Software Reliability

Yashwant K. Malaiya
Computer Science Department
Colorado State University
Fort Collins CO 80523
malaiya@cs.colostate.edu

I. Introduction

Software now controls banking systems, all forms of telecommunications, process control in nuclear plants and factories, as well as defense systems. Even in households without a PC, many of the gadgets and the automobiles are software controlled. The society has developed an extraordinary dependence on software. There are many well known cases of tragic consequences of software failures. In popular software packages used everyday, a very high degree of reliability is needed, 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.

It is not possible to write software which is totally defect free, except possibly for very small programs. All programs must be tested and debugged, until sufficiently high reliability is achieved. Total elimination of all faults in large software systems is infeasible. Software must be released at some point in time, further delay will cause unacceptable 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.

In recent past, enough data has become available to develop and evaluate methods for achieving high reliability. Developing reliable software has become an engineering discipline rather than an art. For hardware systems, quantitative methods for achieving and measuring reliability have been in universal use for a long time. Similar techniques for software are coming in use due to emergence of well understood and validated approaches.

Here we will use the terms *failure* and a *defect* as defined below [1].

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

Defect (or fault): 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.

Some mathematical concepts are applicable to both software and hardware reliability. Hardware faults often occur due to aging. Combined with manufacturing variation in the quality of identical hardware components, the reliability variation can be characterized as exponential decay with time. On the other hand, the software reliability improves during testing as bugs are found and removed. Once released, the software reliability is fixed. The software will fail time to time during operational use when it cannot respond correctly to an input. Reliability of hardware components is often estimated by collecting failure data for a large number of identical units. 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].

II. Development 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.

A. 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.

B. 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.

C. Coding: In this phase, the actual program for each unit is written, generally in a higher level language such as C 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 30 to 60% of the entire development time. It is generally divided into these separate phases.

1. Unit test: In this phase, 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 actual use by the users.

5. Operational use: 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 release.

6. Regression testing: 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.

It should be noted that the exact definition of a test phase and its exit criterion may vary from organization to organization.

Table 1 below shows the typical fraction of total defects introduced and found during a phase [3, 4]. Most defects occur 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. It has been observed that the testing phases can account for 30% to 60% of the entire development effort.

Table 1: Defects introduced and found during different phases.

	Defects (%)		
Phase	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

III. 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.

Durational reliability: Following classical reliability terminology, we can define *reliability* of a software system as:

$$R(t) = Pr \{\text{no system failures during } (o,t)\} \quad (1)$$

Transaction reliability: Sometimes a *single-transaction* reliability measure, as defined below, is more 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).

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

Failure intensity (λ): The expected number of failures per unit time. Note that:

(3)

$$MTTF = \frac{1}{\lambda}$$

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:

(4)

$$A = \frac{\lambda_t}{\lambda_{op}}$$

and is controlled by the test selection strategy and the type of application.

Example 1: For a certain telecommunication application, the acceleration factor A has been found to be 10 in the past. For the current version, the target operational failure intensity has been decided to be 2.5×10^{-3} per second based on market studies. Then the target test failure intensity is:

$$\lambda_t = A \,.\, \lambda_{op} = 10 \times 2.5 \times 10^{-3}$$
$$= 2.5 \times 10^{-2} \text{ per second}$$

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 5 defects/KLOC may be 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. These are some of the common coverage measures.

- Statement coverage: The fraction of all statements actually exercised during testing.
- Branch coverage: The fraction of all branches that were executed by the tests.
- 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 [5]. 100% statement coverage can often be quite easy to achieve. Sometimes a predetermined branch coverage, say 85%, may be used as an acceptance criterion for testing.

IV. What Factors Control Defect Density?

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. The model by Malaiya and Denton [6], based on the data reported in the literature, is given by

$$D = C.F_{ph}.F_{pt}.F_{m}.F_{S} \quad (5)$$

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 thousand source lines of code (KSLOC). We propose 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 is in terms of programmers average experience in years is given by Takahashi and Kamayachi [7]. 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.

Table 3: The Programming Team Factor F_{pt}

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

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.

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

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. We allow other factors to be taken in to account by calibrating the overall model.

Calibrating and using the defect density model: The model given in Equation 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 suggests 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 2: For an organization, the value of C has been found to be between 12 to 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 equation (5), 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.

V. 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.

Some faults are easily detected, i.e. have high *testability*. 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 III. 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 [8]. 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.

