Draft version of Y. K. Malaiya, Assessing Software Reliability Enhancement Achievable through Testing, Recent Advancements in Software Reliability Assurance 2019, pp. 107-138.

Chapter 7: Assessing Software Reliability Enhancement Achievable through Testing

Y. K. Malaiya

7.1 The nature of software defects

The defects in software arise during different phases of software development. The development process can be regarded as a multi-step translation of the software requirements into executable code. In terms of the waterfall model, the steps can be described as (Boem 1988).

0. Business case
1. Requirements
2. Design
3. Development: including the development of units and unit testing
4. Integration and integration testing
5. System testing including debugging and acceptance testing
6. Deployment

In actual practice, the process may be more complex. There may be a partial release to allow external beta testing. Most successful software products undergo periodic revisions for bug fixing and the addition of features/capabilities. This may be termed the maintenance phase which includes regression testing that ensures that the existing functionality is not impaired by the revision. With the emergence of internet-based automated patch delivered, the software can evolve nearly continuously using an agile (Rawat 2017) or DevOps development approach.

Defects arise because of imperfect translation to a lower level (In terms of the levels mentioned above 0 to 1, 1 to 2, 2 to 3 or 3 to 4). Bugs introduced during 0-to-1 translation can arise not only because of the imperfect capture of the business case into requirements, but also shifts in the business case. Modern agile or DevOps approaches attempt to minimize their impact. The 0-to-1 and 1-to-2 translations are higher level and can give rise bugs that are harder to identify and expensive to fix. Available data suggests that most bugs occur at phases 2 and 3. Some defects may occur during integration because the module functionality or the interface requirements may not be clearly understood.

A bug that involves only a single line of code, or a few adjacent lines, are easier to find and debug. Some bugs can arise because of imperfect implementation of a computational task that involves parts of the code that are not adjacent. Such faults may be triggered only under some specific circumstances ("corner cases") and may not be easily detected.

A defect (also termed a bug) is defined as a deviation of a set of statements (not necessarily contiguous) from the higher level need that may be reported as a single *trouble report* and need to be fixed as a single corrective action. We will initially assume that when a defect is encountered during testing, the associated code is fixed ('debugged"). During the actual operation, failures may be recorded (to be addressed by the next patch or release), but are not fixed immediately. Thus during testing reliability growth occurs, during normal operation, the reliability does not change.

When a code segment containing a defect is exercised, an *error* may be generated. The error may cause incorrect data or an improper execution sequence. When the error propagates to the output, a *failure* is said to have occurred.

There are a few measures that are often used to describe software reliability. For any non-trivial real-life software, it is virtually impossible to assure that the software is bug free. It may possible to use formal methods for small and well defined pieces of code. That requires the software to be described with mathematical rigor and then mathematically proven to be correct. For most practical software development that is not feasible. The common software reliability measures used are:

Failure rate: the rate at which the failures are encountered. In the formal software reliability literature, it is represented by the failure intensity.

Defect density: the number of defects per thousand lines of code (without counting the comments).

Transaction reliability: it is the probability that a transaction, with takes a limited amount of execution time, is executed correctly without a failure.

These measures are used in the discussion below.

7.2 Detectability profile

The effectiveness of software testing techniques and the failure rate encountered during normal operation depend on the nature of the defects in the software. Some bugs can be very easy to find. For example, some bugs that violate the syntax requirements may be detected during compilation; they may not even arise when a continuously compiling development environment (such as Eclipse) is used. On the other hand, some bugs may affect computation only when a specific combination of rarely occurring inputs are encountered.

The detectability of a defect may be measured by how hard it is to test. The detectability of a bug must be defined in terms of random testing (Malaiya 1984).An input combination is random if each input value is chosen randomly and independently. In general, the bugs in a program will have different detectability values. The distribution of the detectability values in the program will determine how hard it will be to debug the code to arrive at a target defect density.

If there the total number of distinct combinations is N and j out of them detect the fault f_i then f_i has detectability j/N, since

$$\Pr\{\text{defect } f_i \text{ is detected by a randomly applied input combination}\} = N/j.$$

Thus we can write

$$\Pr(\text{n randomly applied input combinations do not detect the fault } f_i) = (1 - \frac{j}{N})^n \qquad (7.1)$$

Note that multiple faults may have the same detectability value. The discrete *detectability profile* is a vector $\{\pi_1, \pi_2, \pi_3, \ldots \pi_N\}$ which gives the number for faults with detectability values {1/N, 2/N, 3/N…, N/N}. The faults in the set π_1 are hardest to test, since they are each detectable only by a single specific test each. Let the total number of faults be $M = \sum_{i=1}^{N} \pi_i$. Faults in the set π_N are the easiest to test since any randomly chosen test will test for the faults. It can be shown the expected fraction of faults that would be covered by L random tests would be given by (Malaiya 1984).

$$C(L) = 1 - \sum_{k=1}^{N} (1 - \frac{k}{N})^L \frac{\pi_k}{M} \qquad (7.2)$$

If the tests applied are pseudorandom (when tests are chosen without replacement), it can be shown (Wagnor 1987) that the expected coverage is

$$C(L) = 1 - \sum_{k=1}^{N-L} \frac{\binom{N-L}{k}}{\binom{N}{k}} \frac{\pi_k}{M} \qquad (7.3)$$

As a software undergoes testing and removal of bugs, the bugs with a high detectability will be encountered earlier and removed. That will leave bugs with low detectability behind. Thus as testing progresses, the detectability profile of the software will continue to shift. The two expressions above show that once the test coverage is sufficiently high, only the defects with low values of $\frac{k}{N}$ would matter. Sometimes for convenience, the detectability profile may be defined continuously in terms of the test time instead of the number of tests.

At practically any point in time during the development, easy-to-find bugs would have been removed because of some prior testing. During the last phases of testing, the remaining faults are likely to be those which are triggered only by rarely occurring combinations of inputs. The detectability profile would thus shift and become increasing asymmetric. An example can be seen in the data published by Adams for a large IBM project (Adams 1984) as shown in Figure 7.1.

