25. Statistical Models in Software Reliability and Operations Research

Statistical models play an important role in monitoring and control of the testing phase of software development life cycle (SDLC). The first section of this chapter provides an introduction to software reliability growth modeling and management problems where optimal control is desired. It includes a brief literature survey and description of optimization problems and solution methods.

In the second section a framework has been proposed for developing general software reliability models for both testing and operational phases. Within the framework, pertinent factors such as testing effort, coverage, user growth etc. can be incorporated. A brief description of the usage models have been provided in the section. It is shown how a new product sales growth model from marketing can be used for reliability growth modeling. Proposed models have been validated on software failure data sets.

To produce reliable software, efficient management of the testing phase is essential. Three management problems viz. release time, testing effort control and resource allocation are discussed in Sects. 25.2 to 25.4. The operations research approach, i. e. with the help of the models, optimal management decisions can be made regarding the duration of the testing phase, requirement and allocation of resources, intensity of testing effort etc. These optimization problems can be of interest to both theoreticians and software test managers. This chapter discusses both of these aspects viz. model development and optimization problems.

A scientific way of solving decision-making problems arising in large and complex systems involves the construction of a model (usually a mathematical model) that represents the character of the problem. Modeling can be the most practical way of studying the behavior of such systems. A model exhibits relationships between quantitative variables under a definite set of assumptions that portray the system. It allows experimentation with different alternative courses of actions and facilitates the use of sophisticated mathematical techniques and computers for the purpose. Mathematical models have proved to be useful for understanding the structure and functioning of a system, predicting future events and prescribing the best course of actions under known constraints. The success and popularity of operations research, a problem-solving approach started with the above philosophy as its basic working principle, has demonstrated the utility of mathematical modeling. One of the fields where mathematical modeling, particularly stochastic modeling, has been applied widely is

reliability. Stochastic modeling in reliability theory has continued to be an area of extensive research for more than four decades. The subject has traditionally been attached to hardware systems. But with ever increasing use of computers in present times software reliability has also emerged as a discipline of its own. This chapter endeavors to develop new mathematical models for software reliability evaluation and propose methods for efficient management of the testing phase.

The last decade of the 20th century will be noted in history for the incredible growth in information technology. The proliferation of the Internet has gone far beyond even the most outrageously optimistic forecasts. Consequently computers and computer-based systems have invaded every sphere of human activity. As more systems are being automated mankind's dependence on computers is rapidly increasing. Though this technology revolution has made our lives better, concern for safety and security has never been greater. There are already numerous instances where the failure of computer-controlled systems has led to colossal loss of human lives and money. Computer-based systems typically consist of hardware and software. Quality hardware can now be produced at a reasonable cost but the same cannot be said about software. Software development consists of a sequence of activities where perfection is yet to be achieved. Hence there is every possibility that fault can be introduced and can remain in a software. A fault occurs when a human makes a mistake, called an error, in performing some software activity. These faults can lead to failures with catastrophic results. Therefore a lot of emphasis is put on avoiding the introduction of faults during software development and to remove dormant faults before the product is released for use.

The testing phase is an extremely important component of the software development life cycle (SDLC), where around half the developmental resources are consumed. In this phase the software product is tested to determine whether it meets the requirement. It is endeavored to remove faults lying dormant in the software. The theory developed in this chapter primarily addresses the testing phase. The only way to verify and validate the software is by testing. The software testing involves running the software and checking for unexpected behavior of the software output. A successful test can be considered to be one that reveals the presence of latent faults. During testing, resources such as manpower and time are consumed. A very specialized kind of manpower is required for test-case generation, running the test cases and debugging. Time is also a very important resource as software cannot be tested indefinitely and there is always pressure to release the software as early as possible. With the increasing importance of cost and time during software development, efficient management of the testing phase becomes a high-priority issue for an organization. Therefore it is important to understand the failure pattern and faults causing these failures. The chronology of failure occurrence and fault removal can be utilized to provide an estimate of software reliability and the level of fault content. A software reliability model is a tool that can be used to evaluate the software quantitatively, develop test status and monitor the change in reliability performance [25.1]. Numerous software reliability growth models (SRGMs), which relate the number of failures (faults identified) and execution time (CPU time/calendar time), have been discussed in the literature [25.2–9]. These models are used to predict fault content and the reliability of software. The majority of these models can be categorized as nonhomogeneous Poisson process (NHPP) models as they assume a NHPP model to describe the failure phenomenon [25.3,6,8,10]. New models exploit the mean-value function of the underlying NHPP by proposing new forms for it; this chapter takes this modeling approach. Moreover the expected behavior of the users of the software has also been included in the modeling process.

Large software systems contain several million lines of code. The sheer size of the product presents unique problems in terms of the ability of the software designers to achieve software quality rapidly. the testing phase, which consumes the largest portion of software development resources, poses formidable challenges. Manpower from diverse background are involved in the testing process. It is this phase where a closer interaction with the users is a must. It is a fact that if software is tested for a longer period it would result in an increase in reliability. But the cost of testing also increases. Very often test managers work under tight schedules and with limited resources. Therefore, to produce reliable software, efficient management of the testing phase is required. Three such management problems viz. the release-time problem, the testing effort control problem and the resource allocation problem are discussed in this chapter.

To know when to stop testing is a pertinent question during the testing phase. If the time of release of the software for operation can be forecasted beforehand it can help management immensely. The predictive ability of SRGMs can provide a scientific answer. The release time should be an optimal tradeoff between cost and reliability. Due to the obvious importance of the problem it has received a lot of attention from re-

searchers [25.3, 8, 11–16]. In this chapter this problem has been chosen as the first illustration of the application of the methods of operations research in software reliability engineering. The usage-based SRGMs have been applied in more realistic mathematical programming formulations of the problem. Often the target reliability level is fixed for release time during the testing of software. Using SRGMs the reliability of the software can be forecasted for any future time. If it is found that the target cannot be achieved, the testing effort needs to be accelerated. The additional resource requirements can be calculated using

SRGMs [25.17–19]. In Sect. 25.3 we discuss the above testing effort control problem and provide a new solution method through an SRGM specially developed for the purpose. Optimal resource allocation is a problem that bothers all decision makers. Hence the literature in this area is very rich. Many operations researchers are working on new allocation problems arising in systems that are changing due to the proliferation of technology. Module testing in the testing phase is one such activity where optimal resource allocation can be important for obtaining reliable software [25.17, 20, 21]. In this chapter the mathematical programming approach has been suggested for the solution of this problem.

The objective of this chapter is to highlight the importance of modeling and optimization in software reliability engineering. The first part is devoted towards model development and, thereafter, three illustrations of optimization and control are provided.

25.1 Interdisciplinary Software Reliability Modeling

A commercial software developer endeavors to make its software product popular in the market by providing value to its customer and thus generating goodwill. Apart from satisfying customers by meeting all their requirements and attaching additional features, the developer at the same time makes constant efforts to build bug-free software. As manual systems are increasingly being automated, a failure due to software can lead to loss of money, goodwill and even human lives. The competition in the commercial software market is intense and, because of the nature of the applications involved, purchasers look for quality in terms of the reliability of the software. Therefore software developers lay special emphasis on testing their software.

During the testing phase, test cases that simulate the user environment are run on the software and any departure from the specifications or requirements is called a failure and an effort is made immediately to remove the cause of that failure (a fault in the software). Testing goes on until the management is satisfied with the reliability of the software. But software cannot be tested exhaustively within a limited time period. This is the reason why we often hear about failures of software in operation and sometimes even in safety-critical systems. These failures are caused by faults that remain even after testing. Hence it is important to study how these failures occur in the user phase. Selling a software is not a one-off deal. It involves cultivating long-term relationship with the purchasers. Many of the developers come up with newer versions of their software after the launch of their product. These new versions can contain codes of the previous version with some additional modules and modifications. Moreover, some developers give warranties on their products. Hence any fault that is reported by the user is corrected. If the number of faults remaining in the software can be estimated to a reasonable accuracy it can give the management a useful metric to be used for decision making under the situations discussed above. Mathematical modeling can help in developing such a metric.

SRGMs have been widely used to estimate the reliability of software during testing. Many authors have even tried to extend them to represent the failure phenomenon during the operational phase, typically used in the software release-time problem [25.3, 15, 16]. But this approach is not correct when usage of software is different from during testing, which is actually the case for most commercial software packages. Testing is done under a controlled environment. Testing resources such as manpower and consumed (computer) time can be measured and extended further into the future. Mathematical models have been proposed for testing effort itself but they are not suitable for measuring the usage of software in the market. The intensity with which failures would manifest themselves during operational use is dependent upon the number of times the software is used and not much has been done in the literature for this situation [25.21]. An attempt has been made in this chapter to model reliability growth, linking it to the number of users in the operational phase. In this chapter we propose a framework for model development for the operational phase, which can also connect the testing phase, thus providing a unique approach to modeling both the testing and operational phases.

Kenny [25.21] proposed a model to estimate the number of faults remaining in the software during its operational use. He has assumed a power function to represent the usage rate of the software. Though Kenny argues that the rate at which the commercial software is used is dependent upon the number of users, the model proposed by him can fail to capture the growth in the number of users of a software product. A mathematical model to capture the growth of users is integrated into the proposed software reliability growth model.

