Software Reliability Management

Yashwant K. Malaiya
Computer Science Department, Colorado State University, Fort Collins, Colorado, U.S.A.

Abstract
This entry identifies the factors that control software reliability and the approaches that are needed to achieve desired reliability targets. Common reliability measures are defined. The factors that impact defect density and defect finding rates are discussed and software reliability growth modeling is introduced. Both test-time- and test-coverage-based models are introduced. Modeling for security vulnerability discovery process is also presented. Reliability of multicomponent software systems is discussed followed by optimal allocation of test resources to achieve a target reliability level. Finally some of the applicable software tools available today are mentioned.

INTRODUCTION

Software problems are main causes of computer system failures today. There have been numerous well-known cases of software failures in the past few decades. While very high reliability is naturally expected for critical systems, software packages used everyday also need to be highly reliable because the enormous investment of the software developer as well as the users is at stake. Studies have shown software defects in the United States alone cost more than $100 billion in lost productivity and repair costs.[1]

Virtually all nontrivial software developed will have defects.[2] All programs must be tested and debugged, until sufficiently high reliability is achieved. It is not possible to assure that all the defects in a software systems have been found and removed, however, the number of remaining bugs must be very small. Since software must be released within a reasonable time, to avoid loss of revenue and market share, the developer must take a calculated risk and must have a strategy for achieving the required reliability by the target release date.

For software systems, quantitative methods for achieving and measuring reliability are coming in use due to emergence of well-understood and validated approaches. Enough industrial and experimental data is now available which has allowed researchers to develop and validate methods for achieving high reliability. The minimum acceptable standards for software reliability have gradually risen in recent years.

This entry presents an overview of the essential concepts and techniques in the software reliability field. We examine factors that impact reliability during development as well as testing. First, we discuss the reliability approaches taken during different phases of software development. Commonly used software reliability measures are defined next. We discuss what factors control software defect density. Key ideas in test methodologies are presented. Use of a software reliability growth model (SRGM) is discussed and illustrated using industrial data. Use of such models allows one to estimate the testing effort needed to reach a reliability goal. We also see how reliability of a system can be evaluated if we know the failure rates of the components. Finally, the entry presents the type of tools that are available to assist in achieving and evaluating reliability.

Here we will use the terms *failure* and *defect* as defined below.[3]

Failure: a departure of the system behavior from user requirements during execution.

Defect (also termed a fault or bug): an error in system implementation that can cause a failure during execution.

A defect will cause a failure only when the erroneous code is executed, and the effect is propagated to the output. The *detectability* of a defect[4] is defined as the probability of detecting it with a randomly chosen input. Defects with very low detectability can be very difficult to detect, but may have significant impact on highly reliable systems.

Formally, the reliability of a system is defined as the probability that the operation will be correct during the period of interest. The software reliability improves during testing as bugs are found and removed. Once the software is released, its reliability is fixed, as long as the operating environment remains the same, and no modifying patches are applied. The software will fail time to time during operational use when it cannot respond correctly to an input. During operational use, bug fixes are often released that update the software. For a software system, its own past behavior is often a good indicator of its reliability, even though data from other similar software systems can be used for making projections.[5] The availability of a computational system is given by the probability that it will be operational at a given time. The availability is a function of both the system failure rate as well as the time needed for repairs.

Encyclopedia of Library and Information Sciences, Third Edition DOI: 10.1081/E-ELIS3-120043967

RELIABILITY APPROACHES DURING THE SOFTWARE LIFE CYCLE PHASES

Generally, the software life cycle is divided into the following phases. In the simple *waterfall model*, the software development process goes through these phases sequentially. However, it is not uncommon for developers to go to a previous phase, due to requirement changes or a need to make changes in the design. The *V-model* is an extension of the waterfall model that models testing explicitly. It is preferable to catch defects in an earlier phase, since it would be much more expensive to fix them later.

A. **Requirements and definition**: In this phase, the developing team interacts with the customer organization to specify the software system to be built. Ideally, the requirements should define the system completely and unambiguously. In actual practice, there is often a need to do corrective revisions during software development. A review or inspection during this phase is generally done by the design team to identify conflicting or missing requirements. A significant number of errors can be detected by this process. A change in the requirements in the later phases can cause increased defect density.

B. **Design**: In this phase, the system is specified as an interconnection of units, which are well defined and can be developed and tested independently. The design is reviewed to recognize errors.

C. **Coding**: In this phase, the actual program for each unit is written, generally in a higher-level language such as Java or C++. Occasionally, assembly level implementation may be required for high performance or for implementing input/output operations. The code is inspected by analyzing the code (or specification) in a team meeting to identify errors.

D. **Testing**: This phase is a critical part of the quest for high reliability and can take up to 60% of the entire development budget.[6] It is often divided into these separate phases.
   1. Unit test: In this phase of testing, each unit is separately tested, and changes are done to remove the defects found. Since each unit is relatively small and can be tested independently, they can be exercised much more thoroughly than a large program.
   2. Integration testing: During integration, the units are gradually assembled and partially assembled subsystems are tested. Testing subsystems allows the interface among modules to be tested. By incrementally adding units to a subsystem, the unit responsible for a failure can be identified more easily.
   3. System testing: The system as a whole is exercised during system testing. Debugging is continued until some exit criterion is satisfied. The objective of this phase is to find defects as fast as possible. In general, the input mix may not represent what would be encountered during actual operation.
   4. Acceptance testing: The purpose of this test phase is to assess the system reliability and performance in the operational environment. This requires collecting (or estimating) information about how the actual users would use the system. This is also called alpha-testing. This is often followed by beta-testing, which involves use of the beta-version by the actual users.

E. **Operational use and maintenance**: Once the software developer has determined that an appropriate reliability criterion is satisfied, the software is released. Any bugs reported by the users are recorded but are not fixed until the next patch or bug-fix. In case a defect discovered represents a security vulnerability, a patch for it needs to be released as soon as possible. The time taken to develop a patch after a vulnerability discovery, and the delayed application of an available patch contribute to the security risks. When significant additions or modifications are made to an existing version, regression testing is done on the new or "build" version to ensure that it still works and has not "regressed" to lower reliability. Support for an older version of a software product needs to be offered until newer versions have made a prior version relatively obsolete.

