13

Software Reliability: A Quantitative Approach

Yashwant K. Malaiya
Colorado State University

CONTENTS

13.1 Introduction

Software now controls everyday life of individuals involving commerce or social interactions, in addition to critical applications such as aviation or banking. This makes ensuring software reliability a major concern.

Software failures are known to cause enormous damage to the society, estimated to be in excess of a trillion dollar for 2017 by Tricentis [1]. During the past decades, very high reliability was expected mainly in the critical systems. Today software packages used every day also need to be highly reliable, because the enormous investment of the software vendor, as well as the user, is at stake. Studies have found that in software, reliability is considered to be the most important characteristic of a program by the customers.

It is virtually impossible to develop fault tree software unless the code needed is trivially small. All programs developed have defects regardless of how thorough the program development was. While tools have emerged that assist the software developers, program complexity has risen exponentially, sometimes doubling every year for some applications [2]. The programs must be tested for bugs and debugged until an acceptable reliability level is achieved. In any practical software system, the developer cannot assure that all the defects in it have been found and removed. However, the developer must ensure that the remaining bugs are few and do not have a catastrophic impact. Any software project must be released within the expected time to minimize loss of reputation and hence the market share. The developer must assess the risk associated with the residual bugs and needs to have a plan for achieving the target reliability level by the release date.

Quantitative methods are now coming in to use for obtaining and measuring reliability because of the emergence of validated and well-understood approaches. Data from industrial and experimental sources is now available that can be used to develop methods for achieving high reliability and validating them. The acceptable standards for software reliability levels have gradually risen in the past several decades.

This article takes the view that there exist several components and aspects of software reliability and they need to be integrated into a systematic framework. It presents the essential concepts and approaches needed for predicting, evaluating, and managing the reliability of software. The article first discusses the approaches taken during the successive phases of software development. Widely used measures of system software reliability and related metrics are next introduced. Detectability, a defect level metric, is introduced along with its implications. Factors that have been found to control software defect density are examined. Key ideas in test methodologies including the operational profile are presented. Nelson's model is presented in terms of the operational profile. Reliability growth in software is generally described using a software reliability growth model (SRGM). Here growth in reliability is illustrated and analyzed using actual test data. The use of such growth models allows estimation of the testing effort required to achieve the desired reliability goal. We also see how the reliability of a system can be evaluated if we know the failure rates of the components, and how reliability effort can be optimally allocated. Multi-version programming to achieve fault tolerance is also considered. Finally, the chapter presents the type of tools available for assisting in achieving and evaluating reliability.

The terms "failure" and "defect" are formally defined as follows [3]. The distinction is necessary to ensure a clear discussion.

Failure: a system behavior during execution that does not conform to the user requirements.

Defect: an error in software implementation that can cause a failure. It is also termed a fault or bug in literature.

A defect causes a failure when the code containing it is executed, and the resulting error propagates to the output. The probability of detecting a defect using a randomly chosen input is termed *testability* of a defect. Very low testability defects can be very hard to detect but can have a significant, perhaps even critical, impact on a highly reliable system.

As bugs are encountered and removed during testing, the reliability of the software improves. When the software project is released to the users, its reliability stays the same, as long as the same operating environment is used, and no patches are applied to modify the code. Bug fixes are often released in forms of patches updating the software, resulting in the next version of the code. For a program, its own past failure history is often a good indicator of its reliability, although data from other similar software systems may also be used for making projections [4].

13.2 The Software Life Cycle Phases

To achieve high reliability, reliability must be considered from the very beginning of software development. Different approaches are applicable during successive phases. The so-called waterfall model divides the software life cycle into these phases as discussed in the next paragraphs. However, for some development processes, it is common for developers to go to a previous phase, due to a need to make changes in the design or to respond to the required changes. It is much more cost-effective to find defects in an earlier phase because it can be significantly more expensive to address them later.

13.2.1 Requirements

In the requirements phase, the developing team interacts with the customer to understand and document the requirements the program needs. While in an ideal situation the requirements should specify the system unambiguously and completely, in actual practice, often corrective revisions during software development are needed. To identify missing or conflicting requirements, a review or inspection is conducted during this phase. It has been found that a significant number of bugs can be detected by this process. Any changes in the requirements in the later phases just before completion can increase the defect density.

13.2.2 Design

In the design phase, the overall architecture of the system is specified as an interconnection of small blocks termed *units*. The units should be well defined to be developed and tested independently. The design is reviewed to find errors.

13.2.3 Coding

In the coding phase, the actual code for each unit is first developed, generally in a higher level language such as Java, C, or C++. In the past, assembly level implementation was often required for high performance or for implementing input/output operations; however, with the emergence of highly optimizing compilers, the need to use assembly code has declined. The code developed is analyzed in team meetings to identify errors that can be caught visually.

13.2.4 Testing

This testing phase is the most significant phase when high reliability is needed and can take 30%–60% of the overall development effort. It is often divided into four phases:

1. *Unit testing*: In this phase, each unit is separately exercised using many test cases, and the bugs found are removed. As each unit is a relatively small piece of code, which can be tested independently, it can be tested much more exhaustively than a large program.
2. *Integration testing*: During this phase, the units are incrementally assembled, and at each step, partially assembled subsystems are tested. Testing subsystems permit the interaction among modules to be exercised. By incrementally incorporating the units within a subsystem, the unit responsible for a failure can be determined more easily.
3. *System testing*: In this phase, the completely integrated system is exercised. Debugging is continued until an appropriate exit criterion is

satisfied. During this phase, the emphasis is to find defects as quickly as possible. Researchers have pointed out that for highly reliable systems, the inputs should be drawn uniformly from the input partitions rather than using the frequencies that would be encountered during actual operation. However, when only limited testing can be afforded, perhaps due to an approaching deadline, actual operations should be simulated.

4. *Acceptance testing*: The objective of acceptance testing is to assess the system reliability and performance that would be encountered in the actual operational environment. This requires estimating the frequencies describing how the actual users would use the system. It is often termed alpha testing. Beta testing, which involves the use of the beta version released to some potential actual users, often follows next.

13.2.5 Operational Use

When the developer has determined that the suitable reliability criterion is satisfied, the software is released to the customers. In operational use, the bugs reported by the users are recorded so that they can be fixed when the next patch or bug fix is released.

13.2.6 Maintenance

After a release, the developer continues to maintain to code, until the retirement of the program in its current form. When major modifications or additions are made to an existing version, *regression testing* is done on the revised version to ensure that it still works and has not "regressed" to lower reliability because of potential incompatibility that may be injected. When a defect discovered is a security vulnerability, a patch for it needs to be developed as soon as possible. The time taken to develop and test a patch after the discovery of a vulnerability and the delayed application of an available patch can enhance the security risks. Support for an existing version of a software product needs to be offered until a new version has made a prior version obsolete. Eventually, maintenance is dropped when it is no longer cost-effective to the developing organization.

There can be some variation in the exact definitions of specific test phases and their exit criteria among different organizations. Some projects go through incremental refinements; for example, those using the extreme programming approach. Such projects may undergo several cycles of requirements–design–code–test phases.

The fraction of total defects introduced and found during a phase [5,6] will depend on the specific development process used. Table 13.1 gives

TABLE 13.1

Defects Introduced and Removed during the Successive Phases [7]

	Defects (%)		
Phase	**Introduced**	**Found**	**Still Remaining**
Requirements	10	5	5
Design	35	15	25
Coding	45	30	40
Unit testing	5	25	20
Integration testing	2	12	10
System testing	1	10	1

some representative fractions based on various studies. Majority of defects are encountered during design and coding. The fraction of defects found during the system test may be small; however, the phase can take a long time because the remaining defects require much more effort to be found. The testing phases can represent 30%–60% of the entire development effort.

