IST-1999-55017

Software Reliability Model Study

Deliverable A.2

Owner	Michael Grottke
Approvers	Eric David Klaudia Dussa-Zieger
Status	Approved
Date	01/06/01

Contents

1 Introduction

1.1 Project overview

The PETS project is a two-year research and development effort on the part of a consortium of three industrial and one academic bodies, partially funded by the European Commission under the Fifth Framework Agreement (CRAFT project). The objective of PETS is to develop a new, enhanced statistical model which predicts the reliability of SW programs based on test and software maturity results. This new model will be implemented as a prototype which allows small and medium enterprises, especially small software companies, to predict the reliability of their developed and tested software in an easy and efficient manner with a higher accuracy than currently possible.

1.2 This document

In this document existing software reliability growth models are studied. The second chapter discusses some important concepts in the field of software reliability engineering and also describes the rationale for the way in which the software reliability growth models are compared in chapter 3. Since one of the goals of the PETS project is to include process maturity data in a software reliability growth model, chapter 4 explains different existing approaches for linking additional information to software reliability growth models. Chapter 5 discusses the important question whether one model can be used for analyzing all testing-effort and failure data of a software product or if some distinction has to be made either be using more models or by transforming some of the data collected.

2 Important concepts in software reliability engineering

Software reliability is defined as the probability of failure-free software operation in a defined environment for a specified period of time. A *failure* is the departure of software behavior from user requirements. This dynamic phenomenon has to be distinguished from the static *fault* (or *bug*) in the software code, which causes the failure occurrence as soon as it is activated during program execution.

Since software does not deprecate like hardware, the reliability of software stays constant over time if no changes are made to the code or to the environmental conditions including the user behavior. However, if each time after a failure has been experienced the underlying fault is detected and perfectly fixed, then the reliability of software will increase with time. Typically this is the situation during the testing phase(s). As each failure occurrence initiates the removal of a fault, the number of failures that have been experienced by time t, denoted by $M(t)$, can be regarded as a reflected image of reliability growth. Models trying to explain and describe the behavior of $M(t)$ are called *software reliability growth models* (SRGMs). Each SRGM implicitly assumes a certain functional form of $M(t)$. Using the failure times $t_1, t_2, ...$ or the times between failures $\Delta t_1, \Delta t_2, ...$ collected during a testing project, the parameters of a SRGM can be estimated. The *mean value function* $\mu(t)$ of a SRGM represents the expected number of failures experienced by time t according to the model. The derivative of the mean value function with respect to time, $\frac{d\mu(t)}{dt}$, is named *failure intensity* $\lambda(t)$. It represents the limit of the expected number of failures experienced in time interval $[t; t+\Delta t)$ for Δt approaching zero.

The failure intensity should not be mistaken for the *hazard rate of the application*, $z(\Delta t \mid t_{i-1})$, the probability density for the i^{th} failure being experienced at $t_{i-1} + \Delta t$ conditional that the $(i-1)^{st}$ failure has occurred at t_{i-1}. In the field of survival analysis the hazard rate of a component is often denoted by $\lambda(t)$ [10, 11, ch. 9]. Moreover, some researchers in software reliability call the program hazard rate "failure rate $r(t)$" [54, p. 53] or even "failure intensity $\lambda(t)$" [47, p. 88]. The fact that the program hazard rate $z(\Delta t \mid t_{i-1})$ is indeed equal to the failure intensity at time $t_{i-1} + \Delta t$ for software reliability growth models of the Poisson type [41, p. 259] may contribute to this confusion. This study follows the terminology used in [41].

Many software reliability growth models do not only specify the hazard rate of the entire application, but also the *hazard rate of an individual fault*. If each fault is equally dangerous (with respect to its activation frequency, not in regard to the damage caused), then the time until failure occurrence has the same probability density $f_a(t)$. The per-fault hazard rate of each fault is then $z_a(t) = \frac{f_a(t)}{1-F_a(t)}$, where $F_a(t) = \int_0^t f_a(x)dx$. Therefore, the per-fault hazard rate is a measure for the probability that a particular fault will instantaneously cause a failure, given that it has not been activated so far.[1]

The nature of the failure generating process depends on characteristics of the software product and the way in which it is used. The assumptions of many software reliability growth

[1] For more information on the basic concepts in software reliability engineering, please refer to [29] or to [41, pp. 3 -29].

models basically mean that the relative frequencies with which the different functions are used are stable over time. I.e., the different parts of the software are not tested en bloc, but the next test case to be executed is always drawn randomly from the set of specified test cases. Furthermore, if the reliability measure shall be extrapolated to what users will perceive then the relative frequencies of the functions should coincide with the expected frequencies prevailing after release. This methodology is named *operational testing* in short. While its supporters contend that it does not only bring about data which can be used for assessment of the current reliability but also results in detection of those faults that really matter in terms of occurrence probability [39], critics have argued that it is less efficient than *systematic testing techniques* [43] and often not feasible for a number of reasons (cf. the list of arguments in [19]). Simulation studies [9, 21] as well as analytical examinations [13] of the different testing strategies could not prove that one of them is superior to the other.

Like the large majority of industrial organizations [13], the SMEs participating at the PETS project do not employ operational testing. Therefore, it has to be studied if the standard software reliability models can be applied to data collected during systematic testing. As a first step, in chapter 3 different models are compared with respect to their building blocks. On the one hand the identification of a similar structure helps in unifying the models. On the other hand the elements can be interpreted separately. This gives a deeper understanding of the models' assumptions and may guide practitioners in model selection or even in the creation of new models.

3 A unifying view on some software reliability growth models

3.1 Jelinski-Moranda de-eutrophication model

The de-eutrophication model developed by Jelinski and Moranda [24] 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 u_0 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 $(\Delta t_1, \Delta t_2, ... \Delta t_{u_0})$ 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, $z_a(t) = \phi$, being the constant of proportionality:

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

The Jelinski-Moranda model belongs to the binomial type of models as classified by Musa et al. [41, pp. 250 - 252]. 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.:

$$\frac{d\mu(t)}{dt} = u_0 f_a(t) = u_0 \phi \exp(-\phi t) \tag{2}$$

Therefore, the mean value function is

$$\mu(t) = u_0[1 - \exp(-\phi t)]. \tag{3}$$

It can easily be seen from equations (2) and (3) that the failure intensity can also be expressed as

$$\frac{d\mu(t)}{dt} = \phi[u_0 - \mu(t)]. \tag{4}$$

According to equation (4) 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 (4). 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.

3.2 Goel-Okumoto model

The model proposed by Goel and Okumoto in 1979 [16] 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 $(0, t_1), (t_1, t_2), ..., (t_{n-1}, t_n)$ 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 u_0 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

$$\frac{d\mu(t)}{dt} = \phi[N - \mu(t)]. \tag{5}$$

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.

3.3 Models with a time-varying testing-effort

Practically all time-based software reliability growth models implicitly assume that the testing intensity is constant over time. If the time-measure used is calendar time this will often not be the case. In the following, let t^* denote measurements taken in a time scale for which the testing-effort is not constant, while t refers to a time scale for which it is.

Jelinski and Moranda do not only point out the problem of a time-varying testing-effort in their original paper, but they also propose a refinement of their model [24]. They suggest to account for the time-varying testing-effort by a function $E(t^*)$, which they call "exposure rate", and either transform the time scale in accordance with this function or express the program hazard rate after the removal of the $(i-1)^{st}$ fault as

$$z(\Delta t^* \mid t^*_{i-1}) = \phi E(t^*_{i-1} + \Delta t^*)[u_0 - (i-1)]. \tag{6}$$

This second approach leads to the failure intensity function

$$\frac{d\mu(t^*)}{dt^*} = \phi E(t^*)[u_0 - \mu(t^*)] \tag{7}$$

and to the mean value function

$$\mu(t^*) = u_0 \left[1 - \exp\left(-\phi \int_0^{t^*} E(x)dx\right)\right]. \tag{8}$$

In 1986, Yamada, Ohtera and Narihisa [59] extended the Goel-Okumoto model following the same line of thoughts as Jelinski and Moranda. They modified its assumption 2 by stating that the ratio between the expected number of software failures occurring in $(t^*, t^*+\Delta t^*]$ with $\Delta t^* \to 0$ and the current testing-effort expenditures $w(t^*)$ is proportional to the expected number of undetected faults, $N - \mu(t^*)$. Therefore,

$$\frac{d\mu(t^*)}{dt^*} = \phi w(t^*)[N - \mu(t^*)] \tag{9}$$

and

$$\mu(t^*) = N\left[1 - \exp\left(-\phi \int_0^{t^*} w(x)dx\right)\right] = N\left[1 - \exp\left(-\phi W(t^*)\right)\right], \tag{10}$$

where $W(t^*) = \int_0^{t^*} w(x)dx$.

Comparing equations (7) and (9) to equations (4) and (5), respectively, shows that the hazard rate of each fault remaining in the software - $\phi E(t^*)$ or $\phi w(t^*)$ - is not constant in time t^*, but varies due to the changing intensity of testing. Unlike Jelinski and Moranda, Yamada et al. propose particular functional forms for the testing-effort function. In [59] they discuss the exponential density function $w(t^*) = \alpha\beta \exp(-\beta t^*)$ and the Rayleigh density function $w(t^*) = 2\alpha\beta t^* \exp(-\beta {t^*}^2)$, while in [58] the more general Weibull density function $w(t^*) = m\alpha\beta {t^*}^{m-1} \exp\left(-\beta {t^*}^m\right)$ is used. Huang, Kuo and Chen [23] suggest the logistic density function $w(t^*) = \frac{\alpha\beta\gamma \exp(-\gamma t^*)}{(1+\beta \exp(-\gamma t^*))^2}$ as an alternative.

Again, including $w(t^*)$ in equation (9) is equivalent to transforming the time scale with $t = W(t^*)$ and considering $\mu(t) = \mu(W(t^*))$ and $\frac{d\mu(t)}{dt} = \frac{d\mu(W(t^*))}{dW(t^*)}$, because setting

$$\frac{d\mu(W(t^*))}{dW(t^*)} = \phi[N - \mu(W(t^*))] \tag{11}$$

leads to

$$\frac{\frac{d\mu(W(t^*))}{dt^*}}{\frac{dW(t^*)}{dt^*}} = \frac{d\mu(W(t^*))}{dt^*}\frac{1}{w(t^*)} = \phi[N - \mu(W(t^*))], \tag{12}$$

which is equivalent to equation (9).

3.4 Musa basic execution time model

Assuming a certain distributional form of the testing-effort and introducing it into a software reliability growth model or using it to transform the time scale is only one approach to dealing

with a time-varying testing-effort. If the time spent on testing could be measured in units which ensure that the testing-effort is approximately constant, these data could be directly fed into the original software reliability model. Musa's basic execution time model [37] developed in 1975 was the first one to explicitly require that the time measurements are in actual CPU time utilized in executing the application under test (named "execution time" τ in short).[2]

Although it was not originally formulated like that the model can be classified by three characteristics [41, p. 251]:

1. The number of failures that can be experienced in infinite time is finite.
2. The distribution of the number of failures observed by time τ is of Poisson type.
3. The functional form of the failure intensity in terms of time is exponential.