It should be noted that the exact definition of a test phase and its exit criterion may vary from organization to organization. When a project goes through incremental refinements (as in the extreme programming approach), there may be many cycles of requirements-design-code-test phases.

Table 1 shows the fraction of total defects that may introduced and found during a phase,[7–9] based on typical introduction and removal rates. In an actual software development process, the numbers may vary. For example in higher Capability Maturity Model (CMM) maturity organizations, the defects tend to be detected earlier. Most defects are inserted during design and coding phases. The fraction of defects found during the system test is small, but that may be misleading. The system test phase can take a long time because the defects remaining are much harder to find.

Table 1 Defects introduced and found during different phases

	Defects (%)		
Phase	**Introduced**	**Found**	**Remaining**
Requirements analysis	5	1	4
Design	39	19	24
Coding	46	33	37
Unit test	5	24	18
Integration test	3	15	6
System test	2	5	3

SOFTWARE RELIABILITY MEASURES

The classical reliability theory generally deals with hardware. In hardware systems, the reliability decays because of the possibility of permanent failures. However, this is not applicable for software. During testing, the software reliability grows due to debugging and becomes constant once defect removal is stopped. The following are the most common reliability measures used.

Transaction reliability: Sometimes a *single-transaction* reliability measure, as defined below, is convenient to use.

$$R = \Pr\{\text{a single transaction will not encounter a failure}\} \quad (1)$$

Both measures above assume *normal operation*, i.e., the input mix encountered obeys the operational profile (defined below).

Mean-time-to-failure (MTTF): The expected duration between two successive failures. In classical reliability theory, when an object fails, it is not operational until it is repaired. However, a software can continue to be in use even after a bug has been found in it.

Failure intensity (λ): The expected number of failures per unit time. Note that MTTF is inverse of failure intensity. Thus if MTTF is 500 hr, the failure intensity is $1/500 = 0.002$ per hr. For stable software in a stable environment, the failure intensity is stable and is given by

$$\lambda = 1/\text{MTTF} \quad (2)$$

Since testing attempts to achieve a high defect-finding rate, failure intensity during testing λ_t is significantly higher than λ_{op}, failure intensity during operation. Test-acceleration factor A is given by the ratio λ_t/λ_{op}. Thus if testing is 12 times more effective in discovering defects than normal use the test-acceleration factor is 12. This factor is controlled by the test selection strategy and the type of application.

Defect density: Usually measured in terms of the number of defects per 1000 source lines of code (KSLOC). It cannot be measured directly, but can be estimated using the growth and static models presented below. The failure intensity is approximately proportional to the defect density. The acceptable defect density for critical or high volume software can be less than 0.1 defects/KLOC, whereas for other applications 0.5 defects/KLOC is often currently 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 measures: Tools are now available that can automatically evaluate how thoroughly a software has been exercised by a given test suite. The two most common coverage measures are the following:

1. Statement coverage: The fraction of all statements actually exercised during testing.
2. Branch coverage: The fraction of all branches that were executed by the tests.

As discussed below, test coverage is correlated with the number of defects that will be triggered during testing.[10] A 100% statement coverage can often be quite easy to achieve. Sometimes a predetermined branch coverage, say 85% or 90%, may be used as an acceptance criterion for testing, higher levels of branch coverage would require significantly more testing effort.

WHAT FACTORS CAUSE DEFECTS IN THE SOFTWARE?

There has been considerable research to identify the major factors that correlate with the number of defects. Enough data is now available to allow us to use a simple model for estimating the defect density. This model can be used in two different ways. First, it can be used by an organization to see how they can improve the reliability of their products. Secondly, by estimating the defect density, one can use a reliability growth model to estimate the testing effort needed. A few empirical models have been proposed.[11] The model by Malaiya and Denton,[12] based on the data reported in the literature, is given by

$$D = CF_{ph}F_{pt}F_mF_s \quad (3)$$

where the five factors are the phase factor F_{ph}, modeling dependence on *software test phase*; the *programming team factor* F_{pt} taking in to account the capabilities and experience of programmers in the team; the *maturity factor* F_m depending on the maturity of the software development process; the *structure factor* F_s, depending on the structure of the software under development. The constant of proportionality C represents the defect density per KSLOC. The default value of each factor is one. They proposed the following preliminary sub-models for each factor.

Phase factor F_{ph}: Table 2 presents a simple model using actual data reported by Musa et al. and the error profile presented by Piwowarski et al. It takes the default value of one to represent the beginning of the system test phase.

Table 2 Phase factor F_{ph}

At beginning of phase	Multiplier
Unit testing	4
Subsystem testing	2.5
System testing	1 (default)
Operation	0.35

The programming team factor F_{pt}: The defect density varies significantly due to the coding and debugging capabilities of the individuals involved. A quantitative characterization in terms of programmers' average experience in years is given by Takahashi and Kamayachi.[13] Their model can take into account programming experience of up to 7 years, each year reducing the number of defects by about 14%.

Based on other available data, we suggest the model in Table 3. The skill level may depend on factors other than just the experience. Programmers with the same experience can have significantly different defect densities that can also be taken into account here.

The process maturity factor F_m: This factor takes into account the rigor of software development process at a specific organization. The SEI Capability Maturity Model level can be used to quantify it. Here, we assume level II as the default level, since a level I organization is not likely to be using software reliability engineering. Table 4 gives a model based on the numbers suggested by Jones and Keene as well as reported in a Motorola study.

The software structure factor F_s: This factor takes into account the dependence of defect density on language type (the fractions of code in assembly and high-level languages) and program complexity. It can be reasonably assumed that assembly language code is harder to write and thus will have a higher defect density. The influence of program complexity has been extensively debated in the literature. Many complexity measures are strongly correlated to software size. Since we are constructing a model for defect density, software size has already been taken into account. A simple model for F_s depending on language use is given below.

$$F_s = 1 + 0.4a$$

where a is the fraction of the code in assembly language. Here, we are assuming that assembly code has 40% more defects.

