Appendices

Appendix A

Software Reliability Growth Models

A.1 Data Domain Models

These models are described in section 2.7.1. The classification of these models is follows.

A.1.1 Fault-seeding models

In these models, the bugs are seeded into the program source code and then they are identified by the programmer. The description of these models is given in section 2.7.1.1. The classification of these models is follows.

A.1.1.1 Mills's Hypergeometric Model

The hypergeometric distribution was applied to the field of software validation as early 1970 by Mills in an unpublished paper. However, his setup was a capture-recapture sampling, either in the form of error seeding or in the form of a two-stage edit procedure [98]. Tohma et al. later introduced an extended model for n-stage debugging [99]. In both models, program faults are drawn from the same population at each stage, which implies that the parts of the AUT already covered before are possibly tested again.

Building on earlier work [100, 101], Rivers and Vouk [102], derive a hypergeometric density for the number of failures experienced at each stage of testing when the constructs tested are removed from the population, i.e. when there is no retest of constructs covered

before. The constructs under study may be program size metrics like statements, basic blocks, paths, c-uses, p-uses, etc. Rivers and Vouk also consider the number of test cases executed and even execution time or calendar time. The probability that the random variable ΔM_i - the number of faults detected at the i th stage of testing - takes value Δm_i, given that m_{i-1} faults have been discovered at the first $(i-1)$ stages, is given by equation a.1.

$$P(\Delta M_i = \Delta m_i | m_{i-1}) = \frac{\binom{g_i(N-m_{i-1})}{\Delta m_i}\binom{[K-Q_{i-1}]-[g_i(N-m_{i-1})]}{\Delta Q_i - \Delta m_i}}{\binom{K-Q_{i-1}}{\Delta Q_i}} \tag{A.1}$$

for $\Delta m_i \in N$and $\Delta m_i \leq \Delta Q_i$.

In this equation, N stands for the number of faults inherent in the software at the beginning of testing, and K is the number of constructs that are planned to be covered until the end of testing. Q_{i-1}constructs already have been covered at the first $(i-1)$ stages of testing, and ΔQ_i is the number of constructs covered at the i_{th} stage. $g_i = g(\frac{Q_i}{K})$ is a factor accounting for the "visibility" of the $N - m_{i-1}$ faults still remaining in the code. On the one hand, it is influenced by the structure of the code and by the constructs chosen: For example, the execution of several different paths may lead to detection of the same fault, whereas other faults may only be activated if a path is taken more often than just once. On the other hand, g_i may also account for fluctuations in the testers' effectiveness, e.g. due to learning. Therefore, g_i is referred to as "testing efficiency". According to the equation, the probability of finding Δm_i faults is the product of the number of ways in which these faults can be chosen out of $g_i(N - m_{i-1})$ remaining visible faults and the number of ways in which the other $\Delta Q_i - \Delta m_i$ constructs covered can be chosen out of the $[K - Q_{i-1}] - [g_i(N - m_{i-1})]$ constructs that neither have been covered before nor are faulty; this value is divided by the number of ways in which the ΔQ_i constructs can be picked out of the $K - Q_{i-1}$ uncovered statements. The expected number of faults discovered at stage i given that m_{i-1} faults had been detected before is given by equation a.2 [100].

$$E(\Delta M_i|m_{i-1}) = g_i(N - m_{i-1})\frac{\Delta Q_i}{K - Q_{i-1}} = g_i(N - m_{i-1})\frac{\Delta q_i}{1 - q_{i-1}} \tag{A.2}$$

where $\Delta q_i = \frac{\Delta Q_i}{K}$ is the fraction of constructs covered at stage i and $q_{i-1} = \frac{Q_{i-1}}{K}$ is the fraction of constructs covered before that stage. The unconditional expected value of ΔM_i is given by equation a.3.

$$E(\Delta M_i) = g_i[N - E(M_{i-1})]\frac{\Delta q_i}{1 - q_{i-1}} \tag{A.3}$$

A.1.2 Input-domain models

These models are described in section 2.7.1.2. The classification of these models is described as under.

A.1.2.1 Nelson Input Domain Model

The reliability is estimated by running the software for a set of n inputs [40]. The input are chosen randomly from a set $E_i : i = 1, 2, \ldots, N$. Each E_i is a set of data values needed to make a run. The random sampling of n inputs is done according to the probability vector P_i, where P_i is the probability that E_i is sampled. The probability vector $\{P_i : i = 1, 2, \ldots, N\}$ defines the operational profile or the user input distribution. If the number of failures is f, then the estimate of the reliability is given by equation a.4:

$$\hat{R} = 1 - \frac{f}{n} \tag{A.4}$$

A.1.2.2 Brown and Lipow Input Domain Model

The Nelson model explicitly incorporates the usage distribution or the test case distribution. In the model proposed by Brown and Lipow [101], it is assumed that the accomplished testing is representative of the expected user distribution. In this model the entire input domain is partitioned into subdomains. Each E_i from the input domain $\{E_i : i = 1, 2, \ldots, N\}$ represents a specific subdomain. The estimated reliability in this

case is given by equation a.5:

$$\hat{R} = 1 - \sum_{i=1}^{N} \binom{f_j}{E_j} P(E_j) \tag{A.5}$$

Here n_j is the number of test cases sampled from subdomain E_j, f_j is the number of test cases which resulted in an abnormal execution out of n_j test cases, and $P(E_j)$ is the probability that inputs in domain E_j are used in an actual operational environment.

A.1.2.3 Ramamoorthy and Bastani Input Domain Model

This input domain model is concerned with the reliability of critical, real time process control systems. Thus, the number of failures detected in the reliability estimation phase should be zero, leading to the reliability estimate of one. The metric here is the confidence in the reliability estimate rather than the estimate itself; this model provides a conditional probability that the program is correct for all possible set of inputs given that it results in successful execution for a specified set of inputs. It is assumed that Inputs are selected randomly and independently from the input domain according to some probability distribution (which can change with time), which is a relaxed assumption that other general growth models in which we usually assume that the inputs are selected randomly and independently from the input domain according to the operational distribution.