VI. 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 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:

(6)

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

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:

(7)

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

It can be shown that the parameter β_1 is given by:

(8)

$$\beta_1 = \frac{K}{(S.Q.\frac{1}{r})}$$

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 $1x10^{-7}$ to $10x10^{-7}$, when t is measured in seconds of CPU execution time.

Equation (7) can be solved to give:

(9)

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

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

(10)

$$\mu(t) = N(0) - N(t)$$
$$= N(0)(1 - e^{-\beta_1 t})$$

which is generally written in the form:

(11)

$$\mu(t) = \beta_o(1 - e^{-\beta_1 t})$$

where β_o, 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 equation (6), we can obtain an expression for failure intensity using equation (11):

(12)

$$\lambda(t) = \beta_o \beta_1 e^{-\beta_1 t}$$

The exponential model is easy to understand and apply. One significant advantage of this model is that both parameters β_o 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 (1975-80) 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. 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:

(13)

$$\mu(t) = \beta_o \ln(1 + \beta_1 t)$$

or alternatively,

(14)

$$\lambda(t) = \frac{\beta_o \beta_1}{1 + \beta_1 t}$$

Equations (13) and (14) are applicable as long as $m(t) \leq N(0)$. In practice the condition will almost always be satisfied, since 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_o and β_1, do not have a simple interpretation. A possible interpretation is provided by Malaiya and Denton [6]. They have also given an approach for estimating the logarithmic model parameters β_o^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.

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 equation (8). 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 equation (5), 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 x 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:

$$\beta_o = N(O) = 25x10 = 250 \text{ defects,}$$

and from (8)

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

If t_1 is the time needed to achieve a defect density of 2.5/KLOC, then using equation (9),

$$\frac{N(t_1)}{N(O)} = \frac{2.5 \times 10}{25 \times 10} = \exp(-11.2x10^{-4}.t_1)$$

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:

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

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 equation (8) suggests that for the same environment, $\beta_1 x I$ is constant. Thus, if for a prior project with 5 KLOC source code, the final value for β_1 was $2x10^{-3}$ per sec. Then for a new 15 KLOC project, β_1 can be estimated as $2x10^{-3}/3 = 0.66x10^{-3}$ 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 pre-process 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 [1]. For the first 12 hours of

testing, the number of failures each hour is given in Table 5 below.

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

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

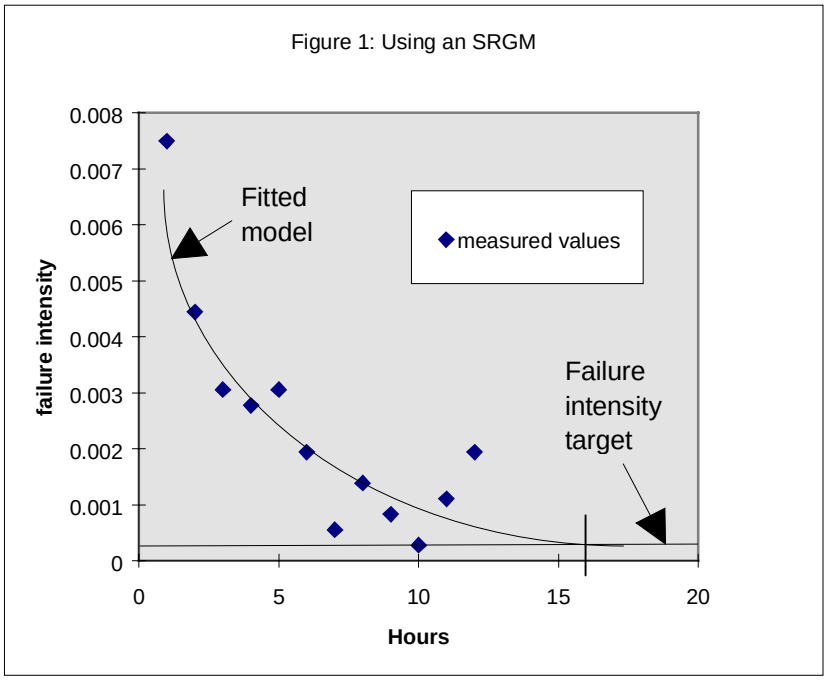
$$\beta_o = 101.47 \text{ and } \beta_1 = 5.22 \times 10^{-5}$$

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:

(15)

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

yielding t_f = 5 6, 4 73 sec., i.e. 15.69 hours, as shown in Figure 1 below.



Investigations with the parameter values of the exponential model suggest that early during testing, the estimated value of β_o tends to be lower than the final value, and the estimated value of β_1 tends to be higher. Thus, the value of β_o tends to rise, and β_1 tends to fall, with the product $\beta_o\beta_1$ remains relatively unchanged. In the equation (15) 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 hours should be used as a lower estimate for the total test time needed.