Table 3 The programming team factor F_{pt}

Team's average skill level	Multiplier
High	0.4
Average	1 (default)
Low	2.5

Table 4 The process maturity factor F_m

SEI CMM level	Multiplier
Level 1	1.5
Level 2	1 (default)
Level 3	0.4
Level 4	0.1
Level 5	0.05

Distribution of module sizes for a project may have some impact on the defect density. Very large modules can be hard to comprehend. Very small modules can have a higher fraction of defects associated with the interaction of the modules. Since module sizes tend to be unevenly distributed, there may be an overall effect of module size distribution.[14] Further research is needed to develop a model for this factor. Requirement volatility is another factor that needs to be considered. If requirement changes occur later during the software development process, they will have more impact on defect density. We can allow other factors to be taken into account by calibrating the overall model.

Calibrating and using the defect density model: The model given in Eq. 5 provides an initial estimate. It should be calibrated using past data from the same organization. Calibration requires application of the factors using available data in the organization and determining the appropriate values of the factor parameters. Since we are using the beginning of the subsystem test phase as the default, Musa et al.'s data suggest that the constant of proportionality C can range from about 6 to 20 defects per KSLOC. For best accuracy, the past data used for calibration should come from projects as similar to the one for which the projection needs to be made. Some of indeterminacy inherent in such models can be taken into account by using a high and a low estimate and using both of them to make projections.

Example 1: For an organization, the value of C has been found to be between 12 and 16. A project is being developed by an average team and the SEI maturity level is II. About 20% of the code is in assembly language. Other factors are assumed to be *average*. The software size is estimated to be 20,000 lines of code. We want to estimate the total number of defects at the beginning of the integration test phase.

From the model given by Eq. 3, we estimate that the defect density at the beginning of the subsystem test phase can range between $12 \times 2.5 \times 1 \times 1 \times (1 + 0.4 \times 0.2) \times 1 = 32.4$/KSLOC and $16 \times 2.5 \times 1 \times 1 \times (1 + 0.4 \times 0.2 \times 1 = 43.2$/KSLOC. Thus the total number of defects can range from 628 to 864.

SOFTWARE TEST METHODOLOGY

To test a program, a number of inputs are applied and the program response is observed. If the response is different from expected, the program has at least one defect. Testing can have one of two separate objectives. During debugging, the aim is to increase the reliability as fast as possible, by finding faults as quickly as possible. On the other hand during certification, the object is to assess the reliability, thus the fault-finding rate should be representative of actual operation. The test generation approaches can be divided into the classes.

A. Black-box (or functional) testing: When test generation is done by only considering the input/output description of the software, nothing about the implementation of the software is assumed to be known. This is the most common form of testing.
B. White-box (or structural) testing: When the actual implementation is used to generate the tests.

In actual practice, a combination of the two approaches will often yield the best results. Black-box testing only requires a functional description of the program; however, some information about actual implementation will allow testers to better select the points to probe in the input space. In *random-testing* approach, the inputs are selected randomly. In *partition testing* approach, the input space is divided into suitably defined partitions. The inputs are then chosen such that each partition is reasonably and thoroughly exercised. It is possible to combine the two approaches; partitions can be probed both deterministically for boundary-cases and randomly for non-special cases.

Testing, specially during integration, should include passing interface data that represents normal cases (n), special cases (s), and illegal cases (i). For two interacting modules A and B, all combinations of {An, As, Ai} and {Bn, Bs, Bi} should be tested. Thus if a value represents a normal case for A and a special case for B, the corresponding combination is (An, Bs). In some cases, specially for distributed and Web-based applications, components of the applications may be developed independently, perhaps using two different languages. Such components are also often updated independently. Interaction of such components needs to be tested to assure interoperability.

Some faults are easily detected, i.e., have high *detectability*. Some faults have very low testability; they are triggered only under rarely occurring input combination. At the beginning of testing, a large fraction of faults have high testability. However, they are easily detected and removed. In the later phases of testing, the faults remaining have low testability. Finding these faults can be challenging. The testers need to use careful and systematic approaches to achieve a very low defect density.

Thoroughness of testing can be measured using a test coverage measure, as discussed before in section "Software Reliability Measures". Branch coverage is a more strict measure than statement coverage. Some organizations use branch coverage (say 85%) as a minimum criterion. For very high reliability programs, a more strict measure (like p-use coverage) or a combination of measures (like those provided by the GCT coverage tool) should be used.

To be able to estimate operational reliability, testing must be done in accordance with the *operational profile*. A *profile* is the set of disjoint actions, operations that a program may perform, and their probabilities of occurrence. The probabilities that occur in actual operation specify the operational profile. Sometimes when a program can be used in very different environments, the operational profile for each environment may be different. Obtaining an operational profile requires dividing the input space into sufficiently small leaf partitions, and then estimating the probabilities associated with each leaf partition. A subspace with high probability may need to be further divided into smaller subspaces.

Example 2: This example is based on the Fone–Follower system example by Musa.[15] A Fone–Follower system responds differently to a call depending on the type of call. Based on past experience, the following types are identified and their probabilities have been estimated as given below.

A.	Voice call	0.74
B.	FAX call	0.15
C.	New number entry	0.10
D.	Data base audit	0.009
E.	Add subscriber	0.0005
F.	Delete subscriber	0.0005
G.	Hardware failure recovery	0.000001
Total for all events:		1.0

Here we note that a voice call is processed differently in different circumstances. We may subdivide event A above into the following.

A1.	Voice call, no pager, answer	0.18
A2.	Voice call, no pager, no answer	0.17
A3.	Voice call, pager, voice answer	0.17
A4.	Voice call, pager, answer on page	0.12
A5.	Voice call, pager, no answer on page	0.10
Total for voice call (event A)		0.74

Thus, the leaf partitions are {A1, A2, A3, A4, A5, B, C, D, E, F, G}. These and their probabilities form the operational profile. During acceptance testing, the tests would be chosen such that a FAX call occurs 15% of the time, a {voice call, no pager, answer} occurs 18% of the time and so on.