This means that the effective error size varies with time even though the program is not changed. This permits a straightforward modeling of the testing process. Let

j = number of failures experienced, k = number of runs since the i_{th} failure, $T_j(k)$ = testing process for the kth run after j failures, $V_j(k)$ = size of residual errors for the kth run after i failures; this can be random.

Now

$$\begin{aligned} P\{\text{success on kth run} \mid \text{j failures}\} &= 1 - V_j(k) \\ &= 1 - f(T_j(k))\lambda_j \end{aligned}$$

where

λ_j = error size under operational inputs; this can be a random variable; $0 \leq \lambda_j \leq 1$;

$f(T_j(k))$ = severity of testing process relative to operational distribution; $0 \leq f(T_j(k)) \leq \frac{1}{\lambda_j}$

Hence

$$\begin{aligned} R_j(k|\lambda_j) &= P\{\text{no failure over k runs}|\lambda_j\} \\ &= \prod_{i=1}^{k} P\{\text{no failure over } i^{th} \text{ run}|\lambda_j\} \end{aligned}$$

since successive test cases have independent failure probability.

Hence

$$R_j(k|\lambda_j) = \prod_{i=1}^{k}[1 - f(T_j(i))\lambda_j] \tag{A.6}$$

i.e.

$$R_j(k) = E_{\lambda_j}[\prod_{i=1}^{k}[1 - f(T_j(i))\lambda_j]] \tag{A.7}$$

where $E_{\lambda_j}[.]$ is the expectation over λ_j.

For many types of software, e.g., operating systems and real-time process control systems, it is difficult to identify "runs" or there may be a very large number of such runs if we assume that each cycle constitutes a run. In these cases, it is simpler to work in continuous time. The above relation becomes

$$R_j(t) = E_{\lambda_j}[\mathrm{e}^{-\lambda_j \int_0^t f(T_j(s))ds}] \tag{A.8}$$

where

λ_j = failure rate after jth failure; $0 \leq \lambda_j \leq \infty$

$T_j(s)$=testing process at time s after jth failure

$f(T_j(s))$=severity of testing process relative to operational distribution; $0 \leq f(T_j(s)) \leq \infty$

Remarks

1. As we have noted above, $f(T_j(.))$ is the severity of the testing process relative to the operational distribution, where the testing severity is the ratio of the probability that a run based on the test case selection strategy detects an error to the probability that a failure occurs on a run selected according to the operational distribution. Obviously, during the operational phase, $f(T_j(.)) = 1$. Thus, a testing strategy which simulates the operational input would be ideal from a reliability modeling point of view since it would simplify the model considerably. But this conflicts with our need to improve the software by detecting as many errors as possible by employing non random testing strategies such as path testing, function testing, and boundary value testing. In general, it is difficult to determine the severity of these test cases and most models assume that $f(T_j(.)) = 1$. However, for some testing strategies we can attempt to quantify $f(T_j(.))$. For example, in function testing the severity increases as we switch to untested functions since these are more likely to contain errors than functions which have already been tested.

2. In the continuous case, the time is the CPU time.

3. The software reliability models discussed below can be applied (in principle) to any type of software. However, their validity increases as the size of the software and the number of programmers involved increases.

A.2 Time-domain models

In these models, the underlying failure process of the software is modeled and the observed failure history is used to estimate the residual number of faults in the software. The classification of this model is given below.

A.2.1 Homogeneous Markov Models

The models belonging to this category assume that initial number of faults in a software product under consideration is unknown but fixed. The number of faults in the system at any time, for the state space of a homogenous Markov chain. The failure intensity of the software or transition rates of the Markov chain depend upon the residual faults in the software. The popular models in this class are: Jelinski-Moranda[2], Goel-Okumoto Imperfect Debugging model[15] etc.

A.2.2 Non-Homogeneous Markov Models

These models assume the number of faults present in a software product to be a random variable most often assumed to display the behavior of a Non-Homogeneous Poisson Process (NHPP). The popular models include GO model[15], Delayed S-shaped[103], etc. This class of models [42] is concerned with the number of faults detected in a given time and hence are referred to as "fault count models". The most popular subclass assumes that the number of failure in the given interval has a Poisson distribution whose parameters take different forms for different models. The mean value function m(t), and the failure intensity $\lambda(t)$, of the software system satisfy the following relations:

$$m(t) = \int_0^t \lambda(s)ds \tag{A.9}$$

$$\frac{dm(t)}{dt} = \lambda(t) \tag{A.10}$$

A non-homogenous Markov chain for a general failure intensity function, $\lambda(t)$ is as shown in the figure a.1.

The NHPP models can be further classified as finite failures and infinite failure models.

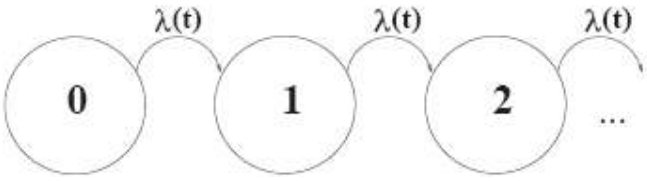

Figure A.1: Non-Homogenous Markov chain.

A.2.3 Semi-Markov Models

The concept of semi-Markov process can be seen in [22]. The number of faults remaining in the software in this case is modeled using a semi-Markov process. As in case of homogeneous Markov models, the number of faults remaining in the software products forms the state space of the Markov chain. However the transition rate from each state depend not only on the number of the remaining faults but also on the time spent in the current state. This class of models assumes that the initial number of faults in the software product is unknown but fixed, and the failure intensity of the software, or the transition rate from a given state depends not only on the residual faults in the software, also on the time elapsed in that state. Schick-Wolverton[104] etc. belongs to this class.