Although commercial software products have been on the market for two decades, identifying the target customers with certainty is impossible. Hence a product, which may be similar in many respect to another one when launched in the market, behaves as a new product or innovation. The Bass model for innovation diffusion [25.22] in marketing has satisfactorily been used for this dynamic market of software products [25.23]. This model explicitly categorizes the customers into innovators and imitators. Innovators have independent decision-making abilities whereas imitators make the purchase decisions after getting first-hand opinion from a user. Here it is assumed that purchasers or users whose number with respect to time can be modeled as an innovation diffusion phenomenon are those who can report a failure caused by the software to the developer. Such a model can correctly describe the growth of users in terms of:

1. a slow start but a gain in growth rate,
2. a constant addition of users,
3. a big beginning and tail off in the usage rate, as pointed out by *Kenny* [25.21].

The model can also describe the situation where a much-hyped product when launched in a market does not fare according to expectation. Once the number of users of the software is known, the rate at which instructions in the software are executed can be estimated. The intensity with which failures would be reported depends upon this usage. The models developed in the software reliability engineering literature can now be used to model the fault exposure phenomenon.

Another important factor that affects software reliability immensely is testing coverage, but very few attempts have been made in the literature to include its impact [25.24, 25]. With the running of test cases and corresponding failure-removal processes during the testing phase, more portions of the software, paths, functions are tested. However, it is also a fact that software cannot be tested exhaustively. As testing coverage increases software becomes more reliable. Hence testing coverage is very important for both software test managers as well as users of the software. The model developed in this chapter for both the testing and operational phases also takes this factor into account, which is another novel feature of the chapter.

Notations:

$m, m(t)$:	Expected number of faults identified in the time interval $(0,t]$ during the testing phase.
$\hat{m}, \hat{m}(t)$:	Expected number of faults identified in the time interval $(0,t]$ during the operational phase.
$e, e(t)$:	Expected number of instructions executed on the software in the time interval $(0,t]$.
$W, W(t)$:	Cumulative testing effort in the time interval $(0,t]$; $\frac{\mathrm{d}}{\mathrm{d}t}W(t) = w(t)$.
a:	Constant representing the number of faults lying dormant in the software at the beginning of testing.
$p, p(W(t))$:	Testing coverage as a function of time testing effort.
$\alpha, \beta, \delta, \gamma$:	Constants.
g, h, k, i, j:	Constants.
$k_i, i = 1, \ldots, 9$:	Constants.
$\bar{W}$:	Constant representing the saturation point for the growth of users of the software.
T:	Release time of the software.
$m^*(t)$:	Number of failures reported during the operational phase, $t > T$.
q:	Factor by which the operational usage rate differs from the testing rate per remaining faults.
$R_{\mathrm{te}}(x\|t)$:	Reliability of the software during the testing phase, $t < T$.
$R_{\mathrm{op}}(x\|t)$:	Reliability of the software during the operational phase, $t > T$.
C_1, C_2:	Costs of testing per unit time and removing a fault, respectively, during the testing phase.
C_3:	Cost of a failure and removing it during the operational phase.
T_{s}:	Scheduled delivery time of the software.
$p_{\mathrm{c}}(t)$:	Penalty cost $\begin{cases} 0, & t \le T_{\mathrm{s}} \\ p_{\mathrm{c}}.t & \text{otherwise} \end{cases}$.
T_{w}:	Warranty period of the software.

25.1.1 Framework for Modeling

As discussed in the Introduction, several quantitative measures of growth in reliability of software during the testing phase have been proposed in the literature and several of these can be classified as NHPP models [25.3, 8, 9]. These NHPP models are based on the assumption that 'Software failures occur at random times during testing caused by faults lying dormant in the software'. The assumption appears true for both the testing and operational phases. Hence NHPP models can be used to describe the failure phenomenon during both of these phases. The counting process $\{N(t), t \geq 0\}$ of an NHPP is given as follows.

$$\Pr[N(t) = k] = \frac{[m(t)]^k}{k!} \mathrm{e}^{-m(t)}, \quad k = 0, 1, 2,$$
$$\text{and } m(t) = \int_0^t \lambda(x)\,\mathrm{d}x\,. \tag{25.1}$$

The intensity function $\lambda(x)$ (or the mean-value function $m(t)$) is the basic building block of all the NHPP models existing in the software reliability engineering literature. These models assume diverse testing environments such as the distinction between failure and removal processes, learning of the testing personnel, the possibility of imperfect debugging and error generation etc. In models proposed by *Yamada* et al. [25.26] and *Trachtenberg* [25.27], the effect of the intensity of testing effort on the failure phenomenon has been studied. Faults if present in the software are exposed when the software is run. During the testing phase, test cases are run and in the operational phase the software is used by the user. Hence the rate at which failures would occur depends upon its usage (i. e. testing effort during testing or number of users in the operational phase [25.21]). Hence SRGMs should incorporate the effect of usage. But this may give rise to more complication and confusion as a number of functions exist in the literature that describes the testing effort or user growth with time. In this chapter an attempt has been made to address this problem. A general framework for model development has been proposed here. Using the basic building blocks of this framework SRGMs for both testing and operational phases can be developed with ease. The proposed approach is based upon the following basic assumptions.

1. Software failure phenomenon can be described by an NHPP. Software reliability growth models of this chapter are the mean-value functions of NHPP.
2. The number of failures during testing/operation is dependent upon the number of faults remaining in the software at that time. It is also dependent upon the rate of testing coverage.
3. Testing coverage increases due to testing effort.
4. As soon as a failure occurs the fault causing that failure is immediately identified. Identified faults are removed perfectly and no additional faults are introduced during the process.
5. The number of instructions executed is a function of testing effort/number of users.
6. Testing effort/number of users is a function of time.

Using the above assumptions the failure phenomenon can be described with respect to time as follows [25.21, 27]:

$$\frac{\mathrm{d}m}{\mathrm{d}t} = \frac{\mathrm{d}m}{\mathrm{d}e}\frac{\mathrm{d}e}{\mathrm{d}W}\frac{\mathrm{d}W}{\mathrm{d}t}\,. \tag{25.2}$$

We discuss below individually each component (fraction) on the right-hand side of the above expression.

Component 1

During testing instructions are executed on the software and the output is matched with the expected results. If there is any discrepancy a failure is said to have occurred. Effort is made to identify and later remove the cause of the failure. The rate at which failures occur depends upon the number of faults remaining in the software [25.10]. As the coverage of the software is increased more faults are removed. The rate at which additional faults are identified is directly dependent upon the rate at which software is covered through additional test cases being run [25.24, 25]. It is also dependent upon the size of the uncovered portion of the software. Based on these facts the differential equation for fault identification/removal can be written as:

$$\frac{\mathrm{d}m}{\mathrm{d}e} = k_1 \frac{p'}{c-p}(a-m)\,, \tag{25.3}$$

where p' is the rate (with respect to testing effort) with which the software is covered through testing, c is the proportion of total software which will be eventually covered during the testing phase, with $0 < c < 1$. If c is closer to 1, one can conclude that test cases were efficiently chosen to cover the operational profile. For a logistic fault removal rate we can assume the following

form for $\frac{p'}{c-p}$:

$$\frac{p'}{c-p} = \frac{g}{1+h\mathrm{e}^{-gW}} . \tag{25.4}$$

$$\text{Hence, } p(W) = c\frac{1-\mathrm{e}^{gW}}{1+h\mathrm{e}^{-gW}} . \tag{25.5}$$

Testing coverage is directly related to testing effort, because with more testing effort we can expect to cover a larger portion of the software. Testing effort can be modeled as a function of time, which will be discussed later in this chapter.

Component 2

The second component of expression (25.2) relates the number of instructions executed with the testing effort or the number of users of the software. For the sake of simplicity we assume it to be constant

$$\frac{\mathrm{d}e}{\mathrm{d}W} = k_2 . \tag{25.6}$$

Substituting (25.3) and (25.6) into (25.2) we have

$$\frac{\mathrm{d}m}{\mathrm{d}t} = k_1\frac{g}{1+h\mathrm{e}^{-gW}}(a-m)k_2\frac{\mathrm{d}W}{\mathrm{d}t} . \tag{25.7}$$

In the next section the mathematical models for the software testing effort (component 3 of (25.2)) are discussed.

25.1.2 Modeling Testing Effort

The resources that govern the pace of testing for almost all software projects [25.6] are

1. Manpower, which includes
   - Failure-identification personnel,
   - Failure-correction personnel.
2. Computer time

In the literature, either the exponential or Rayleigh function has been used to explain the testing effort. Both can be derived from the assumption that, the testing effort rate is proportional to the testing resources available.

$$\frac{\mathrm{d}W(t)}{\mathrm{d}t} = c(t)[\alpha - W(t)] , \tag{25.8}$$

where $c(t)$ is the time-dependent rate at which testing resources are consumed with respect to remaining available resources. Solving (25.8) under the initial condition $W(0) = 0$, we get

$$W(t) = \alpha\left\{1-\exp\left[\int_0^t c(x)\,\mathrm{d}x\right]\right\} . \tag{25.9}$$

When $c(t) = c$, a constant

$$W(t) = \alpha\left(1-\mathrm{e}^{-ct}\right) . \tag{25.10}$$

If $c(t) = ct$, (25.8) gives a Rayleigh-type curve

$$W(t) = \alpha\left(1-\mathrm{e}^{-ct^2/2}\right) . \tag{25.11}$$

Huang et al. [25.28] developed an SRGM, based upon an NHPP with a logistic testing-effort function. The cumulative testing effort consumed in the interval $(0, t]$ has the following form

$$W(t) = \frac{p}{1+r\mathrm{e}^{-lt}} . \tag{25.12}$$

Where p, r and l are constants. SRGMs with logistic testing-effort functions provide better results on some failure data sets.