13.3 Software Reliability Measures

In hardware systems, the reliability gradually declines because the possibility of a permanent failure increases. However, this is not applicable to software. During testing, the software reliability increases due to the removal of bugs and becomes constant when the defect removal is stopped. The most common software reliability measures include the following:

Availability: Using the classical reliability terminology, *availability* of a software system can be given as

$$A(t) = \Pr\{\text{system is operating correctly at instant } (t)\} \tag{13.1}$$

Transaction reliability: For software, the correctness of a single transaction is of major concern. A transaction-based reliability measure is often convenient to use.

$$R = \Pr\{\text{a single transaction will not fail}\} \tag{13.2}$$

Both these measures assume typical operation, that is, the input mix encountered obeys the *operational profile* as defined later.

Mean time to failure (MTTF): The mean time between two successive failures.

Failure intensity (λ): The mean number of failures per unit time is generally termed failure intensity. Note that failure intensity (per unit time) is inverse of MTTF. Thus, if the failure intensity is 0.005 per hour, MTTF is 1/0.005 = 200 hours.

Since testing strategy is specifically designed to find the bugs at, failure intensity λ_t during testing is significantly higher, perhaps by one or two orders of magnitude, than the failure intensity λ_{op} encountered during normal operation. Test acceleration factor describes the relative effectiveness of the testing strategy as measured by the ratio λ_t/λ_{op}. Thus, if testing is 10 times more efficient for finding defects compared with the normal use, the test acceleration factor is 10. This factor is determined by the testing strategy and the type of application being tested.

Defect density: By convention, the software defect density is measured in terms of the number of defects per 1,000 source lines of code (KSLOC), not including the comments. Since many defects will not be found during testing, it cannot be measured directly, but it is possible to estimate it using the static and dynamic models discussed in the later sections. It can also be estimated using sampling approaches. The failure intensity depends on the rate at which the faults are encountered and thus can be assumed to be proportional to the defect density. The target defect density for a critical system or a high volume software can be equal to 0.1 defects/KLOC, or perhaps lower, whereas for other applications 0.5 defects/KLOC may be considered acceptable. Sometimes weights are assigned to defects depending on the severity of the failures they can cause. To keep analysis simple, here we assume that each defect has the same weight.

Test coverage metrics: Several tools are available that can evaluate how thoroughly a program has been exercised by a given test suite. The measurement is generally done by instrumenting the code during compilation. The two most common structural coverage metrics are as follows:

- *Statement coverage*: The fraction of all statements executed during testing. It is related to the metric termed block coverage.
- *Branch coverage*: The fraction of all branches that were taken during the code execution. It is related to the decision coverage metric.

The test coverage is correlated with the number of defects that will be encountered during testing [8]. 100% statement coverage is relatively easy to achieve; however, achieving 100% branch coverage can be hard for large programs. Often a predetermined branch coverage, say 85% or 90%, is used as an acceptance criterion. It becomes exponentially more difficult in terms of the number of test cases needed to achieve higher levels of branch coverage.

13.4 Defect Detectability

A key attribute of a defect is its detectability. The detectability of a specific defect can be defined as the probability that the defect will be tested by a randomly chosen test [9]. While some faults may be easily detected, some faults can be very hard to find. As testing progresses, the remaining faults become harder and harder to find. At any point in time during testing, the probability of detecting new faults depends on the *detectability profile*, which describes the number of remaining faults with the associated detectability values. During the later phases of testing, the larger number of remaining faults are likely to be the ones that are harder to find since easier faults would have already been found and removed [10]. Sometimes the faults found during limited testing are regarded to be representative of all the remaining faults. However, it is likely that they represent the easier-to-find faults and the faults not yet found are likely to be the ones that are harder to find.

The probability that a test will result in the detection of a fault is given by

$$P\{\text{a new fault is detected}\} = \sum_{i=1}^{n} N_i d_i \tag{13.3}$$

The detectability of a fault depends on how frequently the site of the fault is traversed and how easy it is for the resulting error to propagate to the output. Many programs contain code that is only executed when an error is encountered and a recovery is attempted. Faults in such sections of the code have a low probability of discovery with normal usage [11] or with random testing.

13.5 Factors that Impact Software Defect Density

The researchers in the field have attempted to identify the factors that impact the defect density and have evaluated their relative significance. The data available allows us to develop simple models for predicting their impact. Such a model can be used by a developer team to see how they can enhance the process and thus achieve higher reliability of their products. In addition, estimating the defect density can allow one to use a reliability growth model to estimate the testing effort needed. Here we consider the model by Malaiya and Denton [12], which was obtained using the data reported in the literature. Their model incorporates several factors and is given by

$$D = C \cdot F_{ph} F_{pt} F_m F_s F_{cc} F_{ru} \tag{13.4}$$

The following factors are used for estimation:

- The *phase factor* F_{ph}, which takes into account the *software test phase.*
- The *programming team factor* F_{pt}, which is based on the capabilities of the software development team.
- The *maturity factor* F_{m}, which depends on the maturity of the development process as given by the capability maturity model (CMM).
- The *structure factor* F_{s}, which depends on the structure of the software under development.
- The *code churn factor* F_{cc} which models the impact of requirement volatility.
- The *code reuse factor* F_{ru}, which takes in to account the influence of code reuse.

The factor C is a constant of proportionality representing the default defect density per KSLOC. The default value of each factor is 1. We propose the following preliminary submodels for each factor:

The phase factor F_{ph}: A simple model can be constructed using actual data reported by Musa et al. and Piwowarski et al. as shown in Table 13.2, where the multiplier values refer to the beginning of the phase specified. The numbers given indicate the variability in the factor; actual values may depend on the software development process. The numbers imply that the defect density is decreased by an overall factor of about 12 as a result of testing in various test phases. The beginning of the system test phase is taken as the default case with the value of 1.

The programming team factor F_{pt}: The coding and debugging capabilities of the individuals in a development team can vary significantly. Takahashi and Kamayachi [13] conducted a study correlating the number of defects with the programmer's experience in years. Their simple linear model can take into account programming experience of up to 7 years, each year reducing the number of defects by about 14%. Since formal studies on this subject are few, we can use Takahashi and Kamayachi's model as a guide to suggest the model in Table 13.3 based on the average skill level in a team. In addition

TABLE 13.2
Variability in the Phase Factor F_{ph}

Phase	Factor Value at the Beginning of the Phase
Unit testing	4
Subsystem testing	2.5
System testing	1 (default)
Operation	0.35

TABLE 13.3
Variability in the Programming Team Factor F_{pt}

Programmers' Skill Level	Factor Value
Strong	0.4
Average	1 (default)
Weak	2.5

to programming experience, personal discipline and understanding of the problem domain can also influence defect density. Table 13.3 assumes that worst programmer correspond to the most inexperienced programmers, as studied in Ref. [13].

The capability maturity factor F_m: The Software Engineering Institute (SEI) CMM measures the maturity of the software process at an organization, which can range from ad hoc (level 1) to an optimizing one (level 5) where data is continually collected and used to refine the process. Table 13.4 uses the data collected by Jones and Keene as well as that reported in a Motorola study. Most organizations are believed to be at level 1; however, the table uses level II as the default level, since a level I organization is not likely to be using software reliability engineering.

The software structure factor F_s: This factor considers various aspects of the software structure including program complexity, language type (the fractions of code in assembly and high-level languages), and module size distribution. The assembly language code is harder to write and thus naturally will have a higher defect density. The impact of complexity, as measured by several complexity metrics, has been extensively investigated by researchers. Most complexity metrics have been found to be strongly correlated to software size as measured in the number of lines of code (LOCs). Since the defect density is being considered here, which is the number of defects divided by software size, software size has already been discounted.