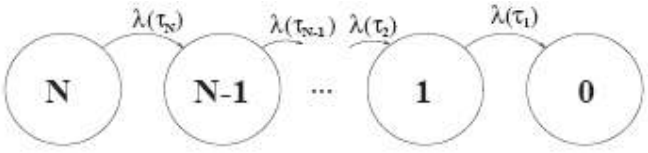

Figure A.2: Semi-Markov chain.

A representative model belonging to this class is discussed below. Figure a.2 shows the semi-Markov chain, where $\lambda(\tau_i)$ indicates the transition rate depends on τ_i, where τ_i is the sojourn time in state i.

A.2.4 Jelinski-Moranda de-eutrophication model

The de-eutrophication model developed by Jelinski and Moranda [2] in 1972 was one of the first software reliability growth models. It consists of some simple assumptions:

1. At the beginning of testing, there are u_0 faults in the software code with u0 being

an unknown but fixed number.

2. Each fault is equally dangerous with respect to the probability of its instantaneously causing a failure. Furthermore, the hazard rate of each fault does not change over time, but remains constant at φ.

3. The failures are not correlated, i.e. given u_0 and φ the times between failures $\Delta t_1, \Delta t_2, \ldots, \Delta t_{u_0}$ are independent.

4. Whenever a failure has occurred, the fault that caused it is removed instantaneously and without introducing any new fault into the software.

As a consequence of these assumptions, the program hazard rate after removal of the $(i-1)^{st}$ fault is proportional to the number of faults remaining in the software with the hazard rate of one fault, $za(t) = \varphi$, being the constant of proportionality:

$$z(\Delta t|t_{i-1}) = \phi[u_0 - M(t_{i-1})] = \phi[u_0 - (i-1)] \tag{A.11}$$

The Jelinski-Moranda model belongs to the binomial type of models as classified by Musa et al. [105]. For these models, the failure intensity function is the product of the inherent number of faults and the probability density of the time until activation of a single fault, $f_a(t)$, i.e.:

$$\frac{d\mu(t)}{dt} = u_0 f_a(t) = u_0 \phi \exp(-\phi t) \tag{A.12}$$

$$\mu(t) = u_0[1 - \exp(-\phi t)] \tag{A.13}$$

It can easily be seen from equations a.12 and a.13 that the failure intensity can also be expressed as

$$\frac{d\mu(t)}{dt} = \phi[u_0 - \mu(t)] \tag{A.14}$$

According to equation a.14 the failure intensity of the software at time t is proportional to the expected number of faults remaining in the software; again, the hazard rate of an individual fault is the constant of proportionality. This equation can be considered

the "heart" of the Jelinski-Moranda model. Moreover, many software reliability growth models can be expressed in a form corresponding to equation a.14. Their difference often lies in what is assumed about the per-fault hazard rate and how it is interpreted. The models in the following sections are presented with respect to this viewpoint.

A.2.5 Goel-Okumoto Imperfect Debugging Model

The model proposed by Goel and Okumoto in 1979 [106] is based on the following assumptions:

1. The number of failures experienced by time t follows a Poisson distribution with mean value function $\mu(t)$. This mean value function has the boundary conditions $\mu(0) = 0$ and $\lim_{t\to\infty} \mu(t) = N < \infty$.

2. The number of software failures that occur in $(t, t + \Delta t]$ with $\Delta t \to 0$ is proportional to the expected number of undetected faults, $N - \mu(t)$. The constant of proportionality is ϕ.

3. For any finite collection of times $t_1 < t_2 < \cdots < t_n$ the number of failures occurring in each of the disjoint intervals $(0, t_1), (t_1, t_2), \ldots, (t_{n-1}, t_n)$ is independent.

4. Whenever a failure has occurred, the fault that caused it is removed instantaneously and without introducing any new fault into the software.

Since each fault is perfectly repaired after it has caused a failure, the number of inherent faults in the software at the beginning of testing is equal to the number of failures that will have occurred after an infinite amount of testing. According to assumption 1, $M(\infty)$ follows a Poisson distribution with expected value N. Therefore, N is the expected number of initial software faults as compared to the fixed but unknown actual number of initial software faults u0 in the Jelinski-Moranda model. Indeed, this is the main difference between the two models. Assumption 2 states that the failure intensity at time t is given by equation a.15.

$$\frac{d\mu(t)}{dt} = \phi[N - \mu(t)] \tag{A.15}$$

Just like in the Jelinski-Moranda model the failure intensity is the product of the constant hazard rate of an individual fault and the number of expected faults remaining in the software. However, N itself is an expected value.

A.2.6 Schick and Wolverton Model

This model [104] is based on the same assumptions as Jelinski-Moranda Model. The transition rate λ_i during the test interval t_i, is assumed to be proportional to the current fault content of the system, and the time elapsed since the last failure and is given by equation a.16.

$$\lambda_i = \phi[N - (i-1)]t_i \tag{A.16}$$

where N and ϕ have the same interpretations as in the JM model. The hazard rate in this case is linear with time in failure interval, returns to zero at the occurrence of a failure, and increases linearly again, at a reduced slope. The decrease in slope is proportional to ϕ. A variation of the above model [98] assumes the transition rate to be parabolic in test time and is given by equation a.17.

$$\lambda_i = \phi[N - (i-1)](-at_i^2 + bt_i + c) \tag{A.17}$$

where a, b, c are constants and N and ϕ have the same interpretations as in the JM model. The transition rate in this case is the superposition of the transition rate of the JM model times a second terms which states that the likelihood of the failure occurring increases rapidly as the test time accumulates within the interval. Note that at the beginning of a testing interval, i.e. $(t_i = 0)$, the transition rate is proportional to that of the JM model, with the constant of proportionality being c.

A.2.7 Finite Failure NHPP Models

This section provides an overview of the finite failure non-homogeneous Poisson process software reliability models. This class of models is concerned with the number of faults detected in a given time and hence are also referred to as "fault count models" [106].