It shares these attributes with the Goel-Okumoto model, and the two models are mathematically equivalent.

In addition to the use of execution time, a difference lies in the interpretation of the constant per-fault hazard rate ϕ. Musa [37] split ϕ up in two constant factors, the linear execution frequency f and the so-called fault exposure ratio K:[3]

$$\frac{d\mu(\tau)}{d\tau} = fK[N - \mu(\tau)]. \tag{13}$$

f can be calculated as the average object instruction execution rate of the computer, r, divided by the number of source code instructions of the application under test, I_S, times the average number of object instructions per source code instruction, Q_x: $f = \frac{r}{I_S Q_x}$ [41, pp. 121 - 124]. The fault exposure ratio relates the fault velocity $f[N - \mu(\tau)]$, the speed with which defective parts of the code would be passed if all the statements were consecutively executed [41, p. 121], to the failure intensity experienced. Therefore, it can be interpreted as the average number of failures occurring per fault remaining in the code during one linear execution of the program.

3.5 Characteristics of the fault exposure ratio

In the context of Musa's basic execution time model K is assumed to be constant over time. Moreover, data published by Musa in 1979 and also listed in [41, p. 123] suggest that the fault exposure ratio might not vary considerably over a wide range of software systems: For 13 projects with known program sizes between 5,700 and 2,445,000 object instructions the fault exposure ratio took values between $1.41 \cdot 10^{-7}$ and $10.6 \cdot 10^{-7}$ with an average value of

[2] The execution time τ can be considered a specific example for a time measure t with constant testing-effort.

[3] In a generalized version of his model, Musa also included the fault reduction factor B, the average rate of fault reduction to failure occurrence, and the testing compression factor C, the ratio of execution time required in the operational phase to execution time required in the test phase [37, 41, p. 178]. However, without additional information the parameters in the equation $\frac{d\mu(\tau)}{d\tau} = BCfK[N-\mu(\tau)]$ cannot be identified. Therefore, most authors do not consider B and C, i.e. they implicitly set them to 1.

$4.2 \cdot 10^{-7}$. Musa [38] gives some explanation why K should be independent of the program size. Likewise, Malaiya et al. [32] argue that K may be relatively independent of the program structure.

Of special importance for the comparison of different software reliability growth models is the question whether the fault exposure ratio is time-invariant. As equations (5) and (13) show, this is equivalent to stating that for a non-growing software code the per-fault hazard rate is constant. If there were a structure of K or ϕ in time, a software reliability model allowing for their time-dependence should be considered.

Malaiya et al. [32, 33] show that if the detectability d_i of fault i, defined as the probability that a random input tests for this fault, is constant but not the same for each fault the fault exposure ratio can be expected to decrease in time. With T_S being the average execution time needed for the execution of a single input the time until the i^{th} fault causes a failure follows the distribution function

$$F_i(\tau) = 1 - \exp\left(-\frac{d_i}{T_S}\tau\right). \tag{14}$$

Accordingly, the expected number of failures experienced until time τ is

$$\mu(\tau) = \sum_{i=1}^{N} F_i(\tau) = N - \sum_{i=1}^{N} \exp\left(-\frac{d_i}{T_S}\tau\right), \tag{15}$$

and the failure intensity is

$$\frac{d\mu(\tau)}{d\tau} = \frac{1}{T_S}\sum_{i=1}^{N}\left[d_i \exp\left(-\frac{d_i}{T_S}\tau\right)\right]. \tag{16}$$

By equations (13) and (15),

$$\frac{d\mu(\tau)}{d\tau} = fK[N - \mu(\tau)] = fK\sum_{i=1}^{N} \exp\left(-\frac{d_i}{T_S}\tau\right). \tag{17}$$

Equating the right hand side of (16) and (17), K can be expressed as

$$K(\tau) = \frac{1}{fT_S}\frac{\sum_{i=1}^{N}\left[d_i \exp\left(-\frac{d_i}{T_S}\tau\right)\right]}{\sum_{i=1}^{N}\exp\left(-\frac{d_i}{T_S}\tau\right)}. \tag{18}$$

Only if all N faults had the same detectability $\overline{d}$ the fault exposure ratio would be constant in time. With different detectabilities $K(\tau)$ decreases as testing proceeds. In contrast to the Musa basic execution time model the ratio of failure intensity after some testing has been done to the failure intensity at the beginning of testing should be lower than equation (13) suggests, because the faults remaining are harder to find. In a refined model, the slope of the mean value function is higher at the outset near $\tau = 0$, but decreases faster. Malaiya et al. [32, 33] suggest to approximate the behaviour of the fault exposure ratio by

$$K(\tau) = \frac{K(0)}{1 + a\tau} \tag{19}$$

where a depends on the detectability profile of the faults in the application under test.

Examination of data from real projects yields the following results [32, 33]:

1. As testing proceeds and the fault density $D(\tau) = \frac{N-\mu(\tau)}{I_S Q_x}$ decreases within a testing project the fault exposure ratio decreases indeed. However, at a fault density of about 2 to 4 faults per KLOC (thousand lines of code) a minimum is reached. For smaller fault densities, $K(D)$ starts to rise again. Of course, there are variations of this behavior for different projects, and there is considerable noise in the data.

2. In a diagram depicting for several projects the fault exposure ratio estimated using Musa's basic execution time model in dependence on the average fault density of the respective application under test a similar curve is obtained. $K(D)$ takes its minimum at about 2 faults per KLOC.

Since the fault density decreases as testing proceeds, the behavior of $K(D)$ implies that $K(\tau)$ and the per-fault hazard rate $fK(\tau)$ also have the shape of a so-called bathtub curve. In the field of hardware reliability, such a hazard rate is often assumed for systems or components [30]: Due to "infant mortality" the hazard rate starts out at high values at the beginning of the lifetime before it stabilizes at an approximately constant value during the operational phase; wear-out is responsible for an increasing hazard rate at the end of the lifetime.

However, one has to bear in mind that $fK(\tau)$ is the hazard rate not of the entire application, but of one individual fault. For a Poisson process model the program hazard rate $\Delta\tau$ time units after the last failure occurrence at τ_{i-1} is equal to the failure intensity at $\tau_{i-1} + \Delta\tau$ [41, p. 259]:

$$z(\Delta\tau \mid \tau_{i-1}) = \lambda(\tau_{i-1} + \Delta\tau) \tag{20}$$

In the context of what could be called a "refined Musa basic execution time model with time-varying fault exposure ratio" the program hazard rate becomes

$$z(\Delta\tau \mid \tau_{i-1}) = fK(\tau)[N - \mu(\tau)], \tag{21}$$

the product of the per-fault hazard-rate $fK(\tau)$ and the expected number of faults remaining in the software code. Due to the second factor the program hazard rate tends to decrease in time. With the bathtub-shaped $fK(\tau)$ the decrease is even amplified at the outset of testing until $K(\tau)$ reaches its minimum. In contrast the trend is partly offset from then on. The fewer faults there are in the software the more dangerous each individual fault becomes. One possible explanation is that testing gets relatively more efficient than what the assumption of random testing implies [33]: the testers try to avoid the execution of redundant test cases and specifically test those areas of the software that have not been executed before or that are still supposed to be error-prone.

To model the observed behavior of K, equation (19) should be modified. Malaiya et al. [32, 33] suggest to extend it by a factor that leads to an increasing K for low fault densities:

$$K(\tau) = \frac{K(0)}{1 + a\tau} \frac{N}{N - \mu(\tau)} \tag{22}$$

As noted before, Musa introduced the constant testing compression factor C - defined as the ratio of execution time required in the operational phase to execution time required in

the test phase [41, p. 178] - to take care of the higher efficiency of testing as compared to normal use of the software in a generalization of his basic model [37]. It seems necessary to let C increase with time, because the fraction of redundant executions gets larger in the operational phase the more and more operations of the software have already been used. Still, the concept of C can be interpreted as a first attempt to account for pretty much the same influences as the factor $\frac{N}{N-\mu(\tau)}$ in the fault exposure ratio (22) proposed by Malaiya et al.

3.6 Musa-Okumoto logarithmic model

Interestingly, the fault exposure ratio in equation (22) leads to another standard software reliability growth model: The failure intensity

$$\frac{d\mu(\tau)}{d\tau} = f\frac{K(0)}{1+a\tau}\frac{N}{N-\mu(\tau)}[N-\mu(\tau)] = fK(0)N\frac{1}{1+a\tau} \tag{23}$$

belongs to the mean value function

$$\mu(\tau) = \frac{fK(0)N}{a}\ln(1+a\tau). \tag{24}$$

This is the functional form of the Musa-Okumoto logarithmic model [42] :

$$\mu(\tau) = \beta_0 \ln(1+\beta_1\tau). \tag{25}$$

Like Musa's basic execution time model the "logarithmic Poisson execution time model" (as it was originally dubbed) by Musa and Okumoto is based on failure data measured in execution time. Its assumptions are as follows [12, 41, 42]:

1. At time $\tau = 0$ no failures have been observed, i.e. $P(M(0) = 0) = 1$.
2. The failure intensity decreases exponentially with the expected number of failures observed, i.e. $\lambda(\tau) = \beta_0\beta_1 \exp\left(-\frac{\mu(\tau)}{\beta_0}\right)$, where $\beta_0\beta_1$ is the initial failure intensity and β_0^{-1} is dubbed failure intensity decay parameter.
3. The number of failures observed by time τ, $M(\tau)$, follows a Poisson process.

As the derivation of the Musa-Okumoto logarithmic model by the fault exposure ratio has shown, the exponentially decreasing failure intensity implies that the per-fault hazard rate has the shape of a bathtub curve.

Expressing the fault exposure ratio of the logarithmic model not in terms of time but in terms of the fault density [26, 27] can be the starting-point for a physical interpretation of the parameters β_0 and β_1 [8, 31] - see section 4.1.1.

3.7 Enhanced Non-Homogeneous Poisson Process (ENHPP) model

Another aspect of testing used to explain why the per-fault hazard rate (and, consequently, the fault exposure ratio) should be time-dependent is the development of test coverage. Test coverage $c(t)$ can be defined as follows [17]:

$$c(t) = \frac{\text{Number of potential fault sites sensitized through testing by time } t}{\text{Total number of potential fault sites under consideration}} \tag{26}$$

Potential fault sites are structural or functional program elements that may contain faults. Specifically, code constructs like blocks, decisions, c-uses and p-uses [22] may be considered as potential faults sites. If the number of fault sites is large, $c(t)$ can be regarded as a continuous, non-decreasing function.

It is obvious that the rate at which failures occur during testing depends on the testers' ability to execute test cases that sensitize fault sites not tested before. The hazard rate of the faults remaining in the software at time t can also be viewed as the danger of them being detected.

In 1996 Gokhale, Philip, Marinos and Trivedi [17] introduced the enhanced non-homogeneous Poisson process (ENHPP) model as a unifying framework for finite failure NHPP models. Its assumptions are as follows:

1. The N faults inherent in the software at the beginning of testing are uniformly distributed over the potential fault sites.
2. At time t the probability that a fault present at the fault site sensitized at that moment causes a failure is $c_d(t)$.
3. The development of test coverage as a function of time is given by $c(t)$ with $\lim_{t\to\infty} c(t) = k \leq 1$.
4. Whenever a failure has occurred, the fault that caused it is removed instantaneously and without introducing any new fault into the software.