Yamada et al. [25.29] described the time-dependent behavior of testing-effort expenditure by a Weibull curve while proposing an SRGM of

$$W(t) = \alpha\left(1-\mathrm{e}^{-\beta t^k}\right) . \tag{25.13}$$

Exponential and Rayleigh curves become special cases of the Weibull curve for $k = 1$ and $k = 2$ respectively. To study the testing-effort process, one of the above functions can be chosen. In the following section we develop an SRGM where the fault-detection rate is a function of the testing effort and can have one of the forms discussed above.

25.1.3 Software Reliability Growth Modeling

Any one of the testing-effort models can be substituted in (25.7) to obtain a general software reliability growth model. Equation (25.7) can be written as follows

$$\frac{(\mathrm{d}m/\mathrm{d}t)}{(\mathrm{d}W/\mathrm{d}t)} = k\frac{g}{1+h\mathrm{e}^{-gW}}(a-m) , \tag{25.14}$$

where, $k = k_1k_2$. Equation (25.14) is a first-order linear differential equation. Solving it with the initial conditions $m(0) = 0$ and $W(0) = 0$ we have,

$$m[W(t)] = a\frac{\left(1+h\mathrm{e}^{-gW(t)}\right)^k-(1+h)\mathrm{e}^{-gkW(t)}}{\left(1+h\mathrm{e}^{-gW(t)}\right)^k} . \tag{25.15}$$

Next it is shown how a similar modeling approach can be used to obtain a failure-count model for the operational phase.

This SRGM is flexible and general in nature. For different parameter values it can reduce to many well-known SRGMs. *Pham* et al. [25.30] have proposed an alternative approach for the development of a general, flexible SRGM, though the impact of testing coverage was not explicitly considered in their model development. Moreover the modeling approach of this chapter can be extended to the operational phase, as shown next.

25.1.4 Modeling the Number of Users in the Operational Phase

During the operational phase failures are reported by the users. Software developers remove faults that cause these failures in future releases of the software. The number of failure reports can depend on the number of users of the software. As the usage grows so does the number of failure reports. Hence usage during the operational phase plays a similar role as testing effort during the testing phase. The failure-count model for the operational phase is based upon the following assumptions:

1. The number of unique failure reports and corresponding fault removals of the software during the operational phase can be described by an NHPP.
2. The number of failures during operation is dependent upon the number of faults remaining in the software. It is also directly proportional to the size of the uncovered portion (at the completion of the testing phase) of the software and the volume of instructions executed.
3. Once a failure is reported, the same failure report by other users is not counted. The SRGM developed can be interpreted as a failure-count model. The debugging process by the developer is assumed to be perfect.
4. The volume of instructions executed is related to the number of users.
5. The number of users of the software is a function of time.

Using the above assumptions the fault-removal phenomenon during the operational phase can be described as a function of time as follows:

$$\frac{\mathrm{d}\hat{m}}{\mathrm{d}t} = [1 - p(T)]\frac{\mathrm{d}\hat{m}}{\mathrm{d}e}\frac{\mathrm{d}e}{\mathrm{d}W}\frac{\mathrm{d}W}{\mathrm{d}t}\,, \tag{25.16}$$

where T is the release time of the software, $[1 - p(T)]$ is the size of the uncovered portion of the software and its value is known at the time of release of the software, $\hat{m}$ is the mean-value function of the failure-count model for the operational phase, i. e., the expected number of faults removed during the operational phase. The other three fractions of the right-hand side of (25.16) can be modeled similarly to the process followed in Sect. 25.1.1. Now fault removal is directly dependent on the number of instructions executed. It is also a fact that additional faults are removed during code checking for failure-cause isolation, but these faults may not have caused failures. *Kapur* and *Garg* [25.31] have discussed this phenomenon. Based upon these arguments the following expression can be written

$$\frac{\mathrm{d}\hat{m}}{\mathrm{d}e} = \left(k_4 + k_5\frac{\hat{m}}{a_1}\right)(a_1 - \hat{m})\,, \tag{25.17}$$

where k_4 is the rate at which remaining faults cause failures. It is a constant, as each one of these faults has an equal probability of causing failure. k_5 is the rate at which additional faults are identified without causing any failures; it is a constant, but also depends upon the number of faults already identified. $a_1 = [a - m(T)]$ is the number of faults present in the software when it was released for use (test time T).

During the debugging process some of the faults might be imperfectly removed and can cause failure in future. If this factor is introduced into the model, (25.17) can be modified as follows:

$$\frac{\mathrm{d}\hat{m}}{\mathrm{d}e} = \left(k_4 + k_5\frac{\hat{m}}{(a_1 + \hat{m})}\right)\left[(a_1 + k_6\hat{m}) - \hat{m}\right]. \tag{25.18}$$

In the above expression k_6 is the rate of imperfect debugging. But finding a closed-form solution for (25.18) is difficult. Therefore we can assume a logistic rate function as discussed (25.17),

$$\frac{\mathrm{d}\hat{m}}{\mathrm{d}e} = \left(\frac{i}{1 + j\,\mathrm{e}^{-it}}\right)(a_1 - k_7\hat{m})\,, \tag{25.19}$$

where, $k_7 = 1 - k_6$. Moreover, assuming that the number of instructions executed is a constant with respect to usage growth, the following expression, which is similar to (25.6), can be written.

$$\frac{\mathrm{d}e}{\mathrm{d}W} = k_8\,. \tag{25.20}$$

25.1.5 Modeling the User Growth

Kenny [25.21] used the power function to describe the growth in the user population of a software

$$W(t) = \frac{t^{(k+1)}}{(k+1)} \,. \tag{25.21}$$

$W(t)$ here is the number of users of the software in the operational phase at time t. The function can correctly describe the users growth in terms of

1. a slow start but a gain in growth rate,
2. a constant addition of users,
3. a big beginning and tail off in the usage rate.

However, in the marketing literature, the power function is seldom used for the purpose as described above. One of the reasons may be that the parameters of the function are not amenable to interpretation. The growth in the number of users with respect to time can also be described by the Bass model [25.22] of innovation diffusion. To apply the Bass model it is assumed that there exists a finite population of prospective users who, with time, increasingly become actual users of the software (no distinction is made between users and purchasers here as the Bass model has been successfully applied to describe the growth in number of both of them). In each period there will be both innovators and imitators using the software product. The innovators are not influenced in their timing of purchase by the number of people who have already bought it, but they may be influenced by the steady flow of nonpersonal promotion. As the process continues, the relative number of innovators will diminish monotonically with time. Imitators are, however, influenced by the number of previous buyers and increase relative to the number of innovators as the process continues.

The combined rate of first purchasing of innovators and imitators are given by the term $\left(\alpha + \beta\frac{W(t)}{\bar{W}}\right)$ and increases through time because $W(t)$ increases through time. In fact the rate of first purchasing is shown as a linear function of the cumulative number of previous first purchasers. However, the number of remaining nonadopters, given by $\left[\bar{W} - W(t)\right]$ decreases through time. The shape of the resulting sales curve of new adopters will depend upon the relative rate of these two tendencies. If a software product is successful, the coefficient of imitation is likely to exceed the coefficient of innovation i. e. $\alpha < \beta$. On the other hand, if $\alpha > \beta$, the sales curve will fall continuously.

The following mathematical model, known as the Bass model [25.22] in the marketing literature, describes this situation.

$$\frac{\mathrm{d}W(t)}{\mathrm{d}t} = \left(\alpha + \beta\frac{W(t)}{\bar{S}}\right)\left[\bar{S} - W(t)\right] . \tag{25.22}$$

The solution of (25.22) for $W(t=0)$ is

$$W(t) = \bar{W}\frac{1-\exp[-(\alpha+\beta)t]}{1+(\beta/\alpha)\exp[-(\alpha+\beta)t]} \,. \tag{25.23}$$

Givon et al. [25.23] have used the modified version of this model to estimate the number of licensed users as well as users of pirated copies of the software. Though it can be reasonably assumed that it is the licensed-copy holders who would report the failures, (25.23) can be used to find the expected number of users at any time during the life cycle of the software. If the new software is expected to go through the same history as some previous software (very likely for versions of the same software) the parameters of an earlier growth curve may be used as an approximation.

The derivative of (25.23) to be used in expression (25.16) has the following form

$$\frac{\mathrm{d}W(t)}{\mathrm{d}t} = \bar{W}\frac{\beta[1+(\beta/\alpha)]\exp[-(\alpha+\beta)t]}{\left\{1+(\beta/\alpha)\exp[-(\alpha+\beta)t]\right\}^2} \,. \tag{25.24}$$

After substitution of all the components, (25.16) is a first-order linear differential equation. Solving it with the initial condition $m(t=0)=0$ we have,

$$\hat{m}(t) = \frac{a_2}{k_9}\left[1 - \left(\frac{(1+j)\,\mathrm{e}^{-\mathrm{i}W(t)}}{1+j\,\mathrm{e}^{-\mathrm{i}W(t)}}\right)^{k_9}\right] , \tag{25.25}$$

where $a_2 = k_8 a_1[1-p(T)]$ and $k_9 = k_7 k_8[1-p(T)]$.

25.1.6 Estimation Methods

The testing-effort data or the data pertaining to the number of users (or usage) of a software can be collected in the form of testing effort/usage, $w_k(-<w_1<w_2<\ldots<w_n)$ consumed in time $(0, t_i]$; $i = 1, 2, \ldots, n$. Then the testing-effort model/usage growth model parameters can be estimated by the method of least squares as follows

$$\text{Minimize} \sum_{i=1}^{n}\left(W_i - \hat{W}\right)^2 \tag{25.26}$$

subject to $\hat{W}_n = W_n$ (i. e. the estimated value of the testing effort is equal to the actual value).