Software failures are assumed to display the behavior of a non-homogeneous Poisson process (NHPP), the stochastic process $N(t), t \geq 0$, with a rate parameter $\lambda(t)$ that is time-dependent. The function $\lambda(t)$ denotes the instantaneous failure intensity. Given $\lambda(t)$, the mean value function $m(t) = E[N(t)]$, where m(t) denotes the expected number of faults detected by time t, satisfies the relation:

$$m(t) = \int_0^t \lambda(s)ds \tag{A.18}$$

and,

$$\lambda(t) = \frac{dm(t)}{dt} \tag{A.19}$$

The random variable N(t) follows a Poisson distribution with parameter m(t), that is, the probability that N(t) takes a given non-negative integer value n is determined by equation a.20.

$$PN(t) = n = \frac{[m(t)]^n \mathrm{e}^{-m(t)}}{n!}, \quad n = 0, 1, \ldots, \infty \tag{A.20}$$

The time domain models which assume the failure process to be a NHPP differ in the approach they use for determining $\lambda(t)$ or m(t). The NHPP models can be further classified into finite failures and infinite failures models. We will be concerned only with finite failure NHPP models in this thesis.

Finite failures NHPP models assume that the expected number of faults detected during an infinite amount of testing time will be finite and this number is denoted by a [16]. The mean value function m(t) in case of finite failure NHPP models can be written as [107]:

$$m(t) = aF(t) \tag{A.21}$$

where F(t) is a distribution function.

The failure intensity $\lambda(t)$ for finite failure NHPP models can be written as:

$$\lambda(t) = aF'(t) \tag{A.22}$$

The failure intensity $\lambda(t)$ can also be written as:

$$\lambda(t) = [a - m(t)]\frac{F(t)}{1 - F'(t)} \tag{A.23}$$

where,

$$h(t) = \frac{F(t)}{1 - F'(t)} \tag{A.24}$$

is the failure occurrence rate per fault or the hazard rate. Referring to equation a.23, $[a - m(t)]$ represents the expected number of faults remaining, and hence is a non-increasing function of testing time. Thus, the nature of the failure intensity is governed by the nature of the failure occurrence rate per fault [108]. Popular finite failure NHPP models can be classified according to the nature of their failure occurrence rate per fault. The Goel Okumoto software reliability growth model has a constant failure occurrence rate per fault [106], the Generalized Goel Okumoto model can exhibit an increasing or a decreasing failure occurrence rate per fault [15], the S-shaped model exhibits an increasing failure occurrence rate per fault [109], and the log-logistic model exhibits an increasing/decreasing failure occurrence rate per fault [17].

A.2.8 State-space view of NHPP

The NHPP models described above provide a closed-form analytical expression for the expected number of faults detected given by the mean value function, m(t). However, the mean value function m(t) can also be obtained using numerical techniques. In this section we present a brief overview of homogeneous and non-homogeneous Poisson process, and describe a method to obtain a solution of the NHPP processes using numerical techniques. A discrete state, continuous-time stochastic process $N(t), t \geq 0$ is called a Markov chain if for any $t_0 < t_1 < \ldots < t_n < t$, the conditional probability mass function of N(t) for given values of $N(t_0) < N(t_1) < \ldots < N(t_n)$, depends only on $N(t_n)$. Mathematically the above statement can be expressed as [42]:

$$P[N(t) = x|N(t_n) = x_n, \ldots, N(t_0) = x(t_0)] = P[N(t) = x|N(t_n) = x_n] \tag{A.25}$$

Equation a.25 is known as the Markov property.

A Markov chain $N(t), t \geq 0$ is said to be time-homogeneous, if the following property holds:

$$P[N(t) = x|N(t_n) = x_n] = P[N(t - t_n) = x|N(0) = x_n] \tag{A.26}$$

Equation a.26 also known as the property of stationary increments implies that the next state of the system depends only on the present state and not on the time when the system entered that state. Thus in case of a homogeneous continuous time Markov chain (CTMC), the holding time in each state is exponentially distributed. Markov chains which do not satisfy the property of stationary increments given by equation a.16 are called as non-homogeneous Markov chains [110]. For a non-homogeneous CTMC, the holding time distribution in state 0 is $1 - \mathrm{e}^{-m(t)}$, whereas the distribution of the holding times in the remaining states is fairly complicated.

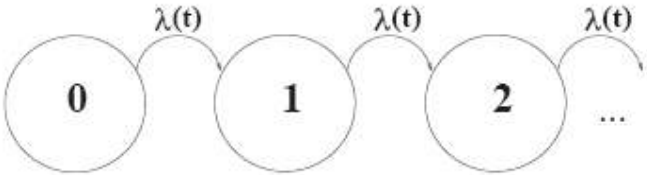

Figure A.3: Non-Homogenous Markov chain.

A homogeneous Poisson process is a homogeneous continuous time Markov chain, whereas a non-homogeneous Poisson process is a non-homogeneous continuous time Markov chain. Figure a.3 shows the representation of a non-homogeneous Poisson process by a non-homogeneous continuous time Markov chain (NHCTMC). In the figure a.3, the state of the process is given by the number of events observed. When the NHCTMC is used to model the fault detection process, each event represents fault detection and hence the state of the process is given by the number of faults detected. The process starts in state 0, since initially at time t = 0 which marks the beginning of the testing process no faults are detected. Upon the detection of the first fault, the process transitions to state 1, and the rate governing this transition is the failure intensity function $\lambda(t)$. Similarly, upon the detection of the second fault the process transitions to state 2 and so on. Note that the time origin of a transition rate emanating from any state is the beginning of system

operation (this is called global time) and not from the time of entry into that state (the so called local time). Globally time dependent transition rates imply a non-homogeneous Markov chain as in the present case whilst locally time dependent transition rates would imply a semi-Markov process. The expected number of faults detected given by the mean value function m(t) can also be computed numerically by solving the Markov chain shown in Figure a.3 using SHARPE [111]. SHARPE stands for Symbolic Hierarchical Automated Reliability and Performance Evaluator, and is a poweful tool that can be used to solve stochastic models of performance, reliability and performability. SHARPE was first developed in 1986 for three groups of users, namely, practising engineers, researchers in performance and reliability modeling, and students in science and engineering courses. Since then several revisions have been made to SHARPE to fix bugs and to adopt new requirements. Details of SHARPE can be obtained from [111]. While using SHARPE to solve the chain, two issues need to be resolved. The first issue is that the chain in figure a.3 has an infinite number of states. In order to overcome this issue, the chain can be truncated to θ states. Assuming that we are unlikely find more faults than six times the expected number of faults present in the beginning, we use:

$$\theta = 6\lceil a \rceil \tag{A.27}$$

where a is the expected number of faults that can be detected given infinite testing time as in case of equation a.21. The second issue is that SHARPE is designed to solve homogeneous CTMCs. We get around this problem using the time-stepping techniques by dividing the time axis uniformly into small sub-intervals, where within each time sub-interval, the failure intensity, $\lambda(t)$, can be assumed to be constant. Thus, within each time sub-interval, the non-homogeneous continuous time Markov chain reduces to a homogeneous continuous time Markov chain which can then be solved using SHARPE. This value of $\lambda(t)$ is used to obtain the state probability vector of the Markov chain at the end of that time sub-interval. These state probabilities then form the initial probability vector for the next time sub-interval. Let $p_i(t)$ denote the probability of being in state i

at time t. The mean value function, m(t), can then be computed as:

$$m(t) = \sum_{i=0}^{\theta} i \times p_i(t) \tag{A.28}$$

The accuracy of the solution produced by solving the NHCTMC using SHARPE will be influenced by the length of the time sub-interval used to approximate the value of the failure intensity function $\lambda(t)$. In general, the smaller the sub-interval, the better will be the accuracy. However, as the length of the sub-interval decreases, the computational cost involved in obtaining a solution increases. Thus, an appropriate choice of the length of the time sub-interval is a matter of trade-off between the computational cost and the desired accuracy of the solution. We demonstrate the computational equivalence of the mean value function obtained using the expressions presented in section A.2.2 (referred henceforth as the closed-form analytical method) and the numerical solution method, using the NTDS data [2, 106].

The NTDS data is from the U.S. Navy Fleet Computer Programming Centre consisting of errors in the development of software for a real-time, multicomputer complex which forms the core of the Naval Tactical Data System (NTDS). The NTDS software consisted of 38 different modules. Each module was supposed to follow three stages: the production (development) phase, the test phase, and the user phase. The parameters of the four NHPP models described above were estimated from the NTDS data using SREPT [112], and then the mean value function is computed for each of these models by solving the Markov chain in figure a.3. Figure a.4 shows the closed-form mean value function and the one obtained using a numerical solution of the non-homogeneous continuous time Markov chain (NHCTMC) for Goel Okumoto, Generalized Goel Okumoto, S-shaped, and log-logistic models respectively.

As observed from figure a.4, the numerical solution of the mean value function obtained using the state-space view of the non-homogeneous Poisson process gives us a very good approximation to the analytical solution. In the next section, we describe how we exploit the flexibility offered by the state-space view to incorporate more realistic features

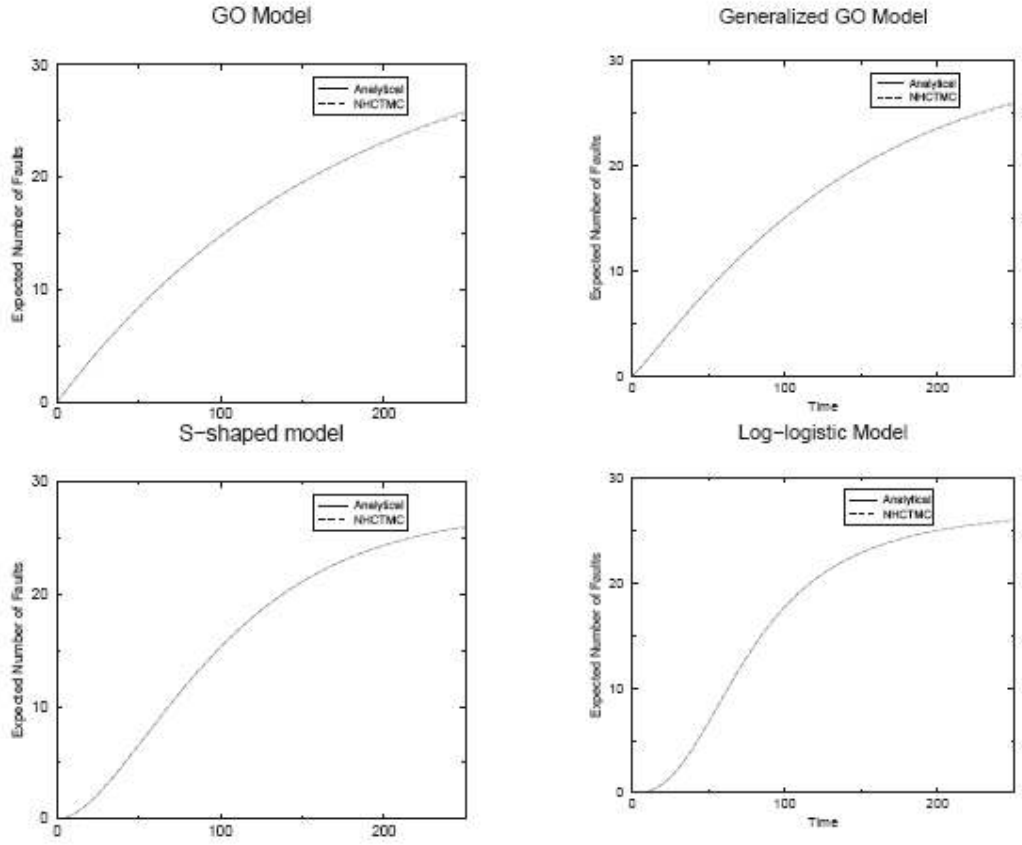

Figure A.4: Comparison of analytical and numerical mean value functions.

into the NHPP software reliability models, which were initially based on oversimplifying assumptions in order to ensure mathematical tractability.