Testing should be done according to the operation profile if the objective is to estimate the failure rate. For debugging, operational profile-based testing is more efficient if the testing time is limited. However if high reliability is desired, testing needs to be more uniform. Defects in parts of the code that is infrequently executed can be hard to detect. To achieve very high reliability, special tests should be used to detect such defects.[16]

During testing, the *test profile* is generally chosen to reveal defects as fast as possible and thus may be significantly different from the operational profile. The failure intensity under the test profile can be significantly higher than under normal operation, the ratio of the two values is referred to as the test compression ratio.[3]

MODELING SOFTWARE RELIABILITY GROWTH

The fraction of cost needed for testing a software system to achieve a suitable reliability level can sometimes be as high as 60% of the overall cost. Testing must be carefully planned so that the software can be released by a target date. Even after a lengthy testing period, additional testing will always potentially detect more bugs. Software must be released, even if it is likely to have a few bugs, provided an appropriate reliability level has been achieved. Careful planning and decision-making requires the use of a SRGM which provides a mathematical relationship between testing time and the number of defects found.

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. The growth of reliability is generally specified in terms of either failure-intensity $\lambda(t)$, or *total expected faults* detected by time t, give by $\mu(t)$. The relationship between the two is given by

$$\lambda(t) = \frac{\mathrm{d}}{\mathrm{d}t}\mu(t) \tag{4}$$

Let the total number of defects at time t be $N(t)$. Let us assume that a defect is removed when it is found.

Here we will derive the most popular reliability growth model, the exponential model.[3] It assumes that at any time, the rate of finding (and removing) defects is proportional to the number of defects present. Using β_1 as a constant of proportionality, we can write

$$-\frac{\mathrm{d}N(t)}{\mathrm{d}t} = \beta_1 N(t) \tag{5}$$

It can be shown that the parameter β_1 is given by

$$\beta_1 = \frac{K}{\left(SQ\frac{1}{r}\right)} \tag{6}$$

where S is the total number of source instructions, Q is the number of object instructions per source instruction, and r is the object instruction execution rate of the computer being used. The term K is called *fault-exposure ratio*, its value has been found to in the range 1×10^{-7} to 10×10^{-7}, when t is measured in seconds of CPU execution time.

Eq. 5 can be solved to give

$$N(t) = N(0)\mathrm{e}^{-\beta_1 t} \tag{7}$$

When $N(0)$ is the initial total number of defects, the total expected faults detected by time t is then:

$$\begin{aligned}\mu(t) &= N(0) - N(t) \\ &= N(0)\left(1 - \mathrm{e}^{-\beta_1 t}\right)\end{aligned} \tag{8}$$

which is generally written in the form:

$$\mu(t) = \beta_0(1 - \mathrm{e}^{-\beta_1 t}) \tag{9}$$

where β_0, the total number of faults that would be eventually detected, is equal to $N(0)$. This assumes that no new defects are generated during debugging.

$$\lambda(t) = \beta_0 \beta_1 \mathrm{e}^{-\beta_1 t} \tag{10}$$

Using Eq. 4, we can obtain an expression for failure intensity using Eq. 9:

The exponential model is easy to understand and apply. One significant advantage of this model is that both parameters β_0 and β_1 have a clear interpretation and can be estimated even before testing begins. The models proposed by Jelinski and Muranda (1971), Shooman (1971), Goel and Okumoto (1979) and Musa (1975–1980) can be considered to be reformulations of the exponential model. The hyperexponential model, considered by Ohba, Yamada, and Lapri assumes that different exponential model parameter values apply for different sections of the software.

Many other SRGMs have been proposed and used. Several models have been compared for their predictive capability using data obtained from different projects. The exponential model fares well in comparison with other models; however, a couple of models can outperform the exponential model. We will here look at the logarithmic Poisson model, (also known as the Musa Okumoto model), which has been found to have a better predictive capability compared with the exponential model.[17,18]

Unlike the exponential model, the logarithmic Poisson model assumes that the fault exposure ratio K varies during testing.[19] The logarithmic Poisson model is a finite-time model, assuming that after a finite time, there will be no more faults to be found. The model can be stated as

$$\mu(t) = \beta_0 \ln(1 + \beta_1 t) \tag{11}$$

or alternatively,

$$\lambda(t) = \frac{\beta_0 \beta_1}{1 + \beta_1 t} \tag{12}$$

Eqs. 11 and 12 are applicable as long as $m(t) \leq N(0)$. In practice, the condition will almost always be satisfied, since testing always terminates when a few bugs are still likely to be present.

The variation in the fault exposure ratio K, as assumed by the logarithmic Poisson model has been observed in actual practice. The value of K declines at higher defect densities, as defects get harder to find. However, at low defect densities, K starts rising. This may be explained by the fact that real testing tends to be directed rather than random, and this starts affecting the behavior at low defect densities.

The two parameters for the logarithmic Poisson model, β_0 and β_1, do not have a simple interpretation. A possible interpretation is provided by Malaiya and Denton.[12] They have also given an approach for estimating the logarithmic Poisson model parameters β_0, β_1, once the exponential model parameters have been estimated.

The exponential model has been shown to have a negative bias; it tends to underestimate the number of defects that will be detected in a future interval. The logarithmic model also has a negative bias however it is much smaller. Among the major models, only the Littlewood-Verral Bayesian model exhibits a positive bias. This model has also been found to have good predictive capabilities, however because of computational complexity, and a lack of interpretation of the parameter values, it is not popular.

An SRGM can be applied in two different types of situations. To apply it before testing requires static estimation of parameters. During testing, actual test data is used to estimate the parameters.[18]

A. Before testing begins: A manager often has to come up with a preliminary plan for testing very early. For the exponential and the logarithmic models, it is possible to estimate the two parameter values based on defect density model and Eq. 6. One can then estimate the testing time needed to achieve the target failure intensity, MTTF or defect density.

Example 3: Let us assume that for a project, the initial defect density has been estimated, using the static model given in Eq. 3, and has been found to be 25 defects/KLOC. The software consists of 10,000 lines of C code. The code expansion ratio Q for C programs is about 2.5; hence, the compiled program will be about $10{,}000 \times 2.5 = 25{,}000$ object instructions. The testing is done on a computer that executes 70 million object instructions per second. Let us also assume that the fault exposure ratio K has an expected average value of 4×10^{-7}. We wish to estimate the testing time needed to achieve a defect density of 2.5 defects/KLOC.