The expected portion of inherent faults that have caused a failure by time t depends on the development of coverage and on the time-varying activation probability $c_d(t)$:

$$\mu(t) = N \int_0^t c_d(x) c'(x) dx \tag{27}$$

If the activation probability is a constant value c_d over time, then equation (27) simplifies to

$$\mu(t) = c(t)\tilde{N}, \tag{28}$$

where $\tilde{N} = N c_d \leq N$ is the number of faults expected to have been exposed at full coverage. The failure intensity can be expressed as

$$\frac{d\mu(t)}{dt} = c'(t)\tilde{N} = \frac{c'(t)}{1-c(t)}[\tilde{N} - c(t)\tilde{N}] = \frac{c'(t)}{1-c(t)}[\tilde{N} - \mu(t)]. \tag{29}$$

Equation (29) is a generalization of equation (5) with respect to two elements:

1. Not all faults in the software can necessarily be detected, even if full coverage has been obtained. However, c_d and N cannot be separately identified; it is only possible to estimate the number of detectable faults, $\tilde{N}$.

2. The per-fault (i.e. per-detectable-fault) hazard rate may change over time. Its variation is explained by the development of test coverage achieved. If $c(0) = 0$ and $\lim_{t\to\infty} c(t) = 1$, then $\frac{c'(t)}{1-c(t)}$ can be interpreted as the hazard rate $z_c(t)$ of the distribution function of the time to full coverage.

By plugging different distributions for $c(t)$ into (29) various standard software reliability growth models can be obtained. Specifically, Gokhale et al. [17] discuss the following distributions:

1. The hazard rate of the exponential function $c(t) = 1 - \exp(-\phi t)$ is the constant ϕ. Assuming this form for the growth of coverage in time obviously leads to the Goel-Okumoto model.

2. By using the Weibull function $c(t) = 1 - \exp(-\phi t^\gamma)$ with hazard rate $z_c(t) = \phi\gamma t^{\gamma-1}$ a model dubbed "Goel generalized NHPP model" [15], "generalized Goel-Okumoto model" [17] or simply "Weibull model" [12] is obtained. This model should not be mistaken for the Goel-Okumoto model extended by a Weibull testing effort (cf. section 3.3), in which the constant hazard rate ϕ of the Goel-Okumoto model is not *replaced* with the *hazard rate* of the Weibull distribution but *augmented* by the Weibull *density function*. The generalized Goel-Okumoto model can be used to model the often found situation that the failure rate initially increases at the outset of testing and then decreases; however, it also includes the exponential Goel-Okumoto model as a special case for $\gamma = 1$.

3. For the S-shaped coverage function $c(t) = 1 - (1 + \phi t)\exp(-\phi t)$ with hazard rate $z_c(t) = \frac{\phi^2 t}{1+\phi t}$ the resulting software reliability model is the one introduced by Ohba [45] for the number of faults removed when there is a delay between failure occurrence and fault removal.

Basically, any NHPP software reliability model with a finite number of failures to be experienced by infinite time can be fit into this framework. Mathematically, it is even possible to determine functions $c(t)$ that lead to NHPP models of the infinite failure category. E.g., the Musa-Okumoto logarithmic model with $\frac{c'(\tau)}{1-c(\tau)} = \frac{fK(0)}{1+a\tau}\frac{N}{N-\mu(\tau)} = \frac{fK(0)}{1+a\tau}\frac{N}{N-Nc(\tau)} = \frac{fK(0)}{1+a\tau}\frac{1}{1-c(\tau)}$ can be obtained by choosing $c(\tau) = \frac{fK(0)}{a}\ln(1 + a\tau)$. However, this $c(\tau)$ cannot be regarded as a test coverage function, because $\lim_{\tau\to\infty} c(\tau) = k \leq 1$ is not satisfied.

While the merit of the ENHPP model lies in the comparison and unification of various software reliability growth models based on an equation of the form $\frac{d\mu(t)}{dt} = \psi(t)[N - \mu(t)]$, the interpretation of $\psi(t)$ as solely depending on the time-varying test coverage seems to neglect other aspects. Test coverage is undoubtedly one driver of the failure occurrences. Experiments by Wong, Horgan, London and Mathur indicate that the correlation between block coverage and fault detection effectiveness is higher than the correlation between the number of test cases (and thus the testing time) and the fault detection effectiveness [56].

Similarly, minimizing the number of test cases in a test set while keeping the all-uses coverage constant caused only relatively small reductions in the fault detection effectiveness [57]. However, at different points in time the same amount of additional test coverage has not to be expected to yield the same number of failures. A study of different testing techniques undertaken by Hamlet and Taylor [21] shows that sampling from domains of the application under test which are likely to be failure-prone rather than trying to attain extensive coverage seems to be a promising strategy for efficiently detecting faults. Moreover, it can easily be proved that after full coverage has been obtained there may still remain faults in the software that have not yet caused a failure. This means that even without any gain in additional coverage during testing, failures may occur. Therefore, test coverage cannot be the only driver of the expected number of cumulative failures.

3.8 Block coverage model

Piwowarski, Ohba and Caruso [48] formulated a model for block coverage in dependence of the number of test cases executed during functional testing, which can easily be extended to become a model of the number of failures experienced in terms of time. The model is based on the following assumptions:

1. The program under test consists of G blocks.
2. Per test case p of these blocks are sensitized on average.
3. The p blocks are always chosen from the entire population. The fact that a block has already been sensitized does not diminish its chances of being sensitized in future.

Let the random variable ΔQ_i denote the number of blocks newly executed by the i^{th} test case. According to the assumptions it follows a hypergeometric distribution with discrete probability function

$$P(\Delta Q_i = \Delta q_i \mid q_{i-1}) = \frac{\left(\begin{array}{c} G - q_{i-1} \\ \Delta q_i \end{array} \right) \left(\begin{array}{c} q_{i-1} \\ p - \Delta q_i \end{array} \right)}{\left(\begin{array}{c} G \\ p \end{array} \right)}, \tag{30}$$

where $q_{i-1} = \sum_{j=1}^{i-1} \Delta q_j$ is the number of blocks actually covered by the first $(i-1)$ test cases. The numerator consists of the product of the number of possible ways to choose Δq_i new blocks out of $G - q_{i-1}$ blocks uncovered so far and the number of possible ways to choose the remaining $p - \Delta q_i$ blocks out of the q_{i-1} blocks sensitized before. This is divided by the number of ways in which p blocks can be picked out of the G blocks the program consists of.

The expected value of ΔQ_i given q_{i-1} is therefore [44, p. 177]

$$E(\Delta Q_i \mid q_{i-1}) = \frac{p}{G}[G - q_{i-1}]. \tag{31}$$

If the actual value of the random variable Q_{i-1} has not yet been observed, the unconditional expected value of ΔQ_i has to be used:

$$E(\Delta Q_i) = \frac{p}{G}[G - E(Q_{i-1})] \tag{32}$$

If the number of test cases and the number of blocks G is very large, this equation can be continuously approximated by

$$\frac{d\xi(i)}{di} = \frac{p}{G}[G - \xi(i)], \tag{33}$$

where $\xi(i) = E(Q_i)$, because $d\xi(i) \to 0$ and therefore $\xi(i) \to \xi(i-1)$ as $di \to 0$. Dividing both sides by G yields the analogue differential equation for the expected block coverage $\kappa(i) = \frac{\xi(i)}{G} = E\left(c(i)\right)$:

$$\frac{d\kappa(i)}{di} = \frac{p}{G}[1 - \kappa(i)], \tag{34}$$

Comparing equations (5) and (34) reveals that the relationship between the number of test cases executed and the expected block coverage has the form of the Goel-Okumoto model. Using three additional assumptions, equation (34) can be transformed into a model of the number of failures experienced by time t, i.e. into a classical software reliability model. The assumptions are as follows:

4. The N faults inherent in the software at the beginning of testing are uniformly distributed over the blocks [48].

5. A fault in the software code causes a failure the first time the block in which the fault is located is sensitized [48]. This fault is removed instantaneously and without introducing any new fault into the software.

6. The test cases are approximately equally sized; the average time needed for execution of one test case is t_t. Therefore, the current time in terms of the number of test cases executed so far is $t(i) = i \cdot t_t$.

Given assumption 6, equation (34) can also be expressed in terms of time:

$$\frac{d\kappa(t)}{dt} = \frac{d\kappa(t(i))}{di}\frac{di}{dt} = \frac{p}{G}[1 - \kappa(t(i))]\frac{1}{t_t} = \frac{p}{t_t \cdot G}[1 - \kappa(t)] \tag{35}$$

Using assumptions 4 and 5, the failure intensity function is obtained:

$$\frac{d\mu(t)}{dt} = N\frac{d\kappa(t)}{dt} = N\frac{p}{t_t \cdot G}[1 - \kappa(t)] = \frac{p}{t_t \cdot G}[N - \mu(t)] \tag{36}$$

Due to the nature of assumptions 4 to 6 the behavior of the expected block coverage in equation (34) has obviously led to the classical Goel-Okumoto model. As a comparison between equation (35) and equation (36) shows, the time-invariant per-fault hazard rate $\phi = \frac{p}{t_t \cdot G}$ is nothing but the hazard function of $\kappa(t)$. This means that equation (36) can also be written as

$$\frac{d\mu(t)}{dt} = \frac{\kappa'(t)}{1 - \kappa(t)}[N - \mu(t)]. \tag{37}$$

This is the form of the ENHPP model in equation (29) with $c_d = 1$; the only difference is that the development of coverage in time is not a deterministic function $c(t)$, but it follows a stochastic process with expected value function $\kappa(t)$. This derivation shows that the special

case of a time-invariant per-fault hazard rate in equation (29) is obtained for the (rather inefficient) sampling of fault sites with replacement. Furthermore, this hazard rate can be interpreted as the percentage of fault sites newly or repeatedly sensitized per test case (i.e. the probability that a certain fault site is chosen) divided by the average duration of one test case execution. Therefore, it is the percentage of sensitized fault sites per time unit.

3.9 Hypergeometric model for systematic testing

Building on earlier work [50, 55], Rivers [49] and Rivers and Vouk [51] derived a hypergeometric model 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, blocks, paths, c-uses, p-uses, etc. The authors also consider the number of test cases executed and even execution time or calendar time.

