A RELIABILITY MODEL FOR A LARGE-SCALE SOFTWARE SYSTEM

by

Fedon Kadifeli

B.S. in CMPE., Boğaziçi University, 1987

Submitted to the Institute for Graduate Studies in Science and Engineering in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

Boğaziçi University

1989

A RELIABILITY MODEL FOR

A LARGE-SCALE SOFTWARE SYSTEM

APPROVED BY

Doç. Dr. Oğuz Tosun .....

(Thesis Supervisor)

Doç. Dr. Selahattin Kuru .....

Doç. Dr. Süleyman Özekici .....

DATE OF APPROVAL 12/09/1989

ACKNOWLEDGEMENTS

I would like to thank Dr. Ufuk Çağlayan who proposed the thesis topic and advised me in all stages of this work. Thanks are also due to Dr. Oğuz Tosun, Neşe Akkaya, and Ayşe Bayazıtoğlu who supplied some of the references used in this thesis. I thank also Dr. Oğuz Tosun, Dr. Selahattin Kuru, and Dr. Süleyman Özekici for their careful reviews.

Fedon Kadifeli

ABSTRACT

In this thesis a model that can be used for conventional software reliability analysis and prediction is described. The proposed model has been applied on a large-scale distributed commercial software system which is already operational. The model is characterized by the gamma distribution having three unknown parameters. The parameters are estimated from failure time data by using the method of maximum likelihood estimation. The model is shown to fit the data better than other well-known models in the absence of additional information on the software system being analyzed.

ÖZET

Bu tezde geleneksel yazılım güvenilirliği çözümlemesi ve önkestiriminde kullanılabilecek bir model anlatılmaktadır. Önerilen model işletimsel durumda olan büyük ölçekli dağıtımlı ticari bir yazılım sistemi üzerinde uygulanmıştır. Model üç bilinmeyen parametresi olan gama dağılımı ile tanımlanmaktadır. Bu parametreler başarısızlık zaman verilerinden en çok olabilirlik kestirimi yöntemi kullanılarak kestirilmektedir. Modelin, çözümlenmekte olan yazılım sistemi üzerine daha fazla bilgi yokluğunda, diğer bilinen modellerden verilere daha iyi uyduğu gösterilmektedir.

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

LIST OF SYMBOLS

In this thesis the notation used by Musa *et al*. [1]* is followed as far as possible.

B	fault reduction factor
$E[X]$	expected value of random variable X
f	failure probability density function or dF/dt; linear execution frequency of a program
F	cumulative failure probability function
I	expected or Fisher information
i, j	indices indicating sequential number of a failure event
k	number of failures per subinterval; index for model parameter
K	fault exposure ratio
l	index for a failure subinterval
L	likelihood function
m	number of failures experienced
m_e	number of failures experienced by time t_e
m_l	number of failures experienced for lth subinterval

---

* References enclosed in brackets refer to the bibliography.

m_t	number of failures experienced by time t
$M(t)$	number of failures experienced by time t (random variable)
$N(t)$	number of faults removed by time t (random variable)
p	total number of failure subintervals; probability of imperfect debugging
$P[E_1]$	probability of event E_1
$P[E_1 \mid E_2]$	probability of event E_1 conditioned on event E_2
r	inflection rate for inflection S-shaped growth model
r_l	observed failure intensity for the lth subinterval
R	reliability or probability of failure-free operation
$R(t_i' \mid t_{i-1})$	reliability following $(i-1)$th failure or during time interval $(t_{i-1}, t_{i-1}+t_i']$
S	sum of squared errors
t	(realization of) cumulative calendar or generic time
$\boldsymbol{t}$	set of times $(t_1, \ldots, t_e)$
t_e	a specific deterministic time denoting the end of failure data
t_i	time of ith failure
t_l	time at midpoint of lth subinterval
t_s	a specific deterministic time denoting the start of failure data
t'	(realization of) time since last failure
t_i'	time interval following $(i-1)$th failure
T	cumulative time (random variable)
T'	time since last failure (random variable)
T_i	time of ith failure (random variable)
u_0	inherent faults for binomial-type model (integer)

U	statistic used to test increasing or decreasing failure intensity
$U(t)$	number of faults remaining at time t (random variable)
w	index of last element in $\boldsymbol{\beta}$ or one less than the number of model parameters
X	a random variable
z	(realization of) hazard rate
z_0	fault detection rate during the first interval or initial hazard rate
$z(t_i'\|t_{i-1})$	hazard rate for the ith failure interval, given that time t_{i-1} has elapsed at the $(i-1)$th failure
α	confidence level; significance of a model or probability of rejecting a correct model
$\boldsymbol{\beta}$	set of model parameters ($\beta_0, \beta_1, \ldots, \beta_w$)
β_k	model parameter (used in different models with no implication of any relationship between their parameters)
γ	third parameter of gamma class model, in this study γ=1, 2, …
Δt	small time interval
$\eta(t)$	expected number of faults removed by time t or $E[N(t)]$
θ	failure intensity decay parameter of logarithmic Poisson execution time model
Θ	expected life or mean time to failure (MTTF)
$\kappa_{1-\alpha}$	$(1-\alpha)$ percentile of a standard normal distribution
λ	parameter of the exponential distribution used in hardware reliability or the constant hazard rate
λ_0	initial failure intensity
$\lambda(t)$	failure intensity function or $d\mu/dt$

$\mu(t)$	mean value function or expected number of failures experienced by time t or $E[M(t)]$
ν_0	expected number of failures experienced in infinite time
$\prod$	product
$\sum$	summation
τ	(realization of) cumulative execution time
ϕ	model parameter representing constant per-fault hazard rate or at zero time
ψ	inflection parameter for the inflection S-shaped growth model
ω_0	number of inherent faults
$\omega(t)$	expected number of faults remaining at time t or $E[U(t)]$
∞	a large time value
(t_1, t_2)	open time interval between t_1 and t_2
$[t_1, t_2]$	closed time interval between t_1 and t_2

I. INTRODUCTION

As computer applications became more diverse and spread through almost every area of everyday life, reliability became a very important characteristic of computer systems. From the first time computers were used until presently, people have been interested in reliable systems.

Since it is a matter of economy to produce a system having reliability above a specific level, it is necessary to measure and control its reliability or unreliability. To do this, a number of models have been and are being developed. New models try to make better predictions and to alleviate the problems resulting from unreasonable assumptions made by earlier ones.

Reliability modeling of a computer system may be considered in two aspects: hardware and software. Although the nature of hardware and software faults resulting in failures is quite different, both may be thought of as occurring randomly. For this reason, variations of hardware reliability models were initially used for software reliability prediction. Currently, models are also being developed specially for software.

In this thesis the reliability of a large operational commercial software system is modeled. In particular, the future failure behavior of this software system is predicted by studying and modeling its past failure behavior assuming that the same behavior will continue. This can be done if the values of model parameters do not change during the period of prediction. Some other metrics, such as the number of faults that were initially present in the software or eventually to be detected, the reliability for a specific time period, the failure intensity, etc., are easily determined from the results obtained. In this study a number of software reliability models are selected. The parameters of the functions described by these models are separately estimated so as the function of each model fits well to the actual data. Then, chosing the model that has obtained the best fit, in other words the one that is able to explain the current and past failure behavior most adequately, the study is pursued further.

In the remaining of the thesis following this introduction, there are six more parts. In the next three a background is provided for the remaining three.

Part II discusses software quality and reliability and gives an overview of hardware and software reliability related concepts.

Part III gives an overview of the most important software reliability models.

Part IV discusses parameter estimation and in particular the maximum likelihood and least squares methods used for fitting a function to actual data.

Part V contains a rather abstract overview of the system being studied. It discusses failure data collection procedures used for this system and explains the data used to conduct this study.

Part VI discusses a few models that can be used to explain the past failure behavior of this software and predict its future behavior. The most appropriate model is chosen for further study.

Part VII gives some concluding remarks on the results obtained and discusses some problems in software reliability modeling.

Appendix A contains two procedures that can be used to estimate parameters for a model. Also a program is given which may be applied directly on failure data collected from a software project.

Appendix B gives an index of some important concepts used in this thesis.

Bibliography gives a list of references used in this study and cited in the text of the thesis. References not cited, but used in this study, are listed separately.

II. SOFTWARE QUALITY AND RELIABILITY

The most important software product characteristics are *level of quality*, *time of delivery* (or, generally, schedule), and *cost*. These attributes are *user-oriented* rather than *developer-oriented*. Also, time of delivery and cost are quantitative, whereas quality obviously cannot be defined quantitatively [1].

As Sommerville [2, p. 296] quotes from Boehm *et al.* [3], some software quality criteria are:

Economy	Correctness	Resilience
Integrity	*Reliability*	Usability
Documentation	Modifiability	Clarity
Understandability	Validity	Maintainability
Flexibility	Generality	Portability
Inter-operability	Testability	Efficiency
Modularity	Re-usability	

Although it has been argued that most of them are difficult to quantify, some attempts to do this have been made. The reader is referred to Mohanty [4] for software quality assessment and to Boehm [5,6] and Mohanty [7] for software cost estimation.

Reliability is one, and probably the most important, aspect of software quality. But before discussing software reliability in particular it is a good idea to talk about reliability in general.

The reliability of any system depends on the correctness of the system design, the correctness of the mapping of the system design to implementation, and the reliability of components making up the system [2]. Since computer systems consist of two parts, hardware and software, it is better to study their reliability separately.

Until the late sixties the major concern in computer systems was on their hardware related performance. The main reasons were that hardware was more expensive than software and less reliable than today's hardware, and software systems were not so

complex. After the early seventies, software costs started to increase while hardware became cheaper and more reliable. For this reason much interest developed in software reliability estimation.

Software systems are quite different than other systems, e.g., hardware. Since they do not contain any moving parts, there is no such thing as wearout in software and there is not any random factor in the output produced. If a routine gives a wrong answer, there is no point in trying to execute it again with the same inputs.

For this reason the same models used for measuring hardware reliability cannot be used for software. Goel [8] notes that hardware exhibits mixtures of decreasing and increasing failure rates. The former is caused from fixing design related hardware failures, while the later is primarily due to hardware component wearout or aging. On the other hand, software exhibits only a decreasing failure rate (assuming that "debugging" a software system does not cause its bug content to increase and there are no changes in the user and computing environment so that the software becomes obsolete). It should always be remembered that software should be considered with its environment; different software reliability measure values for the same software in different environments may be assessed. This is the reason why program testing is considered sometimes more advantageous than program proving for determining the reliability of a program. Program testing gives information about a program's actual behavior in its intended environment, while program proving is limited to conclusions about the program's behavior in a postulated environment which is assumed to be correct.

The purpose for which the measured software reliability can be used is important. It may be used for planning and controlling resources during development, so that a high quality software may be developed. It also gives the user a confidence about the correctness of software. It should be observed that as more faults are uncovered, it becomes more difficult to expose additional faults. So after the reliability, or quality, of software increases above a certain point the software developer may stop the development process.

Software reliability measures can be used in at least four areas: (*a*) system engineering; (*b*) project management during development and particularly test; (*c*) operational phase software management; and (*d*) evaluation of software engineering technologies [1].

2.1. Hardware Reliability

Although software reliability is quite different than hardware reliability, it is necessary to review some hardware reliability evaluation concepts before going into software reliability. The *reliability* of a system is defined as the probability that it will adequately perform its intended function—without failure—for a specified interval of time under stated environmental conditions, which may be defined as the user requirements [1]. Roughly it can be said that, reliability is inversely proportional to the rate at which failures occur [9].

In this section a number of terms related to reliability, such as reliability function, expected life, hazard rate, and failure rate, will be defined and one important reliability function will be mentioned. These concepts apply to software and hardware reliability in general.

If T is a random variable representing the failure time of a system, then the probability that the system will fail by time t, i.e., the *failure probability*, is

$$F(t) = P[T \leq t] = \int_0^t f(x)\, dx. \tag{2.1}$$

Here, $f(t)$ represents the *probability* (or *failure*) *density function* and $F(t)$ the *cumulative distribution function*.

The *reliability function*, i.e., the probability that the system survives until time t, is defined as

$$R(t) = P[T > t] = 1 - F(t) = \int_t^\infty f(x)\, dx. \tag{2.2}$$

In other words, $R(t)$ represents the probability that the system will not have failed by time t assuming it is fault-free at time 0.

The *expected life*, or *mean time to failure (MTTF)*—also denoted by Θ—is simply the mean or the expected value of the failure density function:

$$E[T] = \int_0^\infty t\, f(t)\, dt. \tag{2.3}$$

It can be shown also that

$$E[T] = \int_0^\infty R(t)\, dt. \tag{2.4}$$

Mean time to repair (*MTTR*) is the time during which repair or replacement is occurring. *Mean time between failures* (*MTBF*) is the sum of MTTF and MTTR.

The *failure rate* is the probability that a failure per unit time occurs in an interval such as $[t_1, t_2]$, knowing that a failure has not occurred before t_1:

$$\frac{P[t_1 \le T < t_2 \mid T > t_1]}{t_2 - t_1} = \frac{P[t_1 \le T < t_2]}{(t_2 - t_1)\, P[T > t_1]} = \frac{F(t_2) - F(t_1)}{(t_2 - t_1)\, R(t_1)}. \tag{2.5}$$

The *hazard rate*, on the other hand, is defined as the limit of the failure rate as the interval approaches to zero:

$$z(t) = \lim_{\Delta t \to 0} \frac{F(t + \Delta t) - F(t)}{\Delta t\ R(t)} = \frac{f(t)}{R(t)}. \tag{2.6}$$

So, there is a difference between the hazard rate and failure rate; the hazard rate is an instantaneous rate of failure at time t for a system of age t. The hazard rate changes over the life cycle of a physical system, typically it decreases, remains constant, and then increases with time giving a "bathtub curve."

As an example of a well-known reliability function the *exponential distribution* can be given:

$$\begin{aligned} f(t) &= \lambda \exp(-\lambda t), \quad \lambda > 0, \\ F(t) &= 1 - \exp(-\lambda t), \\ R(t) &= \exp(-\lambda t), \\ z(t) &= \lambda, \qquad \text{constant hazard rate,} \\ E[T] &= 1 / \lambda. \end{aligned} \tag{2.7}$$

Among other important distributions the Weibull and gamma distributions can be mentioned.

2.2. Software Reliability

Software reliability represents a user- (or customer-) oriented view of software quality. It relates directly to operation rather than design of the program, and hence it is dynamic rather that static. For this reason software reliability is interested in failures occurring and not faults in a program. Reliability measures are much more useful than fault measures. Software reliability may be expected to vary during the software development period.

It becomes apparent that the distinction between the terms “failure” and “fault” is an important one.

Failure means a function of the software that does not meet user requirements [1]. It is an external behavior of the system deviating from that required by its specifications. In other words, failure is something dynamic, i.e., occurring at execution time. It is not a bug or fault. It is more general. For example, excessive response time may be considered as a failure if it does not meet the specifications.

On the other hand, a *fault*, or bug, is a defect in a program, that when executed under particular conditions will result in a failure. A fault can be a source of more than one failure. By definition, there cannot be multiple faults causing a single failure.

A fault may result from an *error* made by the programmer. Errors occur because of (*a*) incomplete communication between the people involved in a project or between different times for the same person; (*b*) defective knowledge of the application area, the design methodology, and the programming language; (*c*) incomplete analysis of the possible conditions that can occur at a given point in the program; and (*d*) transcription errors.

The probability, or relative frequency of times, that a given program will work as intended by the user, i.e., without failures, in a specified environment and for a specified duration can be termed as *software reliability*. The aim of a software engineer is to increase this probability and make it one if possible. To do this he or she must measure the reliability of the software. A commonly used approach for measuring software reliability is by using an analytical model whose parameters are generally estimated from available data on software failures. Part III of this thesis discusses such models.

Reliability quantities have usually been defined with respect to time, although it is possible to define them with respect to other variables. Time may be considered in three

different ways [1]:

(A) Execution time (τ), i.e., CPU time;
(B) Calendar time (t); and
(C) Clock time, i.e., the sum of times passed from program start to program end, without counting shut-down periods.

Execution time is considered superior [10].

There are 4 general ways of characterizing failure occurrences in time:

(A) Time of failure
(B) Time interval between failures (incremental) } time-based

(C) Cumulative failures experienced up to a given time
(D) Failures experienced in a time interval } failure-based

All these four quantities are in fact *random variables*, because, (*a*) the locations of faults within a program are unknown; and (*b*) the conditions of program execution are generally unpredictable. Of course, this does not mean that they are completely unpredictable.

A *random process* is a set of random variables, each corresponding to a point in time. In reliability study there are two characteristics of a random process: (*a*) the probability distribution of the random variables, e.g., Poisson; and (*b*) the variation of the process with time. A random process whose probability distribution varies with time is called *nonhomogeneous*.

Two functions can be defined for the time variation of a random process: (*a*) the *mean value function*, μ, as the average cumulative failures associated with each time point; and (*b*) the *failure intensity function*, λ, as the rate of change of mean value function or the number of failures per unit time. Note that the failure intensity function is the derivative of the mean value function.

When there are no changes in the software, i.e., no debugging and software corrections take place, then λ is constant and a *homogeneous random process* takes place. On the contrary, when software corrections occur a nonhomogeneous process as described above takes place. Figure 2.1 illustrates the mean value and related failure intensity functions of such a process. These graphs are very typical in the sense that the mean number of failures experienced increases with time in such a manner that the failure intensity decreases.

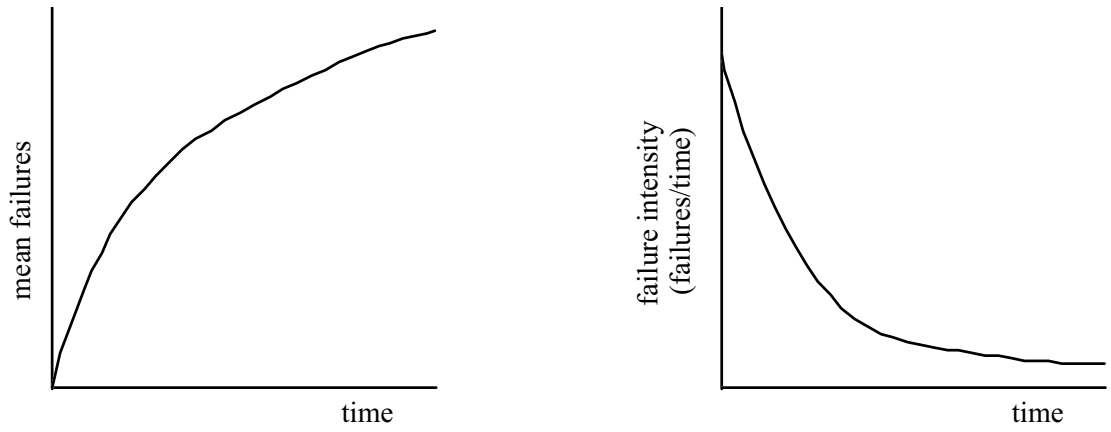

FIGURE 2.1 Typical plots of mean value and failure intensity functions with respect to time

This behavior is affected by two factors: (*a*) the number of faults in the software; and (*b*) the execution environment.