For the exponential model, we can estimate that:

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

and from Eq. 6

$$\beta_1 = \frac{K}{\left(SQ\frac{1}{r}\right)} = \frac{4.0 \times 10^{-7}}{10{,}000 \times 2.5 \times \frac{1}{70 \times 10^6}} = 11.2 \times 10^{-4} \text{ per sec}$$

If t_1 is the time needed to achieve a defect density of 2.5/KLOC, then using Eq. 7,

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

giving us:

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

We can compute the failure intensity at time t_1 to be

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

For this example, it should be noted that the value of K (and hence t_1) may depend on the initial defect density and the testing strategy used. In many cases, the time t is specified in terms of the number of man-hours. We would then have to convert man-hours to CPU execution time by multiplying by an appropriate factor. This factor would have to be determined using recently collected data. An alternative way to estimate β_1 is found by noticing that Eq. 6 suggests that for the same environment, $\beta_1 \times I$ is constant. Thus, if for a prior project with 5 KLOC source code, the final value for β_1 was 2×10^{-3} per sec. Then for a new 15 KLOC project, β_1 can be estimated as $2 \times 10^{-3}/3 = 0.66 \times 10^{-3}$ per sec.

B. During testing: During testing, the defect finding rate can be recorded. By fitting an SRGM, the manager can estimate the additional testing time needed to achieve a desired reliability level. The major steps for using SRGMs are the following:

1. Collect and preprocess data: The failure intensity data includes a lot of short-term noise. To extract the long-term trend, the data often needs to be smoothed. A common form of smoothing is to use *grouped* data. It involves dividing the test duration into a number of intervals and then computing the average failure intensity in each interval.
2. Select a model and determine parameters: The best way to select a model is to rely on the past experience with other projects using same process. The exponential and logarithmic models are often good choices. Early test data has a lot of noise, thus a model that fits early data well, may not have the best predictive capability. The parameter values can be estimated using either least square or maximum likelihood approaches. In the very early phases of testing, the parameter values can fluctuate enormously; they should not be used until they have stabilized.
3. Perform analysis to decide how much more testing is needed: Using the fitted model, we can project how much additional testing needs to be done to achieve a desired failure intensity or estimated defect density. It is possible to recalibrate a model that does not confirm with the data to improve the accuracy of the projection. A model that describes the process well to start with can be improved very little by recalibration.

Example 4: This example is based on the T1 data reported by Musa.[3] For the first 12 hr of testing, the number of failures each hour is given in Table 5.

Thus, we can assume that during the middle of the first hour (i.e., $t = 30 \times 60 = 1800$ sec) the failure intensity is 0.0075 per sec. Fitting all the 12 data points to the exponential model (Eq. 12), we obtain:

$$\beta_0 = 101.47 \text{ and } \beta_1 = 5.22 \times 10^{-5}$$

Let us now assume that the target failure intensity is one failure per hour, i.e., 2.78×10^{-4} failures per second. An estimate of the stopping time t_f is then given by

$$2.78 \times 10^{-4} = 101.47 \times 5.22 \times 10^{-5} e^{-5.22 \times 10^{-5} \times t_f} \quad (13)$$

yielding $t_f = 5\ 6,\ 4\ 73$ sec, i.e., 15.69 hr, as shown in Fig. 1.

Table 5 Hourly failure data

Hour	Number of failures
1	27
2	16
3	11
4	10
5	11
6	7
7	2
8	5
9	3
10	1
11	4
12	7

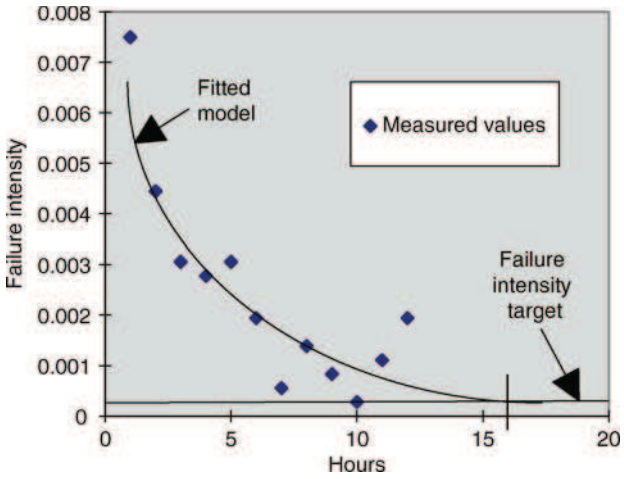

Fig. 1 Using an SRGM.

Investigations with the parameter values of the exponential model suggest that early during testing, the estimated value of β_0 tends to be lower than the final value, and the estimated value of β_1 tends to be higher. Thus, the value of β_0 tends to rise, and β_1 tends to fall, while the product $\beta_0\beta_1$ remains relatively unchanged. In Eq. 13, we can guess that the true value of β_1 should be smaller, and thus, the true value of t_f should be higher. Hence, the value 15.69 hr should be used as a lower estimate for the total test time needed.

In some cases it is useful to obtain interval estimates of the quantities of interest. The statistical methods can be found in the literature to do that.[11,19] Sometimes we wish to continue testing until a failure rate is achieved with a given confidence level, say 95%. Graphical and analytical methods for determining such stopping points are also available.[11]

The SRGMs assume that a uniform testing strategy is used throughout the testing period. In actual practice, the test strategy is changed from time to time. Each new strategy is initially very effective in detecting a different class of faults, causing a spike in failure intensity when a switch is made. A good smoothing approach will minimize the influence of these spikes during computation. A bigger problem arises when the software under test is not stable because of continuing additions to it. If the changes are significant, early data points should be dropped from the computations. If the additions are component by component, reliability data for each component can be separately collected and the methods presented in the next section can be used.

It should be noted that during operational use, the failures would be encountered at a lower rate than during testing. During testing, the software is exercised more thoroughly resulting in a higher failure intensity. The ratio of the failure intensity values during testing and operational use is termed *test compression ratio*, a higher test compression ratio implies more efficient testing.

Test Coverage Based Approach

Several software tools are available that can evaluate coverage of statements, branches, P-uses, etc. during testing. Higher test coverage means the software has been more thoroughly exercised.

