

Software Reliability and Security

Yashwant K. Malaiya

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

INTRODUCTION

Software problems are the main causes of system failures today. There are many well-known cases of the tragic consequences of software failures. In critical systems, very high reliability is naturally expected. Software packages used everyday also need to be highly reliable, because the enormous investment of the software developer is at stake. Studies have shown that reliability is regarded as the most important attribute by potential customers. All software developed will have a significant number of defects. All programs must be tested and debugged, until sufficiently high reliability is achieved. It is not possible to ensure that all the defects in a software system have been found and removed; however, the number of remaining bugs must be very small. As 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 because of the emergence of well-understood and validated approaches. Enough industrial and experimental data are available 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 during testing. First, we discuss the reliability approaches taken during different phases of software development. Integration and interoperability testing are examined. 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 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 discuss a similar model for security vulnerabilities in internet-related software. We also see how reliability of a system can be evaluated if we know the failure rates of the components. Finally, the article 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.^[1]

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

Defect (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 testability of a defect is defined as the probability of detecting it with a randomly chosen input. Defects with very low testability can be very difficult to detect. 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. The software will fail time to time during operational use when it cannot respond correctly to an input. During operational use, bug fixes that update the software are often released. 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.^[2]

RELIABILITY APPROACHES DURING THE SOFTWARE LIFE CYCLE PHASES

A competitive and mature software development organization targets a high reliability objective from the very beginning of software development. Generally, the software life cycle is divided into the following phases. It is not uncommon for developers to go to a previous phase, it may become necessary because of the requirement changes or a need to make changes in the design. It is preferable to catch defects early as it would be more expensive to fix them later.

1. *Requirements and definition*: In this phase, the developing organization 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.

2. *Design*: In this phase, the system is specified as an interconnection of units, such that each unit is well defined and can be developed and tested independently. The design is reviewed to recognize errors.

3. *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.

4. *Testing*: This phase is a critical part of the quest for high reliability and can take 30–60% of the entire development time. It is often divided into the following separate phases.

- a. *Unit test*: In this phase of testing, each unit is separately tested, and changes are done to remove the defects found. As each unit is relatively small and can be tested independently, they can be exercised much more thoroughly than a large program.
- b. *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.
- c. *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 the actual operation.
- d. *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 on how the actual users would use the system. This is also called α -testing. This is often followed by β -testing, which involves use of the β -version by the actual users.

5. *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 typical fraction of total defects introduced and found during a phase.^[3,4] Most defects occur during the 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. It has been observed that the testing phases can account for 30–60% of the entire development effort.

Two types of testing deserve special attention: integration testing and interoperability testing. Integration testing assumes that unit testing has already been done, and thus the focus is on testing for defects that are associated with interaction among the modules. Exercising a unit module requires use of a driver module, which will call the unit under test. If a unit module does not call other units, it is called a terminal module. If a unit module calls other modules, which are not yet ready to be used, surrogate modules called stubs simulate the interaction.

Integration testing can be bottom-up or top-down. In the bottom-up approach, integration starts with attaching the terminal modules to the modules that call them. This requires the use of drivers to drive the higher-level modules. The top-down integration starts with connecting the highest-level modules with the

Table 1 Defects introduced and found during different phases

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

modules called by them. This requires the use of stubs until finally the terminal modules are integrated.

Integration testing should include passing interface data that represent normal cases (n), special cases (s), and illegal cases (i). For two interacting modules A and B, all combinations of $\{A_n, A_s, A_i\}$ and $\{B_n, B_s, B_i\}$ should be tested. Thus, if a value represents a normal case for A and a special case for B, the corresponding combination is (A_n, B_s).

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 ensure interoperability.

Interoperability requires the following.

- *Mutual compatibility of exchanged data units*: The data structures exchanged must use the same format (identical fields and data types). If different formats are used, a suitable translator unit is required.
- *Mutual compatibility of control signals*: In addition to the data units, often control information needs to be exchanged that provides context to the data unit. Exchange of some control signals may sometimes be implicit. Testing must ensure that the control protocol is properly specified (this sometimes requires a formal representation) and control signals are properly exchanged.

Interoperability testing may be considered a more general form of integration testing. Interoperability testing must be performed whenever a component is added or upgraded. Some field-specific interoperability standards have been developed. For example, Z39.50 is an international (ISO 23950) standard defining a protocol for computer-to-computer information retrieval.^[5]

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 because of debugging and becomes constant once defect removal is stopped. The following are the most common reliability measures used:

1. *System availability*: Following classical reliability terminology, we can define availability of a system as:

$$A(t) = \Pr \{\text{system is operational at time } t\} \quad (1)$$

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

$$R = \Pr \{\text{a single transaction will not encounter a failure}\} \quad (2)$$

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

3. *Mean-time-to-failure (MTTF)*: The expected duration between two successive failures.