Let $M(t)$ be a random process representing the number of failures experienced by time t. Then the mean value function is defined as

$$\mu(t) = E[M(t)], \tag{2.8}$$

i.e., the expected number of failures at time t. The failure intensity function of the $M(t)$ process is the instantaneous rate of change of the expected number of failures with respect to time, or

$$\lambda(t) = \frac{d\mu(t)}{dt}. \tag{2.9}$$

The "time" used here may be any one of the above-mentioned three times, but execution time is generally preferred in order to be compatible with hardware reliability.

Principal factors affecting software reliability are fault introduction, fault removal, and environment [1]. *Fault introduction* depends on the characteristics of the code developed, i.e., created or modified—such as its size—and of the development process—such as software engineering technology and tools used and level of experience of programmers. *Fault removal* depends on time, operational profile, and the quality of repair activity. *Environment* is determined by the *operational profile*, which is the set of run types that a program can execute along with the probabilities with which they will occur. It is generally established by enumerating the possible input states and their probabilities of occurrence or by specifying the sequence of program modules executed.

As faults are removed, as in test phase, failure intensity tends to decrease and reliability to increase. When faults are introduced during operation or test, as in cases when new features or design changes are being introduced into the system or when faults predominate repairs during debugging, there tends to be a step increase in failure intensity and a step decrease in reliability. If a system is stable, as in a program that has been released and there are no changes in code, both failure intensity and reliability tend to be constant.

The term mean time to failure (MTTF), which means the average value of next failure interval, is not used so extensively in software reliability as in hardware reliability, since in many cases it is undefined. Instead, *failure intensity*, which is roughly the inverse of MTTF, is preferred.

Software availability is the expected fraction of time during which a software component or system is functioning acceptably. For a constant failure intensity, it is the ratio of up time to the sum of up time and down time, as the interval over which the measurement is made approaches infinity. The down time is the product of the failure intensity and the mean time to repair (MTTR). MTTR for software is defined as the average time required to restore the database for a program, reload the program, and resume execution. It does not include the time spent during fault determination and correction, since it is generally not practical to hold up operation of software during this period.

Maintainability for software relates to the speed and ease with which a program can be corrected. Since repairs to software are not generally performed in a way that idles the program, maintainability does not relate to down time, as it does for hardware.

III. SOFTWARE RELIABILITY MODELING

Littlewood [11] discusses the difficulties that may be encountered in software reliability modeling. In particular, early software reliability models are criticized because they are based on assumptions made for hardware related models. It is suggested that metrics such as MTTF or MTBF must be used with care. This is because they may not exist. One may always obtain a finite average of some data, but this may not always estimate a population mean. For example, if at some point in time one becomes sure that a program is perfect—although it is difficult to be in such a position—then MTTF does not exist—it is infinite—for this program. Better measures such as failure rates are more appropriate.

One should not be concerned too much with the *number* of bugs, i.e., faults, in a program, but with their *effect* on its operation. In other words, the operational reliability of a program is more important than the quality of its state. Littlewood also points out that a relationship between the state of program and its performance is likely to be very complicated and unknown. For this reason, models should be based upon operational reliability. He also says that structural models appropriate to software should be used and a Bayesian approach should be preferred, because of the uncertainties present in the inputs selected from the input space of the program and in the program itself which came into existence during program development and debugging. The first uncertainty can be described by a Poisson process with failure rate λ. The second uncertainty makes this parameter a random variable.

The approach defended by Littlewood (Littlewood-Verrall model [12, 13]) is to improve *faith* in a program, or equivalently decrease its hazard rate, while no failures occur. But once a failure is encountered faith drops immediately, only to raise back below or (hopefully) above its previous level after the program is debugged. In this respect this model is considered superior to models such as of Jelinski and Moranda [14] where the hazard rate changes only at each fault correction, but it is constant between corrections.

Finally, Littlewood advises us not to stop at a reliability analysis, but pursue the model further to evaluate the life-time utility of programs.

In this part some important models are reviewed. A classification of these models made by Musa *et al.* [1] is also given.

Most software reliability models are formulated in terms of random processes. Models consider the probability of failure times or failures experienced and the nature of the variation of the random process with time. So, most models are time-based. Specification of a model generally includes specification of a function of time, such as the mean value or failure intensity function.

The values of the parameters for the specific form of a model are established through either:

(A) *Prediction*: Determination from physical properties of the software product and the development process. This can be done before any execution of the program.
(B) *Estimation*: Statistical inference procedures are applied to failure data taken for the program. This can be done after the program has executed long enough.

Models are not exact and do not totally represent reality. Also in the determination of the parameters of a model some uncertainties are always introduced. For these reasons parameter values should be expressed together with *confidence interval*s, representing the range of values within which a parameter is expected to lie with a certain confidence.

A good software reliability model (*a*) gives good predictions of future failure behavior; (*b*) is capable of estimating and computing useful quantities; (*c*) is simple; (*d*) is widely applicable across different software products; and (*e*) is based on sound assumptions. It can be used (*a*) to evaluate software engineering technology quantitatively; (*b*) to evaluate the development status during the test phases of a project; (*c*) to monitor the operational performance of software and to control new features added and design changes; and (*d*) to enrich insight into the software product and the software development process [1].

Most models are based on a stable program executing in a constant environment. In other words, the code of the program and the operational profile is not changing. Only fault removal may take place. Some models also allow small amount of fault introduction too. This is generally the case for a program that has been released and is operational.

To predict the future failure behavior, the values of model parameters should not change for the period of prediction. If this is not the case, one should either wait until enough failures have occurred—so that model parameters can be re-estimated—or

compensate for the change. Incorporating changes into a model is generally impractical because of the added complexity.

3.1. Reliability Models

Goel and Bastani [15] define two main categories of reliability estimation models: *software reliability growth models* and *statistical models*. The models in the first class estimate the reliability using the error history of the program, whereas the others use the response (success/failure) of a program to a random sample of test cases without making any corrections on the errors discovered.

The models, based mainly on the failure history of software, can be classified according to the nature of the failure process studied as follows [8]:

(A) *Times Between Failures Models*. Generally, it is assumed that the time between (*i*-1)st and *i*th failures follows a distribution with parameters depending on the number of faults remaining in the program during this interval and it is expected that these intervals will get longer as faults are removed. Of course, this may not be true for each pair of successive failure times, because failure times are random variables and observed values are subject to statistical fluctuations. There are a number of models in this class ranging from the simplest (Jelinski and Moranda de-eutrophication model [14]) to more complex (Littlewood-Verrall Bayesian model [12, 13]) ones.

(B) *Failure Count Models*. For this class of models, instead of using the times between failures, the interest is in the number of failures in specified time intervals and it is expected that the observed number of failures per unit time—again a random variable—will decrease as faults are removed.

(C) *Fault Seeding* and *Input Domain Based Models*. These are time-independent models. In fault seeding models a number of faults is “seeded” in the program. The program is then tested, failures resulting from the seeded and indigenous faults are recorded. Using these data the total number of indigenous faults in the program is estimated. In input domain based models the input domain of a program is partitioned into a set of equivalence classes—each one usually associated with a program path—and the program

is tested for each of the above input test cases. The reliability is determined from the number of failures observed during execution of the sampled test cases.

Naturally each model in each class makes different assumptions and can be used only in specific phases of software development. For example, in *design phase* none of them may be used, in *unit testing* time-dependent models do not seem to be applicable, in *integration testing* most of the existing software reliability models are applicable, in *acceptance testing* failure count and input domain based models are generally applicable, and in *operational phase* failure count models are likely to be most applicable.

3.2. Classification of Software Reliability Models

From the late sixties various models have been developed relating reliability to time, failures experienced, or other variables (for example see Yu *et al*. [16]). The most important improvements in this area were the following [1]:

(A) using execution time to simplify models;
(B) distinguishing between fault and failure;
(C) estimation methods for model parameters;
(D) development of comparison criteria to compare different models;
(E) classification of models;
(F) collecting better data;
(G) adapting models for particular circumstances of various applications; and
(H) using failure intensity instead of mean time to failure.

The models developed so far can be classified in terms of their attributes as follows [1]:

(A) *Time domain*: Calendar or execution time.
(B) *Category*: Finite or infinite number of failures experienced in infinite time. For finite failures category models there are a number of *classes* according to the functional form of the failure intensity in terms of time. For infinite failures category there are a number of *families* according to the functional form of the failure intensity in terms of the expected number of failures expected.
(C) *Type*: Distribution of the number of failures experienced by time *t*.

TABLE 3.1 Classification of some software reliability models

Finite failure category			
	Poisson type	**Binomial type**	**Other types**
Exponential class	Musa ’75 [17] Moranda ’75 Schneidewind ’75 Goel-Okumoto ’79 [18]	Jelinski-Moranda ’72 [14] Shooman ’72	Goel-Okumoto ’78 Musa ’79 Keiller-Littlewood ’83 [19]
Weibull class		Schick-Wolverton ’73 Wagoner ’73	
Class 1		Schick-Wolverton ’78	
Pareto class		Littlewood ’81	
Gamma class	Yamada-Ohba-Osaki ’83 [20]		

Infinite failure category				
	Type 1	**Type 2**	**Type 3**	**Poisson type**
Geometric family	Moranda ’75			Musa-Okumoto ’84 [21]
Inverse linear family		Littlewood-Verrall ’73 [12]		
Inverse polyn. (2nd degree) **family**			Littlewood-Verrall ’73 [12]	
Power family				Crow ’74

Table 3.1 tabulates some well-known models. Note that time domain is not considered in the table because the same models may be applied to either time domain. However, models applied to execution time domain are more effective than calendar time models, because execution time, or CPU time, is more closely linked to software’s chance of failure. For example, a software-controlled telephone-switching system may have 10hr execution time per day, while a specialized mathematical subroutine in software library might run only a few minutes each day. Both have calendar time of 24hr, but their execution times are quite different, and their failure rates will depend on their execution times [22].

Some important models, that are shown in the table, are discussed very briefly in the following subsections [8, 1]. Note that most models are times between failures and failure count models according to the classification made by Goel [8].

3.2.1. Jelinski-Moranda Model (1972)

This model [14] assumes u_0 independent software faults at the start of testing which are equally likely to cause a failure during testing. A detected fault is removed with certainty immediately without introducing new faults. In this case the hazard rate is piecewise constant and proportional to number of faults remaining:

$$z(t_i'|t_{i-1}) = \phi[u_0-(i-1)], \tag{3.1}$$

where t_i' is the time between the $(i-1)$st and ith failures, ϕ the proportionality constant between the number of remaining faults and hazard rate (in other words the hazard rate per fault), and u_0 the total number of faults. The failure intensity can be shown to be exponentially decaying in terms of time. Maximum likelihood estimation is used to determine the parameters ϕ and u_0. Figure 3.1 gives an example plot of the hazard rate versus cumulative time.

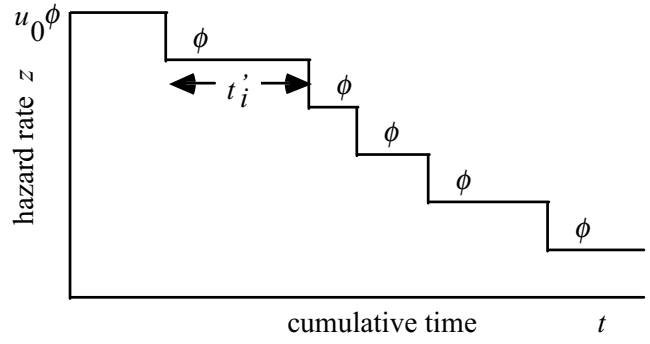

FIGURE 3.1 A typical plot of program hazard rate for the Jelinski-Moranda model

3.2.2. Shooman Model (1972)

This model is essentially similar to the Jelinski-Moranda model.

3.2.3. Schick-Wolverton Model (1973)

It makes the same assumptions as the Jelinski-Moranda model except that the hazard rate is assumed to be proportional to the number of faults remaining as well as the time elapsed since last failure:

$$z(t_i'|t_{i-1}) = \phi[u_0-(i-1)]\ t_i'. \tag{3.2}$$

Figure 3.2 gives an example plot.

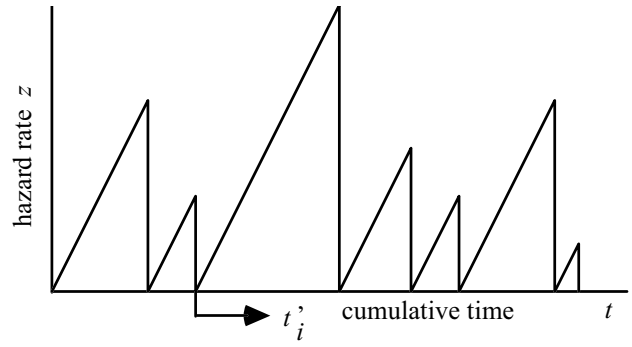

FIGURE 3.2 A typical plot of program hazard rate for the Schick-Wolverton (1973) model

3.2.4. Schick-Wolverton Model (1978)

Later the Schick-Wolverton (1973) model was modified assuming a hazard rate which is parabolic instead of a linear function in time:

$$z(t_i'|t_{i-1}) = \phi\ [u_0-(i-1)]\ (-\beta_1 t_i'^2 + \beta_2 t_i' + \beta_3). \tag{3.3}$$

This indicates that the likelihood of a failure occurring increases rapidly as the test time accumulates within a testing interval.

3.2.5. Schneidewind Model (1975)

In this model fault detections per time interval are viewed as a nonhomogeneous Poisson process with an exponentially decaying intensity function:

$$\lambda(i) = \beta_0 \exp(-\beta_1 i), \quad \beta_0,\beta_1>0, \quad i=1, 2, \ldots. \tag{3.4}$$

3.2.6. Moranda Model (1975)

There are two variations of the Jelinski-Moranda model to describe testing situations where faults are not removed until the occurrence of a fatal one at which time the accumulated group of faults is removed:

(A) *Geometric de-eutrophication process*. In this model the hazard rate decreases in steps that form a geometric progression:

$$z(t_i') = z_0\, k^{l-1}. \tag{3.5}$$

The parameter z_0 is the fault detection rate during the first interval, l the testing interval, and k a constant ($0<k<1$).

(B) *Geometric Poisson*. In this model too hazard rate decreases in geometric progression, but at fixed intervals rather than at each failure correction.

3.2.7. Musa Model (1975)

In this model, also called "*(basic) execution time model*," Musa [17] uses execution time instead of calendar time. Fault correction rate is generally proportional to the fault detection or hazard rate:

$$z(\tau) = f\,K\,[\omega_0 - N(\tau)], \tag{3.6}$$

where τ is the execution time, f the linear execution frequency, K the fault exposure ratio—a proportionality constant which relates fault exposure frequency to the linear execution frequency—ω_0 the total number of faults, and $N(\tau)$ the number of faults corrected during $(0,\tau)$.

This model has a calendar component to relate execution time to calendar time based on the fact that available resources limit the amount of execution time for each calendar day. For example, the pace of execution during testing is affected usually by the number of debuggers, number of test team members, and available computer time, in that sequence.

The failure intensity is given by

$$\lambda(\mu) = \lambda_0 \left(1 - \frac{\mu}{\nu_0} \right), \tag{3.7}$$

where λ_0 is the initial failure intensity, ν_0 the total number of failures that can occur in infinite time, and μ the expected number of failures experienced at a given point in time which is obtained from

$$\mu(\tau) = \nu_0 \left[1 - \exp\left(-\frac{\lambda_0}{\nu_0} \tau \right) \right]. \tag{3.8}$$

Combining (3.7) and (3.8)

$$\lambda(\tau) = \lambda_0 \exp\left(-\frac{\lambda_0}{\nu_0} \tau \right) \tag{3.9}$$

is obtained.

3.2.8. Littlewood-Verrall Model (1973)

In this model [12, 13] a Bayesian approach is used. Software reliability is considered as a measure of strength of belief that a program will operate successfully. The hazard rate is treated as a random variable, varying discontinuously at each failure detection and correction and continuously with the cumulative execution time.

3.2.9. Keiller-Littlewood Model (1983)

This model [19] is similar to the Littlewood-Verrall model, but uses a different parameter of the distribution of the hazard rate to express reliability change.

3.2.10. Littlewood Model (1981)

This is another variant of the Littlewood-Verrall model.

3.2.11. Goel-Okumoto Model (1978)

This model, which is a modification of the Jelinski-Moranda model, considers imperfect debugging. The hazard rate during the interval between the (i-1)st and the ith failures is given by

$$z(t_i'|t_{i-1}) = \phi\,[u_0 - p(i-1)], \tag{3.10}$$

where ϕ is the failure rate per fault and p the probability of imperfect debugging.

3.2.12. Goel-Okumoto Model (1979)

Considering failure detection as a nonhomogeneous Poisson process with an exponentially decaying rate function, the expected number of failures observed by time t is given by [18]

$$\mu(t) = \beta_0\,[1 - \exp(-\beta_1 t)], \tag{3.11}$$

and the failure rate by

$$\lambda(t) = \mu'(t) = \beta_0\,\beta_1\,\exp(-\beta_1 t), \tag{3.12}$$

where β_0 is the expected number of failures to be observed eventually and β_1 the failure detection rate per failure. Figure 3.3 gives typical plots of the mean value and failure intensity functions with respect to time. Note that, when $\beta_0,\beta_1>0$ the basic execution time model of Musa is obtained.