To estimate the parameters of the SRGMs obtained through (25.15) and (25.25), the method of maximum likelihood (MLE) is used [25.3, 6, 8]. The fault-removal

Table 25.1 Fitting of testing effort data

Data sets	α	β	k	R^2
DS-1	2669.9	0.000773	2.068	0.99
DS-2	11710.7	0.0235	1.460154	0.98

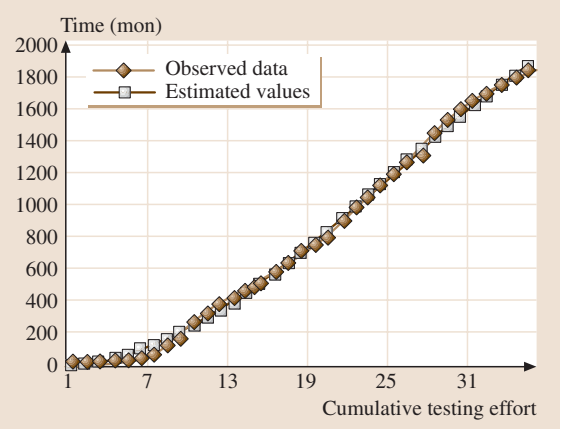

Fig. 25.1 Fitting of the effort curve (DS-1)

data is given in the form of cumulative number of faults removed, y_i in time $(0, t_i]$. Thus the likelihood function is given as

$$L[a_1, a_2, b_o, b_r, q|(y_i, W_i)] = \prod_{i=1}^{n} \frac{[m(t_i) - m(t_{i-1})]^{y_i - y_{i-1}}}{(y_i - y_{i-1})!} e^{-[m(t_i) - m(t_{i-1})]} . \quad (25.27)$$

25.1.7 Numerical Illustrations

To validate the models four real software failure data sets have been chosen. The first two were collected during the testing phase of software while the third and fourth data sets are based on failure reports of software in operational use.

Data set 1 (DS-1): The data are cited from *Brooks* and *Motley* [25.11]. The fault data set is for a radar system of size 124 KLOC (kilo lines of code) tested for 35 months, in which 1301 faults were identified.

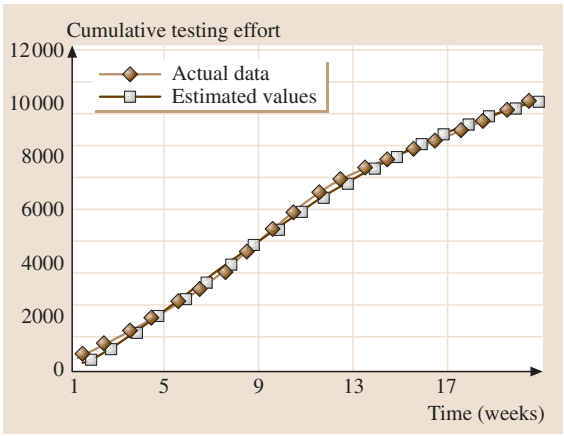

Fig. 25.2 Fitting of the testing effort curve (DS-2)

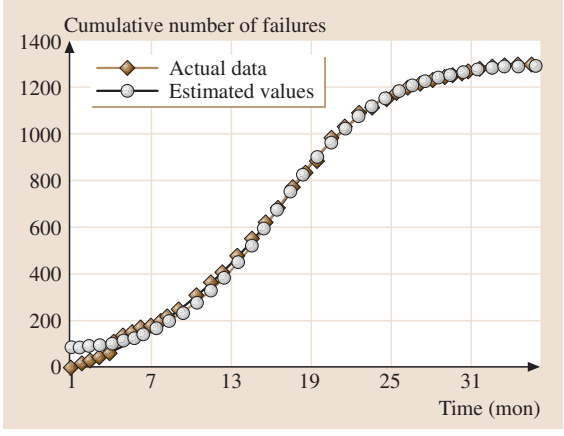

Fig. 25.3 Fitting of the failure curve (DS-1)

Data set 2 (DS-2): The data set pertains to release 1 of the tandem computer project cited in [25.30]. The software test data is available for 20 weeks, during which 100 faults were identified.

In both cases the Weibull function (25.13) gave the best fit to the testing-effort data. The results are presented in Table 25.1 and the curve fits are depicted in Figs. 25.1 and 25.2, respectively.

The estimation results for the parameters of SRGM (25.15) have been summarized in Table 25.2 and are graphically presented in Figs. 25.3 and 25.4.

Table 25.2 Parameter estimation of the SRGM

Data set	a	H	g	k	R^2
DS-I	1305	4.46445	0.003173	1.0363	0.996
DS-II	110	4.8707	0.000392	1.092	0.997

Table 25.3 Estimation result on DS-3

a_2	K_9	j	i	$\overline{W}$	$(\alpha+\beta)$	β/α	R^2
112	0.978	4.352	0.026×10^{-5}	31038400	0.010634	2.4469	0.989

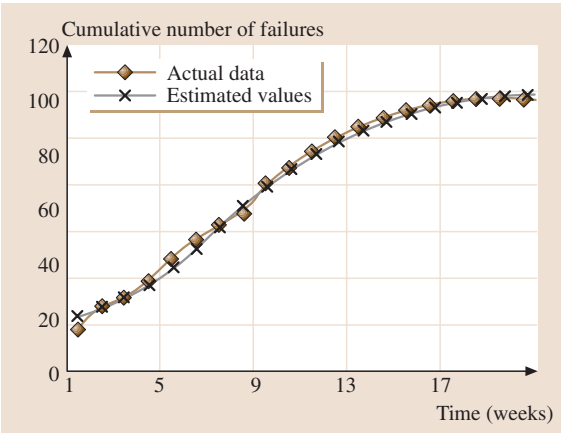

Fig. 25.4 Fitting of the failure curve (DS-2)

Next we estimate the parameters of SRGMs obtained from equation (25.25) and using usage growth functions (25.21) and (25.23). The following data set has been chosen for illustration.

Data set 3 (DS-3): This failure data set [25.32] is for an operating system in its operational phase. The software consists of hundreds of thousands of delivered object code instructions. 112 faults were reported during the observation period of around five months. The Bass model (25.23) could best describe the usage data and hence was chosen. Using the estimated values, the rest of the parameters of the model were estimated. The estimation results are summarized in Table 25.3 and are depicted in Figs. 25.5 and 25.6.

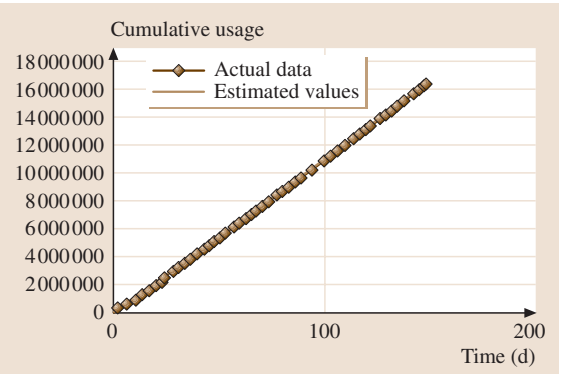

Fig. 25.5 Fitting usage data (DS-3)

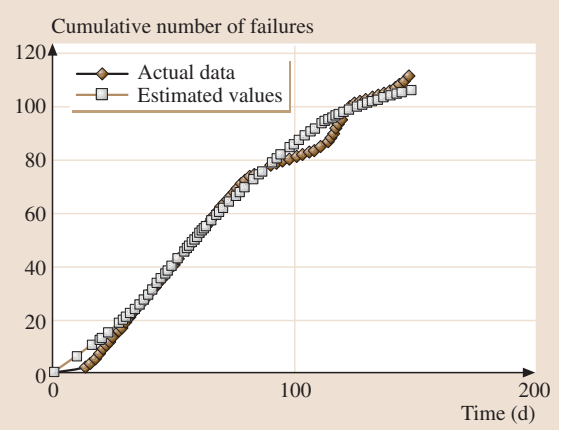

Fig. 25.6 Fitting of number of failures (DS-3)

25.2 Release Time of Software

It is important to know when to stop testing. The optimal testing time is a function of many variables: software size, level of reliability desired, personnel availability, market conditions, penalty cost due to delay in delivery of the product and penalties of in-process failures. If the release of the software is unduly delayed, the software developer may suffer in terms of penalties and revenue loss, while premature release may cost heavily in terms of fault removals to be done after release and may even harm the manufacturer's reputation. Software release-time problems have been classified in different ways. One of them is to find the release time such that the cost incurred during the remaining phases of the life cycle (consisting of the testing and operational phases) of the software is minimized [25.3, 15]. This problem can also be alternatively defined in terms of maximizing gain, where gain is defined as the difference in cost incurred when all faults are removed during the operational phase as against the cost when some faults are removed during the testing phase and others are removed during the operational phase. It can be proved that maximizing gain is the same as minimizing cost. Some

release-time problems are based upon reliability criteria alone. Models that minimize the number of remaining faults in the software or the failure intensity fall under this category [25.3]. Release-time problems have also been formulated for minimizing cost with minimum reliability requirements or maximizing reliability subject to budgetary constraints [25.3]. The bicriterion release policy simultaneously maximizes reliability and minimizes cost subject to reliability and resource constraints. In all these formulations software reliability growth models play a very important role due to their predictive ability. It is a fact that the longer software is tested, the higher its reliability. But it cannot be tested indefinitely, due to time and cost factors. With increasing cost, there is also a loss of opportunity in earning profit. Again software can have scheduled delivery time and the developer may have to pay high penalty costs due to a delay in delivery. Hence an optimal tradeoff between cost and reliability is required to find the termination time of testing. All the costs mentioned above are minimized subject to some constraints. These constraints are primarily related to a certain minimum level of testing reliability. The cost and reliability functions are discussed in detail later in this section.