4. *Failure intensity (λ)*: The expected number of failures per unit time. Note that MTTF is the inverse of failure intensity. Thus, if MTTF is 500 hr, the failure intensity is $1/500 = 0.002 \text{ hr}^{-1}$.

As 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.

5. *Defect density*: This is 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 per KLOC, whereas for other applications 0.5 defects per 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, we assume here that each defect has the same weight.

6. *Test coverage measures*: Tools are now available that can automatically evaluate how thoroughly a software has been exercised. The following are some of the common coverage measures

- a. *Statement coverage*: The fraction of all statements actually exercised during testing.
- b. *Branch coverage*: The fraction of all branches that were executed by the tests.
- c. *P-use coverage*: The fraction of all predicate use (p-use) pairs covered during testing. A p-use pair includes two points in the program, a point where the value of a variable is defined or modified followed by a point where it is used for a branching decision, i.e., a predicate.

The first two are structural coverage measures, while the last is a data-flow coverage measure. As discussed below, test coverage is correlated with the number of defects that will be triggered during testing.^[6] Hundred percent 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 are 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. Second, by estimating the defect density, one can use a reliability growth model to estimate the testing effort needed. The model by Malaiya and Denton, based on the data reported in the literature, is given by^[7]

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

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; and 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 1. We propose the following preliminary submodels for each factor.

Phase Factor F_{ph}

Table 2 presents a simple model using the actual data reported by Musa et al. and the error profile presented by Piwowarski et al. It takes the default value of 1 to represent the beginning of the system test phase.

Table 2 Phase factor F_{ph}

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

Table 3 The programming team factor F_{pt}

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

The Programming Team Factor F_{pt}

The defect density varies significantly because of the coding and debugging capabilities of the individuals involved. A quantitative characterization in terms of the programmer's average experience in years is given by Takahashi and Kamayachi.^[8] Their model can take into account programming experience of up to 7 yr, 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, which can also be taken into account here.

The Process Maturity Factor F_m

This factor takes into account the rigor of the software development process in a specific organization. The SEI Capability Maturity Model level, can be used to quantify it. Here, we assume level II as the default level, as 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 and also 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

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

software size. As 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.

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. As module sizes tend to be unevenly distributed, there may be an overall effect of module size distribution.^[9] 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. As 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 similar to the one for which the projection needs to be made. Some of the 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 (Software Engineering Institute) 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 LOC. 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 $12 \times 2.5 \times 1 \times 1 \times (1 + 0.4 \times 0.2) \times 1 = 32.4$ per KSLOC and $16 \times 2.5 \times 1 \times 1 \times (1 + 0.4 \times 0.2 \times 1) = 43.2$

per 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 following classes:

1. *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.
2. *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 the random-testing approach, the inputs are selected randomly. In the 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 nonspecial cases.

Some faults are easily detected, i.e., they have high testability. Some faults have very low testability; they are triggered only under a 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 remaining faults 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 the section "Software Reliability and Measures." Branch coverage is a stricter measure than statement coverage. Some organizations use branch coverage (say 85%) as the minimum criterion. For very high-reliability programs, a stricter 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.^[10] 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.	Database 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 are infrequently executed can be hard to detect. To achieve very high reliability, special tests should be used to detect such defects.^[11]

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 software reliability growth model (SRGM).

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 , given by $\mu(t)$. The relationship between the two is given by

$$\lambda(t) = \frac{d}{dt}\mu(t) \quad (4)$$

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. 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:

$$-\frac{dN(t)}{dt} = \beta_1 N(t) \quad (5)$$

It can be shown that the parameter β_1 is given by

$$\beta_1 = \frac{K}{(SQ\frac{1}{r})} \quad (6)$$

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 be 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:

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

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

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

which is generally written in the form

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

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.

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

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

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), God 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 sections of the software are separately governed by an exponential model, with different parameter values for different sections.

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 model, proposed by Musa and Okumoto, which has been found to have a better predictive capability compared with the exponential model.

Unlike the exponential model, the logarithmic model assumes that the fault exposure ratio K varies during testing.^[12] The logarithmic model is also 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:

$$\mu(t) = \beta_0 \ln(1 + \beta_1 t) \quad (11)$$

or

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

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

The variation in K , as assumed by the logarithmic 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 model, β_0 and β_1 , do not have a simple interpretation. A possible interpretation is provided by Malaiya and Denton.^[7] They have also given an approach for estimating the logarithmic model parameters β_0^L , β_1^L , 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 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. Applying it before testing requires static estimation of parameters. During testing, actual test data are used to estimate the parameters.

