the Availability Digest

www.availabilitydigest.com

Software Reliability Models

The Use of Defect Density as a Basis for the Prediction of Software Reliability

Dr. Terry Critchley
August, 2013

Hardware reliability can be defined as:

> ***Reliability.*** *The ability of an item to perform a required function under given conditions for a given time period.*

A similar definition is applied to software reliability - performing the function it was designed to do over a period of time. The inclusion of *time* in the definition implies that there will be a failure of some sort or other at some time or other. As a result, most math functions relating to software reliability have a time element in them in a similar way to hardware models, for example, an equation of the form R(t) = ...

A number of models and sub-models, often with esoteric math, attempt to describe the development and subsequent reliability of a piece of software.

Software Reliability Models

Two classes of models are mentioned here, although different authors break down the models in different ways:

- The Software Reliability Growth Model (SRGM) uses time between failures as its working entity.
- The Defect Density (DD) model, or defect density prediction model, uses fault count or failure intensity as its working entity

The latter model is the subject of this discussion.

The Field

The field of reliability is closely packed with models and distributions to predict software reliability characteristics. The models started in about 1972 with models by Shoomna, Tusa and Schneidwind. They now number over 200 types, including *Gompertz, Rayleigh, Gumbel, Gamma-Lomax, Goel-Okumoto, Hossian-Dahiya, Yamada (exponential and Rayleigh distribution forms), Weibull* and *LogNormal* etc.

To make things simpler, most prediction models share the following characteristics;

- assumptions and riders
- factors involved
- a mathematical function which relates the reliability with these factors
- calibration of the model with 'facts' on defects or time elements in existing software
- use of the model and subsequent verification in new situations

None of the models, however many variations and parameters it has, can be used in all situations since no model is complete, completely verified or universally representative. It is a matter of 'choose your weapon' when dueling with any particular software situation.

Defect Density Model

Defect density is a measure of the total of confirmed defects divided by the size of the software entity being measured. There are definitions of *faults*, *errors,* and so on, but in software modeling most people settle for the word *bug*, which is understood by all.

Programs are usually composed of modules which, in turn, are composed of instructions. Size is an important parameter and is often expressed as lines of code (LOC or KiloLOC - KLOC).

Defects can cover a mass of 'sins' or 'bugs,' the origins and breakdown of which are listed in the following table;

Defect Type	% of Faults
Logic	37
Interprocess Communication (IPC)	13
Hardware Interface	2
Functional Description	2
Data Handling	6
Data Definition	4
Computation	2
User Interface	2
Non-reproducible	2
Test Hardware	6
Software Interface	6
Requirements	2
Not a Defect	6
Module/interface Implementation	4
Module design	6

Table 1: Defect Distribution by Type

The Failure/Defect Density Models

The models have quite esoteric names and a selection of them follows;

- **Goel Okumoto Non-homogeneous Poisson Process model (NHPP)**
- **Musa Execution Time model**
- **Musa Okumoto Logarithmic Model**
- Goel Generalized NHPP model
- Shooman Exponential model
- Generalised Poisson model
- IBM Binomial and Poisson model

We will cover only the first three models, with two cases outlined for the Musa models.

The fault count models are concerned with the number of failures seen or faults detected in given testing intervals or 'units' - calendar time, execution time, number of test cases and so on. When faults are removed from the software, the number of failures per unit of measurement will decrease.

Two ways of measuring the reliability of software are:

- measure the trend of cumulative failure count $\mu(\tau)$ – SRGM model
- measure the trend of the number of failures per unit time $\lambda(\tau)$ – defect density

where τ is the time in execution of the programme. We are interested in the second case of *defect density* or *failure count*. Defects occur in software from the first coding stages through to the end of life (EOL) of a program and it is desirable to be able to make prediction about new, developing software based on data from existing, working software.

Common Terms Used in Most Models

N(t), M(t)	Total number of failures experienced by time t
P(N(t)=n)	The probability of there being n failures by time t
$\mu(t)$	Number of failures expected by time t, thus $\mu(t) = E[M(t)]$
$\lambda(t)$	Failure intensity, which is the derivative of the mean value function, that is $\lambda(t) = d\mu/dt$, the number of failures/unit time
t, τ	Elapsed (calendar) time and CPU time respectively
N, N_0	Initial number of faults present in the software prior to testing
z(t)	Per-fault hazard rate, which represents the probability that a fault that had not been activated so far will cause a failure instantaneously when activated.
Φ	A constant value for z(t) assumed by many models.

Goel-Okumoto NHPP Model

This model assumes that failures in a software system occur at random times due to faults. Based on the study of actual failure data across many systems, Goel and Okumoto proposed the following structure to the model;

$$\mu(t) = N(1 - e^{-\varphi t}) \text{ and } \lambda(t) = N\varphi e^{-\varphi t} = \varphi(N - \mu(t))$$

The model assumes the following;

- the expected number of failures observed by time t follows a Poisson distribution with mean value $\mu(t)$
- the number of software failures that occur in interval $(t+\Delta t)$ is proportional to the number of undetected faults, $(N - \mu(t))$
- there is no correlation between the numbers of failures detected in the failure intervals $(0,t_1)$, (t_1,t_2),.... (t_{n-1},t_n); that is they are independent
- the fault removal process when failures are detected is instantaneous and perfect
- the per-fault hazard rate is fixed at a constant given by φ

Musa Models

John Musa's model has two forms, the *basic* and the *logarithmic* (developed with Okumoto). The assumptions in Musa's basic model are**;**

- software faults are independent of each other and distributed with a constant rate of encounter (equidistant in time)
- a mixture of instructions and the execution time between failures is large compared with the instruction execution time
- the sets of input data for each run of the software are selected at random
- all failures are observed
- the fault causing the failure is corrected immediately; otherwise a recurrence of that failure is NOT counted.