The language dependence can be given by a simple model for F_s by considering a, the fraction of the code that is in assembly:

$$F_s = 1 + 0.4a$$

TABLE 13.4
Variability in the Process Maturity Factor F_m

CMM Level	Factor Value
Level 1 (Initial)	1.5
Level 2 (Repeatable)	1 (default)
Level 3 (Defined)	0.4
Level 4 (Managed)	0.1
Level 5 (Optimizing)	0.05

This assumes that assembly language code has 40% more bugs.

Any software project is composed of a collection of modules of various sizes. A module may be a function, a class, a file, and so on. The size of a module has some impact on the defect density. Very large modules may be hard to comprehend. When module sizes are very small, they can have a higher fraction of bugs related to the interaction among them. It has been found that module sizes tend to be asymmetrically distributed with a larger number of small modules. In such a case, module size distribution can impact defect density [14]. Additional research is needed to construct a model for this factor.

The code churn factor F_{cc}: This factor causes changes in the code because of changes in the requirements (also termed requirement volatility) [15,16]. It can be shown that this is a function of the fraction of code f_c that is changed and the time t_c when the change occurs as follows:

$$F_{cc}(t_c) = (1 - f_c) + f_c \; e^{\beta 1 t_c}$$

where $\beta 1$ is a parameter. The impact is higher when the change occurs later.

The code reuse factor F_{ru}: It is common for software projects to use some preexisting code that has already undergone prior testing and thus has a lower defect density [17]. If the defect densities of the reused and new codes are d_r and d_n, respectively, and if u is the fraction of code that is inherited, then it can be shown that factor is given by

$$F_{ru} = 1 - u + u\frac{d_r}{d_n}$$

The resulting defect density is lower if more of the code is reused.

Calibrating the defect density model: The multiplicative model given in Equation (13.4) can provide an early estimate of the defect densities. The factor C needs to be estimated using past data from preferably the same organization for similar projects. Other factors not considered here will be taken into account implicitly by calibrating the overall model. Since the beginning of the subsystem test phase is taken as the default in the overall model, an examination of the Musa's data sets suggests that the constant of proportionality C can range from about 6–20 defects per KSLOC. Some indeterminacy is inherent in such models. It can be accounted for by using an upper and a lower estimate and using both of them to make predictions.

Example 1

The value of the constant C has been found to range between 12 and 16 for an organization. A team in the organization is working on a new project. The team has similar capabilities as other past teams. About 80% of the code is in a high-level language, and the rest 20% is in assembly language. Half of the project reuses the previous code, which has one-fourth the defect density compared with the new code. Other factors are assumed to be what is typical for the organization. The software size is about 20,000 LOC. What is the expected number of total defects at the beginning of the integration test phase?

We note that assembly code increases the defect density by a factor of $(1 + 0.4 \times 0.2) = 1.08$, and reuse decreases the defect density by a factor of $(0.5 + 0.5 \times 0.25) = 0.625$. The integration test phase implies a factor of 2.5 relative to the beginning of the system test phase. From the model given by Equation (13.4), we estimate that the defect density at the beginning of the subsystem test phase can range between $12 \times 2.5 \times 1 \times 1 \times 1.08 \times 1 \times 1 \times 0.625 = 20.25$/KSLOC and $16 \times 2.5 \times 1 \times 1 \times 1.08 \times 1 \times 1 \times 0.625 = 27$/KSLOC. Thus, the total number of defects can range from 405 to 540.

13.6 Testing Approaches

Testing a program involves applying a number of inputs and observing the responses. When the response is different from the expected, the code has some (one or more) defects. During debugging, the objective of software testing is to find faults as quickly as possible and remove them to increase the reliability as fast as possible. However, during certification, the object is to estimate the reliability during actual deployment; thus, the fault encountering rate should be representative of the actual operation. The testing approaches can be divided into these two classes:

A. *Black-box (or functional) testing*: This assumes that nothing about the implementation of the software is known. The test generation is done by only considering the functional input/output description of the software. This is the often most common form of testing.

B. *White-box (or structural) testing*: In this case, the information about the actual implementation is used, for example, the existence of specific blocks or branches. The testing then can target specific structural elements of the code.

A combination of the two approaches will often yield the best results. Black-box testing only needs a functional description of the software. However, specific information about actual implementation will allow testers to better select elements of the structure that need to be exercised. When the inputs are selected randomly, the approach is termed *random testing*. When *partition testing* approach is used, the input space is divided into partitions defined using the ranges of the input variables/structures. The test cases are then chosen to ensure that each partition is exercised thoroughly. Random testing and partition testing are orthogonal approaches and can be combined.

Partitions can be exercised randomly for the usual cases and deterministically for boundary cases.

Many faults have high *testability*, that is, they are easily detected with only limited testing. Some faults that are triggered only under rarely occurring input combination will have very low testability. At the beginning of testing, many faults are found quickly. They are faults that have high testability and are easily detected and removed. Since the easily detectable faults have already been removed, the remaining faults are found in the later phases of fault removal. Finding the hard-to-test faults can take considerable effort. When a very low defect density product is needed, the testers need to use careful and systematic approaches.

A test coverage measure can measure using the thoroughness of testing, as discussed in Section 13.3. Branch coverage is a more thorough measure than statement coverage, since complete branch coverage guarantees complete statement coverage, but not vice versa. A branch coverage of say 85% may be used as a minimum criterion. A stricter measure (like p-use coverage) or a combination of measures (such as those provided by the test coverage tools) should be useful for very high-reliability programs.

Test inputs must be chosen and drawn in accordance with the *operational profile* to be able to estimate the reliability in the field. A *profile* is the set of disjoint operations that a software performs, with their probabilities of occurrence. The *operational profile* is based on the frequencies that occur in actual operation. When a software is used in very different user settings, the operational profile for each setting may be different. For obtaining an operational profile, the input space needs to be divided into sufficiently small partitions and then estimating the probabilities of an input being drawn from a partition. A partition with a high associated probability may need to be further divided into smaller sub-partitions to provide enough resolution.

Example 2

This example is based on an example by Musa [18]. A system responds differently to a call depending on the type of incoming transaction. Using prior experience, the following transaction types (TTs) and maintenance operation types (MOTs) are identified and their probabilities have been determined as follows:

A.	TTA	0.74
B.	TTB	0.15
C.	MOTC	0.10
D.	MOTD	0.009
E.	MOTE	0.0005
F.	MOTF	0.0005
G.	Failure recovery	0.000001
Total for all operation types		1.0

It is noted that TTA occurs 74% of the time, and thus the profile does not have enough resolution. It was found that the transaction type TTA can be further divided depending on the subtypes, given as follows:

A1	TTA1	0.18
A2	TTA2	0.17
A3	TTA3	0.17
A4	TTA4	0.12
A5	TTA5	0.10
Total for all subtypes of TTA		0.74

Thus, the final operational partitions are {A1, A2, A3, A4, A5, B, C, D, E, F, G}. These and their probabilities form the operational profile given by {(A1, 0.18), (A2, 0.17) … (G, 0.000001)}. When acceptance testing is performed, the test operations would be chosen such that a transaction type TTA1 occurs 18% of the time, TTA2 occurs 17% of the time, and so on. While failure recovery occurs only rarely, it is likely to be a critical operation and thus needs at least some test effort.

If the testing objective is to estimate the failure rate, as would be the case during acceptance testing, testing should be done according to the operation profile. Also, operational profile-based testing would be efficient if the testing time is very limited. However, if the project needs to achieve sufficiently high reliability, testing needs to more uniform across operation types. Often code that handles failures and exceptional cases will contain faults that will be hard to detect. For such defects, special tests need to be used so that the code containing them is entered and exercised [19].