The model is based on the following assumptions:

1. For the program under test there are G constructs to be executed.
2. At the i^{th} stage of testing Δq_i constructs are covered. Δq_i is an exogenous variable.
3. Constructs that have been covered are removed from the population. Before the i^{th} stage there are $G - q_{i-1}$ constructs remaining with $q_{i-1} = \sum_{j=1}^{i-1} \Delta q_j$.
4. The N faults inherent in the software at the beginning of testing are uniformly distributed throughout the code.
5. Even when all G constructs have been tested not all faults have necessarily caused a failure. This means that there are faults which are not detectable under a given testing strategy. On the other hand, there may be more constructs (e.g. p-uses or test cases) that lead to the detection of the same fault. The factor $g_i \equiv g(\frac{Q_i}{K})$ is introduced to relate the number of faults remaining in the software at the beginning of the i^{th} testing stage to the number of constructs whose execution causes the activation of one of the faults. g_i therefore accounts for the "visibility" of the faults. Since it also depends on the testers' choice of constructs to be considered in the first place, Rivers and Vouk call it "testing efficiency".
6. If one of those constructs activating a fault is executed a failure occurs. The fault is removed instantaneously and without introducing any new fault into the software.
7. The order of test case execution is ignored.
8. There is a minimal coverage $c_{min} = \frac{q_{min}}{G} > 0$ obtained by the execution of a typical test case. μ_{min} is the number of failures to be expected at this minimal feasible coverage. If some of the constructs cannot be executed at all, there is also a maximum feasible coverage $c_{max} = \frac{q_{max}}{G}$.

According to these assumptions the probability that the random variable ΔM_i - the number of faults detected at the i^{th} stage of testing - takes value Δm_i is

$$P(\Delta M_i = \Delta m_i \mid m_{i-1}) = \frac{\left(\begin{array}{c} g_i(N - m_{i-1}) \\ \Delta m_i \end{array} \right) \left(\begin{array}{c} [G - q_{i-1}] - [g_i(N - m_{i-1})] \\ \Delta q_i - \Delta m_i \end{array} \right)}{\left(\begin{array}{c} G - q_{i-1} \\ \Delta q_i \end{array} \right)} \tag{38}$$

for $\Delta m_i \in \mathbb{N}$ and $\Delta m_i \leq \Delta q_i$. 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 $[G - q_{i-1}] - [g_i(N - m_{i-1})]$ 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 $G - q_{i-1}$ uncovered statements.

The expected value of ΔM_i conditional m_{i-1} is [44, p. 177]

$$E(\Delta M_i \mid m_{i-1}) = g_i(N - m_{i-1}) \frac{\Delta q_i}{G - q_{i-1}} = g_i(N - m_{i-1}) \frac{\Delta c_i}{1 - c_{i-1}}, \tag{39}$$

where $\Delta c_i = \frac{\Delta q_i}{G}$ is the coverage gained at stage i and $c_{i-1} = \frac{q_{i-1}}{G}$ is the coverage at the beginning of that stage.

Likewise, the unconditional expected value of ΔM_i is

$$E(\Delta M_i) = g_i[N - E(M_{i-1})] \frac{\Delta c_i}{1 - c_{i-1}}. \tag{40}$$

There seem to be two shortcomings of River's formulation of the problem.[4] First of all, the probability mass of the hypergeometric distribution (38) is distributed among the integer values from zero to Δq_i. It is not possible to find more faults than the number of constructs covered. While this may be no problem if the constructs measured are p-uses or some other metric dividing the program in a lot of small subdomains, it is unrealistic and restricting if the constructs studied are highly aggregated like test cases. Let us assume that each metric for which $\Delta m_i > \Delta q_i$ seems possible can theoretically be replaced by another, more precise metric with $G^* >> G$ being the number of constructs in the program. If for each stage its relative contribution to overall coverage according to the 'new' metric was the same as its relative contribution according to the 'old' metric, then $\frac{\Delta q_i^*}{G^*} = \frac{\Delta q_i}{G} = \Delta c_i \ \forall i$ and - as a consequence of that - $\frac{q_i^*}{G^*} = \frac{q_i}{G} = c_i \ \forall i$. (This comes close to stating that the different constructs of the 'old' metric were equally sized.) Substituting G^* for G and q_i^* for q_i in (38) clearly yields an altered probability distribution. However, its expected value is

$$\begin{aligned} E(\Delta M_i) &= g_i[N - E(M_{i-1})] \frac{\Delta q_i^*}{G^* - q_{i-1}^*} = g_i[N - E(M_{i-1})] \frac{\frac{\Delta q_i^*}{G^*}}{1 - \frac{q_{i-1}^*}{G^*}} = \\ &= g_i[N - E(M_{i-1})] \frac{\Delta c_i}{1 - c_{i-1}}, \end{aligned} \tag{41}$$

[4] The following discussion and the derivation of the continuous form are taken from [19].

just the same as equation (40).

A second restriction of (38) is that the testing efficiency g_i has to be chosen such that $g_i(N-m_{i-1})$ is an integer value. Therefore, g_i cannot easily be assigned a continuous function of test coverage. To resolve this problem, in equation (38) $g_i(N-m_{i-1})$ should be replaced by a discrete random variable X_i with probability function $f(x_i)$ satisfying $f(x_i) = 0 \; \forall \; x_i \notin \mathbb{N}$ and $E(X_i) = g_i(N - m_{i-1}) \; \forall i$. The resulting equation

$$P(\Delta M_i = \Delta m_i \mid m_{i-1}) = \frac{\begin{pmatrix} X_i \\ \Delta m_i \end{pmatrix} \begin{pmatrix} [G^* - q^*_{i-1}] - X_i \\ \Delta q^*_i - \Delta m_i \end{pmatrix}}{\begin{pmatrix} G^* - q^*_{i-1} \\ \Delta q^*_i \end{pmatrix}} \tag{42}$$

still has expected value (41).

Equation (41) can be used as a starting-point for deriving the mean value function $\mu(c)$, i.e. the expected cumulative number of faults found when $100 \cdot c$ % coverage has been achieved, because $E(\Delta M_i) = \Delta E(M_i) = \Delta \mu_i$.

Since all q_i as well as all q^*_i have to take integer values, (40) and (41) are only defined for Δc_i and c_{i-1} being a multiple of G^{-1} or G^{*-1}, respectively. However, for $G^* >> G$ or $G^* \to \infty$ the change in coverage attained at any stage may approach zero, because $\frac{1}{G^*} \to 0$. At the limit the coverage measure c can take any value between zero and one.

It is therefore possible to transform the difference equation (41) into a differential equation by considering $\Delta c_i \to dc_i \to 0$ and - as a consequence - $c_i \to c_{i-1}$ and $\mu_i \to \mu_{i-1}$. For any coverage value $c \in (0, 1)$,

$$d\mu(c) = g(c)[N - \mu(c)]\frac{dc}{1-c}. \tag{43}$$

From this a continuous approximation of the mean value function can be obtained [49, pp. 38 - 39]:

$$\mu(c) = N - (N - \mu_{min}) \exp\left(- \int_{c_{min}}^{c} \frac{g(\varsigma)}{1-\varsigma} d\varsigma \right). \tag{44}$$

c_{min} and μ_{min} denote the minimum meaningful coverage and the expected number of failures observed for this coverage, respectively (cf. assumption 8).

Rivers considers different functional forms of testing efficiency $g(c)$ and derives the mean value function for each of these special cases [49, pp. 40 - 42]:

1. Assuming a constant testing efficiency, i.e. setting $g(c) = \alpha$ yields

$$\mu(c) = N - (N - \mu_{min}) \left(\frac{1-c}{1-c_{min}} \right)^{\alpha}. \tag{45}$$

2. Using the linear testing efficiency $g(c) = \alpha(1-c)$ results in

$$\mu(c) = N - (N - \mu_{min}) \exp\left(-\alpha(c - c_{min})\right). \tag{46}$$

Clearly, this formulation for the dependence between the number of failures experienced and test coverage is analogous to the Goel-Okumoto model for the number of failure occurrences by time t. In addition to using a different exogenous variable the function has been translated by vector (c_{min}, μ_{min}).

If the coverage achieved was proportional to testing time t, i.e. $c(t) = \beta t$, the classical (translated) Goel-Okumoto model $\mu(t) = N - (N - \mu_{min}) \exp(-\phi(t - t_{min}))$ with $\phi = \alpha\beta$ and $t_{min} = \frac{c_{min}}{\beta}$ would be obtained.

3. In the case of the unimodal testing efficiency $g(c) = (\alpha\gamma q^{\gamma-1})(1 - c)$ the following mean value function is obtained:

$$\mu(c) = N - (N - \mu_{min}) \exp(-\alpha(c^\gamma - c_{min}^\gamma)). \tag{47}$$

It corresponds to the mean value function of the generalized Goel-Okumoto model mentioned in section 3.7. Again, the consideration of minimal meaningful coverage and number of failures leads to a translation by vector (c_{min}, μ_{min}). Assuming that coverage is proportional to testing time leads to its (translated) time-dependent formulation $\mu(c) = N - (N - \mu_{min}) \exp(-\phi(t^\gamma - t_{min}^\gamma))$, where $\phi = \alpha\beta^\gamma$ and $t_{min} = \frac{c_{min}}{\beta}$.

In Rivers' and Vouk's model, the number of constructs covered (and consequently the test coverage, too) is an exogenous variable. The mean value function is some function of this test coverage.

However, the mean value function can also be expressed in terms of time. Let $c(t)$ be a non-decreasing function of time with $\lim_{t\to\infty} c(t) \to k \leq 1$. Dividing both sides of equation (43) by dt then results in the extended model

$$\frac{d\mu(t)}{dt} = g(c(t))[N - \mu(t)]\frac{\frac{dc(t)}{dt}}{1 - c(t)} = \frac{c'(t)}{1 - c(t)} g(c(t))[N - \mu(t)]. \tag{48}$$

3.10 Conclusions

Comparison of the models discussed this chapter shows several similarities:

First of all, for some models (the ones of binomial type [41, p. 251]) the number of faults inherent at the beginning of testing is a fixed but unknown value u_0, whereas for other models (the ones of Poisson type [41, p. 251]) it is a random variable following a Poisson distribution with mean value N. Based on collected failure times the two types of models cannot be distinguished. Furthermore, if only some fraction c_d of the (expected) number of inherent faults can be detected, c_d and N (or u_0) cannot be separately identified. Only the (expected) number of detectable faults - $\tilde{u}_0$ or $\tilde{N}$ - can be estimated. In the following discussion ν denotes either u_0 or N or $\tilde{u}_0$ or $\tilde{N}$.

Moreover, many models can be written in the form $\frac{d\mu(t)}{dt} = \psi(t)[\nu - \mu(t)]$. The difference lies in the mathematical formulation of $\psi(t)$ as well as in its interpretation.

1. The first approach observes the testers' performance in causing the software to fail. It consists of two elements: Their ability to increase test coverage and their skill in

detecting faults in the parts of the code covered, which can also be viewed as the ability to achieve code coverage "in the right way".

The influence of both effects are contained in the extended Rivers-Vouk model (48). A step towards a unifying model is to account for a time-varying testing-effort in the time scale t^* via the transformation $t = W(t^*)$, which also leads to the inclusion of the term $w(t^*)$ on the righthand side. Moreover, test coverage c can be seen as a function of the number of test cases executed, i, which depends on the transformed time $W(t^*)$. Furthermore, by substituting $\kappa = E(c)$ for c it is possible to allow for both stochastic test coverage growth and deterministic test coverage growth, for which $\kappa = E(c) = c$. The following extended equation is obtained:

$$\frac{d\mu(W(t^*))}{dt^*} = w(t^*)\frac{\kappa'(i(W(t^*)))}{1-\kappa(i(W(t^*)))}g(\kappa(i(W(t^*))))[\nu - \mu(W(t^*))] \tag{49}$$

This model can be seen as the generalization of the Jelinski-Moranda model (4), the Goel-Okumoto model (5), the Jelinski-Moranda model with a time-varying testing-effort (7), the Goel-Okumoto model with a time-varying testing-effort (9), the ENHPP model (29), the original block coverage model (34), the extended block coverage model (36), the original Rivers-Vouk model (43) and the extended Rivers-Vouk model (48).