Before Testing Begin

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 per KLOC. The software consists of 10,000 Loc. 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 per KLOC.

For the exponential model, we can estimate that:

$$\beta_0 = N(O) = 25 \times 10 = 250 \text{ defects.}$$

and from Eq. (6)

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

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

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

giving

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

We can compute the failure intensity at time t_1 to be

$$\begin{aligned} \lambda(t_1) &= 250 \times 11.2 \times 10^{-4} e^{-11.2 \times 10^{-4} t_1} \\ &= 0.028 \text{ failure/sec} \end{aligned}$$

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 noting 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 $2 \times 10^{-3} \text{ sec}^{-1}$. Then, for a new 15 KLOC project, β_1 can be estimated as $2 \times 10^{-3}/3 = 0.66 \times 10^{-3} \text{ sec}^{-1}$.

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 as follows:

1. *Collect and preprocess data:* The failure intensity data include a lot of short-term noise. To extract the long-term trend, the data often need 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 have 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.^[1] 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., $t = 30 \times 60 = 1800 \text{ sec}$) the failure intensity is 0.0075 sec^{-1} . Fitting all the 12 data points to the exponential model [Eq. (12)], we obtain:

$$\beta_0 = 101.47$$

and

$$\beta_1 = 5.22 \times 10^{-5}$$

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

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 $t_f = 56,473$ sec., i.e., 15.69 hr, as shown in Fig. 1.

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 that of β_1 tends to fall, with the product $\beta_0\beta_1$ remaining relatively unchanged. In Eq. (13) above, 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 to do that can be found in the literature.^[12,13] 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.^[13]

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.

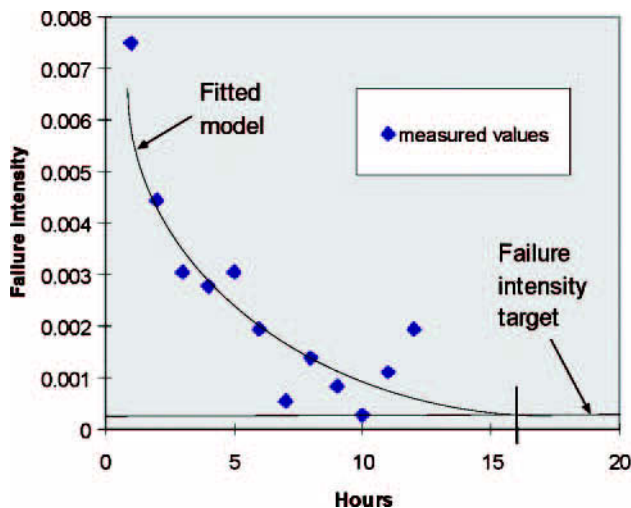


Fig. 1 Using an SRGM. (View this art in color at www.dekker.com.)

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.

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.^[5] The total number of defects found, μ , is linearly related to the test coverage measures at higher values of test coverage. 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 this Eq. (14).

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

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, as 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.

Fig. 2 below plots the 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

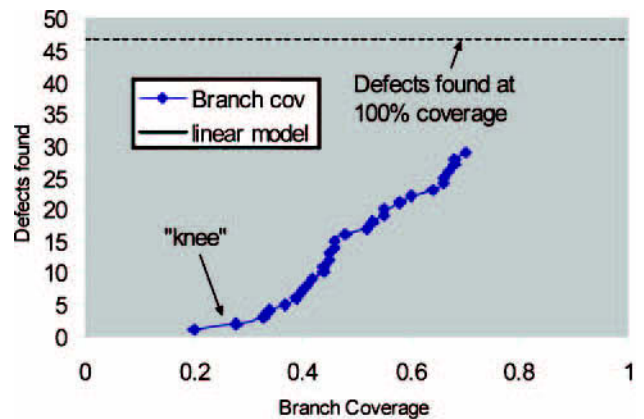


Fig. 2 Coverage based modeling. (View this art in color at www.dekker.com.)