13.7 Input Domain Software Reliability Models

An early input domain model was proposed by Nelson [20]. If there are I distinct possible inputs, and a failure occurs for I_e of them, then the reliability is given by

$$R = 1 - \frac{I_e}{I} \tag{13.5}$$

If the software input space can be partitioned into partitions $j = 1, 2, \ldots, k$, and the probability of an input being chosen from partition j is p_j as given by the operational profile, then Equation (13.5) can be refined as

$$R = 1 - \sum_{j=1}^{k} p_j \, \frac{I_{ej}}{Ij} \tag{13.6}$$

where *Ij* is the number of inputs applicable for partition *j* and I_{ej} is the number of inputs for which an error is encountered. This makes a simplifying assumption that failures for different partitions are statistically dependent. In actual practice, at least some code would be shared, and thus Equation (13.6) should be viewed as an approximation. It can be used to estimate the reliability when the operational profile is known along with the failure rate I_{ej}/I_j for a partition.

The Nelson's model assumes that the software is not changing. When the software is being tested and debugged, the reliability grows as bugs are gradually removed. The relationship between the debugging effort and the resulting reliability growth is discussed in the next section.

13.8 Software Reliability Growth Models

In a software development project, the cost needed to ensure a suitable reliability level can often be 25%–40%, although for very high reliability needs, it has been reported to be as much as 60%. A software needs to be released by a specific target date because of customer requirements and competitive marketing. The manager must ensure that the software has achieved the desired quality level before it can be released. Even after extensive testing, any further testing will always detect result in finding more bugs. The manager must determine the quality level at which the cost of a future encounter with residual bugs is less than the cost of additional testing. The manager thus must determine the amount of testing needed so that the minimum acceptable reliability level will be achieved. Careful planning in terms of time and effort needed requires the use of modeling the defect finding and removal process. A number of such models, termed SRGMs, have been proposed.

An SRGM assumes that reliability will grow with testing time *t*, which can be measured in terms of the CPU execution time used, or the number of man-hours or days. The time can also be measured in terms of the number of transactions encountered. Here we consider the exponential model, which can be thought to represent several models, including the models proposed by Jelinski and Muranda (1971), Shooman (1971), Goel and Okumoto (1979), and Musa (1975–1980). Here it is derived using basic considerations, even though many other models are obtained by noting the trend in the actual data collected.

The SRGMs are generally described using one of these two functions of testing time—*failure intensity* $\lambda(t)$ or *total expected faults* (often termed the *mean value function*) detected by time *t*, given by $\mu(t)$. The failure intensity is the rate of change in the number of a cumulative number of faults detected by time *t*:

$$\lambda(t) = \frac{d}{dt}\mu(t) \tag{13.7}$$

Let us denote the number of yet undetected defects at time t to be *N*(*t*). Here we assume that debugging is perfect, implying that a defect is always successfully removed when it is encountered.

The exponential model assumes that at any time, the rate of finding (and removing) bugs is proportional to the number of undetected defects still present in the software. That can be stated as follows:

$$-\frac{dN(t)}{dt} = \beta_1 N(t) \tag{13.8}$$

where β_1 is a constant of proportionality. The parameter β_1 can be shown to be given by

$$\beta_1 = \frac{K}{\left(S \cdot Q \cdot \frac{1}{r}\right)} \tag{13.9}$$

where *S* is the software size (number of source instructions), *Q* is the number of object instructions per source instruction as a result of compiling a high language code into machine language for the target instruction set architecture, and *r* is the instruction execution rate for the machine instructions of the computer being used. The term *K* called *fault exposure ratio* has been found to be in the range of 1×10^{-7} to 10×10^{-7} when *t* is measured in seconds of CPU execution time. It is notable that *K* does not depend on the software size but can depend on the testing strategy. As a first approximation, we regard *K* to be constant.

The differential equation (13.8) can be solved to yield

$$N(t) = N(0)e^{-\beta_1 t} \tag{13.10}$$

where *N*(0) is the initial number of faults before testing started. Hence the total number of expected faults detected by time *t* is given by

$$\begin{aligned} \mu(t) &= N(0) - N(t) \\ &= N(0)(1 - e^{-\beta_1 t}) \end{aligned} \tag{13.11}$$

In the software reliability engineering literature, it is generally written in the form:

$$\mu(t) = \beta_0(1 - e^{-\beta_1 t}) \tag{13.12}$$

and thus β_0 is the number of faults that would eventually be found, which is equal to *N*(0) when we assume perfect debugging, that is, no new defects are generated during debugging.

Using Equation (13.7), we can obtain an expression for failure intensity given in terms of the two parameters β_0 and β_1:

$$\lambda(t) = \beta_0\beta_1 e^{-\beta_1 t} \tag{13.13}$$

One significant advantage of the exponential model is that it has a simple explanation and both parameters β_0 and β_1 have a clear interpretation and thus can be estimated even prior to testing. The variations of the exponential model include the hyperexponential model, proposed by Ohba, Yamada, and Lapri, which assumes that different sections of the software are separately governed by an exponential model, with the parameter values specific for each section.

Many other SRGMs have been proposed and used [21]. Several studies have compared the models for both fit and predictive capability using the data from different projects. It has been found that the exponential compares well with other models, allthough, a couple of models have been found to outperform the exponential model. We will here look at one of them, the logarithmic model, proposed by Musa and Okumoto (termed logarithmic Poisson model by them), which has been found to have a better predictive capability than the exponential model.

The logarithmic model assumes that the fault exposure ratio K varies during testing [22]. The logarithmic model is also a finite-time model, implying that after a finite (although perhaps long) time, there will be no more faults to be found. The logarithmic model models a logarithmic dependence on time and can be given by

$$\mu(t) = \beta_0 \ln(1 + \beta_1 t) \tag{13.14}$$

or in terms of the failure intensity

$$\lambda(t) = \frac{\beta_0\beta_1}{1 + \beta_1 t} \tag{13.15}$$

The preceding equations apply as long as $\mu(t) < N(0)$. The condition is practically always satisfied since testing has to be terminated while a few bugs are still present, since finding additional bugs gets exponentially harder.

The variation in K, as assumed by the logarithmic model has been observed in actual practice. For higher defect densities, the value of K declines with testing, as defects get harder to find. However, K starts rising at low defect densities. This is explained by the fact that practical testing tends to be directed toward functionality not yet adequately exercised, rather than being random. This impacts the behavior at low defect densities.

For the logarithmic model, the two parameters β_0 and β_1 do not have a straightforward interpretation. An interpretation is provided by Malaiya and Denton [12] in terms of the variation in fault exposure ratio. An approach for estimating the logarithmic model parameters β_0^L, β_1^L has been proposed by relating them to the exponential model parameters.

The exponential model has been shown to underestimate the number of defects that will be detected in a future interval, that is, it tends to have a negative bias. The logarithmic model also exhibits a negative bias, although somewhat smaller. Only the Littlewood–Verral Bayesian model exhibits a positive bias among the major models. This model also has good predictive capabilities; however, it is not popular because it is relatively more complex, and the parameter values do not have a physical interpretation.

An SRGM is needed in two management situations. A manager can use it before testing begins to estimate the testing needed to achieve the desired reliability. That needs an estimation of the parameters using static attributes like the software size. When testing and debugging is being done, actual test data can be collected and used to estimate the parameter values and to make projections.

13.8.1 Planning prior to Testing

Project managers need to come up with a preliminary strategy for testing early so that testers can be hired. For the exponential model, the two values of the two parameters β_0 and β_1 can be designed using defect density model and Equation (13.9) respectively. They can then be used to estimate the testing effort that would be required to reach the target reliability attribute, which can be defect density, failure intensity, or MTTF.

Example 3

For a software development project, the defect density prior to testing has been estimated to be 25 defects/KLOC. The software size is 10,000 lines written in *C*. The code expansion ratio *Q* for *C* code for a target CPU is about 2.5, and thus the compiled object code size will be 10,000 × 2.5 = 25,000. The testing uses a CPU with MIPS rating of 70 (i.e., 70 million machine instructions per second). The fault exposure ratio *K* is expected to be 4×10^{-7}. Obtain the testing time required such that a defect density of 2.5 defects/KLOC will be achieved.