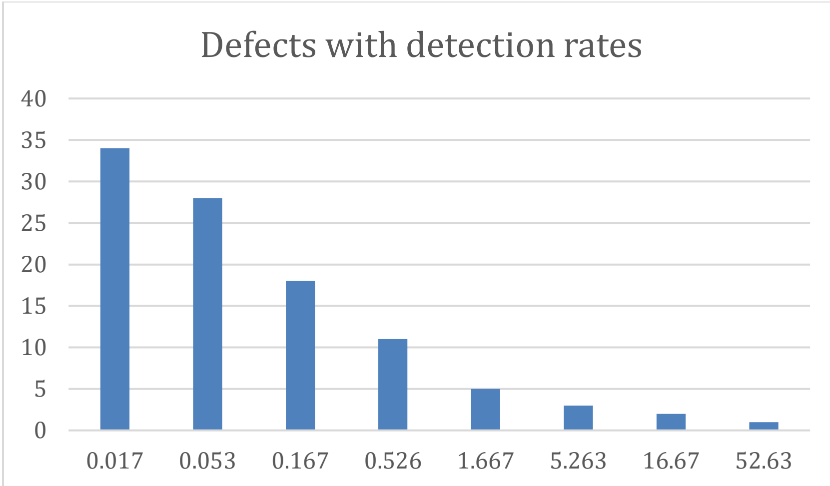

Figure 7.1: Testability Profile for Adam's Data

7.3 Software Partitioning

To ensure that the software is thoroughly exercised during testing, it is generally necessary to partition it to identify tests that would be effective for detecting the defects in different sections of the code. For testing purposes, a program may be partitioned either functionally or structurally.

- *Functional partitioning* refers to partitioning the input space of a program. For example, if a program performs five separate operations, its input space can be partitioned into five partitions. Functional partitioning only requires the knowledge of the functional description of the program, the actual implementation of the code is not required.

- *Structural partitioning* requires the knowledge of the structure at the code level. If a software is composed of ten modules (which may be classes, functions or other types of units), it can be thought of as having ten partitions.

A partition of either type can be subdivided into lower level partitions, which may themselves be further partitionable at a lower level if higher resolution is needed (Elbaum 2001). Dividing a partition into lower partition has the following consequences. Let us assume that a partition p_i can be subdivided into sub-partitions $\{p_{i1}, p_{i2} \ldots p_{in}\}$.

a. Random testing within the partition p_i will randomly select from $\{p_{i1}, p_{i2} \ldots p_{in}\}$. It is possible that some of them will get selected more often in a non-optimal manner.
b. Code within a sub-partition may be correlated relative to the probability of exercising some faults. Thus the effectiveness of testing may be diluted if the same sub-partition frequently gets chosen.
c. Sub-partitioning has a practical disadvantage when the operational profile is constructed, it will require estimating the operational probabilities of the associated sub-partitions.

For structural partitions, a statement or a branch (which are attributes often measured using the software test coverage tools like JCov or Emma) may be regarded as a low level partition. It should be noted that the execution of the statements within a straight-line *block* (containing statements that are always executed one after the other) is completely correlated. This if a single block is partitioned into multiple partitions, their execution and potential discovery of related defects, will be completely correlated.

An *operational profile*, as defined by John Musa (Musa 1993) involves the use of functional partitioning. An operation profile is a set of functional partitions along with a probability associated with each partition which gives the probability that an input is drawn from that partition. More elaborate operational profiles can be constructed by considering the system states using a state diagram, and the transition probabilities (Regnell 2000). The actual resolution used in the operational profile can vary significantly. (Guen 2003) reports the partitions to range from 4 to 143 partitions per thousand lines of code. Operation profile based testing is also sometimes termed statistical usage testing (Runeson 1995).

A simple example of incremental refinement is provided by Musa. A PABX unit has the initial operational profile given in column 1 of the table below. Since partition P1 has a large probability associated with it, it can be divided into sub-partitions as shown in column Refined.

Table 7.1: Sub-partitioning a partition

Initial		Refined	
Operation	Probability	Operation	Probability
P1: Voice call	0.74	Voice call, no pager, answer	0.18
		Voice call, no pager, no answer	0.17
		Voice call, pager, voice answer	0.17
		Voice call, pager, answer on page	0.12
		Voice call, pager, no answer on page	0.10
FAX call	0.15	FAX call	0.15
New number entry	0.10	New number entry	0.10
Data base audit	0.009	Data base audit	0.009
Add subscriber	0.0005	Add subscriber	0.0005
Delete subscriber	0.000499	Delete subscriber	0.000499
Failure recovery	0.000001	Failure recovery	0.000001

When a software project is still in the unit development/testing phase and has not been integrated, each module is can be considered both a functional partition and a structural partition. After integration has taken place, some of the code (and associated defects) may be shared by the functional partitions. If the shared code is small and has been thoroughly tested and debugged, then the bugs associated with each functional partitions may be essentially disjoint, since they are in disjoint structural partitions.

Test and operation profiles: During testing, the testing profile may or may not correspond to the operational profile. Using the operational profile is preferred in these two cases:

1. Acceptance testing, to assess the failure intensity that would be encountered during actual operation.
2. When the testing time available is limited. In this case, the impact of testing would be maximized by drawing more inputs from a partition that is encountered more during actual operation.

Operational profile based testing will not be effective in the following cases.

1. Once most bugs from the frequently executed partitions have been removed, partitions that are exercised less often will tend to have a higher defect density. Testing will become inefficient if these partitions are still exercised less frequently (Li 1994).
2. Some of the partitions may represent reused code, which has already undergone prior testing during the testing for prior releases. In this case, testing should focus on new code (Malaiya 2018, Malaiya 2011).
3. Some partitions may represent critical operations such that their failure may have a high impact.
4. If a partition corresponds to a larger code segment, it will require testing for a longer time to achieve the factor of reduction in defect density.

Example 1: This illustrates the case when there are five partitions P1 to P5. In the table below, the size of each partition (measured in KLOC), the initial defect density and the execution frequencies are given. We assume that the partitions are disjoint.