The SRGMs assume that a uniform testing strategy is used throughout the testing period. In actual practice, the test strategy is changed 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 [9]. 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 has been established that software test coverage is related to the residual defect density and hence reliability [5]. The defect coverage C_D is linearly related to the test coverage measures at higer values of test coverage. For example if we are using branch coverage C_B, we will find that low values of C_B, C_D remains close to zero. However at some value of C_B, C_D starts rising linearly, as given by this equation.

(16)

$$C_D = -a + b.C_B, \quad C_B > 0.5$$

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.

VII. Reliability of Multi-component 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.

Sequential execution: 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 is under execution, then the mean system failure rate is given by:

(17)

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

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

(18)

$$f_i = \frac{e_i . d_i}{T}$$

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 equations (16) and (17), we can write the above as:

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

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

(19)

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

Concurrent execution: Some systems involve concurrently executing modules. They are required to run without failures for the system to operate correctly. In this case, the system failure rate is given by [11]:

(20)

$$\lambda_{sys} = \sum_{j=1}^{m} \lambda_j$$

if there are j concurrently executing modules.

N-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 3-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. Since three different pairs are possible among the three versions, the probability P_{sys} of the system failing is:

(21)

$$P_{sys} = q_3 + 3q_2$$

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 idea case as:

(22)

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

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

Example 5: This example is based on the date collected by Knight and Leveson, and the computations by Hatton [12]. In a 3-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:

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

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

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

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.

VIII. Tools for Software 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.

- Automatic test generations: TestMaster (Teradyne), AETG (Bellcore), ARTG (CSU), etc.
- GUI testing: QA Partner (Seague), WinRunner (Mercury Interactive) etc.
- Memory testing: BoundsChecker (NuMega Tech.), Purify (Relational) etc.
- Defect tracking: BugBase (Archimides), DVCS Tracker (Intersolv), DDTS (Qualtrack) etc.
- Test coverage evaluation: GCT (Testing Foundation), PureCoverage (Relational), ATAC (Bellcore) etc.
- Reliability growth modeling: SMERFS (NSWC), CASRE (NASA), 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.

IX. References:

1. J. D. Musa, A. Ianino and K. Okumoto, *Software Reliability- Measurement, Prediction, Applications*, McGraw-Hill, 1987.
2. Y. K. Malaiya and P. Srimani, Ed., *Software Reliability Models*, IEEE Computer Society Press, 1990.
3. A. D. Carleton, R. E. Park and W. A. Florac, *Practical Software Measurement*, Tech. Report, SRI, CMU/SEI-97-HB-003.
4. P. Piwowarski, M. Ohba and J. Caruso, ‘‘Coverage Measurement Experience during Function Test,’’ Proc. Int. Conference on Software Engineering, 1993, pp. 287-301.

5. Y. K. Malaiya, N. Li, J. Bieman, R. Karcich and B. Skibbe ‘‘The Relation between Test Coverage and Reliability ,’’ Proc. IEEE-CS Int. Symposium on Software Reliability Engineering, Nov. 1994, pp. 186-195.

6. Y.K. Malaiya and J. Denton, ``What do the Software Reliability Growth Model Parameters Represent,’’ Proc. IEEE-CS Int. Symposium on Software Reliability Engineering ISSRE, Nov. 1997, pp. 124-135.

7. M. Takahashi and Y. Kamayachi, ``An Emprical study of a Model for Program Error Prediction,’’ Proc. Int. Conference on Software Engineering, Aug. 1995, pp. 330-336.

8. J. Musa, ``More Reliable, Faster, Cheaper Testing through Software Reliability Engineering’’, Tutorial Notes, ISSRE ’97, Nov. 1997, pp. 1-88.

9. N. Li and Y. K. Malaiya, ``Fault Exposure Ratio: Estimation and Applications,’’ Proc. IEEE-CS Int. Symposium on Software Reliability Engineering, Nov. 1993, pp. 372-381.

10. N. Li and Y. K. Malaiya, ``Enhancing accuracy of Software Reliability Prediction,’’ Proc. IEEE-CS Int. Symposium on Software Reliability Engineering, Nov. 1993, pp. 71-79.

11. P.B. Lakey and A. M. Neufelder, *System and Software Reliability Assurance Notebook*, Rome Lab, FSC-RELI, 1997.

12. L. Hatton, ``N-version Design Versus One Good Design’’ IEEE Software, Nov./Dec. 1997, pp. 71-76.