Of course, in the specific models not all elements of the generalized model are included. First of all, models may start out with an exogenous variable that is considered endogenous in the general model. In the original Rivers-Vouk model, for example, both calendar time and a time measure with constant testing-effort are not included, because the percentage of constructs tested (which is either the percentage of test cases executed or some white box test coverage) is the input variable. Furthermore, models may leave out stages in between. E.g., the ENHPP does not relate the time to the number of test cases but directly to a white box coverage measure. Moreover, the model by Rivers and Vouk is the only one to take into account testing-efficiency as an additional effect on the number of failures experienced besides the development of test coverage; however, testing-efficiency itself is some deterministic function of test coverage. Figure 1 shows the elements considered by the different models. While an X refers to an element included in the original model, (X) depicts an element incorporated in a model extension discussed in one of the last subsections.

This figure also helps to explain why special cases of the ENHPP and of the Rivers-Vouk model and the extended version of the block coverage model result in the Goel-Okumoto model, which directly relates time to the number of failures experienced in the specific form of equation (5):

- The ENHPP model with $\frac{c'(t)}{1-c(t)} = \phi$ implicitly assumes that the white box test coverage measure increases with time in a fashion analogous to the Goel-Okumoto model. Since in the framework of the ENHPP the number of failures is considered proportional to test coverage, the classical Goel-Okumoto model is obtained.

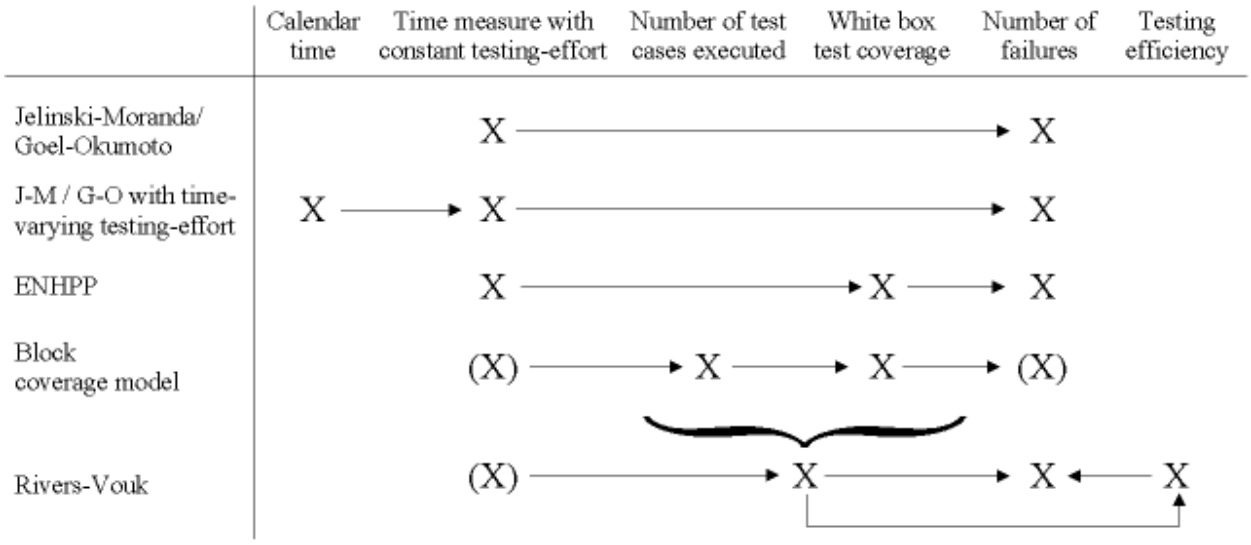

Figure 1: *Elements of the generalized model included in different models*

- The block coverage model shows that it is the highly inefficient testing strategy of sampling blocks with replacement that leads to the function of test coverage in terms of test cases executed which has the form of the Goel-Okumoto model. The specific ENHPP model discussed above can therefore be thought of as implicitly assuming sampling with replacement and a constant duration of each test case execution.
- The Rivers-Vouk model does *not* consider the inefficient sampling with replacement, but the highly efficient sampling out of constructs not covered before, which avoids any redundant execution. However, the linear testing efficiency $g(c) = \alpha(1-c)$ - with an efficiency approaching zero as coverage approaches one - compensates this advantage and gives rise to the Goel-Okumoto model.

2. The second approach sees the number of failure experienced as a result of the tendency of the faults to show themselves, which is expressed by the fault exposure ratio K. As shown above, K can be thought to be influenced by two effects: the change in the detectabilities of the faults and the development of the testers' efficiency. Since the fault exposure ratio is the ratio between the per-fault hazard rate and the linear execution frequency f, it is necessary to determine f if not only relative values of K are to be discussed.

Both approaches consider similar phenomenons from different points of view. However, since the interpretations of the majority of models sees the testers in an active role, more models fit into the first framework.

The aim of the unifying view on software reliability growth models developed in this chapter was to clarify the different effects driving the number of failures observed during testing. On the one hand, using the generalized model (49) it is possible to specify functional forms for each of these effects and build up a new model in that way. On the other hand, existing

models that fit in the framework can be analyzed and chosen if their interpretation seems to agree with the real-world conditions of the testing process. For example, as shown above the Goel-Okumoto either implies a highly inefficient testing strategy or a highly inefficient testing efficiency. If neither of them seem to be that poor in the testing process studied, the Goel-Okumoto probably should not be the model of choice.

4 Linking additional information to software reliability growth models

One of the goals of the PETS project is to include information about the software development process and the software testing process in one or more software reliability growth models. In this chapter different existing approaches for linking additional data to software reliability growth models are studied.

4.1 Physical interpretation of model parameters

For some software reliability growth models it is possible to relate their parameters to characteristics of the software development and the debugging process. There are several advantages of this approach [31]:

1. The parameters of the software reliability growth model can be estimated when failure data have not yet been collected. These so-called "static" estimates permit the "early prediction" [29] of software reliability before testing starts.

2. It is a well-known fact that there is considerable noise in connection with the observed inter-failure times. Especially at the early stage of testing, when only few failure data are available, the "dynamic" parameter estimates calculated using the test data can be unstable or even outside the region of meaningful values. If this happens, the static estimates can be substituted for the dynamic estimates.

3. While sometimes there exist equations for direct computation of the parameter values often numerical techniques have to be used. Generally, the convergence of such techniques will be faster if the starting values chosen are closer to the actual values. The static estimates might be a reasonable choice.

4. Since the parameters have an interpretation with respect to the software development or test process the effect of changes in these processes to software reliability growth can be studied. Moreover, the dynamic parameter estimates can be used for an assessment of process characteristics.

4.1.1 Physical interpretation of the parameters of the Musa basic model and the Musa-Okumoto model

Malaiya and Denton [31] studied the parameters of the Musa basic model and the Musa-Okumoto model.

For the Musa basic model their interpretation is the one given in sections 3.2 and 3.4: Since experience has shown that only about 5 per cent new faults are introduced during debugging of detected faults [41], N is approximately the number of faults inherent in the software at the beginning of testing. $\phi = fK$ is the per-fault hazard rate.

As for the logarithmic Musa-Okumoto model, Malaiya and Denton use earlier work by Li and Malaiya [26, 27] to derive the following interpretation of the parameters:

$$\beta_0 = I_S D_{min} \tag{50}$$

$$\beta_1 = f K_{min} \exp\left(\frac{D_0}{D_{min}} - 1\right), \tag{51}$$

where K_{min} is the minimal value of the bathtub-shaped fault exposure ratio, D_{min} is the fault density for which K attains its minimum and D_0 is the initial fault density. Therefore, β_0 represents the number of faults inherent in the software at that point of testing when the fault exposure starts to rise again, i.e. when testing becomes more efficient. Unfortunately, neither D_{min} nor K_{min} can be directly observed. Based on the studies by Musa et al. [41, pp. 121 - 124] and Malaiya et al. [32, 33] already cited in section 3.5, it seems appropriate to set K_{min} to about $1.4 \cdot 10^{-7}$ and D_{min} to a value between 2 and 4 defects per KLOC. For an initial fault density D_0 larger than 10 faults per KLOC, Malaiya and Denton [31] suggest to set $D_{min} = \frac{D_0}{3}$.

Since the parameter N of the Musa basic model is approximately the initial fault content $I_S D_0$, early prediction both of this model and the logarithmic Musa-Okumoto model requires the determination of the initial fault density D_0. An approach to this is described in the next section.

4.1.2 Relating the fault density to characteristics of the software and the software development process

Malaiya and Denton [31] introduce a factor multiplicative model for D_0:

$$D_0 = C \cdot F_{ph} \cdot F_{pt} \cdot F_m \cdot F_s, \tag{52}$$

where the factors on the right side have the following meaning:

- The phase factor F_{ph} takes into account that at the beginning of later test phases a lot of faults have already been removed. Depending on whether the fault density at the beginning of the unit testing, the subsystem testing, the system testing or the operation testing phase is to be estimated, F_{ph} takes value 4, 2.5, 1 or 0.35, respectively.

- It is obvious that the initial fault density of the program under test depends on the abilities of the people in the development team. To consider this, the programming team factor F_{pt} is set to 0.4 if the team's average skill level is high, 1 if it is average and 2.5 if it is low.

- The process maturity factor F_m is one of the few concepts indirectly relating software development process maturity information to software reliability growth models. It uses the condensed maturity level of the software developing organization according to the capability maturity model (CMM) developed at Carnegie Mellon University [46]. Malaiya and Denton suggest to set F_m to 1.5, 1, 0.4, 0.1 or 0.05 according to the attained CMM maturity level from 1 to 5.

- In a simple approach to model the dependence of initial fault density on the programming language used, F_s is defined as $F_s = 1 + 0.4a$, where a is the percentage of software code in assembly language.

- The constant of proportionality has to be individually estimated for each software developing company and should be determined using projects with characteristics similar to the ones of the project for which D_0 is to be estimated. According to Malaiya and Denton F_s ranges between 6 and 20 faults per KLOC.

Farr [12] mentions another factor multiplicative model developed by the Air Force's Rome Laboratory. It consists of factors like the application type, the development environment, several requirements and design representation metrics and a number of software implementation metrics. Again, for each factor a mapping of the ordinal or qualitative piece and information to an appropriate quantitative value has to be made.

4.2 Bayesian software reliability growth models

The models discussed in chapter 3 follow the frequentist view of statistics, in which model parameters are considered to be fixed but unknown. When data are available, the parameters can be estimated using techniques like the maximum likelihood estimation (MLE) (cf. [18, pp. 129 - 140] and [41, pp. 313 - 343]).

There are several disadvantages of this approach:

1. In the absence of failure data no information about the model parameters can be obtained. Therefore, predictions about the number of failures to be experienced by time t or the reliability of the software at different points in time are not possible. Approaches like the one described in section 4.1 are attempts to overcome this problem.