The Cost Function

Cost functions discussed in the literature include costs of testing, removing faults during testing and that of failures and removals during the operational phase. As testing is done under a controlled environment, costs pertaining to testing, removing faults, documentation etc. can be estimated, but difficulty arises in quantifying the cost of a failure at the user end. As a way out a more realistic approach of warranty cost is being considered [25.33]. In release-time problems, costs of failure and removal of a fault occurring during a limited warranty period immediately after release need also to be included. Failure during the operational phase also amounts to loss of goodwill for the developer. Hence, in the cost function, failures after testing are counted and costs for their removal are estimated. Cost due to delay in delivery [25.34] is normally included in the overall software development cost. Release-time problems should include their affect. The cost function can takes the following form:

$$C(T) = c_1 T + c_2 m(T) + c_3 m^*(T + T_w) + p_c(T) \,. \tag{25.28}$$

It is assumed above that the costs of removing faults during testing and operation are constants: c_2 and c_3, respectively. The functional forms for $m(T)$ and $m^*(T + T_w)$ are also required. SRGMs have been used for $m(T)$ and the model that best describes the reliability growth during the testing phase needs to be chosen. To estimate the number of failures in the warranty period, models for the operational phase should be used. A typical cost function with an S-shaped reliability growth curve can take the form (or a part of it) of the curve, as shown in Fig. 25.7.

In the release-time problems discussed in the literature, it has been assumed that failures will occur in the operational phase in the same manner as they do during testing [25.15]. Though the testing environment is designed such that it best represents the operational phase, the intensity of use of the software may differ. It is shown below how the simple Goel–Okumoto model [25.10] can be modified for the purpose. For the failure phenomenon of software in operational phase, the following differential equation is proposed:

$$\frac{\mathrm{d}}{\mathrm{d}t} m^*(t) = b_1 [a_1 - m^*(t)] \,, \quad t > T \,. \tag{25.29}$$

This equation is based on the assumption that failures during the operational phase are dependent on the number of faults remaining in the software at and after the time of release. Again the rate at which failures will occur with respect to the remaining faults is dependent on perceived usage of the software during this period. It is also assumed that upon a failure the corresponding fault is to be removed, at least during the warranty period. Our primary interest during this phase is to count the failures, as this directly translates to very high costs on account of risk, loss of goodwill and removal of faults or replacement of the entire software. The number of faults remaining in the software at time T is,

$$a_1 = [a - m(T)] = a - m(T) \,. \tag{25.30}$$

It is expected that there would be no fault generation during debugging in this phase. It is also assumed that the

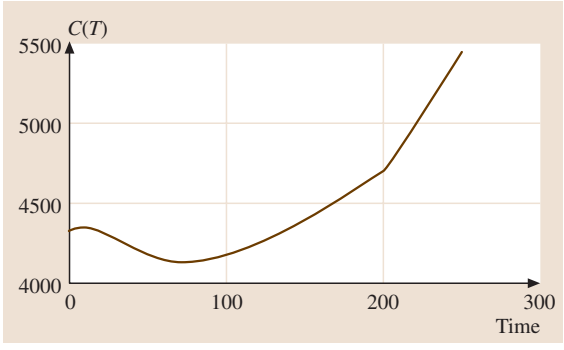

Fig. 25.7 The cost function

usage in the operational phase differs from that during the testing phase by a constant factor q. Hence the new rate is $b_1 = bq$. If $q = 1$, the intensity of use of the software during both these phases is similar. For $q < 1$ ($q > 1$) the software is expected to be used less (more) intensely during the operational phase. The solution of the differential equation (25.29) with the initial condition $m^*(T) = 0$ is

$$m^*(t) = a_1 \left[1 - \mathrm{e}^{-b_1(t-T)}\right], \quad t > T, \tag{25.31}$$

where $m^*(t)$ represents the expected number of failures in operational phase by time 't'. It is assumed that the failure phenomenon is still governed by an NHPP but with a new mean-value function.

Reliability Functions

Reliability expressions for NHPP software reliability models can easily be derived [25.6, 8, 10]. Software reliability is defined as the probability that the software operates failure-free for a specified time interval, on the machines for which it was designed, with the condition that the last failure occurred at a known time epoch. If the fault-detection process follows an NHPP then it can be shown that the software reliability at time t for a given interval $(t, t+x)$ is given by,

$$R_{\mathrm{te}}(x|t) = \mathrm{e}^{-[m(t+x)-m(t)]}. \tag{25.32}$$

Software reliability at time 't' during the user phase is defined as the probability of nonoccurrence of failure in the interval $(t,\ t+x]$, $x \geq 0$, $t > T$; in the operational environment. The definition is similar to the definition for the testing phase. A mathematical expression for the same can be derived using the SRGM (25.31) and the NHPP assumption. The following expressions results [25.6, 22].

$$R_{\mathrm{op}}(x|t) = \mathrm{e}^{-[m^*(t+x)-m^*(t)]}, \quad t > T. \tag{25.33}$$

The reliability curves for different values of q for a particular data set are given in Fig. 25.8. It is also observed that, for particular values of a, b and p, the operational reliability curve lies above the testing reliability curve i. e. $R_{\mathrm{op}}(x|t) \geq R_{\mathrm{te}}(x|t)$, $t \in [0, \infty)$ when $q = 1$. This result agrees with that derived in [25.16] when testing and operational profiles are identical.

25.2.1 Release-Time Problem Formulations

The release-time problem of software is to find a testing termination time T^* from an optimal tradeoff between cost and reliability. Many optimization problems

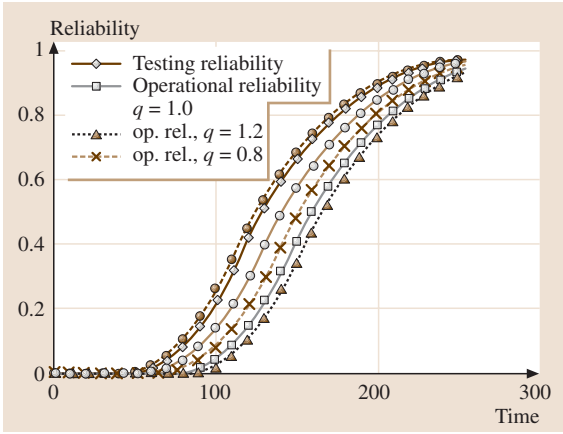

Fig. 25.8 Reliability curves for various q

have been formulated in the literature for this purpose [25.13, 15, 16, 33–36]. These problems select one or more functions from the lists of objective functions and constraints.

Objectives

(O1) Cost function: minimize $C(T)$;
(O2) Reliability functions:

1. Maximize $R_{\mathrm{te}}(x|t)$,
2. Maximize $R_{\mathrm{op}}(x|t)$.

Constraints

(C1) Budget constraint: $C(T) \leq B$;
(C2) Reliability constraints:

1. $R_{\mathrm{te}}(x|t) \geq R_{\mathrm{t}}$,
2. $R_{\mathrm{op}}(x|t) \geq R_{\mathrm{o}}$.

Where R_{t} and R_{o} are minimum reliability requirements for the testing and operational reliabilities respectively. In Table 25.4 some release-time problem formulations [25.3, 8] have been presented.

(R1) is the easiest problem formulation and is applicable for routinely developed software for which requirements are well defined. (R2) should be chosen for safety critical systems, where reliability is of utmost importance. (R4) is the most general problem formulation but is the most difficult to solve. A number of methods, including visual inspection of the cost curve, calculus, nonlinear programming, dynamic programming and neural networks etc. [25.3, 12, 15, 36], have been applied to find the optimal solution. Optimization software packages can also be used for this purpose.

Table 25.4 Release-time problems

Release time problem	Objective(s)	Constraints(s)
(R1) Cost criterion	Cost function (O1)	None
(R2) Reliability criterion	Reliability functions (O2-a) and (O2-b)	Reliability constraints (C2-a) and (C2-b)
(R3-A) Cost-reliability critera	Cost function (O1)	Reliability constraints (C2-a) and (C2-b)
(R3-B) Reliability-cost critera	Reliability functions (O2-a) and (O2-b)	Budget constraint (C1)
(R4) Bicriterion release criteria	Cost function (O1) Reliability functions (O2-a) and (O2-b)	Budget constraint (C1) Reliability constraints (C2-a) and (C2-b)

25.3 Control Problem

Before the release of software, a target reliability level is fixed. A reliable estimate of the fault content of software can also be obtained. Hence the management may desire to remove a certain percentage of it before release. But during the testing phase it is frequently realized that this may not be achievable for a number of reasons, such as inadequacy of the testing effort, inefficiency of the testing team etc. Hence, there is a need to increase the fault-removal rate. The problem of accelerating fault removal to achieve a certain reliability level or to a remove a certain percentage of total fault content of software is known as the testing-effort control problem. *Yamada* and *Ohtera* [25.19] took the software reliability growth modeling approach to solve this problem. In this section a new method to accelerate fault removal using SRGMs is proposed.

Additional Notations Used in this Section

α: Total testing resources to be eventually consumed, a constant;
$W_o(t)$: Cumulative testing effort on failure observation;
$W_r(t)$: Cumulative testing effort on fault removal;
$m_f(t)$: Number of failures observed in $(0, t]$;
b_o, b_r: Constants of proportionality, denoting rates of failure observation and fault removal, respectively;
m^*: Number of faults desired to be removed in time $(0, T_2]$;
$W(t - T_1)$: $W(t) - W(T_1)$, T_1 is the time duration;
a_1: $a - m(T_1)$;
a_2: $a - m_f(T_1)$.