Table 7.2: Table for Example 1 with the five partitions

Partition/Attribute	**P1**	**P2**	**P3**	**P4**	**P5**
Size in KLOC	1	5	3	1	5
DefDensity	5	5	10	20	20
ExecutionFreq	0.1	0.3	0.2	0.1	0.3

Note that the Operational profile is {P1,P2,P3,P4,P5} = {0.1, 0.3, 0.2, 0.1, 0.3} Thus 30% of the time an input from P2 is chosen. The total size of the code 15 KLOC. We use these numbers for examples below.□

Musa, Iannino and Okumoto have defined the testing compression factor (TCF) as the ratio of the execution times needed to cover all the partitions (they use the term *state*) during testing and during normal operation. This ratio can be used as a measure of the effectiveness of testing compared with operational use (Huang10).

$$TCF = \frac{partitions\ exercised\ per\ unit\ time\ during\ testing}{partitions\ exercised\ per\ unit\ time\ during\ operation} \tag{7.4}$$

Musa found that the value for the TCF is between 8 and 21 for many of the programs considered. In a situation where the defects are uniformly distributed among the partitions, the defect finding rate, and hence the failure intensity during testing would be accelerated by a factor of TCF, compared with operational use.

7.4 The significance of model parameters

Here we consider the significance of the parameters of the common exponential model and see how they can be used for optimal test effort distribution for a software. We then examine the variation in the fault exposure ratio and show that the variation results in the Logarithmic Poisson model which has been shown to provide better predictability.

A number of software reliability growth models have been proposed by researchers. The simplest of them can be termed the exponential model, which can be thought to represent several models proposed earlier including Jelinski and Muranda (1971), Shooman (1971), Goel and Okumoto (1979) and Musa (1975-80) (Musa 1987, Yamada 2014). It has the advantage of offering straightforward interpretations of the model parameters in terms of measurable physical quantities.

The exponential model is based on the assumption that the defect finding rate at any point is proportional to the number of defects remaining at that time. Let us denote the number of yet undetected defects at time t to be N (t). The testing effort is measured in terms of time t. It can be the CPU time (as used by Musa), calendar time, or some other measure such as operational coverage (Kansal 2018) or the testing effort function (Peng 2014)

Initially, we assume that debugging is perfect, implying that a defect is always successfully removed when it is encountered.

Let N(t) be the number of defects that has remained undetected at time t. Let T_S be the execution time of a test case, and let k_s be the fraction of faults found during a single test case. Then using the above assumption, we can write

$$-\frac{dN(t)}{dt}T_s = k_s N(t) \qquad (7.5)$$

For the convenience of notation, let us introduce a term K called *fault exposure ratio*, such that

$$K = k_s \frac{T_L}{T_s} \qquad (7.6)$$

Where $T_L = S.Q.\frac{1}{r}$, where S is the source code size, Q is the number of object instructions per source instruction, and r is the instruction execution rate of the computer. Here we assume that the testing time t is measured in terms of the CPU execution time. The fault exposure ratio is a matric that measures the fault exposing capabilities of the testing strategy. Musa has argued that it should be independent of the software size S. The exponential model assumes that the fault exposure ratio remains constant throughout testing. We examine the assumption later.

Using equation(6), the equation (5) can be written as

$$-\frac{dN(t)}{dt} = \frac{K}{T_L}N(t) \quad (7.7)$$

Following Musa's notation, let us indicate the ratio $\frac{K}{T_L}$ by β_1 which will serve as one of the model parameters. Solving the differential equation gives us

$$N(t) = N(0)e^{-\beta_1 t} \quad (7.8)$$

The failure intensity, which describes the defect finding rate is given by

$$\lambda(t) = -\frac{dN(t)}{dt}$$

Thus,

$$\lambda(t) = \beta_0\beta_1 e^{-\beta_1 t} \quad (7.9)$$

Where the initial number of defects N(0) serves as the other parameter β_0. The mean value function $\mu(t)$expresses the cumulative expected number of defects found by time t and is thus given by

$$\mu(t) = \beta_0\left(1 - e^{-\beta_1 t}\right) \quad (7.10)$$

Equations(7.9) and (7.10) express the two forms of the exponential model. The two parameters can be readily interpreted (Malaiya and Denton 1997).

Parameter β_0: In equation (7.9), $\mu(t)$ approaches β_0 as t approaches infinity. Thus it is the total number of defects that would eventually be detected. If no defects were injected during debugging, it will be equal to N(0). In actual practice debugging is imperfect and some bugs are injected during debugging. Studies have shown that the number of such injected defects can be in the range of 5%, causing the final value of β_0 to be somewhat higher. In many projects, the defect density can be estimated using previous projects, and perhaps using some of the static metrics. If the defect density at the onset of testing is D(0), then

$$\beta_0 = D(0).S \quad (7.11)$$

Parameter β_1: Equation (7.6) provides an interpretation of the parameter β_1 which can be written as

$$\beta_1 = \frac{K}{T_L} = \frac{K}{S.Q.\frac{1}{r}} \quad (7.12)$$

The datasets collected by Musa suggest the values of K ranging between $1\text{x}10^{-7}$ to $10\text{x}10^{-7}$. When the data is collected using some other measure of the testing effort (such as person-hours etc.), the value of β_1 should be multiplied by an appropriate factor. Note that Q depends on the high level language and machine architecture, and r is machine dependent. Thus β_1 is proportional to the testing efficiency and inversely proportional to the software size S. Also notable is the fact that since the initial failure intensity is the product $\beta_0\beta_1$, it would be independent of the software size.

Example 2: For the five partitions P1 to P5, we assume that the fault exposure ratio is $5x10^{-7}$, the object instructions per source instruction is 2.5 and the object instruction execution rate for the processor is $7x10^{7}$per second. Then from equations (7.11) and (7.12), we can estimate the two parameters for the exponential model as follows.