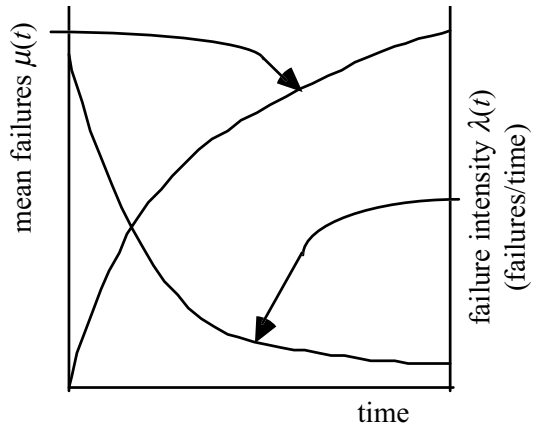

FIGURE 3.3 Typical plots of mean value and failure intensity functions for the Goel-Okumoto (1979) model

In practice, it has been observed that the failure rate first increases and then decreases. In order to model this situation Goel proposed a generalization of the above model with an additional parameter β_2. That is,

$$\mu(t) = \frac{\beta_0}{\beta_1 \beta_2}[1 - \exp(-\beta_1 t^{\beta_2})], \tag{3.13}$$

and

$$\lambda(t) = \mu'(t) = \beta_0 t^{\beta_2 - 1} \exp(-\beta_1 t^{\beta_2}). \tag{3.14}$$

3.2.13. Yamada-Ohba-Osaki Models (1983)

This is a modification of the nonhomogeneous Poisson process to obtain an S-shaped curve for the cumulative number of failures detected [20, 23, 24, 25]. It can be thought as a generalized exponential model where failure rate initially increases and later (exponentially) decays. Thus the mean value function is S-shaped. The software error detection process described by such a growth curve can be regarded as a learning process in which test-team members become familiar with test environment, so their test skills gradually improve. There are two models describing mean value functions that can give such a curve. Generally these models can be used in place of an exponential model to avoid a pessimistic assessment.

(A) *Delayed S-shaped Model* [20]. This is a simple modification of the nonhomogeneous Poisson process described in the previous subsection. A difference between this model and the exponential models is that this model is designed for analysis of the fault isolation data while the exponential software reliability growth models are designed to describe the failure detection process. The mean value function—number of faults isolated—is given by

$$\mu(t) = \omega_0 [1 - (1+\phi t) \exp(-\phi t)], \ \ \omega_0, \phi > 0 \tag{3.15}$$

where ω_0 is the number of inherent faults and ϕ the constant per-fault hazard rate (at zero time). The failure intensity function—fault isolation rate—is given by

$$\lambda(t) = \omega_0 \ \phi^2 t \exp(-\phi t) \tag{3.16}$$

which means $\lambda(0)=\lambda(\infty)=0$ and $\lambda(1/\phi)=\lambda_{max}$. In other words, the time to failure of an individual fault follows a gamma distribution having as shape parameter two. The expected number of faults remaining in the system is

$$\omega(t) = E[N(\infty)-N(t)] = \omega_0(1+\phi t)\exp(-\phi t) \tag{3.17}$$

and the reliability, i.e., the probability of no failures in $(t_{i-1},\ t_{i-1}+t_i']$ given that most recent failure occurred at time t_{i-1}, is given by

$$R(t_i'|t_{i-1}) = \exp[\mu(t_{i-1})-\mu(t_{i-1}+t_i')], \quad i=1, 2, \ldots \tag{3.18}$$

as usual.

(B) *Inflection S-shaped Model* [25]. In this model the underlying concept is that the observed software reliability growth becomes S-shaped if faults in a program are mutually dependent, i.e., some faults are not detectable before some other faults are removed. This is characterized by the following inflection S-shaped growth mean value function

$$\mu(t) = \omega_0 \frac{1-\exp(-\phi t)}{1+\psi\exp(-\phi t)}, \quad \omega_0,\phi,\psi>0 \tag{3.19}$$

where ω_0 is the number of faults to be eventually detected, ϕ the failure detection rate, and ψ the inflection factor defined as

$$\psi(r) = \frac{1-r}{r}, \quad 0<r\leq 1 \tag{3.20}$$

where r, the inflection rate, represents the ratio of the number of detectable faults to the total number of faults. When $r=1$, that is all faults are detectable initially, then the exponential model is obtained since $\psi=0$. Only if $r<0.5$ (or $\psi>1$) the reliability growth curve has an inflection.

The failure intensity function for this model is

$$\lambda(t) = \frac{\omega_0\ \phi\ (1+\psi)\exp(-\phi t)}{[1+\psi\exp(-\phi t)]^2} \tag{3.21}$$

and its maximum, which is also the inflection point for μ, is obtained at

$$t = \frac{\ln \psi}{\phi} \tag{3.22}$$

if $\psi > 1$.

3.2.14. Crow Model (1974)

This model, developed mainly for complex, repairable hardware systems during development testing, is based on a nonhomogeneous Poisson process with failure intensity and mean value functions being power functions of time. It is suggested that it can be applied to software reliability with certain ranges of parameter values.

3.2.15. Musa-Okumoto Model (1984)

This rather simple model [21, 26] is a logarithmic Poisson execution time model based on a nonhomogeneous Poisson process with an intensity function decreasing exponentially with expected failures experienced

$$\lambda(\mu) = \lambda_0 \exp(-\theta\mu), \tag{3.23}$$

where λ_0 represents the initial failure intensity and θ the rate of reduction in the normalized failure intensity per failure. This model incorporates the claim that the repair of early failures reduces the failure intensity more than later ones. The expected number of failures is a logarithmic function of (execution) time:

$$\mu(\tau) = \frac{1}{\theta} \ln(\lambda_0 \theta \tau + 1). \tag{3.24}$$

Combining (3.23) and (3.24)

$$\lambda(\tau) = \frac{\lambda_0}{\lambda_0 \theta \tau + 1} \tag{3.25}$$

is obtained for the relation between the failure intensity and time.

3.3. Model Development

Developing a practical and useful model involves substantial theoretical work, tool building, and practical experience, which means several person years. On the other hand, the application of a well-established model requires a small fraction of project resources. In this thesis the second approach is used, because of the time limitation and the insufficient acquaintance with the system whose reliability is being modeled.

The various steps of the model fitting and decision making process are as follows [8]:

Step 1—Study software failure data in order to gain an insight into the nature of the process being modeled. For most models, such data should be in the form of either times between failures (time-based) or as failure counts (failure-based). To determine the appropriate variables for the model it may be a good idea to plot these as a function of, say, calendar time.

Step 2—Choose an appropriate reliability model based on an understanding of the process studied.

Step 3—Obtain estimates of model parameters using, for example, the method of maximum likelihood, and based on the available data.

Step 4—Obtain the fitted model by substituting the estimated values of the parameters in the chosen model.

Step 5—Conduct some suitable goodness-of-fit test to check the model fit. If the model fits, then proceed with next step, otherwise collect additional data or seek a better, more appropriate model.

Step 6—Compute various quantitative measures, like undetected (remaining) errors, time to next failure, software reliability, etc., (together with confidence bounds), to assess the performance of the software system.

Step 7—Use the developed model to make some decisions about the software system.

In Part VI of this thesis a model is developed roughly following these steps.

IV. PARAMETER ESTIMATION

After choosing a model, there remains the problem of determining its parameters to obtain a specific form of the model. There are two methods: Parameter prediction and parameter estimation.

Parameter prediction tries to establish the parameters of a model form the properties of the software product and the development process. It has the advantage that can be applied before the software is tested, that is before failure data are available. Its disadvantages are that not every model parameter has an interpretation that allows easy parameter prediction and that the accuracy of parameter prediction is limited. An example model where parameter prediction can be used is the basic execution time model which has the following mean value function:

$$\mu(\tau) = \nu_0 \left(1 - \exp\left(-\frac{\lambda_0}{\nu_0}\,\tau\right)\right). \tag{4.1}$$

Here, ν_0, the total number of failures that would be experienced in infinite time, and λ_0, the initial failure intensity, can be predicted using the relations:

$$\nu_0 = \frac{\omega_0}{B} \tag{4.2}$$

and

$$\lambda_0 = f K\, \omega_0. \tag{4.3}$$

B is the fault reduction factor, ω_0 the number of inherent faults, f the linear execution frequency of the program, and K the fault exposure ratio of the program [1]. This subject is not pursued further here, because parameter prediction is not used in this thesis.

Parameter estimation is used in subsystem or system test or operational phase where failure data are available. It is a statistical method trying to estimate model parameters based on failure times or number of failures per time interval. *Point estimation* is used to determine the parameters and *interval estimation* to compute

confidence limits for the parameters that are useful in evaluating the accuracy of the estimates.

In this part, two widely used parameter estimation methods, namely least squares and maximum likelihood estimation, will be covered.

4.1. Least Squares Estimation

For small or medium size samples least squares estimators are considered to be better [1]. Two approaches may be considered:

(A) Estimate the model parameters by fitting the functional relationship of the failure intensity (λ) with respect to mean value function (μ), to the observed failure intensity (r).

(B) Estimate the model parameters by fitting the functional relationship of the failure intensity with respect to time, to the observed failure intensity.

Assume that we have (cumulative) failure times t_i, i=1, …, m_e, in a time interval $(0,t_e]$. This observation interval is partitioned at every kth failure occurrence time so that there are p ($\geq m_e/k$) disjoint subintervals. The observed failure intensity, r_l, for the lth subinterval $(t_{k\times(l-1)}, t_{k\times l}]$ is given by [26, 1]

$$r_l = \begin{cases} \dfrac{k}{t_{k\times l} - t_{k\times(l-1)}}, & l=1, \ldots, p-1 \\ \dfrac{m_e - k(p-1)}{t_e - t_{k\times(l-1)}}, & l=p. \end{cases} \tag{4.4}$$

The estimate of the mean value function for the lth subinterval is given by

$$m_l = k\,(l-1), \quad l=1, \ldots, p. \tag{4.5}$$

Note that the midpoint is not used [26]. The time is calculated as

$$t_l = \begin{cases} \frac{t_{k\times l} + t_{k\times(l-1)}}{2}, & l=1, \ldots, p-1 \\ \\ \frac{t_e + t_{k\times(l-1)}}{2}, & l=p. \end{cases} \tag{4.6}$$

k is selected as 5 as a reasonable compromise between grouping a small number of failures—that will result in large variations in the estimated failure intensity—and grouping a large number of failures—that will result in too much smoothing.

The two approaches try to estimate model parameters by minimizing

$$S(\boldsymbol{\beta}) = \sum_{l=1}^{p} [\ln r_l - \ln \lambda(m_l;\boldsymbol{\beta})]^2 \tag{4.7}$$

or

$$S(\boldsymbol{\beta}) = \sum_{l=1}^{p} [\ln r_l - \ln \lambda(t_l;\boldsymbol{\beta})]^2 \tag{4.8}$$

respectively. Note that the difference of the *logarithms* of the observed failure intensity to the estimated failure intensity is taken. This is equivalent to the minimization of the sum of the squares of the *relative errors*, which is generally preferred since it gives the same weight for any level of failure intensity.

It has been observed that using the functional relationship $\lambda(t,\boldsymbol{\beta})$, i.e., the second approach, does not give better results than the first approach. In particular, the parameters tend to be overestimated.

Examples:

(A) The exponential class models (Subsections 3.2.7 and 3.2.12) have the following functions:

$$\mu(t) = \beta_0 [1 - \exp(-\beta_1 t)] \tag{4.9}$$

$$\lambda(\mu) = \beta_1 (\beta_0 - \mu) \tag{4.10}$$

$$\lambda(t) = \beta_0 \beta_1 \exp(-\beta_1 t). \tag{4.11}$$

From (4.10) and (4.7)

$$S(\beta_0, \beta_1) = \sum_{l=1}^{p} [\ln r_l - \ln \beta_1 - \ln(\beta_0 - m_l)]^2 \tag{4.12}$$

is obtained. Minimizing this expression, results in the least squares estimates $\hat{\beta}_0$ and $\hat{\beta}_1$ for the unknown parameters. This minimization can only be made by numerical methods such as the one described in Appendix A.

Similarly from (4.11) and (4.8)

$$S(\beta_0, \beta_1) = \sum_{l=1}^{p} [\ln r_l - \ln(\beta_0 \beta_1) + \beta_1 t_l]^2 \tag{4.13}$$

is obtained.

(B) The geometric family models (Subsection 3.2.15) have the following functions:

$$\mu(t) = \beta_0 \ln(1+\beta_1 t) \tag{4.14}$$

$$\lambda(\mu) = \beta_0 \beta_1 \exp\left(-\frac{\mu}{\beta_0}\right) \tag{4.15}$$

$$\lambda(t) = \frac{\beta_0 \beta_1}{1 + \beta_1 t} \tag{4.16}$$

From (4.15) and (4.7)

$$S(\beta_0, \beta_1) = \sum_{l=1}^{p} \left[\ln r_l - \ln(\beta_0 \beta_1) + \frac{m_l}{\beta_0}\right]^2 \tag{4.17}$$

is obtained which can be minimized using linear regression [26].

Similarly from (4.16) and (4.8)

$$S(\beta_0, \beta_1) = \sum_{l=1}^{p} [\ln r_l - \ln(\beta_0 \beta_1) + \ln(1+\beta_1 t_l)]^2 \tag{4.18}$$

is obtained.

Interval Estimation: Assuming that the differences of the logarithms of the observed and estimated failure intensity, i.e., ($\ln r_l - \ln \lambda_l$) are independent random variables with zero mean and normally distributed with a common variance, confidence intervals for model parameters can be calculated. These specify the range of values the model parameters may assume given a specific measure of confidence.

4.2. Maximum Likelihood Estimation

Assume that the mean value function $\mu(t)$ includes w+1 model parameters β_k (k=0, …, w). Suppose that the data set on m_e failure occurrence times is observed. The likelihood function for the w+1 unknown parameters in the nonhomogeneous Poisson process model with $\mu(t)$ given $\boldsymbol{t} = (t_1, \ldots, t_e)$ is given by [24, 1]

$$L(\boldsymbol{\beta}) = \exp[-\mu(t_e)] \prod_{i=1}^{m_e} \lambda(t_i). \tag{4.19}$$

The logarithm of the likelihood function yields

$$\ln L(\boldsymbol{\beta}) = \sum_{i=1}^{m_e} \ln \lambda(t_i) - \mu(t_e). \tag{4.20}$$

Then the maximum likelihood estimates $\hat{\beta}_k$ (k=0,…,w) can be obtained by solving the likelihood equations:

$$\frac{\partial \ln L(\boldsymbol{\beta})}{\partial \beta_k} = 0, \quad k=0, \ldots, w. \tag{4.21}$$

In other words,

$$\sum_{i=1}^{m_e} \frac{1}{\lambda(t_i)} \frac{\partial \lambda(t_i)}{\partial \beta_k} - \frac{\partial \mu(t_e)}{\partial \beta_k} = 0, \quad k=0, \ldots, w. \tag{4.22}$$

From Equation (4.22) it can be deduced that, for most Poisson type models, the first parameter is given by

$$\hat{\beta}_0 = \frac{m_e \beta_0}{\mu(t_e;\beta_0,\hat{\beta}_1,\ldots,\hat{\beta}_w)}. \tag{4.23}$$

The other parameters must be determined by finding the roots of the Equation (4.22). If the model has just two parameters, only $\hat{\beta}_1$ need to be determined and this can be done using numerical methods. Such a procedure is explained in Appendix A.

Examples:

(A) For exponential class models using (4.9) and (4.23)

$$\hat{\beta}_0 = \frac{m_e}{1 - \exp(-\hat{\beta}_1 t_e)} \tag{4.24}$$

is obtained. Using (4.22)

$$\frac{m_e}{\hat{\beta}_1} - \frac{m_e t_e}{\exp(\hat{\beta}_1 t_e) - 1} - \sum_{i=1}^{m_e} t_i = 0 \tag{4.25}$$

is obtained. Since in the last equation only $\hat{\beta}_1$ is unknown, using any numerical root finding procedure its value can be determined. Then the value of $\hat{\beta}_0$ is obtained by substituting $\hat{\beta}_1$ in (4.24).

(B) Similarly for geometric family models

$$\hat{\beta}_0 = \frac{m_e}{\ln(1+\hat{\beta}_1 t_e)} \tag{4.26}$$

and

$$\frac{1}{\hat{\beta}_1} \sum_{i=1}^{m_e} \frac{t_i}{1+\hat{\beta}_1 t_i} - \frac{m_e t_e}{(1+\hat{\beta}_1 t_e)\ln(1+\hat{\beta}_1 t_e)} = 0 \tag{4.27}$$

are obtained.

Interval Estimation: When there is only one unknown parameter, β_k, then the maximum likelihood estimator of β_k is asymptotically normally distributed with mean β_k and variance $1/I(\beta_k)$, where

$$I(\beta_k) = E\left[-\frac{\partial^2 \ln L(\beta_k)}{\partial \beta_k{}^2}\right]. \tag{4.28}$$

For example, for exponential class models

$$I(\beta_1) = m_e\left\{\frac{1}{\beta_1{}^2} - \frac{t_e{}^2 \exp(\beta_1 t_e)}{[\exp(\beta_1 t_e) - 1]^2}\right\} \tag{4.29}$$

and a $100(1-\alpha)$ per cent confidence interval for β_1 is given by

$$\hat{\beta}_1 \pm \frac{\kappa_{1-\alpha/2}}{\sqrt{I(\hat{\beta}_1)}} \tag{4.30}$$

where $\kappa_{1-\alpha/2}$ is the appropriate normal deviate. Using the *substitution principle* [1], each limit for $\hat{\beta}_1$ is substituted into Equation (4.24) to obtain a confidence interval for β_0.

V. DESCRIPTION OF COLLECTED DATA

The success of applying and using a software reliability model depends highly on the quality and accuracy of failure data collection which in turn depends on careful planning and organization. To do this, data takers should be motivated; the collection mechanism should be as easy as possible; the data should be collected and scrubbed in real time (missing data should be pursued); and feedback of results obtained should be provided on a regular and timely basis [1]. Of course, these cannot be enforced to the current study since data that were already collected are being used in this thesis.

In this part of the thesis an overview of the software system being modeled is given. Throughout this thesis the pseudonym “System S” is used to refer to this system. The software has been released and it is operational. From time to time modifications are made to it.

5.1. Overview of System S

System S is a distributed system used for telecommunications management in many countries. Control is distributed over numerous modules and is provided through inexpensive microprocessors and memories. Naturally, this system has hardware and software components. For example, it uses processors of the Intel iAPX 8086/8088 family, so the software consists of 8086 Assembly language statements. In this thesis only the software aspect of the system is studied.

System S software is a large and complex system maintained by multiple groups throughout the world. Like many other systems, it is modular. Two levels of modularization can be considered. At the top level there are about 15 important main function modules. Each one consists of several subfunction modules, which in turn are made up of separately assembled and together linked and mapped source programs.

When an unexpected function of the software is detected, a *problem* (or *fault*) *report* is prepared. In response to a problem report, one or more patches are written. Each patch is directly associated with a single problem report. A *patch* is a piece of code overwriting original code in order to correct—remove a fault from—or enhance—add new features to—System S software. A patch file contains a header describing the problem being corrected. Its body consists of 8086 Assembly language statements. A problem report just consists of a header describing the problem.