It has been established that software test coverage is related to the residual defect density and hence reliability.[10,20] The relationship between the total number of defects found μ and the test coverage is nonlinear; however, at higher values of test coverage the relation is approximately linear. For example if we are using branch coverage C_B, we will find that for low values of C_B, μ remains close to zero. However at some value of C_B (we term it a *knee*), μ starts rising linearly, as given by

$$\mu = -a + bC_B, \quad C_B > \text{knee} \quad (14)$$

The values of the parameters a and b will depend on the software size and the initial defect density. The advantage of using coverage measures is that variations in test effectiveness will not influence the relationship, since test coverage directly measures how thoroughly a program has been exercised. For high reliability systems, a strict measure like p-use coverage should be used.

Figure 2 plots actual data from a project. By the end of testing 29 defects were found and 70% branch coverage was achieved. If testing were to continue until 100% branch coverage is achieved, then about 47 total defects would have been detected. Thus according to the model, about 18 residual defects were present when testing was terminated. Note that only one defect was detected when branch coverage was about 25%, thus the knee of the curve is approximately at branch coverage of 0.25.

VULNERABILITIES IN INTERNET-RELATED SOFTWARE

Internet related software systems including operating systems, servers, and browsers face escalating security challenges because Internet connectivity is growing and the number of security violations is increasing. CERT and other databases keep track of reported vulnerabilities. An increasing number of individuals and organizations depend on the Internet for financial and other critical services. That has made potential exploitation of vulnerabilities very attractive to criminals with suitable technical expertise.

Each such system passes through several phases: the release of the system, increasing popularity, peak and stability followed by decreasing popularity that ends with the system eventually becoming obsolete. There is a common pattern of three phases in the cumulative vulnerabilities plot for a specific version of software. We observe a slow rise when a product is first released. This becomes a steady stream of vulnerability reports as the product develops a market share and starts attracting attention of both professional and criminal vulnerability finders. When an alternative product starts taking away the market share, the rate of vulnerability finding drops in an older product. Fig. 3 shows a plot of vulnerabilities reported during January 1999–August 2002 for Windows 98.

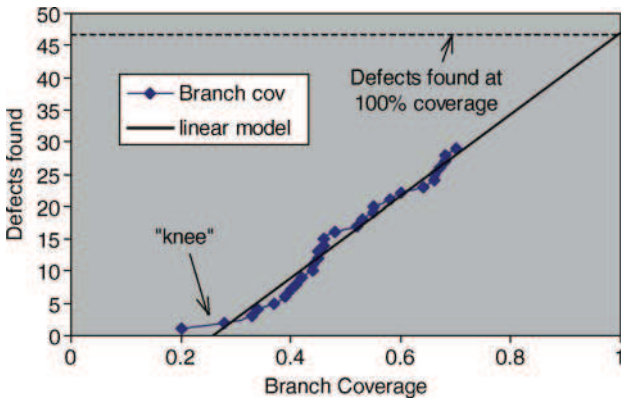

Fig. 2 Coverage-based modeling.

A model for the cumulative number of vulnerabilities y, against calendar time t is given by the equation below.[21,22]

$$y = \frac{B}{BC\mathrm{e}^{-ABt} + 1} \tag{15}$$

where A, B, and C are empirical constants determined from the recorded data. The parameter B gives the total number of vulnerabilities that will be eventually found. The chi-square goodness of fit examination shows that the data for several common operating systems fits the model very well.

The vulnerabilities in such a program are software defects that permit an unauthorized action. The density of vulnerabilities in a large software system is an important measure of risk. The *known vulnerability density* can be evaluated using the database of the reported vulnerabilities. Known vulnerability density V_{KD} can be defined as the reported number of vulnerabilities in the system per unit size of the system. This is given by

$$V_{\mathrm{KD}} = \frac{V_{\mathrm{K}}}{S} \tag{16}$$

where S is the size of software and V_{K} is the reported number of vulnerabilities in the system. Table 6 presents the values based on data from several sources.[21] It gives the known defect density D_{KD}, V_{KD}, and the ratios of the two.

In Table 6, we see that the source code size is respectively 16, 18, and 17 MSLOC (Million source lines of code) for NT and Win 98, approximately the same. The reported defect densities at release are similar for the NT and Win 98, but lower for Linux 6.2. The known vulnerabilities in Table 2 are as of September 2005. We notice that the vulnerability density for Windows NT 4.0 is 0.0113, significantly higher than the other two operating systems. The higher value for NT 4.0 may be due to several factors. First, as a server OS, it may contain more code that handles access mechanisms. Second, because attacking servers would generally be much more rewarding, it must have attracted a lot more testing effort resulting in detection of more vulnerabilities. NT has been around longer than Linux 6.2 resulting in more known vulnerabilities.

The last column in Table 6 gives the ratios of known vulnerabilities to known defects. The values range from 0.84% to 5.63%. It has been assumed by different researchers that the vulnerabilities can be from 1% or 5% of the total defects, the ratios in Table 6 justify the assumptions.

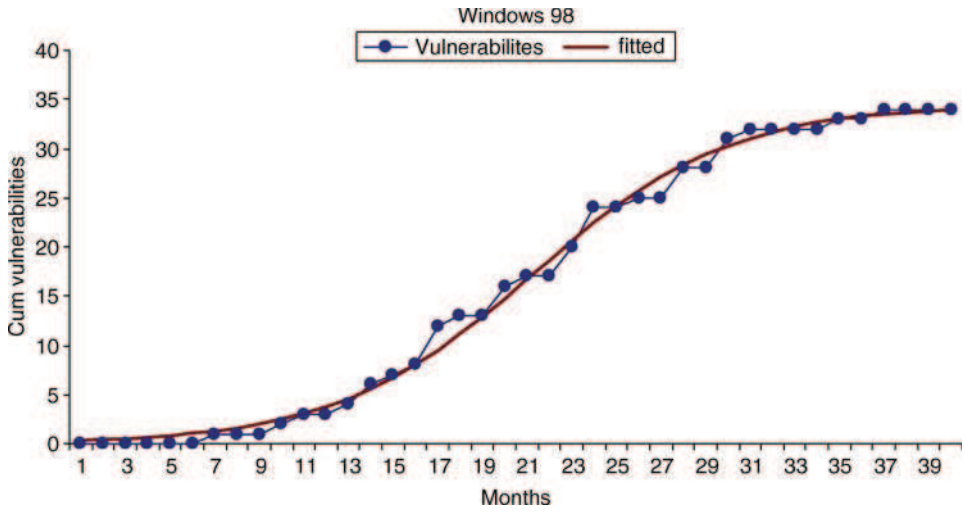

Fig. 3 Windows 98.