Table 7.3: Table for Example 2 illustrating the computation of the parameters values

Partition/Attribute	**P1**	**P2**	**P3**	**P4**	**P5**
Size in KLOC	1	5	3	1	5
Def Density	5	5	10	20	20
Execution Freq	0.1	0.3	0.2	0.1	0.3
$\boldsymbol{\beta_0}$	5	25	30	20	100
$\boldsymbol{\beta_1}$	0.014	0.0028	0.004667	0.014	0.0028

Note that β_0 is simply the number of defects in each partition and β_1 depends inversely on software size. □

Test time required: The *When to stop testing problem* requires obtaining the answer to the question: how much testing is needed to bring the failure intensity (or equivalently, the defect density) down to the acceptable threshold. Normalizing equation (7.8) by dividing both sides by the software size S, we get

$$D(t) = D_0 e^{-\beta_1 t} \qquad (7.13)$$

Where $D_0 = D(0)$ is the initial defect density. If the target defect density is D_T then we can obtain the test time needed as

$$t_F = \frac{-\ln(\frac{D_T}{D_0})}{\beta_1} \qquad (7.14)$$

Equation (7.14) implies that more testing time is needed to reach the target if the initial defect density his higher. Also, since β_1 is inversely proportional to size, a larger module will need to be exercised for a longer time.

Optimal Testing: Testing using the operational profile is not always the most effective approach for debugging and thus achieving higher reliability. This is illustrated here in the next example. Here we assume that the overall failure rate is the weighted sum of the individual failure rates,

$$\boldsymbol{\lambda_{sys}} = \sum_{i=1}^{n} \boldsymbol{f_i \lambda_i} \qquad (7.15)$$

where $\boldsymbol{\lambda_i}$is the failure rate for partition *i* and $\boldsymbol{f_i}$is the fraction of time I is under execution.

Example 3: This example uses the parameter values as estimated in Example 2 above. We can set this up as an optimization problem (Malaiya 2018) with the overall failure rate as the objective function, and maximum allowable testing time (chosen to be 1500 units here) as a constraint. The problem is to allocate the 1500 units of testing to the five partitions. The optimal results can be obtained using the algebraic Lagrange Multiplier technique, which yields a closed form solution, or using an iterative algorithm (for example using the Microsoft Excel Solver).

The table below testing times allocated and gives the resulting system failure rate if operational profile testing is done, and when optimization is done.

Table 7.4: Table for Example 3. Operational Profile based vs. optimal testing

Partition/Attribute	**P1**	**P2**	**P3**	**P4**	**P5**	$\boldsymbol{\lambda_{sys}}$
$\boldsymbol{\beta_0}$	5	25	30	20	100	
$\boldsymbol{\beta_1}$	0.014	0.0028	0.004667	0.014	0.0028	
Op Profile Testing						
Testing time	150	450	300	150	450	
Failure rates	0.0086	0.0199	0.0345	0.0343	0.0794	**0.0410**
Optimal Testing						
Testing time	80.6023	220.5986	303.4676	179.6356	715.6959	
Failure rates	0.0226	0.0377	0.0340	0.0226	0.0377	**0.03397**

It can be seen that the optimal distribution of the test effort is significantly different from the operational profile based testing. Part of this is due to the fact the P1 and P2 have lower defect densities and P4 and P5 have higher defect densities. The code size in each partition also makes a significant impact.

Variation of the fault exposure ratio K: The above discussion uses the simplifying assumption that the fault exposure ratio is constant throughout testing. The assumption can be questioned because of these two facts.

a. As testing progresses, faults that are easy to find are found and removed, leaving faults with lower detectability. This will cause detection efficiency to decline.
b. In truly random testing, each test case applied is chosen regardless of the previous tests applied. In actual practice, the test scheme may remember the partitions that have been exercised in the past and focus on partitions not yet exercised. This will cause the testing efficiency to go up when fewer unexercised partitions remain and testing focuses on them.

Quantitative examination of the test data from several projects suggests that the fault exposure ratio does vary (Malaiya et al. 1993).It declines when the initial defect density is high and eventually starts rising when testing has progressed sufficiently far.

Malaiya et al have examined the variation of the fault exposure ratio K for 13 industrial reliability growth data sets. They observe that at higher fault densities K declines, whereas at lower fault densities K tends to rise as testing progresses. The change appears to occur in the vicinity of density about 2 per KLOC**,** although it is likely to vary depending on the testing approach used.

Considering the fact that the faults remaining undetected tend to be the ones harder to find, it can be shown (Malaiya et al. 1993) that the variation in K can be approximately modelled by

$$K(t) = \frac{K(0)}{1+at}, \quad where\ a > 0 \qquad (7.16)$$

where a is a parameter. The impact of testing becoming more and more focused on the partitions not yet covered, can be assessed by considering the extreme case when the location of the faults is known. Assuming that the application of each test has the same likelihood of revealing the presence of a new fault. In this situation, we have

$$\frac{dN}{dt} = -C$$

where C is a parameter. Based on equation (7.7), we can obtain

$$K = T_L C.\frac{1}{N} \qquad (7.17)$$

Considering both effects, we can hypothesize that the overall variation of K(t) can be represented using a combination of the two factors

$$K(t) = \frac{g}{N(t)(1+at)} \qquad (7.18)$$

where g is a parameter which can be evaluated using $K(0)N(0)$. Substituting this expression for K(t) in equation (7.7), and solving the differential equation, we obtain

$$N = N(0) - \frac{g}{T_L}\ln(1 + at) \qquad (7.19)$$

which confirms with the Logarithmic Poisson model, which has been found to provide better predictability than the exponential model (Malaiya et al. 1992). The correspondence provides an interpretation for the two parameters of the Logarithmic Poison model (for consistency with the literature and for convenience, we are designating the two parameters using the same notation, even though they are different from the Exponential model parameters). They are given by

$$\beta_0 = \frac{g}{T_L} = \frac{K(0)N(0)}{T_L} \qquad (7.20)$$

$$\beta_1 = a \qquad (7.21)$$

The table 7.5 below gives the overall values of K (in units of 10^{-7}) for nine data sets collected by Musa, arranged in the order of decreasing defect densities (in defects per KLOC) [Malaiya93]. The size is given in KLOC.