A.2.9 Infinite Failure Models

The mean value function of this class of models is unbounded, i.e. the expected number of failures experienced in infinite time is infinite. Some of the popular models are described under:

A.2.9.1 Musa-Okumoto Logarithmic Poisson Execution Time Model

This model assumes the number of failures experienced by time τ (execution time), is NHPP, with the mean value function, $m(\tau)$ given by equation a.29,

$$m(\tau) = \frac{1}{\theta} ln(\lambda_0 \theta_\tau + 1) \tag{A.29}$$

where λ_0 denotes the initial failure intensity, and $\theta > 0$, the failure decay parameter.

The failure intensity is given by equation a.30,

$$\lambda(t) = \frac{\lambda_0}{\lambda_0 \theta_\tau + 1} \tag{A.30}$$

A Bayesian approach for predicting the number of failures in a software product using the logarithmic NHPP model has been proposed by Campodonico and Singpurwalla[113]. This model uses, in a formal manner, expert knowledge in the conduct of software testing.

A.2.9.2 Duane Model

This model [114] was originally proposed for hardware reliability studies. Its mean value function, m(t), and the failure intensity function $\lambda(t)$ is given by equations a.31 and a.32 respectively.

$$m(t) = at^b, \quad b > 0 \tag{A.31}$$

$$and$$

$$\lambda(t) = abt^{b-1}, \quad b > 0 \tag{A.32}$$

This model usually overestimates the cumulative number of failures. The main criticism of the model is that its mean value function approaches infinitely very rapidly.

A.2.9.3 Log-Power NHPP Model

This model is a modification of the traditional Duane (1964) model. It has the mean value function is given by equation a.33.

$$\mu(t) = aln^b(1+t) \quad a > 0, b > 0 \tag{A.33}$$

An important property is that the log-power model has a graphical interpretation. If we take the logarithmic on both sides of the mean value function, we have

$$ln\mu(t) = lna + blnln(1+t) \tag{A.34}$$

If the cumulative number of failures is plotted versus the running time, the plot should tend to be on a straight line on a log-log scale. This can be used to validate the model and to easily estimate the model parameters. When the plotted points cannot be fitted with a straight line, the model is probably inappropriate, and if they can be fitted with a straight line, then the slope and intercept on vertical axis can be used as estimates of b and lna, respectively.

Note that the log-power model, as well as the Duane model for repairable system, allows $\mu(t)$ to approach infinity. These models are infinite failure models with the assumption that the expected total number of failures to be detected is infinite. This is valid in the situation of imperfect debugging where new faults are introduced in the process of removing the detected faults.

A.3 Bayesian software reliability growth models

In Bayesian models in the absence of data the parameters are not considered to be fixed at some unknown value, but they are assumed to follow a prior distribution. Let $p(\theta)$ denote the prior density of the parameter vector θ. The still unknown but observable data y - like failure times or the number of failures experienced by time t - are linked to the parameters through the software reliability growth model. Given certain parameter

values, the data follow a conditional distribution $p(y|\theta)$. Since the parameter values are not known, the prior predictive distribution of y is calculated as the integral of the joint distribution of y and θ over the possible values of θ [115]:

$$p(y) = \int p(y, \theta) d\theta = \int p(y|\theta) p(\theta) d\theta \tag{A.35}$$

To put it in words, given a software reliability growth model and an assumption about the distribution of the model parameters, the distribution of quantities to be predicted with the model can be calculated. For example, it is possible to obtain the distribution of the random variable M(t1), the number of failures observed by time t1, and also its expected value $\mu(t1)$, which is the mean value function at t1. All this can be done before any failure has been observed!

The prior distribution of the model parameters is built on the analyst's beliefs, which may also stem from knowledge of structural information of the software or from experience. However, it may still deviate from reality. As soon as data are available, they should be used to update the assumed distribution of the model parameters by calculating the conditional distribution of the model parameters given the observed data. According to Bayes' rule this is [115]:

$$p(\theta|y) = \frac{p(y, \theta)}{p(y)} = \frac{p(y|\theta) p(\theta)}{\int p(y|\theta) p(\theta) d\theta} \tag{A.36}$$

It can easily be seen that only the prior density of $\theta, p(\theta)$, and the likelihood function $\mathcal{L}(\theta; y) = p(y|\theta)$ is used for obtaining the posterior distribution $p(\theta|y)$. Therefore, this updated belief about the parameters is a compromise between the initial assumptions and the evidence of the collected data in connection with the underlying model. Predictions in the light of data draw on this posterior distribution of θ. For example, after the first failure has been observed at t1, one can use y = t1 and the prior distribution $p(\theta)$ to calculate $p(\theta|y)$ via equation a.36 and predict the distribution of $M(t1 + \Delta t)$, the number of failures experienced by $t1 + \Delta t$, based on the posterior distribution of θ. In general, the posterior predictive distribution of some unknown observable $\tilde{y}$ (like the time until

the occurrence of the next failure) is given by [115]:

$$p(\tilde{y}|y) = \int p(\tilde{y}, \theta|y)d\theta = \int p(\tilde{y}|\theta, y)p(\theta|y)d\theta = \int p(\tilde{y}|\theta)p(\theta|y)d\theta \tag{A.37}$$

A lot of different Bayesian software reliability growth models have been proposed over the last decades [116, 117]. Some of them can be considered Bayesian extensions of frequentest models assuming certain prior distributions for their parameters. Although it was not formulated in that way, the Littlewood-Verrall model, for example, puts a Gamma prior distribution on the per-fault hazard rate Φ of the Jelinski-Moranda model while leaving the number of faults in the software U degenerate at some value u0 [116]. On the other hand, assuming that U follows a Poisson distribution with mean N while Φ is fixed at the constant value φ leads to the Goel-Okumoto model. In the past, the prior distributions were often chosen as to ensure mathematical tractability, e.g. in form of the "conjugate prior distributions" [115]. The speed with which today's personal computers can execute Markov Chain Monte Carlo methods, like the Metropolis-Hastings algorithm [117] and specifically the Gibbs sampler [118], gives the analyst more freedom in selecting the prior densities and also enables him to analyze more complex models. Linking maturity information and expert opinion about the software development and the software testing process to software reliability models with the help of prior distributions seems to be a possible approach. There are several options for doing this:

1. The structural information could be used directly for specifying the prior distributions of some or all of the model parameters. On the one hand, the prior densities derived in this way could be used instead of "non-informative" prior distributions - prior distributions with flat densities -, which are frequently chosen in the absence of solid conjectures about the model parameters. On the other hand, in simulation studies the parameters of the prior densities - so-called "hyperparameters" - are often arbitrarily determined. If maturity information and the like could be linked to the Bayesian models, then the particular prior distribution used would have a better technical foundation. Singpurwalla [119] describes a method for incorporating

expert knowledge into the prior densities of the parameters of a Weibull distribution for hardware reliability. It also enables the analyst to include his views about the expert's input, e.g. assumed bias or optimism on the part of the expert with respect to the accuracy of his estimations. However, one obstacle to this method may be that practitioners often find it difficult to express their presumptions in terms of model parameters.

2. Another approach may be based on the results in chapter 3: Instead of assuming prior densities for the parameters of a software reliability growth model one could formulate prior distributions for parameters of the individual effects (like the development of test coverage or the testing efficiency). Probably it is easier to relate structural information to one of these separate factors. Furthermore, they may be more meaningful for a practitioner than the composite model parameters of a SRGM.

3. Campod'onico and Singpurwalla [113, 120] propose to ask an expert about his opinion on the values of the mean value function at k points in time, where k is the number of parameters in the SRGM chosen by the analyst. They describe an application of their method to the analysis of the Musa-Okumoto model.

A.4 Other Models

The Error Complexity Model (ECM) which can neither be classifies as fault seeding model nor an input domain model is presented, next. We also discuss some of the time domain models which can neither be classified as homogeneous Markov or non-homogeneous Markov models.

A.4.1 Error Complexity Model

In the Error Complexity Model (ECM), the faults are classified by fault complexity, which is considered to be a measure of fault detectability. The number of remaining software

faults is estimated from the ratio of complex to simple faults and number of discovered faults. New criteria for classification of faults based on their complexity are proposed.

A.4.2 Littlewood-Verall (LV) Bayesian Model

The Littlewood-Verrall model differs from the models described above in several important ways. The above models assume that all defects contribute equally to the reliability of a program. The Littlewood-Verrall model disposes of this assumption, based on the observation that a program with defects in rarely exercised sections of the code will be more reliable than the same program with the same number of defects in frequently exercised portions of the code. This model also assumes that the failure rate, instead of being constant, is a random variable. Finally, this model attempts to account for defect generation in the correction process by allowing for the probability that the program could be made less reliable by correcting an defect. This is an important departure from the other models described above, all of which assume perfect debugging.

Formally, the assumptions of this model are:

1. Successive execution times between failures, i.e., $X_i, i = 1, 2, 3, \ldots$, are independent random variables with probability density functions

$$f(X_i) = \lambda_i \mathrm{e}^{-\lambda_i X_i} \tag{A.38}$$

where λ_i are the failure rates. X_i is assumed to be exponential with parameter λ_i.

2. The $\lambda_i' s$ form a sequence of random variables, each with a gamma distribution of parameters α and $\Psi(i)$, such that:

$$g(\lambda_i) = \frac{[\Psi_i]^\alpha \lambda_i^{\alpha-1} \mathrm{e}^{-\Psi}(i)\lambda_i}{\Gamma(\alpha)} \tag{A.39}$$

$\Psi(i)$ is an increasing function of the number of defects, i, that describes the "quality" of the programmer and the "difficulty" of the programming task. A good programmer should have a more rapidly increasing function $\Psi(i)$ than a poorer

programmer. By requiring $\Psi(i)$ to be increasing, the condition

$$P(\lambda(i) < x) > P(\lambda(i-1) < x) \tag{A.40}$$

is satisfied for all i. This reflects that it is the intention to make the program better after a defect is detected and corrected. It also reflects the reality that sometimes corrections will make the program worse. For the function $\Psi(i)$, Littlewood and Verrall suggest either of the two forms $\beta_0 + \beta_1 i$ or $\beta_0 + \beta_1 i^2$. Assuming a uniform a priori distribution for α, the parameters β_0 and β_1 can be found by maximum likelihood estimation.

3. During test, the software is operated in a similar manner as the anticipated operational usage. The mean time between the $(i-1)^{th}$ and the i^th failure, $\Theta(i)$, is given by:

$$\Theta(i) = \frac{t_i + \Psi(i)}{\alpha} \tag{A.41}$$

α is a parameter of the gamma distribution for the failure intensities.

$\Psi(i)$ is as defined above.

A.4.3 Littlewood and Keiller (LK) Bayesian Model

This model is similar to LV model, except that the reliability growth is introduced via the shape parameter of the gamma distribution for the failure intensities. The failure intensities have the gamma density function given by

$$p(\lambda_i) = \frac{\beta^{\Psi(i)} \lambda_i^{\Psi(i)-1} \mathrm{e}^{-\beta\lambda_i}}{\Gamma(\Psi(i))} \tag{A.42}$$

The reliability growth in this case occurs when $\Psi(i)$ is a decreasing function of i. the choice of the parametric family for $\Psi(i)$ is up to the user.