Some items in a fault report header are:

(A) Name of fault report.
(B) Date of fault report (year, month, day).
(C) Priority or level of severity of problem. There are four levels of priority:
    (1) System cannot be operational as long as this error exists.
    (2) System can be operational, but the operation can be heavily affected.
    (3) System can be operational with minor restrict.
    (4) Operation is not affected.
(D) Area of software where the problem was first found (operating system, call handling, database management, etc.).
(E) Phase of life cycle during which the problem was found (coding inspection, local integration, system test, module test, customer acceptance, etc.).
(F) Class of error (documentation, code, data, test, etc.).
(G) Build or installation the problem is found.

Some items in a patch header are:

(A) Name of patch.
(B) Test status (tested, untested, or excluded).
(C) Date of patch.
(D) Phase of life cycle during which the patch was provided.
(E) Class of error.
(F) Build or installation where the problem was found at first place.
(G) Application (i.e., whether the patch is official or not).
(H) Module name this patch applies to.
(I) Control element type this patch is entered.
(J) Identification of patches replaced by this patch.
(K) Problem reports or patches associated to this patch.

In this study relevant parts of this information are used.

5.2. System S Failure Data

First of all, it should be stressed that no control on the data collecting procedures was possible by the owner of this thesis. The data that were already recorded had to be used. Also, it was highly probable that duplicate fault reports were present. This can be easily understood from the fact that there were about 16,500 fault reports, while there were about 6750 patches, but each patch must be associated with a single fault report. This can be explained as follows: The same failure may be detected at different sites, so more than one distinct fault reports may be prepared describing the same fault. However this is not considered as a problem during operational phase. (See Section 4.1.2 in Musa *et al*. [1].)

The oldest patch that was found was recorded in the end of March 1983, while the oldest fault report was recorded in the middle of March 1985. This shows that a substantial amount of fault reports have been deleted. Also it has been said that patch header information is more reliable and up to date than fault report header information.

The original number of patch header information retrieved was about 10,500. From these about 500 were data updates. A *data update* is a piece of code overwriting original data (for example, constants of the system) in the software. It can be used for adapting the software to changing needs, or even correct some faults. Unfortunately, these had no headers, so they had to be ignored.

From the remaining about 10,000 patches it was found that there were duplicate patches with exactly the same header information. This can be explained as follows: During the edit-assemble-test cycle, usually more than one time the editor is used for the same patch. The editor does not delete the old version of a file when it makes a change on it. In this way there may be different versions of the same patch in the database. After deleting these, about 6750 patches were obtained.

For fault reports, only about 15 duplicate entries were found. This shows that after a fault report is prepared it is not modified. This is natural, since a fault report does not have a body. The final number of fault reports obtained was about 16,500.

VI. RELIABILITY MODEL

Models are to be used, but not to be believed.
Henri Theil

In this part of the thesis error data obtained for "System S" is modeled by trying various models and then chosing the best one for further study. Confidence intervals are provided for parameters obtained by maximum likelihood estimation on mean value function. Using these values, other quantities, like failure intensity function, total faults initially present in the system, etc., are easily derived. Modeling is done on module basis, but at the end of this part, error data is modeled for the whole system and the results are compared with those obtained previously.

6.1. Defining a Failure

Software reliability modeling rests on the failures experienced. On the other hand, the reason of a failure is a fault in the program. In response to a failure a modification is made on the program, hopefully, correcting the fault so that the same failure does not reoccur.

In other words, there are three different processes which take place at different times and which have not one-to-one relationship. The data obtained from the software system being studied is for the last two—fault reports and patches respectively. Since the quality of data in fault reports is not as good as that for the patches, patch header information has to be used to approximate faults. This also eliminates counting all repetitions of the same failure, since one (or more) patches are written to correct all the failures of the same type. As a patch describes the fault detection and removal process, in this part the general term "error" is used in place of "failure" to avoid confusion.

6.2. Grouping Errors

Instead of modeling the error behavior as a whole, it is a good idea to classify errors and conduct software reliability modeling on groups of errors.

One option is classifying errors by their *severity*. Generally each error is not of the same importance. Classifying errors according to their impact on the operations of the organization is a generally followed practice. Error severity should not be confused with the difficulty to identify and correct the error. The most important classification criteria are: (*a*) cost impact; (*b*) human life impact; and (*c*) service impact [1]. For a telephone switching system, service impact is the most appropriate criterion. In System S, for example, there are four level of "priorities" characterizing the severity of the problem.

Another option is to divide the system into a set of *component*s depending on the physical nature of the system, each of whose reliabilities is easy to measure. Since the error sample will be divided among components, there should not be a large number of them. In System S an appropriate division is to divide it into the main software or hardware modules. Other divisions, such as area of software where the problem was found, may be used.

In this study patches are grouped according to the hardware modules they are entered. The number of groups that are obtained is 94. In summary,
54 groups have less than 10 patches,
21 groups have between 10 and 34 patches,
four groups have between 35 and 100 patches, and
15 groups have more than 100 patches.

Since most of the groups have very few patches, these groups are ignored. Only the groups with more than 100 patches are being considered and these are also the most important ones.

Grouping patches according to the severity of the error being corrected is not done in this study because this information is not kept in the patch headers and has to be retrieved from fault report headers. However very few patches refer to (existent) fault reports to do this.

6.3. Trying Various Models

Five specific models can be considered:

(A) Exponential class (Sections 3.2.7 and 3.2.12):

$$\mu(t) = \beta_0\,[1 - \exp(-\beta_1 t)], \tag{6.1}$$

$$\lambda(t) = \beta_0\,\beta_1 \exp(-\beta_1 t). \tag{6.2}$$

(B) Geometric family (Section 3.2.15):

$$\mu(t) = \beta_0 \ln(1 + \beta_1 t), \tag{6.3}$$

$$\lambda(t) = \frac{\beta_0\,\beta_1}{1 + \beta_1 t}. \tag{6.4}$$

(C) Gamma class (a generalization of delayed S-shaped curve model described in Section 3.2.13) using the Erlangian distribution:

$$\mu(t) = \beta_0 \left[1 - \sum_{i=0}^{\gamma-1} \frac{(\beta_1 t)^i}{i!} \exp(-\beta_1 t) \right], \tag{6.5}$$

$$\lambda(t) = \beta_0 \frac{\beta_1{}^{\gamma}\, t^{\gamma-1}}{(\gamma-1)!} \exp(-\beta_1 t). \tag{6.6}$$

(D) Inflection S-shaped curve (Section 3.2.13):

$$\mu(t) = \frac{\beta_0\,[1 - \exp(-\beta_1 t)]}{1 + \beta_2 \exp(-\beta_1 t)}, \tag{6.7}$$

$$\lambda(t) = \frac{\beta_0\,\beta_1(1 + \beta_2)\exp(-\beta_1 t)}{[1 + \beta_2 \exp(-\beta_1 t)]^2}. \tag{6.8}$$

(E) Weibull class using the following generic function (Section 3.2.12):

$$\mu(t) = \frac{\beta_0}{\beta_1 \beta_2}\,[1 - \exp(-\beta_1 t^{\beta_2})], \tag{6.9}$$

$$\lambda(t) = \beta_0\, t^{\beta_2 - 1} \exp(-\beta_1 t^{\beta_2}). \tag{6.10}$$

The last two models have three parameters. Their estimation is more difficult and so is their interpretation, in other words, the physical meaning of the parameters is less

obvious. Also an advantage of the last model over the others was not observed. The third model has also three parameters, but γ can be viewed as a parameter that gives different models when set to various integer values. For example, when γ=1, the exponential and when γ=2, the delayed S-shaped curve [20] models are obtained. (As an improvement, γ may be considered as a real number, but in this case computations become much more complex.) In this study first γ is set to two, so the delayed S-shaped curve model (C2) is tried out together with the other two-parameter models—namely exponential (A) and geometric (B)—on the data obtained for the 15 groups described in the previous section. The results obtained are shown on Table 6.1.

TABLE 6.1 Maximum likelihood estimation of parameters of three models (modular basis)

MODULE	t_s	t_e	m_e	MODEL	$\hat{\beta}_0$	$\hat{\beta}_1$	SSE
C&T	1578	2893	110	A	454.617	2.1066×10^{-4}	1.443×10^{3}
(Clock and				B	441.614	2.1510×10^{-4}	1.517×10^{3}
tone)				C2	136.079	2.3202×10^{-3}	1.202×10^{3}
SRVC	1214	2893	135	A	-50.791	-7.7242×10^{-4}	3.625×10^{4}
(Service)				B	-164.055	-3.3403×10^{-4}	3.614×10^{4}
				C2	482.380	6.2089×10^{-4}	2.386×10^{4}
TTE	1412	2889	168	A	-5085.864	-2.2003×10^{-5}	8.752×10^{3}
(Trunk)				B	-5800.946	-1.9327×10^{-5}	8.717×10^{3}
				C2	226.623	1.7913×10^{-3}	4.490×10^{3}
LNE	904	2883	174	A	-166.678	-3.6123×10^{-4}	3.165×10^{4}
(Line)				B	-326.407	-2.0879×10^{-4}	3.187×10^{4}
				C2	329.255	8.9487×10^{-4}	1.559×10^{4}
BLNG	1626	2879	178	A	-305.550	-3.6635×10^{-4}	6.815×10^{3}
(Billing)				B	-433.284	-2.6887×10^{-4}	6.939×10^{3}
				C2	287.386	1.6728×10^{-3}	3.895×10^{3}
STD	1707	2879	226	A	515.118	4.9280×10^{-4}	9.257×10^{4}
(Auxiliary for				B	545.678	4.3780×10^{-4}	9.741×10^{4}
trunk)				C2	263.160	2.9442×10^{-3}	3.680×10^{4}
SADM	1037	2876	243	A	-39.420	-1.0708×10^{-3}	2.323×10^{5}
(Adminis-				B	-191.107	-3.9130×10^{-4}	2.331×10^{5}
tration)				C2	11807.691	1.1856×10^{-4}	1.772×10^{5}
PBX	1855	2914	260	A	379.540	1.0909×10^{-3}	8.397×10^{4}
				B	288.444	1.3815×10^{-3}	9.992×10^{4}
				C2	281.291	4.0008×10^{-3}	3.163×10^{4}
SPRX	1637	2897	322	A	702.830	4.8632×10^{-4}	4.799×10^{5}
(Auxiliary for				B	793.648	3.9713×10^{-4}	4.928×10^{5}
prefix)				C2	372.739	2.7760×10^{-3}	2.632×10^{5}

continued on next page

MODULE	t_s	t_e	m_e	MODEL	$\hat{\beta}_0$	$\hat{\beta}_1$	SSE
TTEST	1338	2893	341	A	-1150.065	-1.6699×10^{-4}	6.981×10^{5}
(Trunk test)				B	-2145.403	-9.4506×10^{-5}	6.959×10^{5}
				C2	500.804	1.5124×10^{-3}	4.297×10^{5}
RMLNS	1874	2907	402	A	-400.595	-6.7270×10^{-4}	1.067×10^{5}
(Remote				B	-603.394	-4.7082×10^{-4}	1.043×10^{5}
lines)				C2	749.945	1.7388×10^{-3}	1.726×10^{5}
ETTE	1850	2917	414	A	566.741	1.2288×10^{-3}	9.094×10^{4}
(Trunk)				B	365.656	1.9705×10^{-3}	1.366×10^{5}
				C2	442.056	4.1738×10^{-3}	1.077×10^{5}
DFC	1017	2900	661	A	-424.662	-4.9849×10^{-4}	3.276×10^{6}
(Defence)				B	-1092.530	-2.4107×10^{-4}	3.306×10^{6}
				C2	1507.329	7.9082×10^{-4}	1.794×10^{6}
ELNE	1562	2910	873	A	2314.975	3.5118×10^{-4}	8.111×10^{6}
(Line)				B	2817.509	2.6945×10^{-4}	8.254×10^{6}
				C2	1035.850	2.4565×10^{-3}	3.671×10^{6}
PFLD	1043	2891	1440	A	-906.317	-5.1473×10^{-4}	2.384×10^{7}
(Peripheral				B	-2248.907	-2.5588×10^{-4}	2.447×10^{7}
and load)				C2	3323.955	7.9727×10^{-4}	1.086×10^{7}

The first column gives the module name. The second and third columns give the start and end time, i.e., $[t_s, t_e]$ is the time interval between which the error data was collected. The time is given as days elapsed since December 31, 1980. Table 6.2 may be used to determine the actual date from the times used in this study.

TABLE 6.2 Conversion between first day of a month and number of days elapsed since December 31, 1980

MONTHS:	1	2	3	4	5	6	7	8	9	10	11	12
1981	1	32	60	91	121	152	182	213	244	274	305	335
1982	366	397	425	456	486	517	547	578	609	639	670	700
1983	731	762	790	821	851	882	912	943	974	1004	1035	1065
1984	1096	1127	1156	1187	1217	1248	1278	1309	1340	1370	1401	1431
1985	1462	1493	1521	1552	1582	1613	1643	1674	1705	1735	1766	1796
1986	1827	1858	1886	1917	1947	1978	2008	2039	2070	2100	2131	2161
1987	2192	2223	2251	2282	2312	2343	2373	2404	2435	2465	2496	2526
1988	2557	2588	2617	2648	2678	2709	2739	2770	2801	2831	2862	2892
1989	2923	2954	2982	3013	3043	3074	3104	3135	3166	3196	3227	3257
1990	3288	3319	3347	3378	3408	3439	3469	3500	3531	3561	3592	3622
1991	3653	3684	3712	3743	3773	3804	3834	3865	3896	3926	3957	3987

The fourth column of Table 6.1 shows the number of errors experienced. The fifth column shows the model used. The estimated parameters of the corresponding model are shown in the sixth and seventh columns, together with the error sum of squares (SSE=$\sum_{t=t_s}^{t_e}[m_t-\mu(t)]^2$) in the last column.

Using SSE as a comparison criterion between models, it can be observed that for 13 out of the 15 modules, Model C2 (delayed S-shaped curve model) gives better results. Model A (exponential class) is better for module ETTE, while Model B (geometric family) is the better for module RMLNS. This means that the best between these models for these data is the delayed S-shaped curve model. The exponential model is the second best model in this case.

Since the exponential and delayed S-shaped curve models are special cases of the gamma class model and since these two models are best for 14 out of 15 cases, the gamma class model can be used to model the error behavior of these modules. Also for the module where the geometric model seems better (RMLNS) the exponential model does not give very different results: The error sum of squares are comparable.

Before going further, some observations on these models will prove useful. The values of the parameters for the first two models are sometimes negative. The reason for this is that the exponential and geometric models can accommodate increasing and decreasing failure intensities, but not both at different times. This depends on the sign of the parameter values. Since failure intensity must be always positive, from Equations (6.2) and (6.4), it is deduced that both parameters must have the same sign. When they are positive, the failure intensity decreases; when they are negative it increases; and when $\beta_1=0$ (in this case $\beta_0\to\infty$) there is a constant failure intensity. This is shown in Figures 6.1 and 6.2.

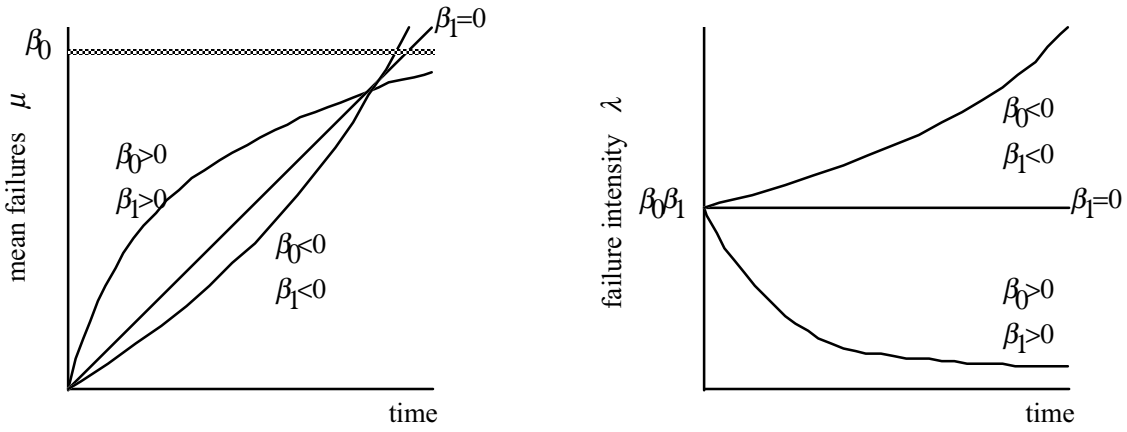

FIGURE 6.1 Behavior of exponential class models

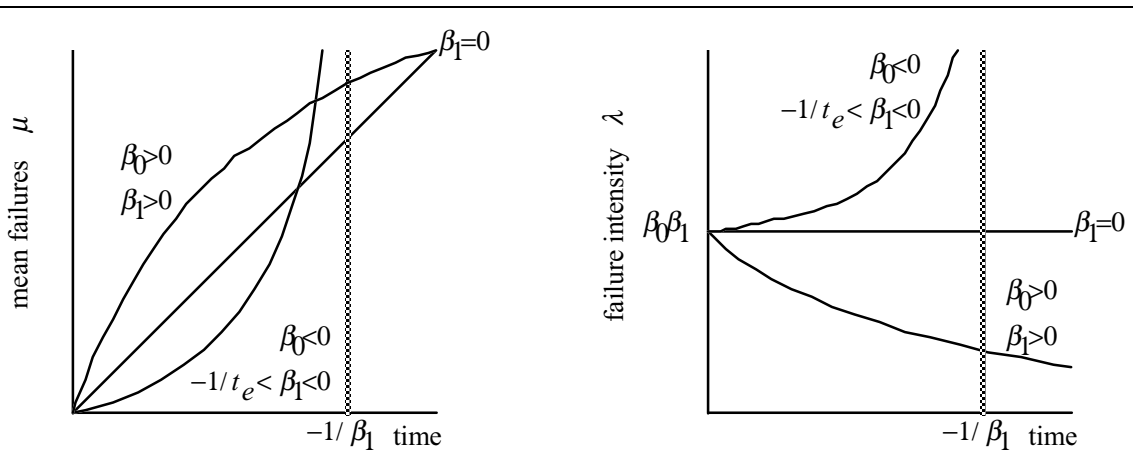

FIGURE 6.2 Behavior of geometric family models

Only in one case the exponential model becomes a finite errors model. This is when its parameters are positive. In this case as $t \to \infty$, $\lambda \to 0$ and $\mu \to \beta_0$, so β_0 is an estimate of the total number of errors that will occur at infinite time. But to have this, the failure intensity must decrease. This can be tested using the statistic [1]

$$U = \frac{\sum_{i=1}^{m_e} t_i - \frac{m_e t_e}{2}}{t_e \sqrt{\frac{m_e}{12}}}. \tag{6.11}$$

Large negative values of U indicate decreasing trend in failure intensity. When U is positive, the parameters for the exponential and geometric models become negative, which indicates an increase in failure intensity and reliability decay. Thus it seems that

for nine of the modules there is an overall increase in failure intensity according to these two models.