Using the exponential model, the value of the first parameter can be estimated as

$$\beta_0 = N(0) = 25 \times 10 = 250 \text{ defects,}$$

The second parameter can be estimated this way:

$$\beta_1 = \frac{K}{\left(S \cdot Q \cdot \frac{1}{r}\right)} = \frac{4.0 \times 10^{-7}}{10,000 \times 2.5 \times \frac{1}{70 \times 10^{6}}}$$

$$= 11.2 \times 10^{-4} \text{per second}$$

If t_1 is the testing time required to reduce the defect density to 2.5/KLOC, then Equation (13.10) can be used to obtain its value

$$\frac{N(t_1)}{N(0)} = \frac{2.5 \times 10}{25 \times 10} = \exp(-11.2 \times 10^{-4} \cdot t_1)$$

giving us

$$t_1 = \frac{-\ln(0.1)}{11.2 \times 10^{-4}} = 2056 \text{ seconds CPU time}$$

The failure intensity at the end of the testing period will be

$$\lambda(t_1) = 250 \times 11.2 \times 10^{-4} e^{-11.2 \times 10^{-4} t_1}$$
$$= 0.028 \text{ failures/second}$$

The example provides a simple illustration. It has been shown that the value of K depends on the testing strategy used. The above example uses CPU seconds as the unit of time and hence the effort. Often the effort is specified in terms of the number of person-hours. The person-hours can be converted into the equivalent CPU execution time by using by an appropriate factor, which would be obtained using past data. A convenient way to estimate β_1 would be to note that Equation (13.9) suggests that $\beta_1 \times I$ should be constant provided only the code size is different. For example, if for a previous project with 10 KLOC source code, the final value for β_1 was 2×10^{-3} per second. Then for a new 20 KLOC project, β_1 can be estimated as $2 \times 10^{-3}/2 = 1.00 \times 10^{-3}$ per second.

USING PARTIAL TEST DATA

During testing, the defect finding rate can be recorded as testing progresses. By using an SRGM, a manager can predict the additional testing time needed to achieve the desired reliability level, or determine the testing effort needed if there is a target release date. The major steps for analyzing the partial data are as follows.

1. *Collection and conditioning of data*: The failure intensity data points always include a lot of short-term variations. The data needs to be smoothed to extract and preserve the long-term trend. Data can be *grouped* by dividing the test duration into a number of intervals and then computing the average failure intensity in each interval.
2. *Model selection and fitting*: The best model that fits the partial data many not be the best choice since the testing effectiveness changes as testing progresses. The preferred way to select a model might be to rely on the past experience with other projects using the same process. The exponential has the advantage that the parameters are amenable to estimation using static data. The logarithmic models have been found to have good

predictive capability across various projects. Early test data is very noisy, and the parameter values may not stabilize until a significant amount of data is available. The parameter values can be determined using either least square or maximum likelihood approaches.

3. Using the parameter values, an analysis can be performed to decide how much more testing is needed. The fitted model can project the testing effort needed to achieve the target failure intensity or defect density. Recalibrating a model that does not conform to the data can significantly improve the accuracy of the projection. A model that describes the process well may not improve significantly by recalibration.

Example 4

The data in this example was collected by Musa [3] for an industrial project with 21,700 instructions. Testing was continued for about 25 hours and 136 defects were found. In Table 13.5, the time is specified in seconds when the next set of 5 defects were found. The failure intensity is computed for each set of 10 defects. This results in some smoothing of data.

Fitting all the data points to the exponential model, the parameter values are obtained as

$$\beta_o = 150.62 \text{ and } \beta_1 = 3.4 \times 10^{-5} \text{ per second}$$

The exact values will depend on the fitting approach used. Let the target failure intensity be one failure per hour, that is, 2.0×10^{-4} failures per second. An estimate of the stopping time t_f is then given by

$$2.0 \times 10^{-4} = 150.92 \times 3.4 \times 10^{-5} e^{-3.4 \times 10^{-4} t_f} \tag{13.16}$$

which gives the total testing time needed as t_f = 95,376 seconds., that is, 26.49 hours. This means an additional testing of about 2 hours. In Figure 13.1, the smooth curve shows the fitted model.

The variation of the parameter values of the exponential model has been studied. Results show that the value of β_0 estimated early during testing and debugging is generally lower than the final value. The final value of β_1 tends to be lower than the initial value. Thus, as testing progresses, the estimated value of β_0 rises and β_1 declines, and the product $\beta_0\beta_1$ remains stable. Thus, the early estimates are likely to be fewer but with higher testing efficiency. In Equation (13.16), the actual value of t_f is likely to be higher. Thus, the answer of 15.69 hours should be taken to represent a low estimate for the test time needed.

In many cases, it is desirable to compute interval estimates of the testing time and the final failure intensity. There exist statistical methods that can be used for calculating the interval estimates [19,20]. Sometimes the testing needs to continue until a failure rate is achieved with a specific confidence level, say 90%. Graphical and analytical approaches for obtaining such stopping points have been developed [21].

TABLE 13.5

Software Failure Data

Defects Found	Time (seconds)	Failure Intensity (per second)
5	342	
10	571	0.015974441
15	968	0.007067138
20	1,986	0.004694836
25	3,098	0.003264773
30	5,049	0.004492363
35	5,324	0.007513148
40	6,380	0.004310345
45	7,644	0.002696145
50	10,089	0.002995806
55	10,982	0.004048583
60	12,559	0.00399361
65	13,486	0.003079766
70	15,806	0.002517623
75	17,458	0.002666667
80	19,556	0.001499475
85	24,127	0.002025522
90	24,493	0.000789141
95	36,799	0.00062162
100	40,580	0.001819174
105	42,296	0.001164009
110	49,171	0.000945269
115	52,875	0.001371366
120	56,463	0.001022913
125	62,651	0.000685871
130	71,043	0.000498728
135	82,702	0.000340155
136	88,682	

The growth models assume that the testing strategy is uniform throughout testing. In actual software testing, the test approach is altered from time to time. Each approach is initially very efficient for detecting a specific class of faults, causing a spike in failure intensity as a result of the switch. This requires a smoothing approach to minimize the impact of these spikes during fitting. The models assume that the software under test is stable during testing; however, often it changes because of additions or changes in an evolving software. To minimize the error due to changes, early data points can be dropped from the calculation, using only the more recent data points. If reliability data for each component added or deleted can be separately assessed, the approaches discussed in the section on multicomponent systems may be used.

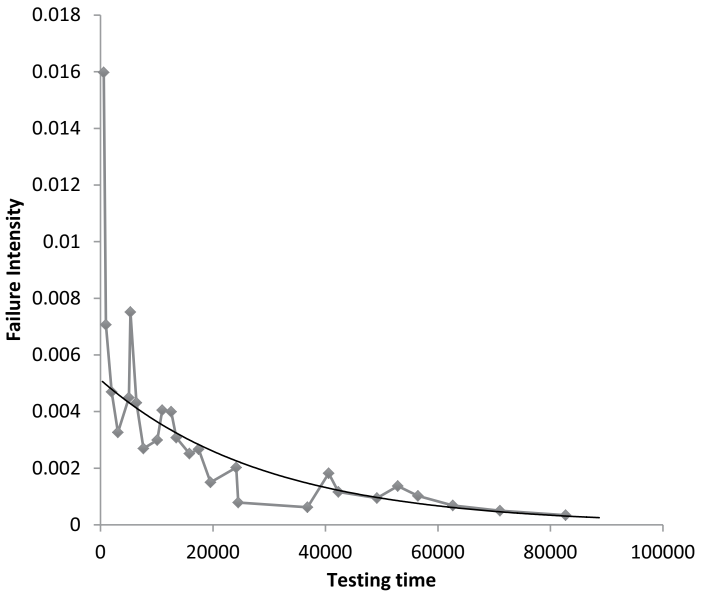

FIGURE 13.1
SRGM fitting.