25.3.1 Reliability Model for the Control Problem

The management of a software development project has time schedules for testing and release of software, but it is ignorant about the number and nature of faults lying dormant in it before the testing is actually done. SRGMs help in this regard after testing has been carried out for a certain period. The estimated parameters of the selected SRGM provide information about the number of faults remaining and the efficiency of the testing effort. Hence the expected number of faults that will be removed at any time in the future can be forecasted if the effort follows a known pattern. Frequently, management aspires to a reliability level at release that can be interpreted in terms of remaining number of faults. When the forecasted number of faults falls below the desired number, the testing effort needs to be controlled [25.18]. One obvious method (method I) is to increase the intensity of the testing effort through the employment of more manpower, computer time etc. But with limited resources available, this may not be feasible. Here it is shown how fault removal can also be accelerated by manipulating the allocation of testing resources to the two processes of failure observation and fault removal (method II). The models developed earlier in this chapter do not distinguish between fault identification and removal phenomenon. For the solution of the control problem we use the following SRGM.

The software testing phase aims to observe the failure process and remove the cause of the failure (the removal process). It is observed that different amounts of testing resources are consumed by each of these

processes. In SRGMs developed in the literature, the time-dependent behavior of the testing effort and the consequent reliability growth has been studied.

Yamada et al. [25.37], have given an SRGM incorporating the time lag between failure observation and fault removal. *Kapur* et al. [25.3] developed an S-shaped SRGM based on an NHPP to model the relationship between fault removal and testing effort. The cumulative testing effort was taken as a weighted sum of resources spent on fault observation and removal processes. We modify these SRGMs here. It is a common experience that, during early stages of testing, a large number of failures are observed, while the corresponding fault removals are lower. On the other hand, during later stages of testing, failures are harder to observe. Hence the failure-detection and the fault-removal processes should be studied distinctly.

Let $q_o(t)$ and $q_r(t)$ be the proportions of testing effort used on the failure-observation and fault-removal processes i. e. $q_o(t) = \frac{W_o(t)}{W_o(t)+W_r(t)} = \frac{W_o(t)}{W(t)}$ and $q_r(t) = \frac{W_r(t)}{W(t)}$. Then $q_o(t)+q_r(t) = 1$. If we assume $q_o(t) = q$ and $q_r(t) = (1-q)$, where q is a constant lying between 0 and 1. Then $qW(t)$ denotes the testing effort on failure observation and $(1-q)W(t)$, the effort on fault correction in the interval $(0, t]$. The NHPP-based SRGM developed below is based on the following assumptions:

1. No new faults are introduced into the software system during the testing phase.
2. The rate of fault removal to the current testing effort on removal is proportional to the number of identified faults that are yet to be removed at that instant.

The assumptions take the form of the following differential equations

$$\frac{m_f'(t)}{qw(t)} = b_o\,[a - m_f(t)]\ , \tag{25.34}$$

$$\frac{m'(t)}{(1-q)w(t)} = b_r\,[m_f(t) - m(t)]\ . \tag{25.35}$$

Solving the above system of equations with the initial conditions $m_f(t=0) = 0$ and $m(t=0) = 0$, we get

$$m(t) = a\left\{1 - \frac{1}{-b_o q + b_r(1-q)} \times \left[b_r(1-q)\mathrm{e}^{-b_o qW(t)} - b_o q\,\mathrm{e}^{-b_r(1-q)W(t)}\right]\right\}. \tag{25.36}$$

Equation (25.36) represents the cumulative number of faults removed, with respect to the testing effort consumed in the interval $(0, t]$. The time-dependent testing-effort function can have any of the forms presented in the preceding subsection. The removal function (25.36) is an S-shaped growth curve, because of the time lag between failure observation, the removal of the corresponding fault and the nature of the effort function.

For $q = 1$, the model reduces to the exponential model due to *Goel* and *Okumoto* [25.10]. In this case the process consists of a single step, i. e. faults are removed as soon as they are identified. With increasing $(1-q)$, the effort on removal increases. Hence the SRGM (25.36), captures the severity in faults present in a software. The model has been validated on actual software reliability data sets [25.17].

25.3.2 Solution Methods for the Control Problem

Method I

Suppose that software has been tested for time T_1 and it is to be released by time T_2, $T_2 > T_1$. Using the test data for the interval $(0, T_1]$ the parameters of the SRGM (25.36), can be statistically estimated. The testing effort in this interval is $W(T_1)$ and the corresponding number of faults that have been removed is $m(T_1)$. Based on the estimates of parameters, the number of faults expected to be removed by time T_2 is,

$$m(T_2) = a\left\{1 - \frac{1}{-b_o q + b_r(1-q)} \times \left[b_r(1-q)\mathrm{e}^{-b_o qW(T_2)} - b_o q\,\mathrm{e}^{-b_r(1-q)W(T_2)}\right]\right\} \tag{25.37}$$

The difference $[m(T_2) - m(T_1)]$ is the number of faults that is expected to be removed in the interval $(T_1, T_2]$. Often the management aspires to a level of reliability for the software at the time of release, which can be translated in terms of the number of faults (m^*) that it

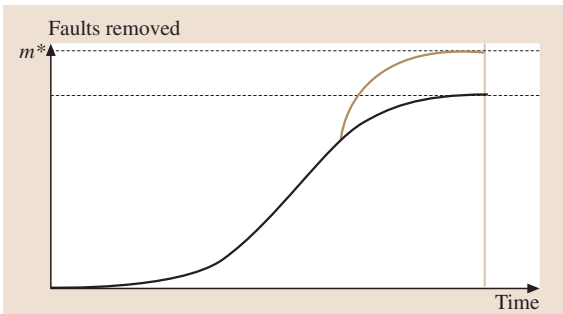

Fig. 25.9 Testing effort control

desired to be removed. If $m^* > m(T_2)$, the fault-removal rate has to be increased. This control problem is depicted in Fig. 25.9.

First the testing effort required to remove $[m^* - m(T_2)]$ faults in the time interval $(T_1, T_2]$ is calculated. Using the assumptions for the SRGM (25.36), the following expression results for $m(t)$, $t > T_2$

$$m(t) = m(T_1) + a_1\left(1 - \mathrm{e}^{-b_\mathrm{r}(1-q)W(t-T_1)}\right) - \frac{a_2 b_\mathrm{r}(1-q)}{-b_\mathrm{o}q + b_\mathrm{r}(1-q)} \times \left(\mathrm{e}^{-b_\mathrm{o}qW(t-T_1)} - \mathrm{e}^{-b_\mathrm{r}(1-q)W(t-T_1)}\right) . \tag{25.38}$$

In the above equation if m^* is substituted for $m(t)$ and W^* for $W(t-T_1)$, the following expression results,

$$m^* = m(T_1) + a_1\left\{1 - \mathrm{e}^{-b_\mathrm{r}(1-q)W^*}\right\} - \frac{a_2 b_\mathrm{r}(1-q)}{-b_\mathrm{o}q + b_\mathrm{r}(1-q)} \times \left\{\mathrm{e}^{-b_\mathrm{o}qW^*} - \mathrm{e}^{-b_\mathrm{r}(1-q)W^*}\right\} . \tag{25.39}$$

With the values of $m^*, m(T_1), a_1, b_o, a_2, b_r$ and q being known, equation (25.39) can be solved numerically to obtain the value of W^*, i. e. the amount of additional resources needed.

Method II

An alternative way to achieve the desired fault-detection level is to change the allocation factor of resources to be spent on the failure-identification and fault-removal processes. During the early stages of the testing phase a large number of failures may be observed, while the corresponding fault-removal rate is lower. This is due to the latency time needed by the removal team to cope with the workload. In this case it is reasonable to allocate resources in order to increase the testing effort of the failure-removal team, which may stimulate removal. On the other hand, during the late stages of testing, failures may be hard to identify and the removal team would have had enough time to remove most of the faults. Thus the fault removal will slow down, due to the lower number of failure observations. Hence it is more logical to assign more resources to the failure-identification team. Again during the later stages of testing it may happen that most of the failures had already been identified but not removed. Ideally the testing effort should now be concentrated on removal.

As discussed above, removal can be accelerated through proper allocation of resources. The optimal proportion of resources to be allocated to the failure-identification process, q^* to remove m^* faults can be found by solving the following equation numerically

$$m^* = m(T_1) + a_1\left(1 - \mathrm{e}^{-b_\mathrm{r}(1-q^*)W(T_2-T_1)}\right) - \frac{a_2 b_\mathrm{r}(1-q^*)}{-b_\mathrm{o}q^* + b_\mathrm{r}(1-q^*)} \times \left(\mathrm{e}^{-b_\mathrm{o}q^*W(T_2-T_1)} - \mathrm{e}^{-b_\mathrm{r}(1-q^*)W(T_2-T_1)}\right) . \tag{25.40}$$

The proportion of testing effort to be spent on fault removal during the time interval $(T_1,T_2]$ is $(1-q^*)$. In the following section, the results derived through the two methods are illustrated numerically.

In the literature only method I has been proposed for the control problem, but frequently management has to deal with limited testing resources. Method II, which increases fault removal through proper segregation of resources, can provide a solution. Moreover, following this method, testing can be done more efficiently by constantly monitoring the effort and fault removal and then allocating the optimal proportion of resources to the testing teams [25.38]. The control problem and the solution methods can be further refined. There can be an upper limit other than the initial fault content on the number of faults that can be removed by changing the allocation factor. Again the sensitivity analysis of the parameters with respect to the estimates and the optimal solution can provide further insight into the optimal allocation of testing resources. An early release of software is always desirable, but this should not undermine its quality. Also with a limited budget, decision making on the duration of testing becomes complicated. Hence cost–reliability criteria for the release of software and control of the testing effort should be jointly considered.