Table 7.5: values of K (units of 10^{-7}) for the nine data sets

Data Set	Size	D0	K
T1	21.7	6.89	1.87
T2	27.7	2.14	2.15
T3	23.4	1.79	4.11
T4	33.5	1.74	10.6
T5	2445	0.374	4.2
T6	5.7	14.08	3.97
T16	126.1	0.357	3.03
T19	61.9	0.675	4.54
T20	115.35	20.89	6.5

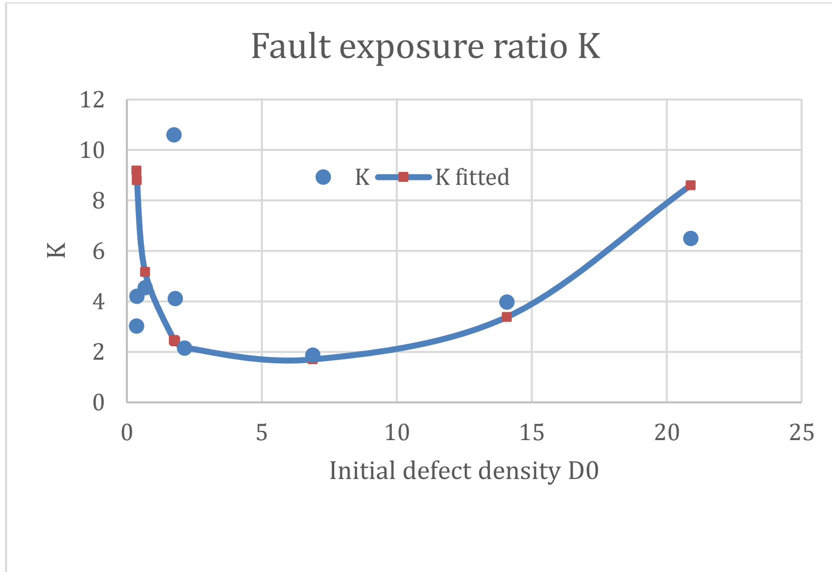

Figure 7.2 Variation of K with Defect Density

The table and the plot illustrate the observation that the fault exposure ration initially declines as faults get harder to find and then starts rising due to the use of directed testing in actual industrial testing.

Equation (7.18) gives K in terms .of testing time. It can be argued that it should be a function of the defect density, where it declines at higher defect densities and later starts rising at lower defect densities. An expression for K(D) can be obtained as (Li and Malaiya 1996) as given below.

$$K = \frac{\alpha_0}{D} e^{\alpha_1 D} \qquad (7.22)$$

Where $\square_0$ and $\square_1$ are applicable parameters. It can be shown that equation also applies for intial defect density $\square_0$ when the model is applied for multiple data sets. The plot in Figure 7.2 gives the actual data points as well as fitted data points.

7.5. Coverage based modeling

During testing, the strategy often changes. It will give rise to bursts in failure intensity. A new strategy may exercise some parts of the code that has not been exercised before. Thus the efficiency of the testing strategy can vary. The SRGMs assume that the testing strategy remains unchanged and uses time as a variable determining the reliability growth. It can be argued that test coverage is a better metric than time since it directly measures the number of test elements exercised.

The statements and branches are among the lowest levels of structural partitions. One measure of test effectiveness can be the coverage of the fraction of statements and branches. Two of the common coverage measure are (Horgan 1996).

- Statement (or block) coverage: the fraction of the total number of statements (blocks) that have been executed by the test data. A block is a segment of the code in which the instructions are always executed together.
- Branch (or decision) coverage: the fraction of the total number of branches that have been executed by the test data.

Weyuker (Weyuker 1993) has shown that the branch coverage subsumes the block coverage, i.e. if all the branches have been exercised, that guarantees that all the blocks would also have been exercised, but not vice versa.

We can obtain a model describing the relationship of the defect coverage with a test coverage metric by combining (i) a model relating defects found and the test time and (ii) a model relating the coverage achieved and the test time. For the first one, we assume that the reliability growth is given by the Logarithmic Poisson model. For the second one also we assume that the coverage growth is also modelled by a Logarithmic Poisson model. For convenience we use superscript 0 to indicate defects covered and superscripts 1 and 2 for the statement and branch coverage. (Malaiya et al. 2002). The test or defect coverage is given by

$$C^i(t) = \frac{1}{N^i}\beta_0^i \ln\left(1+\beta_1^i t\right), \qquad C^i(t) \leq 1 \qquad (7.23)$$

Note that the Logarithmic Poisson is applicable only until all the defects (or statements or branches) have been covered. Here□□ is the total number of enumerables (defects, statements or branches) of type I and □$_0^{□}$ and □$_1^{□}$ are the model parameters. If a single test takes □$_□$, seconds, then the time needed to apply n tests is □$_□$□. Then equation (7.23) can be written as,

$$C^i(n) = \frac{\beta_0^i}{N^i}\ln(1+\beta_1^i T_s n \qquad (7.24)$$

Note that for defect coverage, the parameter values are given by

$$\beta_0^0 = \frac{K^0(0)N^0(0)}{a^0 T_L} \qquad (7.25)$$

$$\beta_1^0 = a^0 \qquad (7.26)$$

For a compact notation, let us denote $\frac{\beta_0^i}{N^i}$ and $\beta_1^i T_S$ by b_0^i and b_1^i respectively, allowing us to write the above as

$$C^i(n) = b_0^i \ln\left(1+b_1^i n\right), \quad C^i(n) \leq 1 \qquad (7.27)$$

Here we can eliminate the number of vectors n in the expression for $C^0(N)$by using the expression for test coverage $C^i(n), i = 1,2$. We get

$$C^0 = b_0^0 \ln\left[1 + \frac{b_1^0}{b_1^i}\left(exp\left(\frac{C^i}{b_0^i}\right) - 1\right)\right], \ \ i = 1..2 \qquad (7.28)$$

Again for convenience, we can denote $b_0^0, \frac{b_1^0}{b_1^0}$, and $\frac{1}{b_0^i}$ by parameters a_0^i, a_1^i and a_2^i respectively to write

$$C^0 = a_0^i \ln\left[1 + a_1^i\left(exp\left(a_2^i C^i\right) - 1\right)\right], \ \ i = 1,2 \qquad (7.29)$$

Equation (7.29) gives an expression for defect coverage in terms of the test coverage. It can be seen that if test coverage is closer to 1, the above equation van be approximated by (Malaiya 1998, Malaiya 2002).

$$C^0 = a_0^i \ln(a_1^i) + \ a_0^i a_2^i C^i, \ i = 1,2, \ \ C^i > C^i{}_{knee} \qquad (7.30)$$

Where $C^i{}_{knee}$ is the test coverage level at which the linear trend begins.