Table 6 Security vulnerabilities in some operating systems

System	Size in MSLOC	Known defects (1000 s)	D_{KD} (/KLOC)	Known vulnerabilities	V_{KD} (/KLOC)	Ratio V_{KD}/D_{KD}
NT 4.0	16	10	0.625	180	0.0113	1.80%
Win 98	18	10	0.556	84	0.0047	0.84%
RH Linux 6.2	17	2.1	0.123	118	0.0069	5.63%

RELIABILITY OF MULTICOMPONENT SYSTEMS

A large software system consists of a number of modules. It is common that in a given version, some modules from the previous version are retained which some modules are freshly added and others are modified from a previous version. It is also possible that the individual modules are developed and tested differently. These variations will result in different defect densities and failure rates. Here, we present methods for obtaining the system failure rate and the reliability if we know the reliabilities of the individual modules. Let us assume that for a system one module is under execution at a time. Modules will differ in how often and how long they are executed. If f_i is the fraction of the time module i that is under execution, then the mean system failure rate is given by Lakey and Neufelder as[23]

$$\lambda_{sys} = \sum_{i=1}^{n} f_i \, \lambda_i \tag{17}$$

where λ_i is the failure rate of the module i.

Let the mean duration of a single transaction be T. Let us assume that module i is called e_i times during T, and each time it is executed for duration d_i, then

$$f_i = \frac{e_i \cdot d_i}{T} \tag{18}$$

Let us define the system reliability R_{sys} as the probability that no failures will occur during a single transaction. From reliability theory, it is given by

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

Using Eqs. 17 and 18, we can write the above as

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

Since $\exp(-d_i\lambda_i)$ is R_i, single execution reliability of module i, we have

$$R_{sys} = \prod_{i=1}^{n} (R_i^{e_i}) \tag{19}$$

RELIABILITY ALLOCATION

When a software is developed as a set of interconnected modules, the testing effort can be allocated to different modules in such a way to minimize the total testing cost while achieving a desired reliability target. This is an example of the reliability allocation problem.

For the exponential SRGM, the failure rate as a function of testing time d_i for a module i is given by Eq. 10 above, which can be rewritten as

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

where $\lambda_{0i} = \beta_{0i}\beta_{1i}$ and β_{1i} are the SRGM parameters applicable to the module i. Thus the test cost d_i is given by a function of the ratio between λ_{0i} and λ_i.

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

Let us assume that a block i is under execution for a fraction f_i of the time where $\Sigma f_i = 1$. Then the reliability allocation problem can be written as

$$\text{Minimize the total cost } C = \sum_{i=1}^{n} \frac{1}{\beta_i} \ln\left(\frac{\lambda_{0i}}{\lambda_i}\right) \tag{20}$$

$$\text{Subject to } \lambda_{ST} \leq \sum_{i=1}^{n} f_i \lambda_i \tag{21}$$

where λ_{ST} is the desired overall failure intensity. The problem posed by Eqs. 10 and 11 is solved by using the Lagrange multiplier approach.[24] The solutions for the optimal failure rates are found as following

$$\lambda_1 = \frac{\frac{\lambda_{ST}}{f_1}}{\sum_{i=1}^{n} \frac{\beta_{1i}}{\beta_{1i}}} \quad \lambda_2 = \frac{\beta_{11} f_1}{\beta_{12} f_2} \lambda_1 \quad \cdots \quad \lambda_n = \frac{\beta_{11} f_1}{\beta_{1n} f_n} \lambda_1 \tag{22}$$

The optimal values of d_1 and d_i, $i \neq 1$ are given by

$$d_1 = \frac{1}{\beta_{11}} \ln\left(\frac{\lambda_{10} f_1 \sum_{i=1}^{n} \frac{\beta_{1i}}{\beta_{1i}}}{\lambda_{ST}}\right) \quad \text{and} \quad d_i = \frac{1}{\beta_{1i}} \ln\left(\frac{\lambda_{i0}\beta_{1i} f_i}{\lambda_1 \beta_{1i} f_1}\right) \tag{23}$$

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

Eq. 6 states that the parameter β_{1i} is inversely proportional to the software size, when measures in terms of the lines of code. The value of f_i can be initially assumed to be proportional to the code size. The values of λ_i and λ_{i0} do not depend on size but depend on the initial defect densities. Thus Eq. 22 states that the optimal values of the posttest failure rates $\lambda_1, \ldots, \lambda_n$ may be equal. Also if the initial defect densities are also all equal for all the blocks, then the optimal test times for each module is proportional to its size. When some of the blocks are more critical, additional weights can be assigned resulting in longer testing times for them.

Example 5: A software system uses five functional blocks B1–B5. We construct this example assuming sizes 1, 2, 3, 10, and 20 KLOC (thousand lines of code) respectively, and the initial defect densities of 20, 20, 20, 25, and 30 defects per KLOC respectively. Let us assume that measured parameter values are given in the top three rows, which are the inputs to the optimization problem. The solution obtained using Eqs. 22 and 23 are given in the two bottom rows. Let us now minimize the test cost such that the overall failure rate is less than or equal to 0.06 per unit time. Here the time units can be hours of testing time, or hours of CPU time used for testing.

Block	B_1	B_2	B_3	B_4	B_5
β_i	7×10^{-3}	3.5×10^{-3}	2.333×10^{-3}	7×10^{-4}	3.5×10^{-4}
λ_{i0}	0.14	0.14	0.14	0.175	0.21
x_i	0.028	0.056	0.083	0.278	0.556
Optimal λ_i	0.06	0.06	0.06	0.06	0.06
Optimal d_i	121.043	242.085	363.128	1.529×10^3	3.579×10^3

Note that the optimal values of λ_i for the five modules are equal, even though they start with different initial values. This requires a substantial part of the test effort allocated to largest blocks. The total cost in terms of testing is 5.835×10^3 hr.