The geometric model, besides of being an infinite failures model, has another disadvantage when the failure intensity increases: At time $t=-1/\beta_1$ the failure intensity tends to infinity. For example, for module RMLNS at time t=1874+2124=3998 (end of year 1991) the failure intensity is undefined.

In summary, the exponential and geometric models are sometimes pessimistic. In gamma class models such problems are not present, since they are more realistic. For example, for the delayed S-shaped curve model the failure intensity increases until time $1/\beta_1$ and then starts to decrease. Generally, for gamma class models failure intensity attains to its maximum at $t=(\gamma-1)/\beta_1$. This is shown in Figure 6.3. Since for all software modules of this study, except SADM, this time is smaller than t_e, it can be deduced that for those modules the failure intensity started to decrease according to the delayed S-shaped curve model.

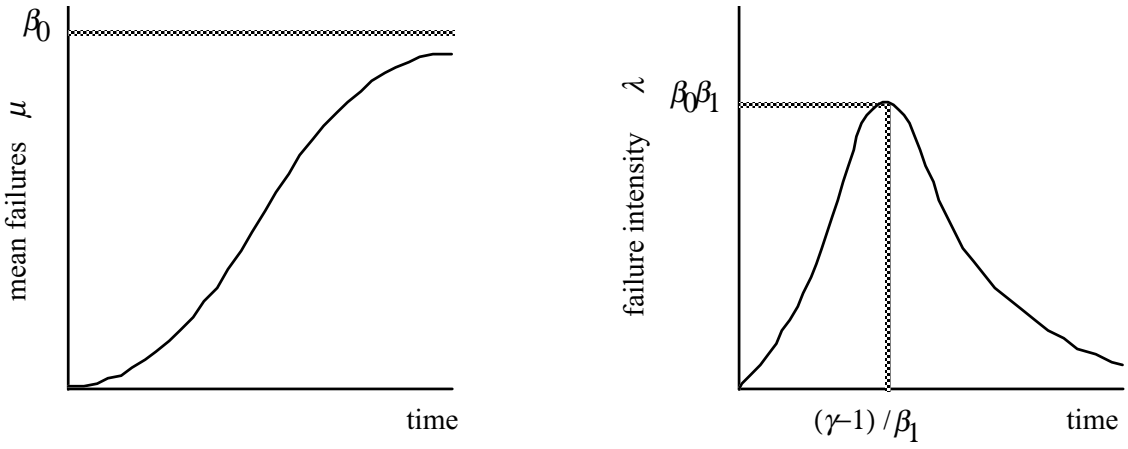

FIGURE 6.3 Behavior of gamma class models

6.4. Using the Gamma Class Model

After having chosen the gamma class model, there remains the determination of which γ value gives the best results. This can be done by trying various integer values for γ and chosing the one which gives minimum error sum of squares. Besides this, it should give a

curve that fits well to the actual data. This can be determined by testing the goodness of fit of the estimation using the Kolmogorov-Smirnov test [27, 28]. This is preferred to other methods such as chi-square or correlation coefficient [25]. The Kolmogorov-Smirnov test determines the significance of the model. For example, a level of significance α=0.10 means that the probability of accepting the hypothesis—the model can be used to explain the actual data—when it is in fact correct is p=0.90. In other words, the probability of rejecting a true hypothesis is α. The larger the level of significance is, the better the fit is.

The results are shown on Table 6.3. The fifth column of this table gives the optimum γ value. The last two columns give the maximum Kolmogorov-Smirnov distance (D_{max}) and the one that must not be exceeded (D_α) if the model is to be accepted at α level of significance. If $D_{max}>D_{0.01}$ the model cannot be accepted.

TABLE 6.3 Maximum likelihood estimation of parameters of gamma class models (modular basis)

MOD-ULE	t_s	t_e	m_e	γ	$\hat{\beta}_0$	$\hat{\beta}_1$	SSE	D_{max}	D_α
C&T	1578	2893	110	2	136.079	2.3202×10^{-3}	1.202×10^{3}	0.06061	$< 0.10202 = D_{0.20}$
SRVC	1214	2893	135	12	139.045	1.1536×10^{-2}	1.461×10^{3}	0.06301	$< 0.09209 = D_{0.20}$
TTE	1412	2889	168	2	226.623	1.7913×10^{-3}	4.490×10^{3}	0.05375	$< 0.08255 = D_{0.20}$
LNE	904	2883	174	5	192.585	4.0704×10^{-3}	7.697×10^{3}	0.07525	$< 0.08112 = D_{0.20}$
BLNG	1626	2879	178	2	287.386	1.6728×10^{-3}	3.895×10^{3}	0.03743	$< 0.08020 = D_{0.20}$
STD	1707	2879	226	3	240.861	5.1246×10^{-3}	2.118×10^{4}	0.09872	$< 0.10843 = D_{0.01}$
SADM	1037	2876	243	12	261.000	9.5057×10^{-3}	6.535×10^{4}	0.13802	$> 0.10456 = D_{0.01}$
PBX	1855	2914	260	2	281.291	4.0008×10^{-3}	3.163×10^{4}	0.08265	$< 0.08434 = D_{0.05}$
SPRX	1637	2897	322	4	332.020	6.7434×10^{-3}	1.359×10^{5}	0.12444	$> 0.09084 = D_{0.01}$
TTEST	1338	2893	341	9	345.300	1.0945×10^{-2}	3.224×10^{4}	0.06483	$< 0.06607 = D_{0.10}$
RMLNS	1874	2907	402	1	-400.595	-6.7270×10^{-4}	1.067×10^{5}	?	$> 0.08130 = D_{0.01}$
ETTE	1850	2917	414	1	566.741	1.2288×10^{-3}	9.094×10^{4}	0.05558	$< 0.05603 = D_{0.15}$
DFC	1017	2900	661	8	693.662	7.0430×10^{-3}	7.082×10^{4}	0.04165	$< 0.04434 = D_{0.15}$
ELNE	1562	2910	873	5	891.380	7.8148×10^{-3}	7.510×10^{5}	0.06729	$> 0.05517 = D_{0.01}$
PFLD	1043	2891	1440	6	1578.723	5.1481×10^{-3}	3.696×10^{5}	0.02535	$< 0.02820 = D_{0.20}$

The results on this table show that,
for six modules the model can be accepted at a level of significance α=0.20,
for two other modules the model can be accepted at a level of significance α=0.15,
for another module the model can be accepted at a level of significance α=0.10,
for another module the model can be accepted at a level of significance α=0.05,
for another module the model can be accepted at a level of significance α=0.01, and
for the remaining four modules the model cannot be accepted at even a level of significance α=0.01 according to Kolmogorov-Smirnov test.

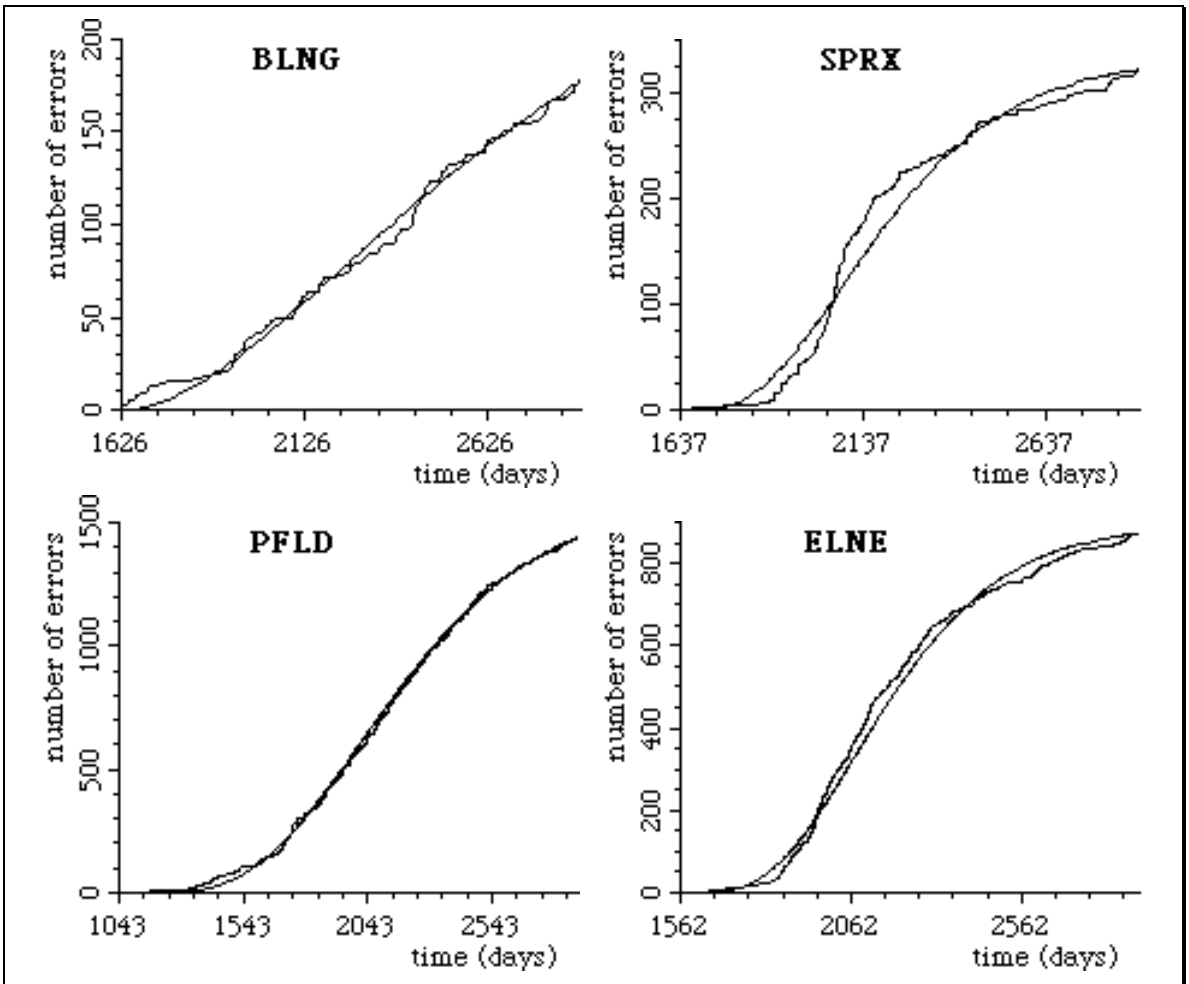

FIGURE 6.4 Gamma class models applied on real data

Figure 6.4 gives graphs for a few modules showing the true errors experienced and the estimated mean value function according to the results of Table 6.3. For example, the worst case according to Kolmogorov-Smirnov test is for module SPRX. The best fit is obtained for module BLNG; the graph confirms this result. On the other hand module ELNE cannot be explained by this model (according to Kolmogorov-Smirnov test), while module PFLD can be explained with a high level of significance. In the graphs for

modules BLNG and ELNE the fit seems equally good, but this is not confirmed by the Kolmogorov-Smirnov test. Also, the fit for module PFLD seems better than the one for BLNG, while the Kolmogorov-Smirnov test shows the contrary. This is, of course, a direct result of a property of this test: As the number of errors, i.e., the number of data points increases the test becomes more strict. Another test for the significance or goodness of fit of a model may lead to a different evaluation.

6.5. Estimating Intervals

The main problem with interval estimation described at the end of Section 4.2 is that in some cases the determined upper and lower bounds are practically meaningless since they depend on the estimated mean value function, but not on the goodness-of-fit index [25]. For example, as shown in the previous section, for some modules the model fits well while for others it does not. However this information is not used in the interval estimation.

In this section interval estimation is done using the method described in Section 4.2, i.e., by using the expected, or Fisher, information. For this purpose, 95 per cent confidence intervals for β_1 are established, then these are substituted separately in

$$\hat{\beta}_0 = \frac{m_e}{1 - \sum_{i=0}^{\gamma-1} \frac{(\hat{\beta}_1 t_e)^i}{i!} \exp(-\hat{\beta}_1 t_e)} \tag{6.12}$$

to obtain confidence intervals for β_0. The results are shown on Table 6.4. The numbers in parentheses show the lower and upper bounds for the corresponding point estimates.

TABLE 6.4 Interval estimation of parameters of gamma class models (modular basis)

MOD-ULE	γ	$\hat{\beta}_0$			$\hat{\beta}_1$		
C&T	2	136.079	(123.279	164.045)	2.3202×10^{-3}	$(1.7537\times10^{-3}$	$2.8868\times10^{-3})$
SRVC	12	139.045	(137.378	141.754)	1.1536×10^{-2}	$(1.0908\times10^{-2}$	$1.2164\times10^{-2})$
TTE	2	226.623	(201.358	276.808)	1.7913×10^{-3}	$(1.3868\times10^{-3}$	$2.1957\times10^{-3})$
LNE	5	192.585	(185.682	203.455)	4.0704×10^{-3}	$(3.7051\times10^{-3}$	$4.4358\times10^{-3})$
BLNG	2	287.386	(238.405	397.185)	1.6728×10^{-3}	$(1.2114\times10^{-3}$	$2.1343\times10^{-3})$
STD	3	240.861	(235.459	249.313)	5.1246×10^{-3}	$(4.6238\times10^{-3}$	$5.6255\times10^{-3})$
SADM	12	261.000	(255.500	268.704)	9.5057×10^{-3}	$(9.0845\times10^{-3}$	$9.9269\times10^{-3})$
PBX	2	281.291	(273.628	293.430)	4.0008×10^{-3}	$(3.5175\times10^{-3}$	$4.4840\times10^{-3})$
SPRX	4	332.020	(328.804	336.685)	6.7434×10^{-3}	$(6.3154\times10^{-3}$	$7.1714\times10^{-3})$
TTEST	9	345.300	(343.957	347.207)	1.0945×10^{-2}	$(1.0532\times10^{-2}$	$1.1359\times10^{-2})$
RMLNS	1	-400.595	(-952.238	-220.577)	-6.7270×10^{-4}	$(-1.0045\times10^{-3}$	$-3.4093\times10^{-4})$
ETTE	1	566.741	(511.303	669.554)	1.2288×10^{-3}	$(9.0270\times10^{-4}$	$1.5550\times10^{-3})$
DFC	8	693.662	(686.787	702.246)	7.0430×10^{-3}	$(6.8183\times10^{-3}$	$7.2677\times10^{-3})$
ELNE	5	891.380	(887.504	896.238)	7.8148×10^{-3}	$(7.5562\times10^{-3}$	$8.0734\times10^{-3})$
PFLD	6	1578.723	(1558.481	1602.299)	5.1481×10^{-3}	$(5.0058\times10^{-3}$	$5.2904\times10^{-3})$

6.6. Deriving Other Quantities

For gamma class models, as $t\to\infty$, $\lambda\to0$ and $\mu\to\beta_0$, so β_0 is an estimate of the total number of errors in the software module. Thus the sixth column of Table 6.3 designates the total number of errors that will be experienced at infinite time. One exception is that module RMLNS has infinite errors, in other words, the data set does not show reliability growth. This may be due to the fact that t_s is the highest among the others modules, so enough data were not collected to adequately fit a model. (Note that the exponential model does not fit according to Kolmogorov-Smirnov test.)

The expected number of errors remaining at time t is given by

$$\omega(t) = \beta_0 - \mu(t) = \beta_0 \sum_{i=0}^{\gamma-1} \frac{(\beta_1 t)i}{i!} \exp(-\beta_1 t) \tag{6.13}$$

By substituting the estimated parameters into Equation (6.6), an estimate of the failure intensity as a function of time can be obtained. (Note that before using time in such equations t_s must be subtracted from it.) Estimated failure intensity started to decrease, according to gamma class models, for all modules of this study except for the problematic module RMLNS.

Table 6.5 gives the expected number of remaining errors and the failure intensity as a function of future time. As shown the failure intensity and number of remaining errors is expected to drop except for module RMLNS.

TABLE 6.5 Expected number of remaining errors and failure intensity (modular basis)

MODULE	$\omega(t)$				$\lambda(t)$			
start of	1989	1990	1991	1992	1989	1990	1991	1992
C&T	25	13	6	3	4.35×10^{-2}	2.37×10^{-2}	1.23×10^{-2}	6.22×10^{-3}
SRVC	3	0	0	0	1.93×10^{-2}	2.40×10^{-3}	2.12×10^{-4}	1.46×10^{-5}
TTE	56	34	21	12	7.34×10^{-2}	4.74×10^{-2}	2.94×10^{-2}	1.78×10^{-2}
LNE	17	7	3	1	4.02×10^{-2}	1.77×10^{-2}	7.07×10^{-3}	2.64×10^{-3}
BLNG	104	67	42	26	1.19×10^{-1}	8.29×10^{-2}	5.49×10^{-2}	3.52×10^{-2}
STD	13	3	1	0	4.71×10^{-2}	1.23×10^{-2}	2.86×10^{-3}	6.22×10^{-4}
SADM	15	3	0	0	6.26×10^{-2}	1.36×10^{-2}	2.22×10^{-3}	2.90×10^{-4}
PBX	21	6	2	0	6.70×10^{-2}	2.09×10^{-2}	6.08×10^{-3}	1.70×10^{-3}
SPRX	9	1	0	0	4.17×10^{-2}	7.53×10^{-3}	1.17×10^{-3}	1.64×10^{-4}
TTEST	4	0	0	0	2.25×10^{-2}	2.17×10^{-3}	1.58×10^{-4}	9.36×10^{-6}
RMLNS	?	?	?	?	5.46×10^{-1}	6.98×10^{-1}	8.92×10^{-1}	1.14
ETTE	152	97	62	39	1.86×10^{-1}	1.19×10^{-1}	7.60×10^{-2}	4.85×10^{-2}
DFC	30	7	1	0	1.13×10^{-1}	2.94×10^{-2}	6.38×10^{-3}	1.21×10^{-3}
ELNE	17	2	0	0	8.93×10^{-2}	1.33×10^{-2}	1.66×10^{-3}	1.82×10^{-4}
PFLD	127	42	13	4	3.60×10^{-1}	1.34×10^{-1}	4.33×10^{-2}	1.27×10^{-2}

From Equation (2.6)

$$R(t) = \exp\left[-\int_0^t z(x)dx\right] \tag{6.14}$$

is obtained. Since program hazard rate is the same as the failure intensity function for Poisson-type models, the following is obtained [1]

$$R(t_i'|t_{i-1}) = \exp\{-[\mu(t_{i-1}+t_i')-\mu(t_{i-1})]\}, \quad i=1, 2, \ldots \tag{6.15}$$

for the probability of failure-free operation during time interval t_i' between the ith and (i-1)th failure. Since failure intensity is decreasing, except for module RMLNS, it can be concluded that reliability is increasing for all the other modules.