25.4 Allocation of Resources in Modular Software

Large software consists of modules. Modules can be visualized as independent pieces of software performing predefined tasks, mostly developed by separate teams of programmers and sometimes at different geographical locations. These modules are later integrated to form complete software. In module testing, each module is

tested independently and the software environment is simulated [25.39]. Typically this phase consumes 25% of the total development effort. In this phase the objective is to remove the maximum number of faults lying dormant in the modules. Though no conclusion can be drawn on system reliability at this stage, it is definitely enhanced with each fault removal. However, the testing has to be concluded within a specified time, which calls for proper allocation of limited resources among modules. This gives rise to the management problem of maximization of total fault removal within a finite time period or testing resource budget. In this section, we have formulated it as a mathematical programming problem. Again all modules are not equally important neither do they contain an equal number of faults. The severity of a fault in each module can also differ. In the light of this we have proposed another mathematical programming problem in that section. To arrive at a solution to this problem a mathematical relationship between testing resource consumption and fault removal is required. SRGMs have been used as a tool to monitor the progress of the testing phase by quantifying various reliability measures of the software system such as reliability growth, remaining number of faults, mean time between failures, testing effort etc. One approach is due to *Musa* et al. [25.6], where they have assumed that the resource consumption isan explicit function of the number of faults removed and calendar time. We develop a new model for the testing effort, which is solely dependent upon fault removal. The mathematical model is based upon the marginal testing effort function (MTEF), is defined as the effort required to remove an additional fault at any time. MTEFs are different in each module, depending on the severity of the faults in it, and using them we determine the optimal allocation of testing resources among modules; this has never been used before for optimal allocation of resources.

Using a simpler form of the MTEF a closed-form solution for optimal allocation of testing resource is obtained. It is very important to give a plausible form, but, as seen in another optimization problem, when more practical constraints are added to the same problem, no closed-form solution could be obtained. The problem is then solved as a nonlinear programming problem using a software package.

Additional Notations for this Section

m_i: Number of faults removed in the *i-th* module;
c_i: Cost of unit testing effort in the *i-th* module;
α_i: Relative importance of module i, $\sum_{i=1}^{N} \alpha_i = 1$;
$w(m)$: Marginal testing effort when m faults have been removed;
$W(m)$: Cumulative testing effort when m faults have been removed;
$W_i(m)$: Cumulative effort to remove m faults in the *i-th* module;
$w_i(m)$: Marginal testing effort function in the *i-th* module;
B: Total testing resource available; budget;
r_i: Minimum number of faults desired to be removed in the *i-th* module;
N: Number of modules;
k', p, q, k: Constants of proportionality;
p_i, q_i, k_i: Parameters of marginal testing effort function for the *i-th* module;
a_i: Number of faults in the *i-th* module;
d_i: $c_i k_i$;
δ, γ: Probability of imperfect (perfect) debugging, $0 \le \delta, \gamma \le 1$;
δ_i, γ_i: Probability of imperfect (perfect) debugging, $0 \le \delta_i, \gamma_i \le 1$.

25.4.1 Resource-Allocation Problem

Consider software with 'N' modules. During module testing each module is tested independently. We assume that the modules have a finite number of faults and we aspire to remove the maximum number of them. Testing resources such as manpower and computer time are used and the management has to allocate limited testing resources among the modules. This problem of optimal allocation of testing resources among modules can be formulated as a mathematical programming problem, which is given below.

$$\text{maximize} \quad \sum_{i=1}^{N} m_i \,,$$

subject to

$$\sum_{i=1}^{N} c_i W_i(m_i) = B \,. \tag{25.41}$$

Kubat and *Koch* [25.18] have used SRGMs and through the method of Lagrangian multipliers have obtained solution to the above problem. However, this method does not rule out the possibility of negative allocation of resources to some modules. To correct this, algorithms that are similar in nature have been proposed in these

papers; they sequentially give zero allocations to these modules and distribute the values among the others. But this proposition is also not suitable, as testing cannot be stopped abruptly. As a way out, management can decide upon some minimum number of faults that it expects to remove from each module.

All modules of software are not equally important. The relative importance of modules can be determined based upon the frequency with which modules are expected to be called for execution in the actual user environment. Accordingly weights can be attached to each module. Incorporating this, the optimization problem above (25.41) can be reformulated as

$$\text{maximize} \quad \sum_{i=1}^{N} \alpha_i m_i$$

subject to

$$m_i \geq r_i \,, \quad i = 1, \ldots, N \,,$$
$$\sum_{i=1}^{N} c_i W_i(m_i) = B \,. \tag{25.42}$$

A functional relationship between the testing effort and fault removal is needed before we solve (25.41) or (25.42). SRGMs can be used for this purpose. We adopt the reverse approach and develop a model for resource consumption vis-à-vis fault removal in the next section.

25.4.2 Modeling the Marginal Function

Most SRGMs depict reliability growth with reference to execution time. Only a few SRGMs incorporate the effect of a time-dependent testing-effort pattern. Testing-effort components such as manpower utilization are dependent on the outcome of testing. Hence how the resources are consumed with each failure and removal attempt is a very important factor during decision making on resource allocation. Therefore we formulate an MTEF that gives a functional relationship between testing and fault removal. The time factor is not explicitly present in the model. Marginal testing effort (MTE) is the amount of effort required to remove an additional fault at any given time. Hence, if m faults have already been removed from the software, the MTE is the testing effort required to remove the $(m+1)$-th fault. We propose a mathematical relationship between the MTE and the number of faults removed based upon the assumption that the MTE is inversely proportional to the remaining faults in the software, i. e. the more faults we remove, the greater effort would be required to remove the next fault.

Mathematically this can be written as

$$w(m) = \frac{k'}{a-m} \tag{25.43}$$

$$\text{and} \quad W(m) = k' \ln \frac{a}{a-m} \,. \tag{25.44}$$

In this expression it is also implicitly assumed that the software contains a finite number of faults at the initiation of testing, that fault removal is perfect and that no new faults are introduced in the process. These assumptions are similar to those used by *Goel* and *Okumoto* [25.10] for their SRGM, i. e. the rate of fault removal is proportional to the remaining faults at any given time. The SRGM is with respect to execution time and it is the mean-value function of the underlying stochastic process described by an NHPP. As the optimization problems being studied in this chapter are with respect to testing resource consumption, MTEF is better suited for our purpose. But the variability in the nature of relationship between the variables i. e. resource consumption and fault removal needs to be captured in a MTEF. It should also include the effect of learning on the testing team. With each additional fault removed, some more faults lying on the execution path are removed and the testing team also gains insight into the software. To incorporate this, we assume that, the MTE is also inversely proportional to a linear function of the number of faults removed. Hence (25.44) can be written as,

$$w(m) = \frac{k}{(p+qm)(a-m)} \tag{25.45}$$

$$\text{and} \quad W(m) = \frac{k}{p+aq} \ln \frac{a(p+qm)}{p(a-m)} \,. \tag{25.46}$$

A higher value of q denotes a higher rate of learning of the testing team and implies a growth rate in the MTE. In expression (25.45) it is assumed that, on a failure, the fault causing that failure is immediately removed with unit probability. Though every care is taken to correct the cause of a failure, the possibility of imperfect debugging and fault generation cannot be ruled out [25.3, 6, 40]. If the fault remains, even after debugging, then it is said to be imperfectly debugged. New faults can also be introduced during the removal process. In both ways the fault content enhances the chance of failure in future. As this phenomenon is a reality, the MTEF should ideally contain the effect of imperfect debugging and fault generation. We modify (25.45) through the assumption that the number of faults imperfectly debugged and generated during the debugging phenomenon is dependent

upon the number of removal attempts already made. The following expression results from these assumptions:

$$w(m) = \frac{k}{(p+qm)} \frac{1}{(a+\delta m - m)} = \frac{k}{(p+qm)} \frac{1}{(a-\gamma m)} , \quad (25.47)$$

$$W(m) = \frac{k}{p\gamma + aq} \ln \frac{a(p+qm)}{p(a-\gamma m)} . \quad (25.48)$$

The modules of a piece of software are independent pieces of software themselves. Hence the least-squares method suggested above can be used to estimate the parameters of the MTEFs of the different modules. For this it is required that modules have already been tested for some time and data pertaining to failures and resource consumption has been recorded.

25.4.3 Optimization

During module testing, modules are tested independently, i. e. the testing teams are different. Again each module can be visualized as independent software and hence distinct MTEFs can be used to describe their testing resource consumption. After the modules have been tested for some time, the parameters of the MTEF viz. a_i, p_i, q_i, k_i can be estimated. Based upon these estimates optimal allocation of resources among modules can be calculated.