USING TEST COVERAGE TO EVALUATE RELIABILITY GROWTH

Several software tools are available that can evaluate coverage of statements, branches, and other structural and data flow elements (such as P-uses). In large programs, it may not be feasible to achieve complete (100%) coverage, still, a higher test coverage implies the code has been exercised more thoroughly.

The software test coverage is directly related to the residual defect density and hence the reliability [8]. It can be shown that μ, the total number of defects found, is linearly related to the test coverage measures at higher test coverage. For example, if the branch coverage C_B is being evaluated, it will be found that for low values of C_B, μ remains close to zero. However, for higher values of C_B (after a bend in the curve termed a *knee*), μ starts rising linearly. The model is given by

$$\mu = -a + b \cdot C_B, \quad C_B > \text{knee} \tag{13.17}$$

In the preceding equation, the parameters a and b will be influenced by the software size and the defect density at the beginning. It is notable that the variations in test effectiveness will not influence the relationship since the test coverage directly evaluates the thoroughness of testing of a program. When the system is expected to have a high reliability, a stricter metric like branch coverage should be used rather than statement coverage.

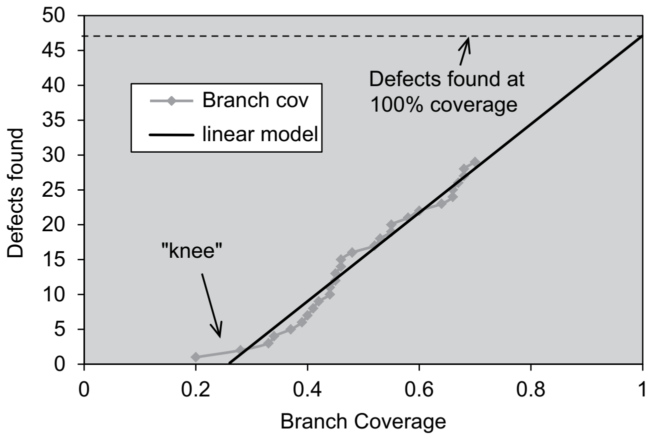

FIGURE 13.2
Coverage-based modeling.

Figure 13.2 gives the actual data from a European Space Agency project. When testing was stopped, 29 defects were found and 70% branch coverage was obtained. According to the model, if the testing had continued until 100% branch coverage was obtained, 47 total defects would have been found. Thus, about 18 residual defects were present when testing was stopped. According to the test data, only one defect was detected when branch coverage was about 25%; thus, the knee of the curve can be considered to be at a branch coverage of 0.25.

13.9 Security Vulnerabilities in Internet Software

Software systems such as operating systems, servers, and browsers can present major security challenges because systems are now generally connected using a network or the Internet and the number of security breaches is increasing. The software defects in such programs that permit an unauthorized action are termed *vulnerabilities*. Reported vulnerabilities are tracked by the CERT and other databases. Modern society, from individuals to commercial and national organizations depend on the Internet connectivity. That makes the exploitation of vulnerabilities very attractive to criminals or adversaries with the technical expertise. Vulnerabilities and their exploits can now be bought and sold in the international arena.

Each software system goes through several phases: the release, increasing use, and eventual stability followed by a decline in the market share when it is replaced by a new version or a new system. The three phases in the cumulative vulnerabilities plot for a specific version of the software can often be clearly observed. When a product is first released, there is a slow rise. As the product develops a market share, it starts attracting the attention of both white hat and black hat vulnerability finders. This creates a steady pace of vulnerability discovery reports. When an alternative program takes away the market share, the vulnerability finding rate declines in an older program. Figure 13.3 gives a plot of vulnerabilities reported for Windows 98 during January 1999–August 2002.

A model proposed by Alhazmi and Malaiya [22] for the cumulative number of vulnerabilities, y, against calendar time *t*, can be given by

$$y = \frac{B}{BCe^{-ABt} + 1} \tag{13.18}$$

In the equation, *A*, *B*, and *C* are the parameters to be determined from data. The parameter *B* represents the total number of vulnerabilities present that will be eventually found. The data for several common operating systems fit the model as determined by the goodness of fit values.

Just as the overall defect density is a major reliability metric, the *vulnerability density* in a large software system is an important measure of risk. The known vulnerability density can be computed using the complete data about the reported vulnerabilities. Known vulnerability density V_{KD} is the reported number of vulnerabilities in the software divided by the software size. Thus,

$$V_{KD} = \frac{V_K}{S} \tag{13.19}$$

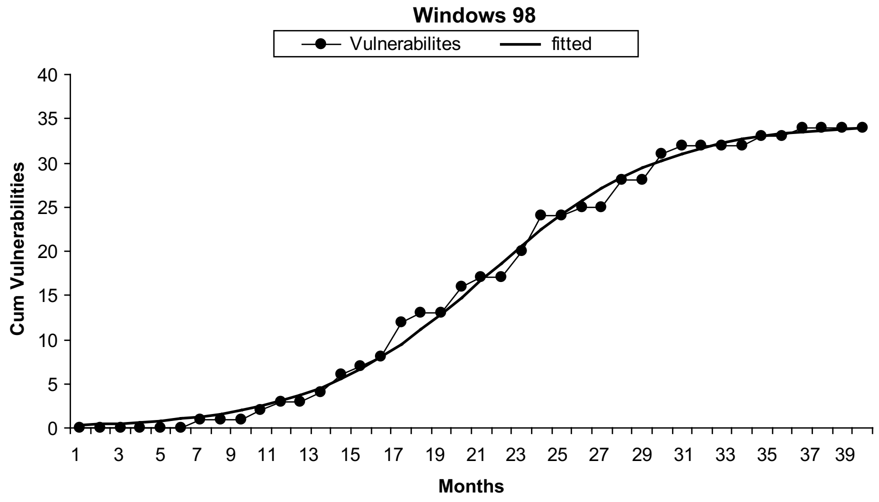

FIGURE 13.3
Vulnerability discovery in Windows 98.

TABLE 13.6
Vulnerability Densities and Ratios for Windows NT 4.0 and Windows 98

System	MSLOC	Known Defects (1,000s)	D_{KD} (per KLOC)	Known Vulnerabilities	V_{KD} (per KLOC)	Ratio (V_{KD}/D_{KD})
Windows NT 4.0	16	10	0.625	206	0.0129	2.06%
Windows 98	18	10	0.556	84	0.0047	0.84%

where V_K is the reported number of vulnerabilities and S is the size of software in the number of LOC. Table 13.6 gives the values based on data from several sources [22]. It gives the known defect density D_{KD}, V_{KD}, and the ratios of the two.

In Table 13.6, we see that the source code size is 16 and 18 MSLOC (million source lines of code) for Windows NT 4.0 and Windows 98, respectively, which is approximately the same. The reported defect densities at release, 0.625 and 0.556, are also similar for the two OSs. The known vulnerabilities in Table 13.6 are as of the period 1999–2008 (no vulnerabilities were reported after that). For Windows 98, the vulnerability density was 0.0047, whereas for Windows NT 4.0 it is 0.0129, nearly three times. The higher vulnerability density for Windows NT 4.0 is likely due to two factors. First, since it is a server OS, it contains more code that handles external accesses. Second, attacking servers would generally be much more rewarding, and thus it must have attracted more attention by the vulnerability finders causing detection of more vulnerabilities.

The last column in the table gives the ratios of known vulnerabilities to known defects. For the two systems, the ratios are 1.06% and 0.84%. A few researchers had presumed that of the total defects, vulnerabilities can be 1% or 5%. The values given in Table 13.6 show that 1%–2% may be closer to reality.