TOOLS FOR SOFTWARE TESTING AND RELIABILITY

Software reliability has now emerged as an engineering discipline. It can require a significant amount of data collection and analysis. Tools are now becoming available that can automate several of the tasks. Here names of some of the representative tools are mentioned. Many of the tools may run on specific platforms only, and some are intended for some specific applications only. Installing and learning a tool can require a significant amount of time, thus a tool should be selected after a careful comparison of the applicable tools available. Some open-source tools are considered to be comparable to commercial tools.

- Automatic test generations: Conformiq Test Generator, ModelJUnit (open source), TestMaster (Teradyne), AETG (Bellcore), ARTG (CSU), etc.
- GUI testing: WinRunner (Mercury Interactive), RFT (IBM), eValid, etc.
- Memory testing: Purify (Relational), BoundsChecker (NuMega Tech.), etc.
- Defect tracking: BugBase (Archimides), Bugzilla (Mozilla), DVCS Tracker (Intersolv), DDTS (Qualtrack), etc.
- Test coverage evaluation: NCover, PureCoverage (IBM Relational), XSUDS (Bellcore), etc.

- Reliability growth modeling: CASRE (NASA), SMERFS (NSWC), ROBUST (CSU), etc.
- Defect density estimation: ROBUST (CSU).
- Coverage-based reliability modeling: ROBUST (CSU).
- Markov reliability evaluation: HARP (NASA), HiRel (NASA), PC Availability (Management Sciences), etc.
- Fault-tree analysis: RBD (Relax), Galileo (UV), CARA (Sydvest), FaultTree+ (AnSim), etc.

CONCLUSIONS

The reliability of a program is determined by the development process used and the thoroughness of testing. Enough software reliability data is now available that allows models to be validated. These models can be used to assess the reliability levels achieved as well as estimation of further measures needed to achieve the target reliability levels. This entry presents a summary of the approaches available today to manage software reliability.

REFERENCES

1. Callaghan, D.; O'Sullivan, C. Who should bear the cost of software bugs? Comput. Law Security Rep. **2005**, *21* (1), 56–60.
2. Holzmann, G.J.; Joshi, R. Reliable software systems design. In *Grand Challenge in Verification, Verified Software: Theories, Tools, Experiments*; Zurich, 2005. http://spinroot.com/gerard/pdf/zurich2005_1.pdf.
3. Musa, J.D. *Software Reliability Engineering: More Reliable Software, Faster Development and Testing*; McGraw-Hill: New York, 1999.
4. Malaiya, Y.K.; von Mayrhauser, A.; Srimani, P. An examination of fault exposure ratio. IEEE Trans. Software Eng. **1993,** 1087–1094.
5. Malaiya, Y.K., Srimani, P., Eds. *Software Reliability Models*; IEEE Computer Society Press: Los Alamitos, CA, 1991.
6. Engel, A.; Last, M. Modeling software testing costs and risks using fuzzy logic paradigm. J. Syst. Softw. **2007**, *80* (6), 817–835.
7. Carleton, A.D.; Park, R.E.; Florac, W.A. *Practical Software Measurement*, Tech. Report, SRI, CMU/SEI-97-HB-003.
8. Piwowarski, P.; Ohba, M.; Caruso, J. Coverage measurement experience during function test. Proc. Int. Conf. Software Eng. **1993**, 287–301.
9. Christof E. Technical controlling and software process improvement. J. Syst. Softw. **1999**, *46* (1), 25–39.
10. Malaiya, Y.K.; Li, N.; Bieman, J.; Karcich, R.; Skibbe, B. The relation between test coverage and reliability. Proc. IEEE-CS Int. Symp. Softw. Reliab. Eng. **1994**, 186–195.
11. Lyu, M.R., Ed.; *Handbook of Software Reliability Engineering*; McGraw-Hill: New York, 1996.
12. Malaiya, Y.K.; Denton, J. What do the software reliability growth model parameters represent. Proc. IEEE-CS Int. Symp. Softw. Reliab. Eng. (ISSRE) **1997**, 124–135.
13. Takahashi, M.; Kamayachi, Y. An empirical study of a model for program error prediction. Proc. Int. Conf. Softw. Eng. **1995**, 330–336.
14. Malaiya, Y.K.; Denton, J. Module size distribution and defect density. Proc. Int. Symp. Softw. Relaib. Eng. **2000**, 62–71.
15. Musa, J. More reliable, faster, cheaper testing through software reliability engineering. Tutorial Notes, ISSRE'97 **1997**, 1–88.
16. Yin, H.; Lebne-Dengel, Z.; Malaiya, Y.K. Automatic test generation using checkpoint encoding and antirandom testing. Proc. Int. Symp. Softw. Reliab. Eng. **1997**, 84–95.
17. Malaiya, Y.K.; Karunanithi, N.; Verma, P. Predictability of software reliability models. IEEE Trans. Reliab. **1992**, 539–546.
18. Almering, V.; Genuchten, M.; Cloudt, C.; Sonnemans, P.J.M. Using a software reliability growth model in practice. IEEE Softw. **2007**, 82–88.
19. Li, N.; Malaiya, Y.K. Fault exposure ratio: Estimation and applications. Proc. IEEE-CS Int. Symp. Softw. Reliab. Eng. **1993**, 372–381.
20. Lyu, M.R. Software reliability engineering: A roadmap. Future Softw. Eng. 23–25, **2007**, 153–170.
21. Alhazmi, O.H.; Malaiya, Y.K.; Ray, I. Measuring, analyzing and predicting security vulnerabilities in software systems. Comput. Security J. **2007**, *26* (3), 219–228.
22. Alhazmi, O.H.; Malaiya, Y.K. Application of vulnerability discovery models to major operating systems. IEEE Trans. Reliab. **2008**, 14–22.
23. Lakey, P.B.; Neufelder, A.M. *System and Software Reliability*. Assurance Notebook, Rome Lab, FSC-RELI 1997.
24. Malaiya, Y.K. Reliability allocation. In *Encyclopedia of Statistics in Quality and Reliability*; John Wiley & Sons: Hoboken, NJ, 2008.