2. The maximum likelihood method does not make use of structural information that may be available in addition to the failure data collected. Even if the model parameters are estimated using known facts about the software and its development process, these estimates are in no way joined to the maximum likelihood estimates obtained from failure data. Basically, the analyst has to make a decision for one of the estimates, e.g. for the "early estimates" in case that MLE produces illegal or untrustworthy values [31].

3. Many studies have shown that MLE often results in unstable and illegal parameter estimates, especially when only relatively few failure data are available [8, 36].

4. Uncertainty about the estimates in frequentist approaches can only be expressed indirectly. The so-called "confidence interval" with a confidence level of $100\cdot(1\text{-}\alpha)\%$ for a parameter is *not* the range of values containing the true parameter value with the probability of $100\cdot(1\text{-}\alpha)\%$; since the parameter is assumed to be fixed but unknown it is either an element of the confidence interval, or it is not. The correct interpretation is rather that if the experiment (i.e. the testing of the software and the collection of failure data) was repeated very often, the confidence interval constructed can be expected to contain the true parameter value in approximately $100\cdot(1\text{-}\alpha)\%$ of the samples [18, p. 153]. However, this explanation is not very intuitive.

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 $p(y \mid \theta)$. 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 [14, p. 8]:

$$p(y) = \int p(y, \theta) d\theta = \int p(y \mid \theta) p(\theta) d\theta \tag{53}$$

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 $M(t_1)$, the number of failures observed by time t_1, and also its expected value $\mu(t_1)$, which is the mean value function at t_1. 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 [14, p. 8]

$$p(\theta \mid y) = \frac{p(y, \theta)}{p(y)} = \frac{p(y \mid \theta) p(\theta)}{\int p(y \mid \theta) p(\theta) d\theta}. \tag{54}$$

It can easily be seen that only the prior density of θ, $p(\theta)$, and the likelihood function $\mathcal{L}(\theta; y) = p(y \mid \theta)$ is used for obtaining the *posterior distribution* $p(\theta \mid y)$. 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 t_1, one can use $y = \{t_1\}$ and the prior distribution $p(\theta)$ to calculate $p(\theta \mid y)$ via equation (54) and predict the distribution of $M(t_1 + \Delta t)$, the number of failures experienced by $t_1 + \Delta t$, 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 [14, p. 9]:

$$p(\tilde{y} \mid y) = \int p(\tilde{y}, \theta \mid y) d\theta = \int p(\tilde{y} \mid \theta, y) p(\theta \mid y) d\theta = \int p(\tilde{y} \mid \theta) p(\theta \mid y) d\theta \tag{55}$$

A lot of different Bayesian software reliability growth models have been proposed over the last decades [53, 54]. Some of them can be considered Bayesian extensions of frequentist models assuming certain prior distributions for their parameters. Although it was not formulated in that way, the Littlewood-Verrall model [28], 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 u_0 [25]. 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.[5]

In the past, the prior distributions were often chosen as to ensure mathematical tractability, e.g. in form of the "conjugate prior distributions" [14, pp. 36 - 38]. The speed with which today's personal computers can execute Markov Chain Monte Carlo methods, like the Metropolis-Hastings algorithm [5] and specifically the Gibbs sampler [4], gives the analyst more freedom in selecting the prior densities and also enables him to analyze more complex models.

Linking maturity information, information from models like in section 4.1 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 [52] 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. Campodónico and Singpurwalla [2, 3] 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.

[5] Originally, the Goel-Okumoto model was not seen in a Bayesian context; the parameters were calculated using MLE [16]. The Littlewood-Verrall model was a parametric empirical Bayes model, because some of its parameters were treated as fixed, unknown quantities and also estimated by the maximum likelihood method [34]. For both of them a fully Bayesian analysis is also possible [34, 54].

4.3 Parametric regression models for survival times

Models trying to explain the time until some event of interest occurs by a set of covariates have been applied to problems in bio-statistics, social sciences and economics [10], but also to the field of hardware reliability [35, ch. 17]. However, they have hardly been employed in connection with software reliability data.

The covariates used can either be time-invariant or time-varying. Models for both cases are discussed in the following two subsections.

4.3.1 Models with time-invariant covariates

There are two different approaches for including time-invariant covariates in a survival model:

1. Transformation models [10] are constructed in analogy to classical regression models: Some transformation of the survival time T of one unit is considered to depend on a function h of the vectors of covariates $\mathbf{x}$ and regression parameters $\boldsymbol{\beta}$ and on a disturbance term ϵ multiplied by a scale parameter σ:

$$g(T) = h(\mathbf{x}; \boldsymbol{\beta}) + \sigma\epsilon \tag{56}$$

   A simple choice for $h(\mathbf{x}; \boldsymbol{\beta})$ is the linear function $\mathbf{x}'\boldsymbol{\beta}$. Since $\mathbf{x}'\boldsymbol{\beta}$ and ϵ can take negative values if no particular attention is given to their specification while only positive values of T are reasonable, $g(T)$ ought to map T from $\mathbb{R}_0^+$ to $\mathbb{R}$. In most practical applications $g(T) = \ln(T)$. Models with the structure

$$\ln(T) = \mathbf{x}'\boldsymbol{\beta} + \sigma\epsilon \tag{57}$$

   are called "accelerated failure time models".

   The functional form of the hazard rate $z(t)$ of the unit depends on the distribution of ϵ. Among those assumed are the extreme value, the logistic and the standard normal distribution [10].

2. The semi-parametric proportional hazards model [10, 35, pp. 455 - 458] expresses the hazard rate of a unit as the product of some unspecified, time-varying baseline hazard rate $z_0(t)$ and a positive term depending on the covariates $\mathbf{x}$:

$$z(t \mid \mathbf{x}) = z_0(t)h(\mathbf{x}; \boldsymbol{\beta}) \tag{58}$$

   The reason for its name is that the hazard rate of a unit with covariate values $\mathbf{x_1}$ at time t is proportional to the hazard rate of another unit with covariate values $\mathbf{x_2}$ at the same point in time, i.e. their ratio is some constant c:

$$\frac{z(t \mid \mathbf{x_1})}{z(t \mid \mathbf{x_2})} = \frac{z_0(t)h(\mathbf{x_1}; \boldsymbol{\beta})}{z_0(t)h(\mathbf{x_2}; \boldsymbol{\beta})} = \frac{h(\mathbf{x_1}; \boldsymbol{\beta})}{h(\mathbf{x_2}; \boldsymbol{\beta})} = c \tag{59}$$

   This means that the proportional influence of a covariate does not change in time. For some applications this property of the model may be too restrictive, but the model can be refined by defining different strata [10].

In the proportional hazard model by Cox [10, 18, pp. 997 - 999] $h(\mathbf{x};\boldsymbol{\beta})$ is chosen to be $\exp(\mathbf{x}'\boldsymbol{\beta})$:

$$z(t \mid \mathbf{x}) = z_0(t) \exp(\mathbf{x}'\boldsymbol{\beta}) \tag{60}$$

It is possible to show that the linear model of $\ln(T)$ in $\mathbf{x}$ with disturbances following the extreme value distribution, discussed as one of the transformation models, is a special case of the Cox proportional hazard model [10]. Its hazard rate can be written as

$$z(t \mid \mathbf{x}) = \frac{1}{\sigma} t^{\frac{1}{\sigma}-1} \exp(\mathbf{x}'\boldsymbol{\beta}^*), \tag{61}$$

where $\boldsymbol{\beta}^* = -\frac{\boldsymbol{\beta}}{\sigma}$; this is of the same form as equation (60). However, while the functional form of the hazard rate in the transformation models is fully specified - which can also be seen from equation (61) - the semi-parametric proportional hazard rate models normally do not specify the baseline hazard rate $\lambda_0(t)$. Using the method of maximum partial likelihood estimation, the vector $\boldsymbol{\beta}$ can be estimated without requiring estimation of $\lambda_0(t)$ [10, 11].

When applying these models to the field of software reliability, the question arises for what units the survival time is modelled. In analogy to medicine, where a defined number of patients are studied, one might consider each software fault a unit of its own and model the time until it causes a failure. However, in divergence with a medical survey, the total number of faults is not known. Therefore, after i failures have occurred, we do not know how many faults are remaining in the software. In formulating the likelihood function the term for units with "censored survival times", i.e. with times to failure exceeding the period of time during which the software has been tested, cannot be specified. Moreover, the objective of including exogenous variables is to model the different behavior of units. Therefore, it would be necessary to classify each fault with respect to factors that may influence the time until its activation. Probably this categorization needs information - e.g. about the structure of the respective software component - from the programming team. But it remains questionable whether the faults could be classified appropriately even then. In any case, for those faults that have not yet been detected the values of such covariates are not available.

Another approach to the problem could be the modelling not of the time to failure of one fault, but of the application itself. Indeed, this is the strategy Pham [47, pp. 210 - 220] proposes (see below section 4.3.2). However, since the covariates then just have influence on the behavior of the entire program and since only one application is studied at a time, e.g. in the Cox proportional hazard rate model, the baseline hazard rate and the factor depending on exogenous variables cannot be separately identified. Therefore it seems that such a model for the program hazard rate only makes sense when the covariates are time-varying. Such a model is discussed in the next section.

4.3.2 Models with time-varying covariates

Besides the time-invariant covariates, like information about the software development environment used for one release, there are variables that may change during the testing stage, while the failure data are collected (e.g. variables related to the testing-efficiency). Both the

accelerated failure time models and the proportional hazard rate models can be extended to include such variables. Since a continuous model of the covariate process in time may cause difficulties for the numerical analysis, it should be considered changing at discrete points in time. Moreover, there are problems in connection with internal variables, i.e. covariates that are generated by the units under study themselves. So far these problems have not been entirely investigated by researchers. However, it seems that the software faults that have not been detected do not influence the way in which the software is tested. Therefore, variables representing aspects of the testing process probably are external. (For more detailed information on these issues see [10, pp. 331 - 338].)

Pham [47, pp. 210 - 220] describes the "enhanced proportional hazard Jelinski-Moranda" (EPJM) model as an application of the Cox proportional hazard model to software reliability. However, some parts of this concept are in divergence with the proportional hazard model:

1. Probably due to the problems discussed in section 4.3.1 Pham does not model the hazard rate of a single fault but the program hazard rate $z(t_i \mid \mathbf{x_i}) = z(t_{i-1} + \Delta t \mid \mathbf{x_i})$. In opposition to what it is named, $\mathbf{x_i} = \mathbf{x}(t_i)$ does not contain the environmental factors of the i^{th} fault, but the covariates coming to effect between the $(i-1)^{st}$ and the i^{th} failure occurrence. Therefore, it is a mixture of the covariates for all the faults still remaining in the application at $t_{i-1} + \Delta t$.