6.7. Modeling the Whole System

Without doing the decomposition on error data done in Section 6.2 the whole system can be modeled. In this section two different models will be used for this purpose: The gamma class model which has proven to be satisfactory on module basis and the inflection S-shaped curve model whose parameters are difficult and time consuming to compute. (Also the third parameter, β_2, must be given to the estimating procedure as input [25].)

The results of the gamma model are given on Table 6.6. The estimated time at which failure intensity is maximum is also shown. Estimated values for parameters are given together with their 95 per cent confidence intervals. Also the expected number of remaining errors and failure intensity is given as a function of future time. Figure 6.5 gives the actual and estimated cumulative number of errors as a function of time and Figure 6.6 gives the estimated failure intensity as a function of time. As it is seen, quantities like failure intensity, reliability, and total number of errors in the system can easily be derived. For example, the reliability started to increase and the expected number of errors at infinite time is 7084 according to this model. The number of errors experienced so far, 6738, make up 95 per cent of the total expected errors.

TABLE 6.6 Maximum likelihood estimation on System S error data using gamma class model

t_s	=	820		
t_e	=	2917		
m_e	=	6738		
γ	=	13		
$\hat{\beta}_0$	=	7084.163	(7060.767	7109.127)
$\hat{\beta}_1$	=	9.2966×10^{-3}	(9.2246×10^{-3}	9.3686×10^{-3})
$t_{\lambda_{max}}$	=	2111		

start of	**1989**	**1990**	**1991**	**1992**
$\omega(t)$	338	67	11	1
$\lambda(t)$	1.39	3.18×10^{-1}	5.58×10^{-2}	8.03×10^{-3}

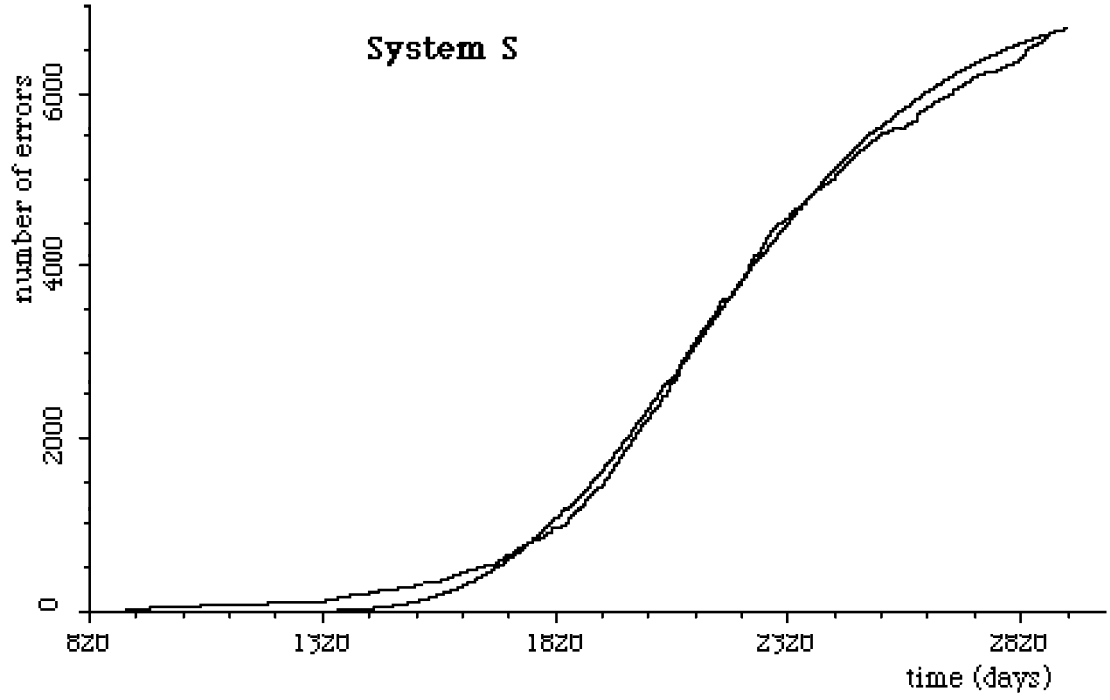

FIGURE 6.5 Gamma class model applied on System S error data

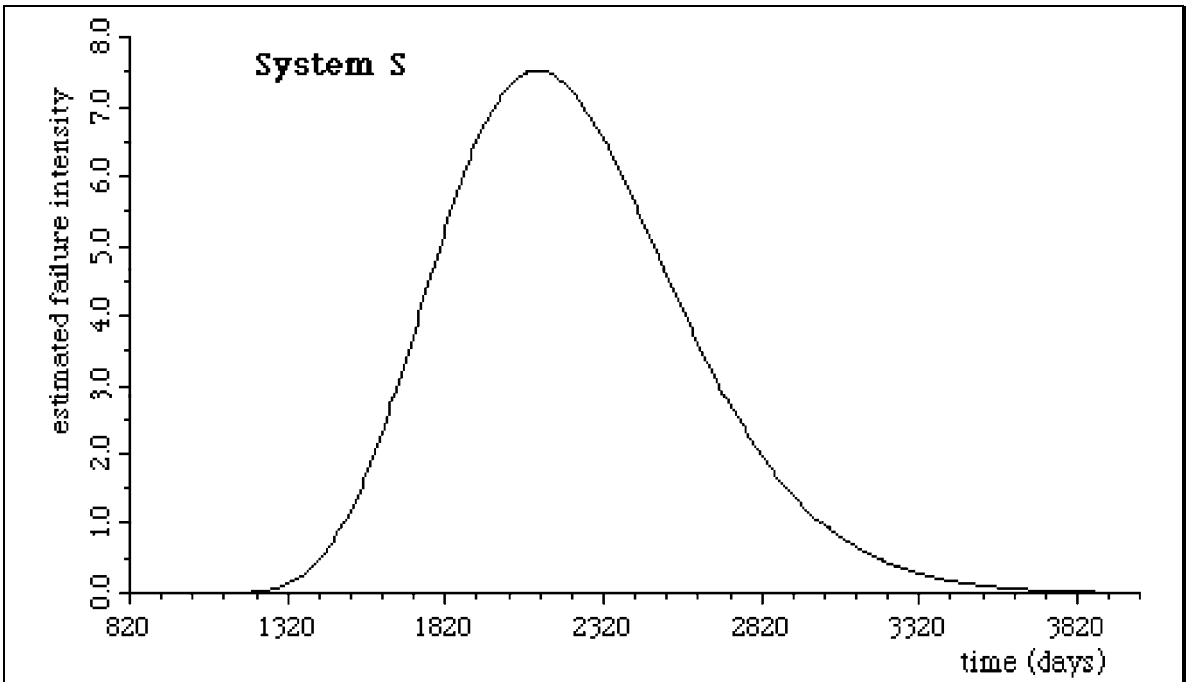

FIGURE 6.6 Estimated failure intensity of System S using gamma class model

A similar study was conducted using inflection S-shaped curve model. Various values for the third parameter, β_2, were tried and the optimum one was chosen. The results are given on Table 6.7. The fit for this model is slightly worse than the fit for gamma class model and the derived quantities, like total number of errors, time of maximum failure intensity, expected number of remaining errors, estimated failure intensity, are slightly different. For example, according to this model 96 per cent of the total expected errors have been experienced. On the other hand this quantity is 91 per cent on module basis and 95 per cent on system basis using gamma class models. This means that roughly 90-95 per cent of the total errors have been corrected by the end of year 1988.

TABLE 6.7 Maximum likelihood estimation on System S error data using inflection S-shaped curve model

t_s	=	820		
t_e	=	2917		
m_e	=	6738		
$\hat{\beta}_0$	=	7037.912	(7018.060	7059.170)
$\hat{\beta}_1$	=	4.3657×10^{-3}	(4.3330×10^{-3}	4.3983×10^{-3})
β_2	=	420		
$t_{\lambda_{max}}$	=	2204		

start of	**1989**	**1990**	**1991**	**1992**
$\omega(t)$	292	61	13	3
$\lambda(t)$	1.22	2.66×10^{-1}	5.48×10^{-2}	1.12×10^{-2}

VII. CONCLUSIONS

This study has shown the relative advantage of the S-shaped reliability growth models especially in the absence of additional information about the software system being modeled. In particular, the three-parameter gamma class model is the most appropriate, at least for the system under study. The disadvantage of the two-parameter models—like exponential or geometric—is that they lack flexibility. They cannot accommodate reliability decay and growth at different times during debugging, something which is generally encountered in real life and can be modeled by a gamma class model. Although the gamma class model assumes eventual reliability growth, in one case—that is, when the third parameter becomes one, so that the exponential model is obtained—it can provide reliability decay. But in this case no useful results can be derived as happened for one software module of this study.

The theory behind the advantage of a gamma class model—e.g., delayed S-shaped growth model—over the exponential growth models is discussed in some detail by Ohba [25]. The most important difference is in the definition of errors; the exponential software reliability growth models are designed for modeling a failure detection process while the gamma class models are more appropriate for a test process analyzed as a fault isolation process. Also, during a fault isolation process, some faults may be removed without failure detection by a test team. All these conditions are also valid for System S data, which explains why the gamma class models fit better than other models to these data.

Some general problems in software reliability modeling are briefly discussed below [1].

Firstly execution time should be used during modeling and then this should be converted to calendar time. However this requires knowledge of the planned and resource usage parameters as explained in Subsection 3.2.7. Since these data were not available for this study, and it is believed that in operational phase this is not as important as in testing—i.e., calendar time *is* proportional to CPU time—this was not done. If it had been done, possibly, the exponential model could fit better.

Another problem is the definition of a failure. This is discussed in Section 6.1. Besides this, should requests of new features be considered as failures? This also brings about the problem of evolving programs which is discussed below.

There is also the problem of time uncertainties or assigning multiple failures to the same time point. This results in unpredictable, or even infinite, failure intensities at some time points which causes problems in estimations (like least squares estimation discussed in Section 4.1) which depend on failure intensity. One solution is to assign random times within a day—as can be done for System S data—to each failure occurred at that day. In maximum likelihood estimation, which depends on the number of failures experienced, this does not cause a problem and such random assignment of failure times within a day produces less than one per cent improvement in the goodness of fit of the estimation.

When a software is used and tested in multiple installations with different processors, failure times should be interleaved by adjusting all execution times and failure intensities to a "reference computer." In the system under study these are similar processors with same instruction execution rates.

Reliability models assume stable program, except changes resulting from failure correction. But in most systems, and in System S too, there are requirement changes that result in new features being added to the system. Solutions to this problem are:

(A) Think changes as additions or removals of independent subsystems, so do modeling on each "subsystem" and then combine the results.

(B) Adjust failure times to what they would have been if the complete final program had been present and then think the data as coming from a stable program.

(C) Ignore changes and allow model to adapt its parameters.

The disadvantage of the first two options is the need of additional data about the software, while for the third option estimation errors are larger.

Changes in environment or operational profile is another problem that should be accounted for.

All these problems contribute into uncertainties in the estimated parameters of a model which should be reflected on the results obtained, that is the lower and upper bounds of the model parameters. "Nonclassical" or Bayesian approaches, like the

Littlewood-Verrall (Section 3.2.8) may prove useful for this purpose, but estimation is much more complex.

APPENDIX A. OPTIMIZATION ALGORITHM

To estimate the unknown parameters of a specific software reliability model it is generally necessary to optimize a function. This may be for *least squares estimation* the minimization of the sum of squares function, and for *maximum likelihood estimation* (MLE) the maximization of the log-likelihood function. There is no essential difference between the maximization and minimization problems, because the values of x_j which maximize $f(\boldsymbol{x})$ also minimize $-f(\boldsymbol{x})$.

Since there is presently not a single recommended method for solving every general optimization problem, nor is there ever likely to be, it is important to take advantage of the special features that a given problem may posses. For software reliability modeling, it is suggested [1] that either the Newton-Raphson root finding procedure, or the Nelder and Mead searching procedure, or both be used.

The Newton-Raphson method is a numerical root finding procedure using gradient information. It has the advantage of converging very rapidly if the initial estimate for a variable is close enough to the final solution. The main disadvantage is that it may not always converge, especially when the initial estimate is not good enough. This problem becomes more severe as the number of unknown parameters to be estimated increases.

The Nelder and Mead method is a simple to program direct search method which is based neither on gradients (first-order derivatives) nor on quadratic forms (second-order derivatives). No assumptions are made about the surface defined by the function except that it is continuous and has a unique minimum in the area of the search. This method is one of the most efficient pattern search methods and has been found to work well when the number of variables does not exceed five or six. It is robust and always converges to a local minimum. The main disadvantage is that it is slow, especially in the neighborhood of a minimum, when compared with methods which depend on arguments applicable to quadratic forms.

These two methods are discussed in more detail in the following sections.

A.1. Newton-Raphson Root Finding Procedure

As a numerical root finding procedure for systems of nonlinear simultaneous equations the Newton-Raphson procedure can be used. This is a widely used technique for solving equations where a direct algebraic solution is not possible. If a root for the equation $f(x)=0$ is sought, the formula

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \quad n=0, 1, \ldots \tag{A.1}$$

can be used to improve the first approximation x_0 to root to whatever degree of accuracy that is required.

More generally, we want to solve multiple equations for roots β_k (k=0, 1,…, w), i.e., we have the equation $\mathbf{U}(\boldsymbol{\beta})=0$, where $\mathbf{U}(\boldsymbol{\beta})$ is a $(w+1)\times 1$ column vector with elements $\mathbf{U}_k(\boldsymbol{\beta})=f_k(\boldsymbol{\beta})$, k=0, 1, …, w. Then we have the formula

$$\boldsymbol{\beta}' = \boldsymbol{\beta} - \mathbf{H}^{-1}(\boldsymbol{\beta}) \times \mathbf{U}(\boldsymbol{\beta}) \tag{A.2}$$

where $\boldsymbol{\beta}'$ is closer to root than the previous approximation $\boldsymbol{\beta}$. The matrix $\mathbf{H}(\boldsymbol{\beta})$, also called the *Hessian matrix*, is a $(w+1)\times(w+1)$ square matrix with elements

$$\mathbf{H}_{kl}(\boldsymbol{\beta}) = \frac{\partial f_k(\boldsymbol{\beta})}{\partial \beta_l} = f_{kl}(\boldsymbol{\beta}), \quad k, l=0, 1, \ldots, w. \tag{A.3}$$

The above "matrix" version of the original Newton-Raphson formula is applied repeatedly by replacing $\boldsymbol{\beta}$ with the newly found vector $\boldsymbol{\beta}'$, until successive estimates agree to a specified tolerance on an element by element basis.

If w=0, then $\boldsymbol{\beta}$ is a single-element vector and we obtain the original formula with x_n replaced by β_0, x_{n+1} by β_0', $f(x_n)$ by $f_0(\beta_0)$, and $f'(x_n)$ by $f_{00}(\beta_0)=\dfrac{\partial f_0(\beta_0)}{\partial \beta_0}$.

If w=1, then we have

$$\begin{bmatrix}\hat{\beta}_0\\ \hat{\beta}_1\end{bmatrix} = \begin{bmatrix}\beta_0\\ \beta_1\end{bmatrix} - \begin{bmatrix}f_{00} & f_{01}\\ f_{10} & f_{11}\end{bmatrix}^{-1} \times \begin{bmatrix}f_0\\ f_1\end{bmatrix} = \begin{bmatrix}\beta_0\\ \beta_1\end{bmatrix} - \frac{\begin{bmatrix}f_{11} & -f_{01}\\ -f_{10} & f_{00}\end{bmatrix} \times \begin{bmatrix}f_0\\ f_1\end{bmatrix}}{\begin{vmatrix}f_{00} & f_{01}\\ f_{10} & f_{11}\end{vmatrix}}$$

$$= \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix} - \frac{1}{\begin{vmatrix} f_{00} & f_{01} \\ f_{10} & f_{11} \end{vmatrix}} \begin{bmatrix} \begin{vmatrix} f_0 & f_{01} \\ f_1 & f_{11} \end{vmatrix} \\ \\ \begin{vmatrix} f_{00} & f_0 \\ f_{10} & f_1 \end{vmatrix} \end{bmatrix} \tag{A.4}$$

where $f_{kl}(\boldsymbol{\beta})$ is shown as f_{kl} for simplicity.

In most cases w does not exceed 1 and in some cases it is 0, so we need not bother ourselves with more complex situations.

In reliability estimation we have a function to be maximized. For example, in maximum likelihood estimation we have the likelihood function $L(\boldsymbol{\beta})$. To maximize it, its derivatives are set to zero, i.e.,

$$\frac{\partial \ln L(\boldsymbol{\beta})}{\partial \beta_k} = 0, \quad k=0, 1, \ldots, w. \tag{A.5}$$

So the elements of vector $\mathbf{U}(\boldsymbol{\beta})$ are

$$\mathbf{U}_k(\boldsymbol{\beta}) = f_k(\boldsymbol{\beta}) = \frac{\partial \ln L(\boldsymbol{\beta})}{\partial \beta_k} = 0, \quad k=0, 1, \ldots, w, \tag{A.6}$$

and of matrix $\mathbf{H}(\boldsymbol{\beta})$ are

$$\mathbf{H}_{kl}(\boldsymbol{\beta}) = f_{kl}(\boldsymbol{\beta}) = \frac{\partial^2 \ln L(\boldsymbol{\beta})}{\partial \beta_k \, \partial \beta_l}, \quad k, l=0, 1, \ldots, w. \tag{A.7}$$

The Newton-Raphson procedure converges rapidly if it does, which generally happens if the initial estimate is close enough to $\hat{\boldsymbol{\beta}}$. If the initial estimate is poor it may diverge. Also, the evaluation of the Hessian matrix, and its inversion, may pose formidable computational problems.