13.10 Evaluation of Multicomponent System Reliability

A large software system is always implemented using a number of modules [24], which may be separately developed by different teams. The individual modules are likely to have been developed and tested differently, some possibly reused from a previous version, causing a variation in defect densities and failure rates. Here we discuss methods for computing the system level failure rate and the reliability if we know the reliabilities attributes of the individual modules. In a system where only one thread executes at a time, one specific module is under execution at a time. Some modules are invoked more often, and some may remain under execution for a longer time. If f_i is the fraction of the time module *i* being executed, then the average system failure rate is

$$\lambda_{sys} = \sum_{i=1}^{n} f_i \lambda_i \tag{13.20}$$

where λ_i represents the failure rate of module *i*.

Let the execution time of a single transaction be *T*. Let us assume that module *i* is invoked e_i times the transaction time *T*, and each time execution takes for d_i time, then its execution frequency is

$$f_i = \frac{e_i \cdot d_i}{T} \tag{13.21}$$

The system reliability R_{sys} can be specified as the probability that no failures will occur during a single transaction. From the reliability theory, the system reliability is given by

$$R_{sys} = \exp(-\lambda_{sys} \cdot T)$$

Using the above mentioned equations, we can write the expression as

$$R_{sys} = \exp\left(-\sum_{i=1}^{n} e_i d_i \lambda_i\right)$$

The single execution reliability of module *i* is R_i is $\exp(-d_i\lambda_i)$. Thus, we have

$$R_{sys} = \prod_{i=1}^{n} (R_i)^{e_i} \tag{13.22}$$

Multiple-version programming: In the critical domains, like defense or avionics, sometimes redundant versions of the program are employed. To reduce the probability of multiple numbers of them failing at the same time, each version is developed independently by a separate team. The implementation can use triplication and voting on the result to achieve fault tolerance. The system is operating correctly as long as two of the versions are working correctly. The scheme assumes the voting mechanism to be defect free. If the failures in the three versions are independent as ideally expected, the reliability can be improved by a large factor. However, it has been shown that that there is a significant probability of correlated failures that must be considered.

Here is a simple analysis. In a three-version system, let the probability of all three versions failing for the same input be q_3. Also, let the probability that any two versions will fail at the same time be q_2. As three distinct pairs are possible among the three versions (three choose two), the probability P_{sys} of the system failure for a transaction is

$$P_{sys} = q_3 + 3q_2 \tag{13.23}$$

If the failures are statistically independent, and if the probability of one version failing is p, Equation (13.23) can be written as

$$P_{sys} = p^3 + 3(1-p)p^2 \tag{13.24}$$

Experiments have demonstrated that in reality, the failures have some statistical correlation. The values of q_3 and q_2 then need to be experimentally determined for system reliability evaluation.

Example 5

The potential improvement in system reliability in an n-version system can be illustrated using the experimental data collected by Knight and Leveson, as has been pointed out by Hatton [25]. For the experimental data, the probability of a version failing for one transaction was 0.0004. If there is no correlation, then a three-version system would have an overall failure probability given by P_{sys}:

$$\begin{aligned} P_{sys} &= (0.0004)^2 + 3(1-0.0004)(0.0004)^2 \\ &= 4.8\times10^{-7} \end{aligned}$$

This represents an improvement by a factor of $0.0004/4.8\times10^{-7} = 833.3$. However, a significant correlation was observed in the experiments, causing the probability of all three modules failing at the same time to be $q_3 = 2.5\times10^{-7}$ and the probability of any two modules failing to be $q_2 = 2.5\times10^{-6}$. Using these values, we get

$$P_{sys} = 2.5\times10^{-7} + 3\times2.5\times10^{-6} = 7.75\times10^{-6}$$

Thus, the realistic improvement factor achieved is $0.0004/7.75\times10^{-6} = 51.6$.

Hatton argues that the best software testing approaches have been found to reduce defect density only by a factor of 10. Thus, an improvement factor 52 is not achievable by additional testing; it can only be obtained by using a redundant n-version software implementation.

13.11 Optimal Reliability Allocation

A large program is always developed as a set of separate units that are eventually integrated to form the complete system. The separate modules will have different sizes and defect densities, especially if some of them are reused. That gives rise to the problem of allocating the test effort to different modules in a way that minimizes the total testing cost, while still obtaining the target reliability level. This is an application of the reliability allocation problem that arises in mechanical and electrical systems also. The problem is easier for a software system because the reliability can be continuously improved by using additional testing, and the impact of testing can be modeled by the SRGMs.

Assuming that the exponential SRGM is applicable, the failure rate can be specified as a function of testing time d_i for a module i as discussed. The failure rate can be rewritten as

$$\lambda_i(d) = \lambda_{0i} \exp(-\beta_{1i} d_i)$$

where the SRGM parameters applicable to the module i are $\lambda_{0i} = \beta_{0i}.\beta_{1i}$ and β_{1i}. Thus, the test cost, measured in terms of the testing time needed, d_i, can be expressed as a function of the module's failure intensity:

$$d(\lambda_i) = \frac{1}{\beta_i} \ln\left(\frac{\lambda_{0i}}{\lambda_i}\right)$$

To obtain the overall system failure rate, we assume that the block i is under execution for a fraction f_i of the time ($\Sigma f_i = 1$). The reliability allocation problem can be formally stated as an optimization problem

$$\text{Minimize the total cost } C = \sum_{i=1}^{n} \frac{1}{\beta_i} \ln\left(\frac{\lambda_{0i}}{\lambda_i}\right) \tag{13.25}$$

$$\text{subject to } \lambda_{ST} \le \sum_{i=1}^{n} f_i \lambda_i \tag{13.26}$$

where the desired overall failure intensity is given by λ_{ST}. The optimization problem given by Equations (13.25) and (13.26) can be solved by using the Lagrange multiplier approach [26] to obtain a closed form solution. The optimal failure rates and the testing times needed are obtained as follows:

$$\lambda_1 = \frac{\frac{\lambda_{ST}}{f_1}}{\sum_{i=1}^{n} \frac{\beta_{11}}{\beta_{1i}}} \quad \lambda_2 = \frac{\beta_{11} f_1}{\beta_{12} f_2} \lambda_1 \quad \cdots \quad \lambda_n = \frac{\beta_{11} f_1}{\beta_{1n} f_n} \lambda_1 \tag{13.27}$$

The analysis results in the optimal values of d_1 and d_i, $i \neq 1$ as given by

$$d_1 = \frac{1}{\beta_{11}} \ln \left(\frac{\lambda_{10} f_1 \sum_{i=1}^{n} \frac{\beta_{11}}{\beta_{1i}}}{\lambda_{ST}} \right) \quad and \quad d_i = \frac{1}{\beta_{1i}} \ln \left(\frac{\lambda_{i0} \beta_{1i} f_i}{\lambda_1 \beta_{1i} f_1} \right) \tag{13.28}$$

Note that the testing time for a block must be nonnegative. The value of d_i is positive if $\lambda_i \leq \lambda_{i0}$.

The parameter β_{1i} of the exponential model as given by Equation (13.9) is inversely proportional to the software size as discussed earlier. As a first assumption, the value of f_i can be assumed to be proportional to the module size. Also, we note that the values of λ_i and λ_{i0} depend on the initial defect densities, but not on size. Thus, Equation (13.27) implies that the optimal values of the failure rates $\lambda_1, \dots \lambda_n$, after testing and debugging, will be equal. If the initial defect densities are same for all the blocks, then the optimal test times for each module would be proportional to its size. In many applications, some of the blocks are more critical. That can be modeled by assigning additional weights resulting in longer testing times for them.

