0	120	22	1			

The property of this model is that the failure rate is constant, and the software stage is unchanged during the testing.

Based on the data given in Table 5, the estimates of the two parameters N and ϕ using MLE are as follows:

$$\hat{N} = 142$$
$$\hat{\phi} = 3.48893 \times 10^{-5}$$

Therefore, the current reliability of the software system is given by

$$R(t_{137}) = e^{-\hat{\phi}[\hat{N}-(137-1)]t_{137}}$$

Now we want to predict the future failure behavior using only data collected in the past after 136 errors have been found. For example, the reliability of the software for the next 100 s after 136 errors are detected is given by

$$\begin{aligned} R(t_{137} = 100) &= e^{-\hat{\phi}[\hat{N}-(137-1)]t_{137}} \\ &= e^{-(0.0000348893)[142-136](100)} \\ &= 0.979284 \end{aligned}$$

Similarly, the reliability of the software for the next 1000 s is given by

$$\begin{aligned} R(t_{137} = 1000) &= e^{-(0.0000348893)[142-136](1000)} \\ &= 0.811123 \end{aligned}$$

Assume that we use the maximum partial likelihood approach to estimate the environmental factor parameter for the EPJM model. Because there is a factor in this application, we can easily obtain the estimated parameter, using the statistical software package SAS:

$$\beta_1 = 1.767109$$

with a significance level of 0.0001. Then the estimates of N and ϕ are given as follows:

$$\hat{N} = 141$$
$$\hat{\phi} = 3.28246 \times 10^{-5}$$

Therefore,

$$E_i = e^{\beta_1 z_{1i}} = \begin{cases} 5.853905235 & \text{for } z = 1 \\ 1 & \text{for } z = 0 \end{cases}$$

The current reliability of the software system is given by

$$\begin{aligned} R(t_{137}) &= e^{-\hat{\phi}E_{137}[\hat{N}-(137-1)]t_{137}} \\ &= e^{-9.6076048.10^{-4}t_{137}} \quad \text{for } z = 1 \\ &\quad e^{-1.64123.10^{-4}t_{137}} \quad \text{for } z = 0 \end{aligned}$$

Assuming that

$$P(Z = 1) = \frac{28}{136} = 0.20588$$
$$P(Z = 0) = \frac{108}{136} = 0.79412$$

then the reliablity of the software for the next 100 s is given by

$$\begin{aligned} R(t_{137} = 100) \\ &= 0.908394931 \quad \text{for } z = 1 \; with\ probability = 0.20588 \\ &\quad 0.983721648 \quad \text{for } z = 0 \; with\ probability = 0.79412 \end{aligned}$$

and therefore,

$$R(t_{137} = 100) = 0.95375$$

Similarly, the reliability of the software for the next 1000 s is given by

$$\begin{aligned} R(t_{137} = 1000) \\ &= 0.382601814 \quad \text{for } z = 1 \; with\ probability = 0.20588 \\ &\quad 0.848637633 \quad \text{for } z = 0 \; with\ probability = 0.79412 \end{aligned}$$

or

$$R(t_{137} = 1000) = 0.74021$$

It is worthwhile to note that one may want to consider the environmental factor variables Z_{ji} as a function of time. In this case, a mathematical generalized form of the failure intensity function is given by

$$\lambda(t_i; z_i) = \lambda_0(t_i)e^{[\sum_{j=1}^{m} \beta_j z_{ji}(t_i)]}$$

The techniques discussed in the previous section can be used to estimate the parameters of environmental factors and the baseline failure intensity function.

FURTHER READING

There are many survey papers on software reliability that can be read at an introductory stage. Interested readers are referred to the review papers by Ramamoorthy and Bastani (1982), Goel (1985), and Cai (1998).

Software Reliability by H. Pham, Springer-Verlag, 1999, *Handbook of Software Reliability Engineering* by M. Lyu (ed.), McGraw-Hill and IEEE CS Press, 1996, the book *Software-Reliability-Engineered Testing Practice* by J. Musa, McGraw-Hill, 1997, and *Software Assessment: Reliability, Safety, Testability* by Friedman and Voas (Wiley, New York, 1995), are recently new and good textbooks for students, researchers, and practitioners.

In addition, the edited books, *Software Reliability Models: Theoretical Developments, Evaluation, and Application,* by Malaiya and Srimani, IEEE Computer Society Press, 1991 and *Software Reliability and Testing* by H. Pham, IEEE Computer Society Press, 1995 recently reprinted many classic and quality papers on the subject.

This list is by no means exhaustive, but it will help readers get started learning about the subject.

BIBLIOGRAPHY

1. H. Pham, *Fault-Tolerant Software Systems: Techniques and Applications,* Los Alamitos, CA: IEEE Computer Society Press, 1992.

2. H. Pham, *Software Reliability and Testing,* Los Alamitos, CA: IEEE Computer Society Press, 1995.
3. B. W. Boehm, *Software Engineering Economics.* Englewood Cliffs, NJ: Prentice-Hall, 1981.
4. H. Pham and X. Zhang, A software cost model with warranty and risk costs, *IEEE Trans. Comput.,* **48**: 1999 (in press).
5. S. Yamada, M. Ohba, and S. Osaki, S-shaped software reliability growth models and their applications, *IEEE Trans. Reliab.,* **R-33**: 289–292, 1984.
6. S. Yamada and S. Osaki, Optimal software release policies for a nonhomogeneous software error detection rate model, *Microelectron. Reliab.,* **26**: 691–702, 1986.
7. A. L. Goel and K. Okumoto, Time-dependent error-detection rate model for software and other performance measures, *IEEE Trans. Reliab.,* **R-28**: 206–211, 1979.
8. H. Pham, A software cost model with imperfect debugging, random life cycle and penalty cost, *Int. J. Syst. Sci.,* **27**: 455–463, 1996.
9. M. Ohba and S. Yamada, S-shaped software reliability growth models, *Proc. 4th Int. Conf. Reliability Maintainability,* Perros Guirec, France, 1984.
10. H. Pham and L. Nordmann, A generalized NHPP software reliability model, *Proc. 3rd Int. Conf. Reliability and Qual. in Design,* Anaheim, CA, 1997.
11. H. Pham and X. Zhang, An NHPP software reliability model and its comparison, *Int. J. Reliab., Quality Safety Eng.,* **4** (3): 1997.
12. T. Furuyama, Y. Arai, and K. Iio, Fault generation model and mental stress effect analysis, *Proc. 2nd Int. Conf. Achieving Quality in Software,* Venice, Italy, 1993.
13. W. W. Everett and M. Tortorella, Stretching the paradigm for software reliability assurance, *Software Qual. J.,* **3**: 1–26, 1994.
14. H. Pham and X. Zhang, A study of environmental factors in software development, prepared for the U.S. D.O.T. Federal Aviation Administration, Atlantic City Int. Airport, NJ, October, 1998.
15. H. Pham, A generalized software reliability model with environmental factors, IE Working Paper, Rutgers Univ., 1998.
16. D. R. Cox, Partial likelihood, *Biometrika,* **62**: 1975.
17. Z. Jelinski and P. B. Moranda, Software reliability research, in W. Freiberger (ed.), *Statistical Computer Performance Evaluation,* New York: Academic Press, 1972.
18. J. D. Musa, A theory of software reliability and its applications, *IEEE Trans. Softw. Eng.,* **SE-1**: 312–327, 1975.

HOANG PHAM
Rutgers University