Using the MTEF (25.44), the optimization problem (25.41) can be formulated as

$$\text{maximize} \quad \sum_{i=1}^{N} m_i$$

$$\text{subject to} \quad \sum_{i=1}^{N} c_i k_i \ln \frac{a_i}{a_i - m_i} = B . \quad (25.49)$$

We can solve this problem by the method of Lagrangian multipliers. Defining the Lagrange function as

$$L(m_1, m_2, \ldots, m_N, \theta) = \sum_{i=1}^{N} m_i - \theta \left(\sum_{i=1}^{N} c_i \int_0^{m_i} \frac{k_i}{a_i - x} \mathrm{d}x - B \right) , \quad (25.50)$$

we get the following optimality conditions

$$\frac{c_i k_i}{a_i - m_i} = \text{constant} \quad \forall i \quad (25.51)$$

$$\text{and} \quad \sum_{i=1}^{N} c_i k_i \ln \frac{a_i}{a_i - m_i} = B . \quad (25.52)$$

After some algebraic simplifications, from (25.52) and (25.52) we obtain

$$W_i^* = \frac{B - \ln \prod_{i=1}^{N} \left(\frac{a_j d_i}{d_j a_i} \right)^{d_j}}{\sum_{j=1}^{N} \frac{d_j}{d_i}} , \quad i = 1, \ldots, N . \quad (25.53)$$

Which is the optimal allocation of testing resources for the *i-th* module in terms of B, a_i and d_i. We have used here the simplest among the MTEFs proposed above. Though obtaining closed-form solution such as (25.53) is always desirable, arriving at one becomes nearly impossible if (25.41) is made more complex. Even with the other two MTEFs in (25.45) and (25.47) the method of Lagrangian multipliers does not directly provide a solution. As the objective here is to highlight the use of marginal effort modeling in allocation problems, we formulate these optimization problems as nonlinear programming problems that could be solved by any of the known methods.

In the solution obtained through (25.53) some modules can receive zero allocations. Hence the minimum number (percentage) of faults that are desired to be removed from each module should be added as a constraint, as in (25.42). Consider the following optimization problem where (25.45) and (25.47) have been substituted into the problem (25.42).

$$\text{Maximize} \quad \sum_{i=1}^{N} \alpha_i m_i$$

subject to

$$m_i \geq r_i , \quad i = 1, \ldots, N ,$$

$$\sum_{i=1}^{N} \frac{c_i k_i}{p_i + a_i q_i} \ln \frac{a_i (p_i + q_i m_i)}{p_i (a_i - m_i)} = B . \quad (25.54)$$

With the MTEF (25.47) the resource constraint takes the following form (25.55), the objective function and the other constraint remaining the same in the problem:

$$\sum_{i=1}^{N} \frac{c_i k_i}{p_i \gamma_i + a_i q_i} \ln \frac{a_i (p_i + q_i m_i)}{p_i (a_i - \gamma_i m_i)} = B . \quad (25.55)$$

Equations (25.54) and (25.55) are nonlinear programming problems and any of the standard methods can be used to solve them. But when the number of modules increases, deriving the solution manually becomes difficult. We have solved the problem above with

the help of a software a packages for higher numbers of modules. Once the optimal m_i are found, they can be substituted into (25.44) or (25.48) to find the optimal allocation of resources to the modules. Optimization techniques such as dynamic programming and fuzzy mathematical programming have also been used by the authors for solving more complex resource-allocation problems [25.20, 41].

References

25.1 X. Zhang, H. Pham: An analysis of factors affecting software reliability, J. Syst. Softw. **50**, 43–56 (2000)

25.2 S. Bittanti, P. Bolzern, E. Pedrotti, R. Scattolini: *A flexible modelling approach for software reliability growth*, ed. by G. Goos, J. Harmanis (Springer Verlag, Berlin Heidelberg New York 1988) pp. 101–140

25.3 P. K. Kapur, R. B. Garg, S. Kumar: *Contributions to hardware and software reliability* (World Scientific, Singapore 1999)

25.4 P. K. Kapur, A. K. Bardhan, O. Shatnawi: On why software reliability growth modeling should define errors of different severity, J. Indian Stat. Assoc. **40(2)**, 119–142 (2002)

25.5 M. R. Lyu (Ed.): *Handbook of Software Reliability Engineering* (McGraw Hill, New York 1996)

25.6 J. D. Musa, A. Iannino, K. Okumoto: *Software Reliability: Measurement, Prediction, Applications* (McGraw Hill, New York 1987)

25.7 M. Ohba: Software reliability analysis models, IBM J. Res. Dev., **28**, 428–443 (1984)

25.8 H. Pham: *Software Reliability* (Springer Verlag, Singapore 2000)

25.9 M. Xie: *Software reliability modelling* (World Scientific, Singapore 1991)

25.10 A. L. Goel, K. Okumoto: Time dependent error detection rate model for software reliability and other performance measures, IEEE Trans. Reliab. R **28(3)**, 206–211 (1979)

25.11 W. D. Brooks, R. W. Motley: *Analysis of discrete software reliability models, Technical Report RADC-TR-80-84* (Rome Air Development Center, New York 1980)

25.12 T. Dohi, Y. Nishio, S. Osaki: Optimal software release scheduling based on artificial neural networks, Ann. Softw. Eng. **8**, 167–185 (1999)

25.13 P. K. Kapur, R. B. Garg: Optimal software release policies for software reliability growth models under imperfect debugging, Recherche Operationnelle – Oper. Res. **24(3)**, 295–305 (1990)

25.14 M. Kimura, T. Toyota, S. Yamada: Economic analysis of software release problems with warranty cost and reliability requirement, Reliab. Eng. Syst. Safety **66**, 49–55 (1999)

25.15 S. Yamada, S. Osaki: Optimal software release policies with simultaneous cost and reliability requirements, Eur. J. Oper. Res. **31(1)**, 46–51 (1987)

25.16 B. Yang, M. Xie: A study of operational and testing reliability in software reliability analysis, Reliab. Eng. Syst. Safety **70**, 323–329 (2000)

25.17 A. K. Bardhan: Modelling in software reliability and its interdisciplinary nature. Ph.D. Thesis (Univ. of Delhi, Delhi 2002)

25.18 P. Kubat, H. S. Koch: Managing test procedures to achieve reliable software, IEEE Trans. Reliab. **39(2)**, 171–183 (1993)

25.19 S. Yamada, H. Ohtera: Software reliability growth model for testing effort control, Eur. J. Oper. Res. **46**, 343–349 (1990)

25.20 P. K. Kapur, P. C. Jha, A. K. Bardhan: Optimal allocation of testing resource for a modular software, Asia Pac. J. Oper. Res. **21(3)**, 333–354 (2004)

25.21 G. Q. Kenny: Estimating defects in a commercial software during operational use, IEEE Trans. Reliab. **42(1)**, 107–115 (1993)

25.22 F. M. Bass: A new product growth model for consumer durables, Man. Sci. **15(5)**, 215–224 (1969)

25.23 M. Givon, V. Mahajan, E. Muller: Software piracy: estimation of lost sales and the impact on software diffusion, J. Market. **59**, 29–37 (1995)

25.24 S. Inoue, S. Yamada: Testing-coverage dependent software reliability growth modeling, Int. J. Qual. Reliab. Safety Eng. **11**(4), 303–312 (2004)

25.25 H. Pham, X. Zhang: NHPP software reliability and test models with testing coverage, Eur. J. Oper. Res. **145**, 443–454 (2003)

25.26 H. Yamada, H. Ohtera, H. Narihisa: Software reliability growth models with testing effort, IEEE Trans. Reliab. **R-35(1)**, 19–23 (1986)

25.27 M. Trachtenberg: A general theory of software reliability modeling, IEEE Trans. Reliab. **39(1)**, 92–96 (1990)

25.28 C-Y. Huang, S-Y. Kuo, J. Y. Chen: Analysis of a software reliability growth model with logistic testing effort function, Proc. 8th Int. Symp. Softw. Reliab. Eng., November 1997, pp. 378–388

25.29 S. Yamada, J. Hishitani, S. Osaki: Software-reliability growth model with a Weibull test effort: a model and application, IEEE Trans. Reliab. **42(1)**, 100–106 (1993)

25.30 H. Pham, L. Nordmann, X. Zhang: A general imperfect software-debugging model with S-shaped fault detection rate, IEEE Trans. Reliab. **R-48**, 169–175 (1999)

25.31 P. K. Kapur, R. B. Garg: A software reliability growth model for an error removal phenomenon, Softw. Eng. J. **7**, 291–294 (1992)

25.32 www.dacs.dtic.mil: Software reliability data; Data and Analysis Center for software, USA

25.33 H. Pham: A software cost model with warranty and risk costs, IEEE Trans. Comput. **48(1)**, 71–75 (1999)

25.34 P. K. Kapur, R. B. Garg, V. K. Bahlla: Release policies with random software life cycle and penalty cost, Microelectr. Reliab. **33(1)**, 7–12 (1993)

25.35 P. K. Kapur, R. B. Garg: Cost–reliability optimum release policies for software system under penalty cost, Int. J. Syst. Sci **20**, 2547–2562 (1989)

25.36 S. Yamada: Software reliability measurement during operational phase and its application, J. Comput. Softw. Eng. **1(4)**, 389–402 (1993)

25.37 S. Yamada, M. Ohba, S. Osaki: S-shaped software reliability growth modelling for software error detection, IEEE Trans. Reliab. **R-32(5)**, 475–484 (1983)

25.38 P. K. Kapur, A. K. Bardhan: Testing effort control through software reliability growth modelling, Int. J. Modelling Simul. **22(1)**, 90–96 (2002)

25.39 P. Kubat: Assessing reliability of modular software, Oper. Res. Lett. **8**, 35–41 (1989)

25.40 S. Yamada, K. Tokuno, S. Osaki: Imperfect debugging models with fault introduction rate for software reliability assessment, Int. J. Syst. Sci. **23(2)**, 2241–2252 (1992)

25.41 P. K. Kapur, P. C. Jha, A. K. Bardhan: Dynamic programming approach to testing resource allocation problem for modular software, J. Ratio Math. **14**, 27–40 (2003)