A.2. Nelder and Mead Simplex Method

The Nelder and Mead method [29] is largely based on the method developed by Spendley, Hext, and Himsworth (1962). (See Chapter 3 in Walsh [30].) It can be used to minimize a function of n variables. The function may be considered as a mapping from a

point $\mathbf{p}_i$ in the n-dimensional space E^n to a single value in $\Re$. An initial estimate point $\mathbf{p}_0$ is chosen together with n additional points $\mathbf{p}_i$ (i=1,…,n) preferably in n different directions away from $\mathbf{p}_0$. These (n+1) points are the vertices of a *simplex* which is not necessarily *regular*. For example, in E^2 a simplex is a triangle and a regular simplex is an equilateral triangle; in E^3 a (regular) simplex is a (regular) tetrahedron, and so on. At each stage of the application of the method the point $\mathbf{p}_h$ (with the highest function value) is replaced by a new (and more optimal) point. In this way the simplex moves towards the minimum of the function.

The advantage of the Nelder and Mead method to the one described earlier by Spendley *et al*. is that here "the simplex adapts itself to the local landscape, elongating down long inclined planes, changing direction on encountering a valley at an angle, and contracting in the neighborhood of a minimum." To accomplish this, three operations are used: *reflection*, *contraction*, and *expansion*. The factor by which the volume of the simplex is changed by these operations is given by the coefficients α, β, and γ respectively. The complete method in algorithm form is given below.

Definitions:

Solution to be found:	$\mathbf{p}^*$	a column vector denoting a point in Euclidean n-space E^n, $[x_1, x_2, \ldots, x_n]$.
Function to be minimized:	$f(\mathbf{p})$	a function from E^n to $\Re$.
Initial estimate of $\mathbf{p}^*$:	$\mathbf{p}_0$	
Vertices of simplex:	$\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$	such that $\mathbf{p}_i = \mathbf{p}_0 + h_i\mathbf{e}_i$ (i=1, …, n) where $\mathbf{e}_i$ are the unit coordinate vectors and the scalars h_i are chosen so as to equalize $\lvert f(\mathbf{p}_0+h_i\,\mathbf{e}_i) - f(\mathbf{p}_0)\rvert$ as far as possible.

Also define

$$\mathbf{y}_i = f(\mathbf{p}_i)$$

$$\mathbf{y}_h = \max_{i=0}^{n} (y_i) = f(\mathbf{p}_h)$$ highest function value,

$\mathbf{y}_s = \max_{\substack{i=0 \\ i \neq h}}^{n} (y_i) = f(\mathbf{p}_s)$ second highest function value,

$\mathbf{y}_l = \min_{i=0}^{n} (y_i) = f(\mathbf{p}_l)$ lowest function value,

$\mathbf{pc} = \frac{1}{n} \sum_{\substack{i=1 \\ i \neq h}}^{n+1} \mathbf{p}_i$ centroid of all vertices except $\mathbf{p}_h$.

Algorithm:

Calculate initial $\mathbf{p}_i, y_i$ (i=0, …, n);

while not converged do
    determine l, h, s, and **pc**;
    pp = (1+α) **pc** − α $\mathbf{p}_h$; /* reflection */
    $yy = f(\mathbf{pp})$;
    if ($y_l \le yy \le y_s$) /* case 1 */
        $\mathbf{p}_h$ = **pp**;
        $y_h = f(\mathbf{p}_h)$;
        start next iteration;
    if ($yy < y_l$) /* case 2: expansion */
        ppp = (1−γ) **pc** + γ **pp**;
        $yyy = f(\mathbf{ppp})$;
        if ($yyy < y_l$)
            $\mathbf{p}_h$ = **ppp**;
        else
            $\mathbf{p}_h$ = **pp**;
        $y_h = f(\mathbf{p}_h)$;
        start next iteration;
    if ($yy > y_s$) /* case 3: contraction */
        if ($yy < y_h$)
            ppp = (1-β) **pc** + β **pp**;
        else
            ppp = (1-β) **pc** + β $\mathbf{p}_h$;
        $yyy = f(\mathbf{ppp})$;

```
        if (yyy < y_h and yyy < yy)
                p_h = ppp;
                y_h = f(p_h);
                start next iteration;
        else
                for (i=0, …, n and i≠l)
                        p_i = (p_i + p_l) / 2;
                        y_i = f(p_i);
                start next iteration;
endwhile;
p* = p_l;
```

Convergence is tested with $\sqrt{\{\sum(y_i-y)^2/n\}} < \varepsilon$ or $\{\sum y_i^2-(\sum y_i)^2/(n+1)\}/n < \varepsilon^2$. The constants are defined as follows (suggested values are given inside parentheses): $\alpha>0$ ($\alpha=1$), $0<\beta<1$ ($\beta=1/2$), $\gamma>1$ ($\gamma=2$), and $0<\varepsilon<<1$ ($\varepsilon=10^{-8}$).

A.3. Recommended Procedure

Musa *et al*. [1], after pointing out the problems with the above two methods suggest a method which combines these two. The strategy is to use the simplex procedure until the solution is close enough to the roots for the Newton-Raphson procedure. In other words, the simplex procedure is not allowed to converge completely. The obtained estimate is used by the Newton-Raphson procedure to get the final answer. If it converges after a specific number of iterations we are done. Otherwise, the simplex procedure is used again to improve the estimate more. This is given again to the Newton-Raphson procedure and the process is repeated until convergence is obtained or program time limit is exceeded.

If the initial estimate is good the performance of this method will be also good. From practical experience it is suggested that the value $1/t_e$ be used as the initial estimate for the parameter β_1 in the maximum likelihood estimation for both the exponential class and geometric family of Poisson models. This is a good initial estimate for other models too.

A.4. Problems

There are at least two problems that should be considered when using this maximization procedure. First, we should make sure that we have found a maximum (for maximum likelihood estimation) and not a minimum, and second, that this is a global maximum and not a local one.

The first concern is not so difficult to avoid since we can always test the second derivative at the solution point. Also the Nelder and Mead procedure is always directed to a maximum (of the negative) of the function.

The second problem occurs when we have a small sample to estimate multiple parameters. Once the sample size exceeds 25 this problem becomes rare. But it is always a good idea to use the procedure with different starting values and use the best solution if there are more than one.

A.5. Program

By using the methods outlined in this appendix, a program was developed which may be used to estimate parameters for a specific model using data obtained from a software project. Maximum likelihood estimation on mean value function is used in this program since this is preferred over the least squares estimation. It should be pointed out that a similar program using least squares estimation on failure intensity was tried, but the results were not satisfactory.

Four models are implemented in this program: Exponential class, geometric family, gamma class, and (inflection) S-shaped curve reliability growth models. The last two models can be considered as generalizations of the exponential model with an additional parameter; when this parameter is set to one, for the gamma model, and to zero, for the inflection S-shaped curve model, the exponential model is obtained. All these models are Poisson type. Also a Binomial type exponential model is provided. New models can easily be added to this program.

The function specific to each model is coded in the program together with its derivatives. The program finds the roots for this function. The Nelder and Mead procedure, which is actually a function minimization method, tries to minimize the square of the function, while the Newton-Raphson procedure finds the roots of the function using as initial estimate the value obtained by the Nelder and Mead procedure.

```
/***  MLE - Maximum likelihood estimation of the parameters of
 *        some software reliability models using Nelder and Mead
 *        (N-M) method and Newton-Raphson (N-R) procedure.
 *
 *    Written by Fedon Kadifeli. 1989.
 *
 *    Input - This program reads from a file observed times of
 *        failures. If known, the start and end test times can
 *        also be given as negative numbers in the beginning of
 *        the file. Otherwise the smallest and largest time
 *        values will be used for test start and end times,
 *        respectively.
 *
 *    Output - Estimated values of the parameters of the chosen
 *        model together with 95 per cent confidence intervals
 *        are displayed after some statistics about the failure
 *        data.
 *
 *    Options - The models supported by this program are:
 *        Exponential class (Poisson and binomial),
 *        geometric family (Poisson),
 *        gamma class (Poisson), and
 *        inflection S-shaped reliability growth model.
 *        For three-parameter models the third parameter is
 *        supplied by the user.
 */

char usage [] = "\
\n    MLE - Estimate parameters for a model using MLE on mean\
\n        value function.\
\n\
\n    Command call:\
\n\
\n    MLE  dataf\
\n\
\n    dataf: file containing failure time data (need not be\
\n        sorted).";

#define N       1       /* number of parameters for N-M */
#define ALPHA   1.0     /* constant for N-M */
#define BETA    0.5     /* " " */
#define GAMMA   2.0     /* " " */
#define EPSILON 1e-10   /* convergence limit for N-R */
#define MAXSST  200     /* max no of N-M steps before stop */
#define MAXRST  15      /* step limit for N-R */
#define SST     20      /* step limit for N-M */
#define K_95    1.96    /* normal variate for 95% conf. int. */
```

```
typedef double vector [N];

char   *fn;             /* failure data file name */
int    model;           /* class/family of model */
float  t[10000];        /* observed times read from data file */
double b2;              /* third parameter of infl. S-sh. model */
int    gamma;           /* third parameter of gamma model */
double me, te, sti, mete, metete, sx1, sx2, mete_g, fact_g;
double (*f)(), (*fp)(); /* MLE function and its derivative */
double (*b0_b1)(), (*x_b0)(), (*x_b1)(),
       (*b1_x)();       /* conversion functions */

#include "file.h"       /* file open and close routines */
#include <float.h>
#include <math.h>

double  error ()        /* for unimplemented functions */
{
  fprintf (stderr, "This option is not implemented.");
  exit (2);
}  /* error */

double  null (double x) { return  x; } /* do nothing function */

double  powi (x,i)      /* x**i */
double  x;
register  i;
{
  double p = 1.0;
  if (i<0) {
    i = -i;
    x = 1.0/x;
  }
  while (i-- > 0)
    p *= x;
  return  p;
}  /* powi */


/*****  M o d e l   f u n c t i o n s  *****/


/***** Binomial type models *****/

/*** Expb --- Exponential class (Binomial-type) model ***/
/* x == b0 (== u0) */
double  b0_b1_expb(double b0) { return me/(sti+b0*te-mete); }
#define  x_b1_expb      b0_b1_expb
double   b1_x_expb(double b1) { return me/b1/te + me - sti/te; }

sx_expb (x)
double  x;
{
  register  i, lv = (int)(me+0.5);
  double  d;
```

```
  sx1 = sx2 = 0.0;
  for (i = 0; i < lv; i++)  {
    d = 1.0 / (x-i);
    sx1 += d;
    sx2 += d*d;
  }  /* for */
}  /* sx_expb */

double  f_expb (x)      /* MLE function */
double  x;
{
  sx_expb(x);
  return  - mete/(sti+te*x-mete) + sx1;
}  /* f_expb */

double  fp_expb (x)     /* derivative of MLE function */
double  x;
{
  double  d = sti + te*x - mete;
  sx_expb(x);
  return  metete/d/d - sx2;
}  /* fp_expb */


/***** Poisson type models *****/

/*** Expp --- Exponential class (Poisson-type) model ***/
/* x == b1 */
double  x_b0_expp(double x) { return me/(1.0-exp(-x*te)); }

double  f_expp (x)      /* MLE function */
double  x;
{
  return  me/x - mete/(exp(x*te)-1.0) - sti;
}  /* f_expp */

double  fp_expp (x)     /* derivative of MLE function */
double  x;
{
  double  d = exp(x*te);
  return  - me/x/x + metete*d/(d-1.0)/(d-1.0);
}  /* fp_expp */


/*** Geo --- Geometric family model ***/
/* x == b1 */
double   x_b0_geo(double x)  { return me/log(1.0+x*te); }

sx_geo (x)
double  x;
{
  register  i, lv = (int)(me+0.5);
  double  d;

  sx1 = sx2 = 0.0;
  for (i = 0; i < lv; i++)  {
    d = 1.0 / (1.0 + x*t[i]);
```

```
    sx1 += d;
    sx2 += t[i]*d*d;
  }  /* for */
}  /* sx_geo */

double  f_geo (x)       /* MLE function */
double  x;
{
  double  d = 1.0 + x*te;
  sx_geo(x);
  return  sx1/x - mete/d/log(d);
}  /* f_geo */

double  fp_geo (x)      /* derivative of MLE function */
double  x;
{
  double  d = 1.0 + x*te;
  double  ld = log(d);

  sx_geo(x);
  return  - (sx1/x + sx2)/x + metete*(1.0+ld)/d/d/ld/ld;
}  /* fp_geo */


/*** Gamma --- Gamma class model ***/
/* x == b1 */
sx_gamma (x)
double  x;
{
  register i=1;
  double   d1=1.0, d2=x*te;

  sx2 = 0.0;
  while (i<gamma) {
    sx2 += d1;
    d1 *= d2/i++;
  }
  sx1 = sx2+d1;
}  /* sx_gamma */

double x_b0_gamma(double x) { sx_gamma(x);
  return  me/(1.0-exp(-x*te)*sx1); }

double  f_gamma (x)     /* MLE function */
double  x;
{
  sx_gamma(x);
  return  gamma*me/x - sti
    - mete_g*powi(x,gamma-1)/fact_g/(exp(x*te)-sx1);
}  /* f_gamma */

double  fp_gamma (x)    /* derivative of MLE function */
double  x;
{
  double  d1 = exp(x*te);
  double  d2 = d1 - sx1;
```

```
  sx_gamma(x);
  return  - gamma*me/x/x + mete_g*powi(x,gamma-2)
    *(x*te*(d1-sx2)-(gamma-1)*d2)/fact_g/d2/d2;
}  /* fp_gamma */


/*** Gexp --- General exponential (inflection S-shaped curve)
                class model ***/
/* x == b1 */
double  x_b0_gexp(double x) { double d = exp(-x*te);
  return  me*(1.0+b2*d)/(1.0-d); }

sx_gexp (x)
double  x;
{
  register  i, lv = (int)(me+0.5);
  double  d1, d2;

  sx1 = sx2 = 0.0;
  for (i = 0; i < lv; i++)  {
    d1 = b2*exp(-x*t[i]);
    d2 = 1.0/(1.0 + d1);
    sx1 += t[i]*(1.0-d1)*d2;
    sx2 += t[i]*t[i]*d1*d2*d2;
  }  /* for */
}  /* sx_gexp */

double  f_gexp (x)      /* MLE function */
double  x;
{
  double  d = exp(x*te);
  sx_gexp(x);
  return  me/x - mete*(1.0+b2)/(d-1.0+b2*(1.0-1.0/d)) - sx1;
}  /* f_gexp */

double  fp_gexp (x)     /* derivative of MLE function */
double  x;
{
  double  d1 = exp(x*te);
  double  d2 = d1 - 1.0 + b2*(1.0-1.0/d1);

  sx_gexp(x);
  return  - me/x/x + metete*(1.0+b2)*(b2/d1+d1)/d2/d2 - 2*sx2;
}  /* fp_gexp */


/*****  E n d   o f   m o d e l   f u n c t i o n s  *****/


double  f2 (vp)         /* function to be minimized by N-M */
vector  vp;
{
  double  d = f(vp[0]);
  return  d*d;          /* use the square of the MLE function */
}  /* f2 */

getinp ()               /* failure time data from file */
```

```
{
  FILE      *df;
  double    ti, ts, trs, tre;
  int       sf=1, ef=1;
  register  ime;

  df = fop (fn, "r", NULL);
  fprintf (stderr, "\n\nEnter time range (start, end)\n");
  /* may be used to skip data outside a "time window" */
  scanf ("%lf%lf", &trs, &tre);
  fprintf (stderr, "\nLoading data.  Please wait...\r");
  fscanf (df, "%lf", &ti);
  if (ti<=0.0) {        /* try to see if start and...*/
    te = -ti;           /* end test time is given    */
    ef = 0;
    fscanf (df, "%lf", &ti);
    if (ti<=0.0) {
      ts = te;
      sf = 0;
      te = -ti;
      fscanf (df, "%lf", &ti);
    } else  ts = tre;
  } else {              /* otherwise use the first...*/
    ts = tre;           /* and last failure time     */
    te = trs;
  }  /* else */
  sti = 0.0;
  ime = 0;              /* failure count */
  do {
    if (trs<=ti && ti<=tre) {
      if (sf && ts>ti)  ts = ti;
      if (ef && te<ti)  te = ti;
      t [ime] = ti;
      sti += ti;
      if (++ime >= sizeof(t)/sizeof(*t))
        break;
    }  /* if */
  } while (fscanf(df, "%lf", &ti)==1);
  me = ime;
  fprintf (stderr, "\r\t\t\t\t\t\t\t\t\t\r");
  if (!feof(df)) {
    fprintf (stderr, "The whole file was not read!\a");
    exit(3);
  }
  fcl (df, NULL);
  fprintf (stderr, "\rt        = (%g, %g)  ->  (0, ", ts, te);
  te -= ts;
  sti -= me*ts;
  mete = me*te;
  metete = mete*te;
  while (--ime>=0)      /* adjust times */
    t [ime] -= ts;
  fprintf (stderr, "%g)\n", te);
  fprintf (stderr, "me       = %.15g\n", me);
  fprintf (stderr, "sti      = %.15g\n", sti);
  fprintf (stderr, "me*te    = %.15g\n", mete);
  fprintf (stderr, "me*te*te = %.15g\n", metete);
```

```
  fprintf (stderr, "U        = %.3f\n",
        (sti-me/2.0*te)/te/sqrt(me/12.0) );
}  /* getinp */

initialize (p, y)       /* initial estimation */
vector  p[N+1];
double  y[N+1];
{
  register  i, j;
  double  x;

  getinp ();
  if (model==3) {       /* if infl. S-shaped model */
    fprintf (stderr,
      "\nEnter third parameter of model\n\t\t\tb2 = ");
    scanf ("%lf", &b2);
  } else if (model==2) {        /* if gamma model */
    fprintf (stderr,
      "\nEnter third parameter of model\n\t\t\tgamma = ");
    scanf ("%i", &gamma);
    if (gamma<1) {
      fprintf (stderr, "Gamma must be a positive integer!\a");
      exit (4);
    }
    mete_g = me*powi(te,gamma);
    fact_g = i = 1;
    while (i<gamma)
      fact_g *= i++;
  }
  /* p[0] : initial estimation from b1 = - sign (U) / te */
  i = (me*te*0.5<sti) ? -1 : +1;
  if (model==3 || (model==2 && gamma>1))
    i = +1;
  x = p[0][0] = b1_x((double)(i)/te);
  fprintf (stderr, "Initial estimation is\tx  = %g\n\
This means\t\tb0 = %g\n\t\t\tb1 = %g\n\
\nYou can enter another initial estimation if you want.\n\
\t\t\tx  = ", x, x_b0(x), x_b1(x));
  scanf ("%lf", &p[0][0]);      /* note: no error check */
  y[0] = f2(p[0]);
  for (i=1; i<N+1; i++) {       /* initialize p[i] and y[i] */
    for (j=0; j<N; j++)
      if (i!=j+1)
        p[i][j] = p[0][j];
      else
        p[i][j] = (p[0][j]==0.0) ? 1.0 : p[0][j]*1.1;
    y[i] = f2(p[i]);
  }  /* for */
}  /* initialize */

assign (d, s)           /* vector assignment */
vector  d, s;
{
  register  j;
  for (j=0; j<N; j++)
    d[j] = s[j];
}  /* assign */
```

```
eval (p, q, r, w)       /* weighted vector addition */
vector  p, q, r;
double  w;
{
  register  j;
  for (j=0; j<N; j++)
    p[j] = (1.0-w)*q[j] + w*r[j];
}  /* eval */

results (p, y, l)       /* intermediate results from N-M */
vector  p[N+1];
double  y[N+1];
{
  register  i, j;
  for (i=0; i<N+1; i++) {
    fprintf (stderr, "%c f2 (%.15g", (i==l)?'*':' ', p[i][0]);
    for (j=1; j<N; j++)
      fprintf (stderr, ", %.15g", p[i][j]);
    fprintf (stderr, ") = %.15g\n", y[i]);
  } /* for */
}  /* results */

lohish (l, h, s, y)     /* low, high, and second high in y */
int  *l, *h, *s;
double  y[N+1];
{
  register  i, min, max, max2;
  /* determine min, max, and max2 of first two elements */
  min = max2 = !(max = y[0]<y[1]);
  for (i=2; i<N+1; i++)
    if (y[i] < y[min])
      min = i;
    else if (y[i] > y[max])
      { max2 = max;  max = i; }
    else if (y[i] > y[max2])
      max2 = i;
  *l = min;
  *h = max;
  *s = max2;
}  /* lohish */


main (argc, argv)       /* Maximum Likelihood Estimation */
char  **argv;
{
  vector  p[N+1],       /* vertices of simplex */
        pp,
        ppp,
        pc;             /* centroid */
  double  y[N+1],       /* f2(p[i]) */
        yy,             /* f2(pp) */
        yyy,            /* f2(ppp) */
        dif, x;
  int   btype,          /* model type; 0:Poisson, 1:Binomial */
        i, j, k,
        l /* low */, h /* high */, s /* second high */,
```

```
        ssc = 0,        /* step count for N-M */
        rsc;            /* step count for N-R */

/* --- Constant Tables --- */
  static char *et[]
    = {
  "0 : exponential  mu(t) = b0*(1-exp(-b1*t))\
\n\t\t lambda(t) = b0*b1*exp(-b1*t)",
  "1 : geometric    mu(t) = b0*ln(1+b1*t))\
\n\t\t lambda(t) = b0*b1/(1+b1*t)",
  "2 : gamma (S-sh) \
mu(t) = b0*(1-S(i=0,gamma-1,(b1*t)**i/i!)*exp(-b1*t))\
\n\t\t lambda(t) = b0*b1**gamma*t**(gamma-1)*exp(-b1*t)/(gamma-1)!",
  "3 : infl. S-sh.  mu(t) = b0*(1-exp(-b1*t))/(1+b2*exp(-b1*t))\
\n\t\t lambda(t) = b0*b1*(1+b2)*exp(-b1*t)/(1+b2*exp(-b1*t))**2",
  };
#define MNO     (sizeof(et)/sizeof(*et))

  static double  (*ft[2][MNO])() = {    /* MLE functions */
    {     f_expp,      f_geo,    f_gamma,     f_gexp  },
    {     f_expb,      error,      error,      error  }
  };

  static double  (*fpt[2][MNO])() = {   /* derivatives */
    {    fp_expp,     fp_geo,   fp_gamma,    fp_gexp  },
    {    fp_expb,      error,      error,      error  }
  };

  static double  (*b0_b1t[2][MNO])() = {
    {      error,      error,      error,      error  },
    { b0_b1_expb,      error,      error,      error  }
  };

  static double  (*x_b0t[2][MNO])() = {
    {  x_b0_expp,   x_b0_geo, x_b0_gamma,  x_b0_gexp  },
    {       null,      error,      error,      error  }
  };

  static double  (*x_b1t[2][MNO])() = {
    {       null,       null,       null,       null  },
    {  x_b1_expb,      error,      error,      error  }
  };

  static double  (*b1_xt[2][MNO])() = {
    {       null,       null,       null,       null  },
    {  b1_x_expb,      error,      error,      error  }
  };

  if (argc!=2) {
    fprintf (stderr, usage);
    exit (1);
  }
  fprintf (stderr, "\n--- DATA FILE NAME: %s\n", fn=argv[1]);
  do {
    fprintf (stderr, "\nChoose a model\n\n");
    for (model=0; model<MNO; model++)
      fprintf (stderr, "%s\n", et[model]);
```

```
  fprintf (stderr,
      "\nwhere t is (t-ts)\n\n\tYour choice (0-%d): ", MNO-1);
  scanf ("%d", &model);
} while (model < 0 || model >= MNO);
do {
  if ((btype=getchar())=='\n')
    fprintf (stderr, "\nType of model (Binomial/Poisson): ");
  btype = toupper (btype);
} while (btype != 'B' && btype != 'P');
btype = btype == 'B';          /* 0: Poisson, 1:Binomial */
f = ft [btype][model];         /* MLE function */
fp = fpt [btype][model];       /* its derivative */
b0_b1 = b0_b1t [btype][model];/* conversion function */
x_b0 = x_b0t [btype][model];  /* " " */
x_b1 = x_b1t [btype][model];  /* " " */
b1_x = b1_xt [btype][model];  /* " " */
initialize (p, y);
lohish (&l, &h, &s, y);
results (p, y, l);

/* main loop */
do {
/* Nelder and Method method */
  fprintf (stderr, "Simplex\n");
  while (++ssc%SST != 0) {
    lohish (&l, &h, &s, y);   /* determine l, h, and s */

    for (j=0; j<N; j++) {     /* calculate centroid...*/
      pc[j] = 0;              /* of pts with i!=h     */
      for (i=0; i<N+1; i++)
        if (i!=h)
          pc[j] += p[i][j];
      pc[j] /= N;             /* / (N+1-1) */
    }  /* for */

    fprintf(stderr, "R");     /* reflection */
    eval (pp, pc, p[h], -ALPHA);
    yy = f2(pp);

    if (y[l]<=yy && yy<=y[s]) {  /* case 1 */
      assign (p[h], pp);
      y[h] = f2(pp);
      continue;               /* end of case 1 */
    }  /* if */

    if (yy<y[l]) {            /* case 2 : expansion */
      fprintf(stderr, "E");
      eval (ppp, pc, pp, GAMMA);
      yyy = f2(ppp);
      assign (p[h], (yyy<y[l]) ? ppp : pp);
      y[h] = f2(p[h]);
      continue;               /* end of case 2 */
    }  /* if */

    fprintf(stderr, "C");     /* case 3 : contraction */
    eval (ppp, pc, (yy<y[h]) ? pp : p[h], BETA);
    yyy = f2(ppp);
```

```
      if (yyy>=y[h] || yyy>=yy) {
        for (i=0; i<N+1; i++)
          if (i!=l) {
            eval (p[i], p[i], p[l], 0.5); /* take mean */
            y[i] = f2(p[i]);
          }  /* if */
      } else {
        assign (p[h], ppp);
        y[h] = f2(ppp);
      }  /* else */
      continue;                 /* end of case 3 */
    }  /* while */

    lohish (&l, &h, &s, y);
    fprintf (stderr, "\n");
    results (p, y, l);
  /* Newton-Raphson procedure */
    x = p[l][0];
    fprintf (stderr, "Newton-Raphson\n");
    for (rsc=0; rsc<MAXRST; rsc++) {
      dif = f(x)/fp(x);
      x = x - dif;
      fprintf (stderr, "%25.20g %25.20g\n", x, dif);
    } /* for */
    if (dif<0)
      dif = -dif;
  } while (dif>EPSILON && ssc<MAXSST);
  if (ssc >= MAXSST) {
    fprintf (stderr,
      "Convergence not obtained. (Error: %g)\a", dif);
    exit (5);
  }  /* if */

  {  /* printing final results */
#define b1_b0(b1)  x_b0(b1_x(b1))
    double  b0 = x_b0(x), b1 = x_b1(x); /* estim. parameters */
    double  db1 = K_95/sqrt(-fp(x));

    fprintf (stderr, "b0    = %.3f\n", b0);
    fprintf (stderr, "b1    = %.4e\n", b1);
    if (btype)          /* if Binomial type */
      fprintf (stderr, "b1_r  = %.4e\n", b0_b1((int)(b0+0.5)));
    if (model==3)       /* if infl. S-sh. model */
      fprintf (stderr, "b2    = %g\n", b2);
    else if (model==2)  /* if gamma model */
      fprintf (stderr, "gamma = %d\n", gamma);
    printf ("%.3f\t(%.3f\t%.3f)\t", b0,
      b1_b0(b1+db1), b1_b0(b1-db1));
    printf ("%.4e\t(%.4e\t%.4e)\n", b1,
      b1-db1, b1+db1);
  }  /* results */
}  /* main */
```

```
/** file.h - File open and close functions.
 */

#include <stdio.h>

/* fop: Open file 'fn' with mode 'fm' using as buffer 'buf'.
 *      If BUFSIZE is not defined or buf==NULL, the file
 *      will not be buffered. If fm[0]=='w', an error will be
 *      given if the file already exists.
 */
FILE *fop (fn, fm, buf)
char *fn, *fm, *buf;
{
  FILE *fp;

  if (fm[0]=='w' && access(fn,0)==0) {
    fprintf (stderr, "File %s already exists.\n", fn);
    exit (255);
  }
  if ((fp=fopen(fn,fm))==NULL)
    fprintf (stderr, "Cannot open file %s with mode %s.\n",
        fn, fm);
#ifdef BUFSIZE
  else if (buf==NULL)
    return fp;
  else if (setvbuf (fp, buf, _IOFBF, BUFSIZE))
    fprintf (stderr, "Cannot buffer file %s.\n", fn);
#endif
  else
    return fp;
  exit (255);
}  /* fop */


/* fcl: Close file stream 'fp' which is using buffer 'buf'.
 *      If BUFSIZE is not defined or buf==NULL no attempt
 *      will be made to unbuffer the file.
 */
void fcl (fp, buf)
FILE *fp;
char *buf;
{
#ifdef BUFSIZE
  if (buf!=NULL && setvbuf(fp,NULL,_IONBF,0)) {
    fprintf (stderr, "Cannot unbuffer file.\n");
    exit (254);
  }
#endif
  if (fclose (fp)) {
    fprintf (stderr, "Cannot close file.\n");
    exit (254);
  }
}  /* fcl */
```