Appendix B

TimeNET TOOL

B.1 Introduction

TimeNET is a graphical and interactive toolkit for modeling with stochastic Petri nets (SPNs) and stochastic colored Petri nets (SCPNs). TimeNET has been developed at the Real-Time Systems and Robotics group of Technische Universit¨at Berlin, Germany (http://pdv.cs.tu-berlin.de/). The project has been motivated by the need for powerful software for the efficient evaluation of timed Petri nets with arbitrary firing delays. TimeNET and its predecessor DSPNexpress were influenced by experiences with other well-known Petri net tools, e.g., GreatSPN and SPNP. For the latest information about TimeNET, check the tool's home page at http://pdv.cs.tu-berlin.de/ timenet/. The version TimeNET 4.0 (stable version available since 2007) includes a completely rewritten JAVA graphical user interface and provides support of the Microsoft Windows operating system [121]. It supports a new class of stochastic colored Petri nets (SCPNs). A standard discrete-event simulation has been implemented for the performance evaluation of SCPN models. SCPNs allow arbitrary distributions of firing delays including zero delays, complex token types, global guards, time guards, and marking dependent transition priorities. Petri net classes are defined by an extendable XML schema in TimeNET 4.0 which affects the behavior of the graphical user interface. A model is a well-formed XML document which is validated automatically. Some of the keys features of TimeNET are

as follows.

1. a generic graphical user interface in JAVA based on an XML net class representation, easily extendable for most graph-like modeling formalisms.
2. user interface and evaluation algorithms run under Windows and Linux operating system environments.
3. modeling and simulation of stochastic colored Petri nets.
4. graphical display of SCPN simulation results.
5. independent components for modeling, simulation, analysis, and result output allowing the GUI to be run on a different computer than the analysis modules.

B.2 System Requirements

TimeNET 4.0 is currently supported on three runtime environments:

1. Windows Windows XP (a version for Windows Vista will be compiled when a MinGWport for Vista is available)
2. Linux on personal computers (recommended distribution: Debian 3.1)
3. SunOS 5.6-5.9 (Solaris) on Sun workstations: TimeNET 3.0 runs in these environments

B.3 Downloading TimeNET

TimeNET is free of charge for non-commercial use. Companies should contact Armin Zimmermann for a commercial license. For commercial use of TimeNET, please copy, fill out, and sign the registration form from the TimeNET web pages are required to print, fill and sign. Send it by fax or mail to

Technische Universit¨at Berlin

Prozessdatenverarbeitung und Robotik

Armin Zimmermann

Sekr. FR 2-2

Franklinstr. 28/29

10587 Berlin, Germany

Fax: +49-30-31.42.11.16

After your successful registration, we will send an email with login name and password for the TimeNET download page. Use your web browser to download the files you need. According to your target architecture, you should download one of the available precompiled TimeNET files.

Please note that the password may be changed without further notice. If you are a registered user of TimeNET, you can obtain the current password by email at any time.

B.4 How to Install the Tool

The tool can be installed by extracting all files from the downloaded archive into a directory TimeNETinstall as described in the following steps:

1. Permenently remove TNETHOME and MODELDIR environment variables from your system which were required by previous versions of TimeNET. (Applies to users of older version only)

2. Create a TimeNETinstall directory in C:\Program Files on Windows or in /usr/local/bin or /home/username on Linux.

3. Copy the downloaded compressed archive (TimeNETxxxx_windows.zip or TimeNETxxxx_linux.tgz) into the newly created TimeNETinstall directory.

4. Switch to the TimeNETinstall directory.

5. On Linux, use tar to extract the directories and files from the archive. tar -xvzf TimeNETxxx_linux.tgz

6. On Windows, right-click the archive TimeNETxxx_windows.zip (or use WinZip) and extract all files into the current directory.

B.5 Configure a multi-user installation

One tool installation can be used by several users on a machine. The TimeNETinstall directory contains two directories "TimeNET" and "Models" after file extraction. If you want to use TimeNET from different user accounts, you should copy the "Models" directory to each local user directory, which is My Documents on Windows and /home/username on Linux.After starting TimeNET as described later, each user can create and use own models in its local "Models" directory.

B.6 Starting the Tool

If all previous steps have been successfully completed, the tool can be started using the command:

1. On Linux: /usr/local/bin/TimeNETinstall/TimeNET/bin/startGUI
2. On Windows: $C:\backslash ProgramFiles\backslash TimeNETinstall\backslash TimeNET\backslash bin\backslash startGUI.bat$

B.7 Configuring the User Interface

Several graphical appearance options of the user interface can be changed with the menu entry File/Settings.

B.8 Upgrading to TimeNET 4.0

It is not possible to update older versions. Please install TimeNET 4.0 in a new directory.

B.8.1 Conversion of old Model Files

If you need to convert model files from one version of the tool to another, here are some hints:

1. TimeNET 1.4 / 2.0 models Filename extension: .TN Format explanation: see manual of TimeNET 2.0 TimeNET 4.0 allows to import and export models in the old .TN format. After loading a model in the old .TN format, it is automatically converted to the new XML format. The internal analysis modules for eDSPN models still use the old .TN format, into which it is possible to export model files.

2. TimeNET 3.0 models Filename extension: .net Format explanation: see manual of TimeNET 3.0 TimeNET 3.0 stores models in a net class-independent format. It is not possible to load models in the .NET format of TimeNET 3.0 into TimeNET 4.9. Such models must be converted to the .TN format first using TimeNET 3.0 (File/Export and File/Import).

3. TimeNET 4.0 Beta realease models The general XML format of TimeNET models has slightly changed, so that models created with the 4.0 beta version must be converted by following a few simple steps. Open the old model in your favourite text editor and change the first lines depending on the type of the model (this is also explained in the web pages, from where you can copy and paste the text).

- GMPN models: Remove the first lines until GMPN.xsd”> and add the following lines at the same place:

<net id="0" netclass="SCPN"
xmlns="http://pdv.cs.tu-berlin.de/TimeNET/schema/SCPN"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://pdv.cs.tu-berlin.de/TimeNET/schema/SCPN
etc/schemas/SCPN.xsd">

- eDSPN models: Remove the first lines until eDSPN.xsd"> and add the following lines at the same place:

<net id="0" netclass="eDSPN"

xmlns="http://pdv.cs.tu-berlin.de/TimeNET/schema/eDSPN"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation=

"http://pdv.cs.tu-berlin.de/TimeNET/schema/eDSPN etc/schemas/eDSPN.xsd">