2. The baseline hazard rate $z_0(t)$ does not remain unspecified, but it is assigned the functional form of the the program hazard rate in the Jelinski-Moranda model (1). This results in the equation

$$z(t_i \mid \mathbf{x_i}) = \phi[u_0 - (i-1)] \exp(\mathbf{x_i}'\boldsymbol{\beta}) . \tag{62}$$

Since Pham treats the application like one unit for which n events occur, in the likelihood function only the values of the density function at the failure times $\{t_1, t_2, ..., t_n\}$ have to be considered; there is no censoring [47, pp. 211 - 212]. However, this raises doubts about the possibility of employing maximum partial likelihood estimation as proposed by Pham [47, pp. 212 - 213]. For this technique the risk set, i.e. the collection of units that have not yet left the study due to the event or due to censoring, at each of the failure occurrences has to be known. If the entire application is considered to be the only unit, it is questionable whether the concept of a "risk set" is reasonable at all.

5 Different kinds of failure and testing data

Another issue is whether all failures and all testing-efforts can be considered within one software reliability growth model or whether different models should be used for subdomains of failures and testing intervals. The partitioning of the data would be necessary if the failure times in the subsets were created by different mathematical processes and if the factors that cause the dissimilarities could not be identified and included in an extended model. There are various criteria according to which the separation could be made. They are discussed in the following subsections.

5.1 Failures with different severities

As noted above, software reliability is the probability of failure-free operation of the piece of software in a defined environment for a specified duration. In this definition the *impact* of the deviation from user requirements is not taken into account. Therefore, all failure times are normally treated equally in software reliability calculations, regardless of their severity. Since these range from typographic errors to system crashes, it seems to be a drawback of software reliability growth models that no distinction is made. One proposition is to fit one separate software reliability growth model to the failures of each severity class. However, it has been noted that the smaller number of observations in the sets results in higher variances of the model parameter estimates and in broader confidence intervals for predictions of software reliability and failure intensity [40, p. 68]. This is an important aspect, because generally inter-failure times are already considerably noisy.

5.2 Current and past data from the same test cycle

Obviously, even for the failure data measured during the same test cycle (see below section 5.5), the same test phase (see below chapter 5.4) and the same type of testing (see below section 5.3) the last inter-failure times measured are more closely related to the current failure rate of the software under test than older observations. While practically all software reliability models use all failure data from one test cycle, a model developed by Schneidewind based on failure counts in equally-sized time periods either ignores the first $s-1$ failure counts or uses the cumulative number of failures in these $s-1$ periods as the first data point [12]. This approach should be considered if the (mathematical) process generating the times of failure occurrence changed gradually over time or if there were structural breaks within the test cycle, e.g. due to the removal of a fault blocking a huge part of the software code. Farr [12] notes that this method could also be used to eliminate learning curve effects. While this might indeed decrease the need for S-shaped software reliability growth models, it seems that it destroys valuable information like systematic changes in the testers' efficiency at the same time.

5.3 Failure and testing data from different types of testing

Since execution time τ, i.e. the CPU time used by the application under test, is not available at the SMEs participating at the PETS project, another measure of time ensuring an approximately constant testing effort had to be chosen. The test plan management tool *ProDok* developed by imbus AG has been enhanced by the Chair of Statistics and Econometrics of the University of Erlangen-Nuremberg to permit the logging of the time intervals in which the application under test is executed by a tester. This extension was named *Teddi* (*Te*st *d*ata *d*etermination and *i*nterpretation). Still, apart from the "normal" testing, i.e. the execution of the application according to the test specification, there are other types of software usage during testing which cannot be totally neglected, because failures potentially could occur; however, the failure rates might be quite different. The following types of testing have been identified:

- Test preparation:

  Sometimes a specified test case can only be executed after additional operations that ascertain that preconditions of the test case are met. E.g., in a customer database it is not possible to test the deleting of customer records if no customer record has yet been created. While an efficient ordering of test cases reduces the need for such preparing actions that are not test cases themselves, they may be necessary even then. Since all program usage puts the software under certain stress, failures potentially can occur during test preparation. However, because it is not directly aimed at detecting faults one might expect that the failure intensity is smaller than for "normal" testing.

- Failure reproduction:

  In order to help the development team in detecting the faults that caused a certain failure, the failure report should describe as precisely as possible the state of the application and the tester's actions that led to the failure occurrence. For being able to isolate the relevant information the tester sometimes has to repeat parts of a test case. This means that exactly the same operations are executed another time. Since the repeated occurrence of the same failure is not counted and since it is unlikely that a different failure is experienced, such time intervals should be logged separately.

- Retest of supposedly fixed bugs:

  After faults have been removed by the development team the testers are often asked to check whether the correcting actions have been successful. It may be expected that this re-execution of test cases does not produce any failure. However, some of the faults may still be in the software, if the developers were not able to reproduce the failure or find the defect that caused it. Furthermore, the fault correction may only have been partially successful, i.e. the set of inputs activating the fault has been reduced but not to size of zero. Moreover, it is possible that new faults have been spawned. Again, these testing intervals cannot be ignored, but the failure intensity may be supposed to be smaller than during "normal" testing.

- Automated testing:

  Test cases cannot only be executed manually, but also automatically with the help of test tools. The navigation through the application under test and the filling of information in dialogs can be expected to be faster if done by a test script rather than by a human. Therefore, the failure intensity should be higher than during manual testing. However, there are counteracting effects. First of all, the test tools themselves use CPU resources, which may slow down the application under test. Moreover, a test script can basically only test for the identity of observed and expected values. I.e., the programmer of the script has to specify which output variables or which parts of a dialog box to monitor and what to compare them with. A typographic error in a dialog, for example, will not be detected if no matching between the output and the orthographically correct text has been implemented. Furthermore, in order to determine or record the expected values the tester often has to execute the application manually at least once. If failures should occur, they are reported right away. This means that only a subset of software faults can cause the first occurrence of a failure during automated testing.

While the failure intensities during these special types of testing may be supposed to differ from the failure intensity observed for "normal" testing, the degree and sometimes even the overall trend of the deviation cannot be determined analytically. In order to make it possible to analyze the types of testing separately as well as to study the differences in their failure intensities, for each interval during which the software is being tested the type of testing is also logged in *Teddi*. Please note that one interval can only belong to one of the types: test preparation, (normal) test execution, failure reproduction and retest of supposedly fixed bugs. However, the classification manual–automated test interval is orthogonal to the other one. I.e., there may be manual test preparation, automated test preparation, manual normal test execution, automated normal test execution, etc.

Analysis of the data for different projects will show whether it is necessary and reasonable to make the distinction between all these types of testing. Conclusions can only be drawn, however, if there are enough testing intervals from each category available in the data sets.

5.4 Failure and testing data from different test phases

Early software development methodologies, like the waterfall model [1, pp. 99 - 101], considered the software development process to consist of several separate activities that were executed in strict linear succession. Accordingly, testing was done after the software had been completely developed and either lead to its acceptance and distribution or to its rejection and its return to the programming team. A drawback of this model was that deviations of the product from the requirements and other problems were only detected late in the entire development process.

In newer methodologies, e.g. the V-model [1, pp. 101 - 113], testing takes place at different stages during software development. Testing phases could occur after the programming of individual modules or objects (unit testing), after integration of several modules/objects

(subsystem testing), after integration of all modules/objects (system testing) and after installation of the software system in an environment corresponding to field conditions (operation testing).

Some software quality measurement techniques like "Orthogonal Defect Classification" [6, 7] use the development of the shares of different failure and fault types over all the testing phases as an indicator of testing progress and risk. Since classical software reliability engineering requires testing to be done according to the operational profile, the application under test should be complete; therefore, data from system or - even better - operation testing should be used. Apart from the fact that often no operational profile can be determined for modules [20] the reduced amount of failure data for small software components makes software reliability growth modelling inappropriate for individual software units [37]. Changes in the software code, like the addition of new modules to a system, or changes in environmental conditions, e.g. in the usage pattern, lead to alterations in the failure generating process. Therefore, generally only failure data from one testing phase should be used for fitting a software reliability growth model. However, some models - the Musa-Okumoto model, for example [37] - seem to be able to cope with evolving code at least to some extent. Musa, Iannino and Okumoto [41] describe an extended version of Musa's basic execution time model for failure data of a program growing at several discrete points in time.

Instead of fitting one curve to failure times from different stages, information about the testing done and the extent of changes in the code between the phases could be used for early prediction of the model parameters of the software reliability growth model at the next stage (cf. section 4.1) or for the estimation of their prior distributions (cf. section 4.2). How thoroughly testing in earlier phases has been done can either be directly determined from test data of these earlier phases or from the number of failures that occur during the acceptance test at the beginning of the new test phase.

5.5 Failure and testing data from different test cycles

In between two test cycles, i.e. the (system) tests for two releases of the same application, various changes have taken place:

- Faults that were detected in the earlier release have been removed by the development team. However, the removal may have been imperfect, and possibly new faults have been introduced into the software.
- New functions have been implemented in addition to or as a substitute for functions available in the earlier release.

Again, it does not seem reasonable to feed all the failure data into one model. However, the old data could be used for predicting (the prior distribution of) the initial fault density or the initial failure intensity of the new release:

- Even if fault correction was imperfect for "old" code the fault density and failure intensity at the beginning of the system testing phase should still be lower than in the previous release. Moreover, the older the code is the larger the portion of the remaining faults with low detectabilities should be.

- The initial fault density and failure intensity of "new" parts of the code can be estimated using failure data from the first test cycle, when the whole application was new. The detectability profile can be expected to be more uniform than for parts of the software that have been tested several times before.

These calculations require that it is possible to distinguish between executions of new and old code. This could be achieved by classifying test cases as newly executed or repeatedly executed.

References

[1] Balzert, H.: *Lehrbuch der Software-Technik - Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*, Heidelberg, Berlin, 1998

[2] Campodónico, S.; Singpurwalla, N. D.: *A Bayesian Analysis of the Logarithmic-Poisson Execution Time Model Based on Expert Opinion and Failure Data*, IEEE Trans. Software Engineering 20 (1994), pp. 677 - 683

[3] Campodónico, S.; Singpurwalla, N. D.: *Inference and Predictions From Poisson Point Processes Incorporating Expert Knowledge*, Journal of the American Statistical Association 90 (1995), pp. 220 - 226

[4] Casella, G.; George, E. I.: *Explaining the Gibbs Sampler*, The American Statistician, Vol. 46 (1992), No. 3, pp. 167 - 174

[5] Chib, S.; Greenberg, E.: *Understanding the Metropolis-Hastings Algorithm*, The American Statistician, Vol. 49 (1995), No. 4, pp. 327 - 335

[6] Chillarege, R.: *Chapter 9: Orthogonal Defect Classification*, in: Lyu, M. R. (ed.): *Handbook of Software Reliability Engineering*, New York, San Francisico, et al., 1996, pp. 359 - 400

[7] Chillarege, R.; Bhandari, I. S.; Chaar, J. K.; Halliday, M. J.; Moebus, D. S.; Ray, B. K.; Wong, M.-Y.: *Orthogonal Defect Classification - A Concept for In-Process Measurements*, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., 1992, URL = http://chillarege.com/odc/articles/odcconcept/odc.html (site visited 2001-05-31)

[8] Denton, J. A.: *Accurate Software Reliability Estimation*, Thesis, Colorado State University, 1999, URL = http://www.cs.colostate.edu/∼denton/jd_thesis.pdf (site visited 2001-05-31)

[9] Duran, J. W.; Ntafos, S. C.: *An Evaluation of Random Testing*, IEEE Trans. Software Eng. 10 (1984), pp. 438 - 444