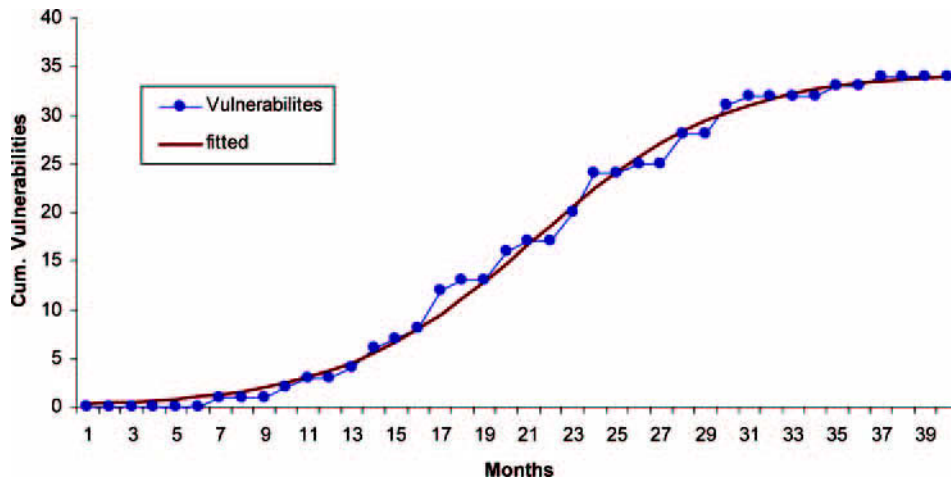


Fig. 3 Windows 98 vulnerabilities. (View this art in color at www.dekker.com.)

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 the reported vulnerabilities. An increasing number of individuals and organizations depend on the Internet for financial and other critical services. This 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 below shows a plot of vulnerabilities reported during January 1999 to August 2002 for Windows 98.

A model for the cumulative number of vulnerabilities y , against calendar time t is given by Eq. (15).^[14]

$$y = \frac{B}{BCe^{-ABt} + 1} \quad (15)$$

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 χ -square goodness of fit examination shows that the data for several common operating systems fit 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:

$$V_{KD} = \frac{V_K}{S} \quad (16)$$

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.^[14] 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 16 and 18 MSLOC (million source lines of code) for NT and Win 98, respectively, approximately the same. The reported defect densities at release, 0.625 and 0.556,

Table 6 Security vulnerabilities in Windows NT 4.0 and Windows 98

System	MSLOC	Known defects (1000s)	$D_{KD}/(KLOC)$	Known vulnerabilities	$V_{KD}/(KLOC)$	Ratio (%) V_{KD}/D_{KD}
NT 4.0	16	10	0.625	162	0.0101	1.62
Win 98	18	10	0.556	58	0.0032	0.58

are also similar for the two OSs. The known vulnerabilities in Table 2 are as of July 2004. We notice that for Windows 98 the vulnerability density is 0.0032, whereas for Windows NT 4.0 it is 0.0101, significantly higher. The higher values for NT 4.0 may be due two 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.

The last column in Table 6 gives the ratios of known vulnerabilities to known defects. For the two OSs the values are 1.62% and 0.58%. It has been assumed by different researchers that the vulnerabilities can be 1% or 5% of the total defects. The values given in Table 6 suggest that 1% may be closer to reality.

RELIABILITY OF MULTICOMPONENT SYSTEMS

A large software system consists of a number of modules. It is possible that the individual modules are developed and tested differently resulting in different defect densities and failure rates. Here, we will 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 under execution, then the mean system failure rate is given by:^[16]

$$\lambda_{\text{sys}} = \sum_{i=1}^n f_i \lambda_i \quad (17)$$

where λ_i is the failure rate of the module i . Here, each major interoperability interface should be regarded to be a virtual module.

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

$$f_i = \frac{e_i d_i}{T} \quad (18)$$

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

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

Using Eq. (17) and (18), we can write the above as:

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

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

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

Multiple-Version Systems

In some critical applications, like defense or avionics, multiple versions of the same program are sometimes used. Each version is implemented and tested independently to minimize the probability of a multiple number of them failing at the same time. The most common implementation uses triplication and voting on the result. The system is assumed to operate correctly as long as the results of at least two of the versions agree. This assumes the voting mechanism to be perfect. If the failures in the three versions are truly independent, the improvement in reliability can be dramatic; however, it has been shown that correlated failures must be taken into account.

In a three-version system, let q_3 be the probability of all three versions failing for the same input. Also, let q_2 be the probability that any two versions will fail together. As three different pairs are possible among the three versions, the probability P_{sys} of the system failing is:

$$P_{\text{sys}} = q_3 + 3q_2 \quad (20)$$

In the ideal case, the failures are statistically independent. If the probability of a single version failing is p , the above equation can be written for an ideal case as:

$$P_{\text{sys}} = p^3 + 3(1 - p)p^2 \quad (21)$$

In practice, there is a significant correlation, requiring estimation of q_3 and q_2 for system reliability evaluation.