Example 6

A software system uses five functional modules M1 to M5. To illustrate optimization, we construct this example assuming sizes 1, 1, 3, 00, and 20 KLOC, respectively, for the five modules, with the initial defect densities of 5, 10, 10, 15, and 20 defects per KLOC. The assumptions result in the parameter values are given in the top three rows, which are the inputs to the optimization problem. The optimal solution obtained using the aforementioned equations are given in the two bottom rows. The same result would be obtained if a numerical optimization is done (e.g., using Excel Solver). Let us now compute the optimal distribution of the additional testing time required for each module to achieve the overall system failure rate equal to 0.15 while minimizing the total test cost. Here the time units can be hours of testing time or hours of CPU time used for testing.

Module	**M_1**	**M_2**	**M_3**	**M_4**	**M_5**
β_i	5×10^{-3}	5×10^{-3}	1.67×10^{-3}	2.5×10^{-4}	2.5×10^{-4}
λ_{i0}	0.2	0.4	0.4	0.4	0.8
x_i	0.022	0.022	0.067	0.444	0.444
Optimal λ_i	0.15	0.15	0.15	0.15	0.15
Optimal d_i	57.49	196.1	588.5	3923	6696

The results show that the final values of λ_i for all the modules are equal, even though they begin with different initial values. This requires a larger fraction of the test effort allocated to blocks that are larger (compare d_2 and d_3) or with higher defect densities (compare d_1 and d_2). The cost in terms of total additional testing effort needed is 11,461.41 hours.

13.12 Tools for Software Testing and Reliability

Software reliability is an engineering discipline. Thus, it requires collection, analysis of data, and the evaluation of proposed designs. Many tools are now becoming available that can automate several of the tasks such as testing, evaluation of testing, assessing potential reliability growth and failure rates. Some of the representative types of tools are mentioned here, with an example to illustrate. Some of the tools are in public domain. Installing and learning to use a tool can require a significant amount of time; thus, a tool should be selected after a careful comparison of the applicable tools available. Tables are available in the literature that compare the features or performance of different tools [27,28].

- Automatic test generations: JCONTRACT
- Graphical User Interface (GUI) testing: AscentialTest
- Memory testing: Valgrind
- Defect tracking: Bugzilla
- Test coverage evaluation: JCover
- Reliability growth modeling: Statistical Modeling and Estimation of Reliability Functions for Systems (SMERFS)
- Defect density estimation: ROBUST
- Coverage-based reliability modeling: ROBUST
- Markov reliability evaluation: Symbolic Hierarchical Automated Reliability and Performance Evaluator (SHARPE)
- Fault tree analysis: Eclipse Modeling Framework based Fault-Tree Analysis (EMFTA)

13.13 Conclusions

This chapter presents an overview of the basic concepts and approaches for the quantitative software reliability field. It considers the software life cycle and discusses the factors that impact the software defect density. Use of an SRGM is presented and illustrated using actual data. It also discusses a model for the discovery of security vulnerabilities in Internet-related software. Reliability evaluation of a system using the failure rates of the components is presented, and an approach is given for optimal allocation of reliability. The article also mentions the type of tools that can be used to assist in achieving and evaluating reliability.

References

1. W. Platz, *Software Fail Watch*, 5th ed., Tricentis, Vienna, 2018.
2. B. St. Clair, Growing complexity drives need for emerging DO-178C standard, *COTS The Journal of Military Electronics and Computing*, 2009.
3. J. D. Musa, *Software Reliability Engineering*, McGraw-Hill, New York, 1999.
4. Y. K. Malaiya and P. Srimani (Ed.), *Software Reliability Models*, IEEE Computer Society Press, Los Alamitos, CA, USA, 1990.
5. D. Carleton, R. E. Park and W. A. Florac, Practical software measurement, Tech. Report, SRI, CMU/SEI-97-HB–003.
6. P. Piwowarski, M. Ohba and J. Caruso, Coverage measurement experience during function test, *Proceedings, International Conference on Software Engineering*, 1993, pp. 287–301.
7. Y. K. Malaiya, Software reliability management, In: S. Lee, (Ed.), *Encyclopedia of Library and Information Sciences*, 3rd ed., Vol. 1: Taylor & Francis Group, Milton Park, 2010, pp. 4901–4912
8. Y. K. Malaiya, N. Li, J. Bieman, R. Karcich and B. Skibbe, The relation between test coverage and reliability, *Proceedings, IEEE-CS International Symposium on Software Reliability Engineering*, Nov. 1994, pp. 186–195.
9. Y. K. Malaiya, A. von Mayrhauser and P. Srimani, An examination of fault exposure ratio, *IEEE Transactions on Software Engineering*, 19(11), pp. 1087–1094, 1993.
10. E. N. Adams, Optimizing preventive service of software products, *IBM Journal of Research and Development*, 28(1), pp. 2–14, 1984.
11. H. Hecht and P. Crane, Rare conditions and their effect on software failures, *Proceedings, IEEE Reliability and Maintainability Symposium*, Los Angeles, CA, Jan. 1994.
12. Y. K. Malaiya and J. Denton, What do the software reliability growth model parameters represent, *Proceedings, IEEE-CS International Symposium on Software Reliability Engineering ISSRE*, Nov. 1997, pp. 124–135.
13. M. Takahashi and Y. Kamayachi, An empirical study of a model for program error prediction, *Proceedings, International Conference on Software Engineering*, Aug. 1995, pp. 330–336.
14. Y. K. Malaiya and J. Denton, Module size distribution and defect density, *Proceedings, International Symposium on Software Reliability Engineering*, Oct. 2000, pp. 62–71.
15. N. Nagappan and T. Ball, Use of relative code churn measures to predict system defect density. *Proceedings, 27th International conference on Software engineering (ICSE '05)*, ACM, 2005, pp. 284–292.
16. Y. K. Malaiya and J. Denton, Requirements volatility and defect density, *Proceedings, 10th International Symposium on Software Reliability Engineering (Cat. No.PR00443)*, Boca Raton, FL, 1999, pp. 285–294.
17. T. J. Ostrand and E. J. Weyuker, The distribution of faults in a large industrial software system. *Proceedings, ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '02)*, 2002, pp. 55–64.
18. J. Musa, More reliable, faster, cheaper testing through software reliability engineering, Tutorial Notes, ISSRE '97, 1997, pp. 1–88.

19. H. Yin, Z. Lebne-Dengel and Y. K. Malaiya, Automatic test generation using checkpoint encoding and antirandom testing, *International Symposium on Software Reliability Engineering*, 1997, pp. 84–95.
20. T. A. Thayer, M. Lipow and E. C. Nelson, *Software Reliability*, North-Holland Publishing, TRW Series of Software Technology, Amsterdam, 1978.
21. M. R. Lyu (Ed.), *Handbook of Software Reliability Engineering*, McGraw-Hill, Hightstown, NJ, USA, 1996, pp. 71–117.
22. N. Li and Y. K. Malaiya, Fault exposure ratio: Estimation and applications, *Proceedings, IEEE-CS International Symposium on Software Reliability Engineering*, Nov. 1993, pp. 372–381.
24. P.B. Lakey and A. M. Neufelder, System and software reliability assurance notebook, Rome Lab, FSC-RELI, 1997.
25. L. Hatton, N-version design versus one good design, *IEEE Software*, 14(6), pp. 71–76, 1997.
26. Y. K. Malaiya, *"Reliability Allocation" Encyclopedia of Statistics in Quality and Reliability*, John Wiley & Sons, Hoboken, NJ, 2008.
27. Comparison of GUI testing tools, (updated time to time) available at https://en.wikipedia.org/wiki/Comparison_of_GUI_testing_tools.
28. S. Wang and J. Offutt, Comparison of unit-level automated test generation tools, in *Fifth Workshop on Mutation Analysis*, 2009, pp. 2–10.