The plot in Figure 7.1 below demonstrates the model given in equations (7.29) and (7.30). The data is from a European Space Agency project with 6100 lines of C code. It is seen that the growth in defects found is very linear after a branch coverage of about 25%. The testing was terminated at branch coverage of 71% with 20,000 tests applied because no additional defects were found after having applied 1240 tests. The branches not covered were part of the code that would get exercised only rarely (Pasquini 96). Had the testing continued, the model projects finding about 43 defects, provided all of the code is reachable.

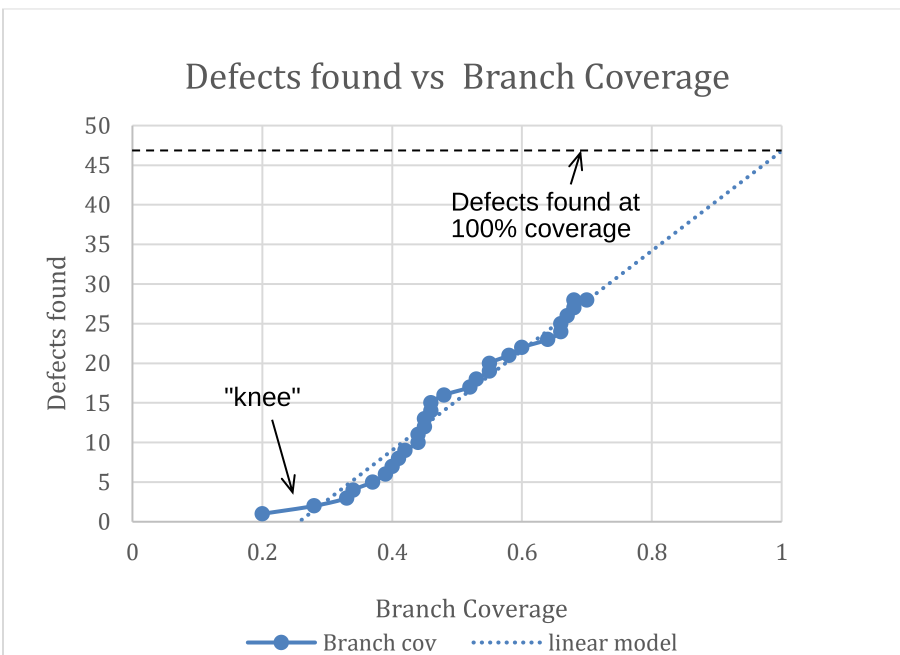

Figure 7.3: Coverage based modeling

The value of $C^i{}_{knee}$ is significant. The model in equations (7.29) and (7.30) suggest that very few defects are detected until the knee is encountered. After the knee, defects found rise linearly with the rise in coverage. In Pasquini's data, the knee occurs at approximately 25% coverage, as can be seen in the plot. It can be shown (Malaiya, 1998) that the knee occurs at this value

$$C^i{}_{knee} = 1 - ZD_0 \qquad (7.31)$$

Where the parameter Z depends on the attributes of the fault exposure ratio and the corresponding coverage item exposure ratio. The equation (7.31) suggests that the knee occurs very early when the defect density is high. After some testing, the faults that are easier to detect are removed and the remaining faults would only get detected at a higher coverage level. Thus the knee shifts to the higher coverage side when the initial defect density is low.

Defect density and failure rate: Pasquini et al (1996) have also collected data for the same project that allows computation of the failure rate (per input applied) when each fault found is removed. It is given in Figure 7.4 below.

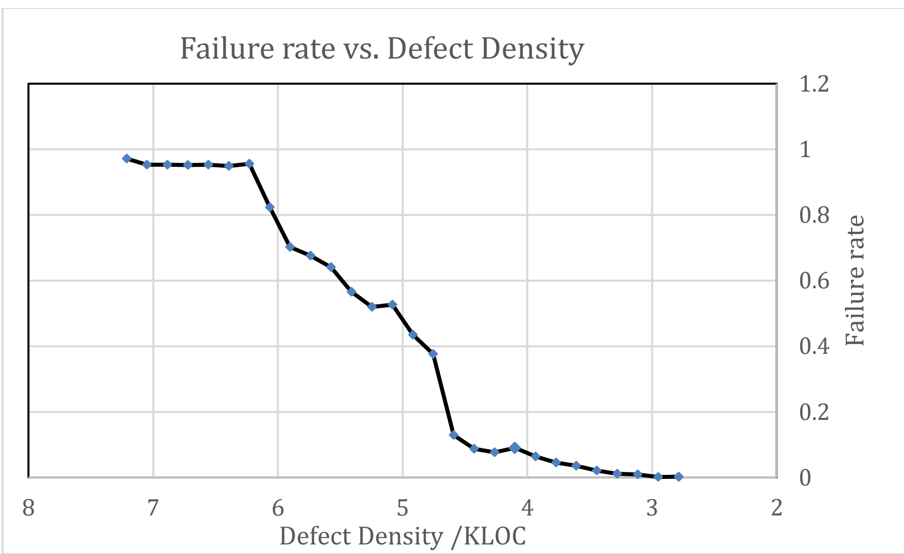

Figure 7.4: Failure rate variation with Defect Density

From Figure 7.4 it can be noted that code almost always fails until the first seven defects are removed. Removing the first six defects, one after the other, does not significantly decrease the failure rate. It is likely that these faults have a significant test correlation, i.e. they are triggered by many common tests, and thus removing one the first fault makes no significant difference in the failure rate. That should be expected for faults that have a very high testability. This demonstrates that the equation (7.15) for the system failure rate does not apply when the individual failure rates are very high, it would be a good approximation when detectability of the remaining faults is low.

At the lower defect density end, the failure rate decreases only gradually as seen at defect densities of about 3. That is because of the fact that these faults have very low testability and thus they are triggered by very few tests.