Example 5: This example is based on the data collected by Knight and Leveson, and the computations by Hatton.^[15] In a three-version system, let the probability of a version failing for a transaction be 0.0004. Then, in the absence of any correlated failures, we can achieve a system failure probability of:

$$\begin{aligned} P_{\text{sys}} &= (0.0004)^2 + 3(1 - 0.0004)(0.0004)^2 \\ &= 4.8 \times 10^{-7} \end{aligned}$$

which would represent a remarkable improvement by a factor of $0.0004/4.8 \times 10^{-7} = 833.3$. However, let us assume that experiments have found $q_3 = 2.5 \times 10^{-7}$ and $q_2 = 2.5 \times 10^{-6}$, then

$$P_{\text{sys}} = 2.5 \times 10^{-7} + 3 \times 2.5 \times 10^{-6} = 7.75 \times 10^{-6}$$

This yields a more realistic improvement factor of $0.0004/7.75 \times 10^{-6} = 51.6$.

Hatton points out that the state-of-the-art techniques have been found to reduce defect density only by a factor of 10. Hence, an improvement factor of about 50 may be unattainable except by using N-version redundancy.

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, the 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.

- *Automatic test generations*: TestMaster (Tera-dyne), AETG (Bellcore), ARTG (CSU), etc.
- *GUI testing*: QA Partner (Seague), WinRunner (Mercury Interactive), etc.
- *Memory testing*: Purify (Relational), Bounds Checker (NuMega Tech.), etc.
- *Defect tracking*: BugBase (Archimides), DVCS Tracker (Intersolv), DDTS (Qualtrack), etc.
- *Interoperability*: Interoperability tool (AlphaWorks).
- *Test coverage evaluation*: Jcover (Man Machine Systems), GCT (Testing Foundation), PureCoverage (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.

REFERENCES

1. Musa, J.D. *Software Reliability Engineering*; McGraw-Hill, 1999.
2. Malaiya, Y.K.; Srimani, P. *Software Reliability Models*; IEEE Computer Society Press, 1990.

3. Carleton, A.D.; Park, R.E.; Florac, W.A. *Practical Software Measurement*. Technical Report; SRI, CMU/SEI-97-HB-003.
4. Piwowarski, P.; Ohba, M.; Caruso, J. Coverage measurement experience during function test. Proceedings of International Conference on Software Engineering, 1993; 287–301.
5. International Standard Z39.50 Maintenance Agency; <http://www.loc.gov/z3950/agency> (accessed Mar 21 2005).
6. Malaiya, Y.K.; Li, N.; Bieman, J.; Karcich, R.; Skibbe, B.; The relation between test coverage and reliability. Proceedings IEEE-CS International Symposium on Software Reliability Engineering, Nov 1994; 186–195.
7. Malaiya, Y.K.; Denton, J. What do the software reliability growth model parameters represent. Proceedings IEEE-CS International Symposium on Software Reliability Engineering ISSRE, Nov 1997; 124–135.
8. Takahashi, M.; Kamayachi, Y. An empirical study of a model for program error prediction. Proceeding International Conference on Software Engineering, Aug 1995; 330–336.
9. Malaiya, Y.K.; Denton, J. Module size distribution and defect density. Proceedings IEEE International Symposium Software Reliability Engineering, Oct 2000; 62–71.
10. Musa, J. More Reliable, Faster, Cheaper Testing through Software Reliability Engineering, Tutorial Notes, ISSRE '97, Nov 1997; 1–88.
11. Yin, H.; Lebne-Dengel, Z.; Malaiya, Y.K. Automatic test generation using checkpoint encoding and antirandom testing. International Symposium on Software Reliability Engineering, 1997; 84–95.
12. Li, N.; Malaiya, Y.K. Fault exposure ratio: estimation and applications. Proceedings IEEE-CS International Symposium on Software Reliability Engineering, Nov 1993; 372–381.
13. Lyu, M.R., Ed.; *Handbook of Software Reliability Engineering*; McGraw-Hill, 1996; 71–117.
14. Alhazmi, O.H.; Malaiya, Y.K. Quantitative vulnerability assessment in systems software. Proceedings IEEE Reliability & Maintainability Symposium, Jan. 2005; 615–620.
15. Hatton, L. *N-version Design Versus One Good Design*; IEEE Software; Nov/Dec 1997; 71–76.
16. Lakey, P.B.; Neufelder, A.M. *System and Software Reliability Assurance Notebook*; Rome Lab, FSC-RELI, 1997.