APPENDIX B. INDEX

BIBLIOGRAPHY

1. John D. Musa, Anthony Iannino, and Kazuhira Okumoto. *Software Reliability*: *Measurement, Prediction, Application*. Whippany, NJ: McGraw-Hill, 1987.

2. Ian Sommerville. *Software Engineering*. Wokingham, England: Addison-Wesley, 1985.

3. Barry W. Boehm, J. R. Brown, H. Kaspar, Myron Lipow, G. Macleod, and M. Merrit. *Characteristics of Software Quality*. TRW Series of Software Technology, Amsterdam: North-Holland, 1978.

4. Siba N. Mohanty, "Models and measurements for quality assessment of software," *ACM Computing Surveys*, Vol. 11, No. 3, pp. 251-275, September 1979.

5. Barry W. Boehm, "Software engineering economics," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 1, pp. 4-21, January 1984.

6. Barry W. Boehm. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

7. Siba N. Mohanty, "Software cost estimation: Present and future," *Software-Practice and Experience*, Vol. 11, No. 2, pp. 103-121, February 1981.

8. Amrit L. Goel, "Software reliability models: Assumptions, limitations, and applicability," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 12, pp. 1411-1423, December 1985.

9. Tom Anderson and Pete A. Lee. *Fault Tolerance, Principles and Practice*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

10. John D. Musa, "Validity of execution-time theory of software reliability," *IEEE Transactions on Reliability*, Vol. R-28, No. 3, pp. 181-191, August 1979.

11. Bev Littlewood, “How to measure software reliability and how not to,” *IEEE Transactions on Reliability*, Vol. R-28, No. 2, pp. 103-110, June 1979.

12. Bev Littlewood and John L. Verrall, “A Bayesian reliability growth model for computer software,” *Applied Statistics* (J. Roy. Statist. Soc., Series C), Vol. 22, No. 3, pp. 332-346, 1973.

13. Bev Littlewood and John L. Verrall, “A Bayesian reliability model with a stochastically monotone failure rate,” *IEEE Transactions on Reliability*, Vol. R-23, No. 2, pp. 108-114, June 1974.

14. Zygmunt Jelinski and Paul B. Moranda, “Software reliability research,” (W. Freiberger, Editor), *Statistical Computer Performance Evaluation*, NY: Academic, pp. 465-484, 1972.

15. Amrit L. Goel and Farokh B. Bastani, “Foreword: Software reliability,” *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 12, pp. 1409-1410, December 1985.

16. Tze-Jie Yu, Vincent Y. Shen, and Hubert E. Dunsmore, “An analysis of several software defect models,” *IEEE Transactions on Software Engineering*, Vol. 14, No. 9, pp. 1261-1270, September 1988.

17. John D. Musa, “A theory of software reliability and its application,” *IEEE Transactions on Software Reliability*, Vol. SE-1, No. 3, pp. 312-327, September 1975.

18. Amrit L. Goel and Kazu Okumoto, “Time-dependent error-detection rate model for software reliability and other performance measures,” *IEEE Transactions on Reliability*, Vol. R-28, No. 3, pp. 206-211, August 1979.

19. P. A. Keiller, Bev Littlewood, Douglas R. Miller, and A. Sofer, “On the quality of software reliability prediction,” (J. K. Skwirzynski, Editor), *Electronic Systems Effectiveness and Life Cycle Costing*, NATO ASI Series, F3, Springer-Verlag, Heidelberg, pp. 441-460, 1983.

20. Shigeru Yamada, Mitsuru Ohba, and Shunji Osaki, “S-shaped reliability growth modeling for software error detection,” *IEEE Transactions on Reliability*, Vol. R-32, No. 5, pp. 475-478, 484, December 1983.

21. John D. Musa and Kazuhira Okumoto, “A logarithmic Poisson execution time model for software reliability measurement,” *Proceedings Seventh International Conference on Software Engineering*, Orlando, pp. 230-238, 1984.

22. John D. Musa, “Tools for measuring software reliability,” *IEEE Spectrum*, Vol. 26, No. 2, pp. 39-42, February 1989.

23. Shigeru Yamada, Mitsuru Ohba, and Shunji Osaki, “S-shaped software reliability growth models and their applications,” *IEEE Transactions on Reliability*, Vol. R-33, pp. 289-292, October 1984.

24. Shigeru Yamada and Shunji Osaki, “Software reliability growth modeling: Models and applications,” *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 12, pp. 1431-1437, December 1985.

25. Mitsuru Ohba, “Software reliability analysis models,” *IBM Journal Research and Development*, Vol. 28, No. 4, pp. 428-443, July 1984.

26. Kazuhira Okumoto, “A statistical method for software quality control,” *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 12, pp. 1424-1430, December 1985.

27. A. E. Green and A. J. Bourne. *Reliability Technology*. NY: John Wiley & Sons, 1972.

28. Nancy R. Mann, Ray E. Schafer, and Nozer D. Singpurwalla. *Methods for Statistical Analysis of Reliability and Life Data*. NY: John Wiley & Sons, 1974.

29. J. A. Nelder and R. Mead, “A simplex method for function minimization,” *Computer Journal*, Vol. 7, No. 4, pp. 308-313, January 1965.

30. Gordon R. Walsh. *Methods of Optimization*. NY: John Wiley & Sons, 1975.

REFERENCES NOT CITED

Anderson, David R., Dennis J. Sweeney, and Thomas A. Williams. *Introduction to Statistics*. St. Paul, MN: West Publishing Co., 1981.

Currit, P. Allen, Michael Dyer, and Harlan D. Mills, "Certifying the reliability of software," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 1, pp. 3-11, January 1986.

Dingiloğlu, Handan, Engin Sungur, and İsmail Erdem, "Stochastic modelling and statistical inferences on computer software reliability," in *Second International Symposium on Computer and Information Sciences*, Boğaziçi University, Istanbul, pp. 532-548, October 1987.

Ehrlich, Willa K. and Thomas J. Emerson, "Modeling software failures and reliability growth during system testing," *Proceedings Ninth International Conference on Software Engineering*, Monterey, CA, pp. 72-77, March 1987.

Gaffney, John E., Jr., "The impact on software development costs of using HOL's," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 3, pp. 496-499, March 1986.

Halstead, M. H., "Forum: On lines of code and programming productivity," *IBM Systems Journal*, Vol. 16, No. 4, pp. 421-422, 1977.

Jewell, William S., "Bayesian extensions to a basic model of software reliability," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 12, pp. 1465-1471, December 1985.

Leveson, Nancy G., "Software safety: What, why, and how," *ACM Computing Surveys*, Vol. 18, No. 2, pp. 125-163, June 1986.

Littlewood, Bev and John L. Verrall, "Likelihood function of a debugging model for computer software reliability," *IEEE Transactions on Reliability*, Vol. R-30, No. 2, pp. 145-148, June 1981.

Mazzuchi, Thomas A. and Nozer D. Singpurwalla, "Some Bayesian approaches for estimating the failure rate," *ca*. 1981.

Ross, Sheldon M., "Software reliability: The stopping rule problem," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 12, pp. 1472-1476, December 1985.

Scholz, F. - W., "Software reliability modeling and analysis," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 1, pp. 25-31, January 1986.

Singpurwalla, Nozer D. and Refik Soyer, "Assessing (software) reliability growth using a random coefficient autoregressive process and its ramifications," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 12, pp. 1456-1464, December 1985.

Troy, Robert and Ramadan Moawad, "Assessment of software reliability models," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 9, pp. 839-849, September 1985.

Walston, Claude E. and Charles P. Felix, "A method of programming measurement and estimation," *IBM Systems Journal*, Vol. 16, No. 1, pp. 54-73, 1977.