Coverage vs. Mutation testing: In mutation testing, defects (termed mutants) are automatically injected to evaluate the effectiveness of a test strategy. The number of mutants detected then can be taken as a measure of test effectiveness. Its key limitation is that the injected faults may not represent a realistic distribution of faults, especially at low defect densities. On the other hand when branch coverage is measured, covering a branch does not necessarily imply certain detection of all the associated defects. Approaches have been proposed that uses coverage to make mutation testing more efficient (Oliveira 2018), and conversely mutation has been used to make coverage more effective.

7.6. High reliability Software

Developing techniques for achieving high reliability in software has long been the aim of the researchers in the field. In real projects, the developers face deadlines for getting the software ready for release. The challenge thus is to achieve high reliability within a reasonably short time. The potential approaches can be classified as below.

1. **Low defect density by design**: use of tools and development discipline can reduce the initial defect density. The techniques include the following.

   High level development: Developing software at a high level and using automatic translation can reduce defect density. Assembly language code is more defect prone, and with modern optimizing compilers, the need to write time-critical code in assembly has been significantly reduced. Using well tested library code for common functions (such as graphics) reduces the need to write the corresponding functions in a programming language. Reusing an existing code component from an earlier version is likely to have a lower defect density, provided its functionality and interface are well defined.

   Integrated development environments (IDEs): IDEs allow better visualization, use of breakpoints for debugging, use of refactoring to automate code modifications. Continuous compilation virtually eliminates syntax errors.

   Compliance tools: tools such as those which automatically detect actual or potential memory leaks can reduce some troublesome run-time issues.

2. **Effective Testing**: Testing can significantly reduce the number of bugs and the failure rate. Increasing reliability using testing become increasing more expensive as the remaining bugs become harder and harder to find. Random testing becomes increasingly ineffective. For very high reliability, the major approaches that can help are(i) use coverage based testing that uses the structural information, and (ii) testing for rarely occurring input combinations.

3. **Redundant design**: There have been some investigations into the use of redundancy to implement fault tolerance. Experiments have found that there is a significant correlation among redundant implementations (N-version programming). Hatton (1997) has provided a simple analysis using the experimental data obtained by Knight and Leveson. Knight and Leveson found these probabilities for an input transaction:

    - A version failing: 0.0004
    - Any two modules failing at the same time (correlated failures): 2.5 x 10^{-6}
    - Three versions failing at the same time(correlated failures): 2.5 x 10^{-7}

    Were all the failures statistically independent, the probability failure of a Triple Modular Redundancy scheme with voting failing would be

    $$failure\ probability = \Pr\{all\ three\ fail\} + \Pr\{any\ two\ fail\} =$$

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

    In the presence of correlation the probability of failing would be higher:

    $$failure\ probability = 2.5 \times 10^{-7} + 3 \times 2.5 \times 10^{-6} = 7.75 \times 10^{-6}$$

    Thus while an improvement factor due to redundancy of 0.0004/$4.8 \times 10^{-7} = 833.3$ is not achievable, an improvement by 0.0004/7.75×10^{-6} = 51.6 is still achievable. Hatton argues that none of the testing approaches can reduce the defect density by that factor, and thus redundant design may be an alternative worth considering in some critical situations. It should be noted that Neufelder had found that on average testing reduces the defect density by only a factor of 5.1 (Neufelder 2007). Triple modular redundancy increases the cost by a factor of more than three (considering the overhead of voting mechanism) but may be considered for systems which need to be highly reliable.

Is ultra reliable software possible?: Butler and Finelli (2004) have argued that it would be hard to quantify the reliability of an ultra reliable software just by testing using the operational profile because the number of failures recorded within a reasonable time would not be statistically significant. They also argue that using probabilistic testing approaches, such as those assumed by the common SRGMs, will not be able to achieve a failure rate of 10^{-7} per hour or better.

Probabilistic testing methods lose effectiveness when they are used to further reduce the failure rates or the defect densities to very low values. Some form of directed testing would need to be used using structural test coverage or fine-grained functional partitioning to apply rarely used input combinations (Hecht 1993).

Is fault free software possible? There have been claims of fault free software having been achieved. For examples, it has been claimed that “A few projects - for example, the space-shuttle software - have achieved a level of 0 defects in500,000 lines of code using a system of format development methods, peer reviews, and statistical testing” (McConnel 2004). That is however misleading. The Space Shuttle software was a project that lasted 30 years from 1981 to 2011, and the last three version were found to have one error each [Fishman96]. It has been claimed that Formal Methods may yield defect free software. However, a study by Groote et al (2011) found that the use of formal methods was able to obtain defect densities as low as 0.5 per KLOC. By comparison, the original space shuttle software was found to have a defect density of 0.1 per KLOC, which is regarded as a verified standard (Binder 1997). Formal methods are infeasible for most projects because of the effort and the high degree of expertise required but may help in special cases.

7.7. Conclusions and future work

We have critically examined the impact of software testing by examining the mathematical modeling approaches using the testability of defects. These include both time-based as well as coverage based approaches. It is noted that as testing progresses, the remaining faults become harder and harder to find with random testing. For achieving low defect densities more efficiently structural testing and low-level partitioning need to be used. This may be especially important for finding security vulnerabilities which are security related defects, some of which can be very hard to find. The mean time a vulnerability remains undetected has recently been found to be 5.7 year (Ablon and Bogart 2017). Finding them sooner is a major challenge. Testing constitutes a major fraction of the development and maintenance costs. Methods for making testing more effective, for accurately modelling their behavior, as well as the tools for automating software testing need to be continually refined to approach the elusive aim of defect free software.

References

Ablon, Lillian, and Andy Bogart. *Zero Days, Thousands of Nights: The Life and Times of Zero-Day Vulnerabilities and Their Exploits*. Rand Corporation, 2017.

Adams, Edward N. "Optimizing preventive service of software products." *IBM Journal of Research and Development* 28, no. 1 (1984): 2-14.

B. Regnell, P. Runeson and C. Wohlin, "Towards Integration of Use Case Modelling and Usage-Based Testing", Journal of Software and Systems, Vol.50, No. 2, pp. 117-130, 2000.

Binder, Robert V. "Can a manufacturing quality model work for software?." *IEEE Software* 14, no. 5 (1997): 101-102.