[10] Fahrmeir, L.; Hamerle, A.; Tutz, G.: *Regressionsmodelle zur Analyse von Verweildauern*, in: Fahrmeir, L.; Hamerle, A.; Tutz, G. (ed.): *Multivariate statistische Verfahren*, 2nd edition, Berlin, New York, 1996, pp. 301 - 356

[11] Fahrmeir, L.; Tutz, G.: *Multivariate Statistical Modelling Based on Generalized Linear Models*, New York, Berlin, et al., 1994

[12] Farr, W.: *Chapter 3: Software Reliability Modeling Survey*, in: Lyu, M. R. (ed.): *Handbook of Software Reliability Engineering*, New York, San Francisico, et al., 1996, pp. 71 - 117

[13] Frankl, P. G.; Hamlet, R. G.; Littlewood, B.; Strigini, L.: *Evaluating Testing Methods by Delivered Reliability*, IEEE Trans. Software Eng. 24 (1998), pp. 587 -601

[14] Gelman, A.; Carlin, J. B.; Stern, H. S.; Rubin, D. B.: *Bayesian Data Analysis*, London, Glasgow, et al., 1995

[15] Goel, A. L.: *Software Reliability Models: Assumptions, Limitations, and Applicability*, IEEE Trans. Software Eng. 11 (1985), pp. 1411 - 1423

[16] Goel, A. L.; Okumoto, K.: *Time-Dependent Error-Detection Rate Model for Software Reliability and Other Performance Measures*, IEEE Trans. Reliability 28 (1979), pp. 206 - 211

[17] Gokhale, S. S.; Philip, T.; Marinos, P. N.; Trivedi, K. S.: *Unification of Finite Failure Non-Homogeneous Poisson Process Models through Test Coverage*, Technical Report 96-36, Center for Advanced Computing and Communication, Department of Electrical and Computer Engineering, Duke University, 1996, URL = ftp://ftp.eos.ncsu.edu/pub/ccsp/papers/ppr9636.PS (site visited 2001-05-31)

[18] Greene, W. H.: *Econometric Analysis*, 3rd edition, New Jersey, 1997

[19] Grottke, M.; Dussa-Zieger, K.: *Systematic vs. Operational Testing: The Necessity for Different Failure Models*, to appear in: Proc. Fifth Conference on Quality Engineering in Software Technology, Nuremberg, 2001

[20] Hamlet, R. (D.): *Random Testing*, Portland State University, Department of Computer Science, URL = ftp://ftp.cs.pdx.edu/pub/faculty/hamlet/random.ps.Z (site visited 2001-05-31)

[21] Hamlet, D.; Taylor, R.: *Partition Testing Does Not Inspire Confidence*, IEEE Trans. Software Eng. 16 (1990), pp. 1402 - 1411

[22] Horgan, J. R.; London, S.; Lyu, M. R.: *Achieving Software Quality with Testing Coverage Measures*, IEEE Computer, Sept. 1994, pp. 60 - 69

[23] Huang, C.-Y.; Kuo, S.-Y.; Chen I.-Y.: *Analysis of a Software Reliability Growth Model with Logistic Testing-Effort Function*, Proc. Eighth International Symposium on Software Reliability Engineering, Albuquerque, New Mexico, 1997, pp. 378 - 388

[24] Jelinski, Z.; Moranda, P.: *Software Reliability Research*, in: Freiberger, W. (ed.): *Statistical Computer Performance Evaluation*, New York, 1972, pp. 465 - 484

[25] Langberg, N.; Singpurwalla, N. D.: *A Unification of Some Software Reliability Models*, SIAM Journal of Scientific and Statistical Computing 6 (1985), pp. 781 - 790

[26] Li, N.; Malaiya, Y. K.: *Empirical Estimation of Fault Exposure Ratio*, Technical Report CS-93-113, Department of Computer Science, Colorado State University, 1993, URL = http://www.cs.colostate.edu/~ftppub/TechReports/1993/tr-113.pdf (site visited 2001-05-31)

[27] Li, N.; Malaiya, Y. K.: *Fault Exposure Ratio - Estimation and Applications*, Technical Report CS-96-130, Department of Computer Science, Colorado State University, 1996, URL = http://www.cs.colostate.edu/~ftppub/TechReports/1996/tr96-130.pdf (site visited 2001-05-31)

[28] Littlewood, B.; Verrall, J. L.: *A Bayesian Reliability Growth Model for Computer Science*, Journal of the Royal Statistical Society, Ser. A (Applied Statistics) 22 (1973), pp. 332 - 346

[29] Lyu, M. R.: *Chapter 1: Introduction*, in: Lyu, M. R. (ed.): *Handbook of Software Reliability Engineering*, New York, San Francisico, et al., 1996, pp. 3 - 25

[30] Lyu, M. R.: *Appendix B: Review of Reliability Theory, Analytical Techniques, and Basic Statistics*, in: Lyu, M. R. (ed.): *Handbook of Software Reliability Engineering*, New York, San Francisico, et al., 1996, pp. 747 - 779

[31] Malaiya, Y. K.; Denton, J.: *What Do the Software Reliability Growth Model Parameters Represent?*, Technical Report CS-97-115, Department of Computer Science, Colorado State University, 1997,
URL = http://www.cs.colostate.edu/~ftppub/TechReports/1997/tr97-115.pdf (site visited 2001-05-31)

[32] Malaiya, Y. K.; von Mayrhauser, A.; Srimani, P. K.: *The Nature of Fault Exposure Ratio*, Proc. Third International Symposium on Software Reliability Engineering, Research Triangle Park, NC, October 1992, pp. 23 - 32

[33] Malaiya, Y. K.; von Mayrhauser, A.; Srimani, P. K.: *An Examination of Fault Exposure Ratio*, IEEE Trans. Software Eng. 19 (1993), pp. 1087 -1094

[34] Mazzuchi, T. A.; Soyer, R.: *A Bayes Empirical-Bayes Model for Software Reliablity*, IEEE Trans. Reliability 37 (1988), pp. 248 - 254

[35] Meeker, W. Q.; Escobar, L. A.: *Statistical Methods for Reliability Data*, New York, Chichester, et al., 1998

[36] Meinhold, R. J.; Singpurwalla, N. D.: *Bayesian Analysis of a Commonly Used Model for Describing Software Failures*, The Statistician 32 (1983) , pp. 168 - 173

[37] Musa, J. D.: *A Theory of Software Reliability and Its Application*, IEEE Trans. Software Eng. 1 (1975), pp. 312 - 327

[38] Musa, J. D.: *Rationale for Fault Exposure Ratio K*, ACM SIGSOFT Software Engineering Notes vol. 16 no. 3 (July 1991), p. 79

[39] Musa, J. D.: *Operational Profiles in Software-Reliability Engineering*, IEEE Software, March 1993, pp. 14 - 32

[40] Musa, J. D.: *Software Reliability Engineering - More Reliable Software, Faster Development and Testing*, New York, St. Louis, et al., 1998

[41] Musa, J. D.; Iannino, A.; Okumoto, K.: *Software Reliability - Measurement, Prediction, Application*, New York, St. Louis, et al., 1987

[42] Musa, J. D.; Okumoto, K.: *A Logarithmic Poisson Execution Time Model for Software Reliability Measurement*, 1984, in: Malaiya, Y. K.; Srimani, P. K. (ed.): *Software Reliability Models - Theoretical Developments, Evaluation & Applications*, Los Alamitos, 1990, pp. 23 - 31

[43] Myers, G. J.: *Methodisches Testen von Programmen*, 6th edition, München, Wien, et al., 1999

[44] Newbold, P.: *Statistics for Business and Economics*, 3rd edition, Englewood Cliffs, 1991

[45] Ohba, M.: *Software reliability analysis models*, IBM Journal of Research and Development 28 (1984), pp. 428 - 443

[46] Paulk, M. C.; Weber, C. V.; Garcia, S. M.; Chrissis, M. B.; Bush, M.: *Key Practices of the Capability Maturity Model, Version 1.1*, Technical Report CMU/SEI-93-TR-25 ESC-TR-93-178, Software Engineering Institute, Carnegie Mellon University, 1993

[47] Pham, H.: *Software Reliability*, New York, Berlin, et al., 2000

[48] Piwowarski, P.; Ohba, M.; Caruso, J.: *Coverage Measurement Experience During Function Test*, Proc. Fifteenth International IEEE Conference on Software Engineering (ICSE), 1993, pp. 287 - 301

[49] Rivers, A. T.: *Modeling Software Reliability During Non-Operational Testing*, Ph.D. thesis, North Carolina State University, 1998, URL = http://renoir.csc.ncsu.edu/Faculty/Vouk/Papers/Rivers/Thesis/Rivers.Thesis.pdf.zip (site visited 2001-05-31)

[50] Rivers, A. T.; Vouk, M. A.: *An Empirical Evaluation of Testing Efficiency during Non-Operational Testing*, Proc. Fourth Software Engineering Research Forum, Boca Raton, 1995, pp. 111 - 120, URL = http://renoir.csc.ncsu.edu/Faculty/Vouk/Papers/SERF96.ps (site visited 2001-05-31)

[51] Rivers, A. T.; Vouk, M. A.: *Resource-Constrained Non-Operational Testing of Software*, Proc. Ninth International Symposium on Software Reliability Engineering, Paderborn, 1998, pp. 154 - 163

[52] Singpurwalla, N. D.: *An Interactive PC-Based Procedure for Reliability Assessment Incorporating Expert Opinion and Survival Data*, Journal of the American Statistical Association, Vol. 83 (1988), No. 401, pp. 43 - 51

[53] Singpurwalla, N. D.; Wilson, S. P.: *Software Reliability Modeling*, International Statistical Review, Vol. 62 (1994), No. 3, pp. 289 - 317

[54] Singpurwalla, N. D.; Wilson, S. P.: *Statistical Methods in Software Engineering - Reliability and Risk*, New York, Berlin, et al., 1999

[55] Vouk, M. A.: *Using Reliability Models During Testing With Non-Operational Profiles*, 1992, URL = http://renoir.csc.ncsu.edu/Faculty/Vouk/Papers/non_op_testing.ps (site visited 2001-05-31)

[56] Wong, E. W.; Horgan, J. R.; London, S.; Mathur, A. P.: *Effect of Test Set Size and Block Coverage on the Fault Detection Effectiveness*, Technical Report SERC-TR-153-P, Software Engineering Research Center, 1994,
URL = http://www.serc.net/TechReports/files/TR153P.PS (site visited 2001-05-31)

[57] Wong, W. E.; Horgan, J. R.; London, S.; Mathur, A. P.: *Effect of Test Set Minimization on Fault Detection Effectiveness*, Proc. Seventeenth International IEEE Conference on Software Engineering (ICSE), 1995, pp. 41 - 50

[58] Yamada, S.; Hishitani, J.; Osaki, S.: *Software-Reliability Growth with a Weibull Test-Effort: A Model & Application*, IEEE Trans. Reliability 42 (1993), pp. 100 - 106

[59] Yamada, S.; Ohtera, H.; Narihisa, H.: *Software Reliability Growth Models with Testing-Effort*, IEEE Trans. Reliability 35 (1986), pp. 19 - 23