Boehm, Barry W. "A spiral model of software development and enhancement." *Computer* 21, no. 5 (1988): 61-72.

Butler, Ricky W., and George B. Finelli. "The infeasibility of quantifying the reliability of life-critical real-time software." *IEEE Transactions on Software Engineering* 19, no. 1 (1993): 3-12.

de Oliveira, Andre Assis Lobo, Celso Goncalves Camilo-Junior, Eduardo Noronha de Andrade Freitas, and Auri Marcelo Rizzo Vincenzi. "FTMES: A Failed-Test-Oriented Mutant Execution Strategy for Mutation-Based Fault Localization." In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 155-165. IEEE, 2018.

Elbaum, Sebastian, and S. Narla. "A methodology for operational profile refinement." In *Reliability and Maintainability Symposium, 2001. Proceedings. Annual*, pp. 142-149. IEEE, 2001.

Fishman, C. “They Write the Right Stuff,” Fast Company, December 31, 1996. Retrieved Feb. 28, 2019, from https://www.fastcompany.com/28121/they-write-right-stuff
Groote, Jan Friso, Ammar Osaiweran, and Jacco H. Wesselius. "Analyzing the effects of formal methods on the development of industrial control software." 2011 27th IEEE International Conference on Software Maintenance (ICSM), Williamsburg, VI, 2011: 467–472.
H. Le Guen and T. Thelin, "Practical experiences with statistical usage testing," Eleventh Annual International Workshop on Software Technology and Engineering Practice, Amsterdam, 2003, pp. 87-93.
Hatton, Les. "N-version design versus one good version." *IEEE Software* 14, no. 6 (1997): 71-76.
Hecht, Herbert. "Rare conditions-an important cause of failures." In *Computer Assurance, 1993. COMPASS'93, Practical Paths to Assurance. Proceedings of the Eighth Annual Conference on*, pp. 81-85. IEEE, 1993.
Horgan, J., and A. Mathur. "Software testing and reliability." *The Handbook of Software Reliability Engineering* (1996): 531-565.
Huang, Chin-Yu, and Chu-Ti Lin. "Analysis of software reliability modeling considering testing compression factor and failure-to-fault relationship." *IEEE Transactions on Computers*59, no. 2 (2010).
Kansal, Yogita, Parmod Kumar Kapur, and Uday Kumar. "Coverage-based vulnerability discovery modeling to optimize disclosure time using multiattribute approach." *Quality and Reliability Engineering International* (2018).
Malaiya, Yashwant K., and Shoubao Yang. "The coverage problem for random testing." In *Proceedings of the 1984 international test conference on The three faces of test: design, characterization, production*, pp. 237-245. IEEE Computer Society, 1984.
Malaiya, Yashwant K., Nachimuthu Karunanithi, and Pradeep Verma. "Predictability of software-reliability models." *IEEE Transactions on Reliability* 41, no. 4 (1992): 539-546.
Malaiya, Yashwant K., Anneliese Von Mayrhauser, and Pradip K. Srimani. "An examination of fault exposure ratio." *IEEE Transactions on Software Engineering* 11 (1993): 1087-1094.
Malaiya, Yashwant K., and Jason Denton. "What do the software reliability growth model parameters represent?." In *Software Reliability Engineering, 1997. Proceedings., The Eighth International Symposium on*, pp. 124-135. IEEE, 1997.
Malaiya, Yashwant K., and Jason Denton. "Estimating the number of residual defects." In *hase*, p. 98. IEEE, 1998.
Malaiya, Y., Naixin Li, J. and Bieman, Rick Karcich,. *Software test coverage and reliability*. ," IEEE Trans. Reliability, pp. 420-426, Dec. 2002.
Malaiya, Yashwant K. "Reliability allocation" *Wiley Encyclopedia of Operations research and Management Science, John Wiley & Sons*, Jan. 14, 2011.
Malaiya, Yashwant K. "Software Reliability: A Quantitative Approach." In *System Reliability Management*, pp. 221-252. CRC Press, 2018.
McConnell, Steve. *Code complete*. Pearson Education, 2004.
Musa, John D., Anthony Iannino, and Kazuhira Okumoto. "Software Reliability: Measurement, Prediction, Application. 1987." (1987).
Musa, John D. "Operational profiles in software-reliability engineering." *IEEE software* 2 (1993): 14-32.
Li, Naixin and Y.K. Malaiya, "On Input Profile Selection for Software Testing," Proc. Int. Symp. Software Reliability Engineering, Nov. 1994, pp. 196-205.

Li, Naixin and Yashwant K. Malaiya. "Fault exposure ratio estimation and applications." In *Software Reliability Engineering, 1996. Proceedings., Seventh International Symposium on*, pp. 372-381. IEEE, 1996.

Neufelder, A. M. “Current Defect Density Statistics”, 2007, http://www.softrel.com/Current%20defect%20density%20statistics.pdf

P. Runeson and C. Wohlin, "Statistical Usage Testing for Software Reliability Control", Informatica, Vol. 19, No. 2, pp. 195-207, 1995.

Pasquini, Alberto, Adalberto Nobiato Crespo, and Paolo Matrella. "Sensitivity of reliability-growth models to operational profile errors vs. testing accuracy [software testing]." *IEEE Transactions on Reliability* 45, no. 4 (1996): 531-540.

Peng, Rui, Y. F. Li, W. J. Zhang, and Q. P. Hu. "Testing effort dependent software reliability model for imperfect debugging process considering both detection and correction." *Reliability Engineering & System Safety* 126 (2014): 37-43.

Rawat, Shubham, Nupur Goyal, and Mangey Ram. "Software reliability growth modeling for agile software development." *International Journal of Applied Mathematics and Computer Science* 27, no. 4 (2017): 777-783.

Weyuker, Elaine J. "More experience with data flow testing." *IEEE transactions on software engineering* 19, no. 9 (1993): 912-919.

Wagner, Kenneth D., Cary K. Chin, and Edward J. McCluskey. "Pseudorandom testing." *IEEE transactions on computers* 3 (1987): 332-343.

Yamada, Shigeru. *Software reliability modeling: fundamentals and applications*. Vol. 5. Tokyo: Springer, 2014.