There are different treatments of the Musa models.[1] One is covered in the following discussion in subsection A and the standard treatment of the two Musa models in B. and C, followed by a schematic graph in subsection D.

[1] http://incoming-proxy.ist.edu.gr/stfs_public/cs/msc/ReadingMaterial_MMSE-SEPE_oct2011/Software%20Quality/Software%20Reliability%20Model%20Study.pdf

A. Initial Software Reliability

The 'owner' of the paper from which this mathematics is quoted is the only identifier I have for the author of the paper referenced in the footnote.

The Rome Reliability Toolkit[2] suggests the following equation for the initial software reliability:

$$\lambda_0 = \frac{rKN_0}{I} \qquad Failures\ per\ CPU\ second$$

where;

r = processor speed in instructions/second
K = fault exposure ratio (a constant as far as we are concerned)
N_0 = Total no. of faults in the initial program (at $\tau = 0$)
I = No. of ***object*** instructions which is determined by LOC x expansion ratio

Object instructions are in essence ***machine*** instructions. The number of such instructions in say 'L' lines of code will depend on which language the instructions are in. Higher level languages like COBOL will generate lower level instructions in Assembler code, so a 5000-statement programme in COBOL will generate more machine (or object) instructions than an assembler program of the same size. This is illustrated in the table below (taken from the Rome Reliability Toolkit 1993).

Programming Language	Expansion Ratio
Assembler	1 (base unit)
Macro Assembler	1.5
C	2.5
COBOL	3
FORTRAN	3
Ada[3]	4.5

Table 2: Musa Model: Expansion Ratios

Because of the age of these figures (1993), it is hard to see how 4GLs would be handled in these models. Following this, the procedure develops as follows:

Let us call the ratio r/I = k, which is used in the following equations. The failure intensity in Musa's Basic Execution model is given by;

$$\lambda(t) = Kf(N - \mu(t))$$

This looks the same as the *Goel-Okumoto* equation except the factor **Kf** replaces the **φ** of that model.

B. Musa's Basic Execution Model

This uses the assumption that the decrease in failure intensity (rate) is *constant* and states that the failure intensity is a function of the average number of failures experienced at any point in time, that is, the failure probability. This is expressed in the equations following.

Assumption for this model: $$\lambda(\mu) = \lambda_0[1 - \frac{\mu}{\nu_0}]$$

That is, the failure intensity is a function of the average number of failures experienced at any time given point in time.

[2] Obtainable from www.quanterion.com
[3] .. which begat B language which begat C. Small world.

$$\mu(\tau) = \lambda_0[1 - \frac{\mu}{\nu_0}] \quad and \quad \lambda(\tau) = \lambda_0 \, exp[-\frac{\lambda_0 \tau}{\nu_0}]$$

Equation 1: Musa Basic Model Failure Rate/Intensity

where $\lambda(\mu)$ = failure intensity
λ_0 = initial failure intensity at the start of execution
μ = average numbers of failures/unit time at a given point in time
ν_o = total number of failures over an infinite time

C. Musa-Okumoto Logarithmic Model

A base assumption for this model is expressed as a decreasing rate with time, that is, expressed as follows;

$$\lambda(\mu) = \lambda_0 \, e^{-\theta\mu}$$

$$\mu(\tau) = \nu_0[1 - e^{\frac{\lambda_0}{\nu_0}\tau}] \qquad Basic\ Model$$

$$\mu(\tau) = \frac{1}{\theta} \, e^{(\lambda_0 \theta \tau + 1)} \qquad Logarithmic\ Model$$

Equation 2: Musa-Okumoto Logarithmic Model Failure Rate

θ is a parameter representing a *non-linear* drop in failure intensity in this model.

D. Graphical Representation

A graphical representation of these models showing some of the parameters used in the equations above can be found in Figure 2.

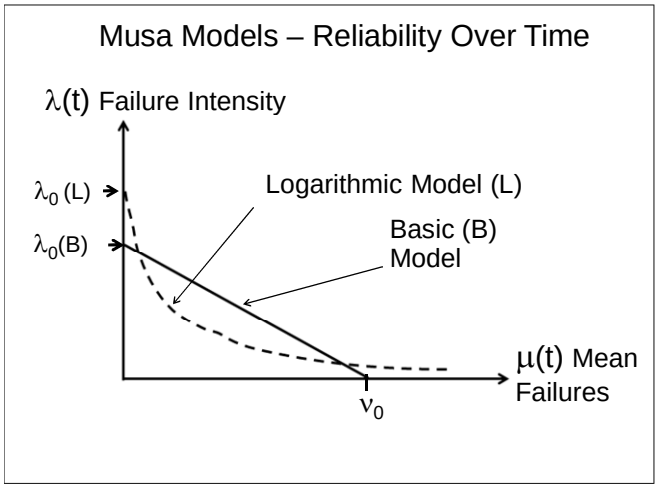

Figure 2: Basic and Logarithmic Models - Failure Intensity vs. Time

The relationship between the failure intensity (rate) and the reliability of the software is given by the expression (similar to its exponential hardware cousin);

$$R(\tau) = e^{-\lambda\tau}$$

There are a number of other models in this class which are listed below;

- Goel Generalized NHPP model
- Shooman Exponential model
- Generalized Poisson model
- IBM Binomial and Poisson model

These can be found in numerous papers and presentations on the Internet.

Times Between Failure Models

This is another group of models, each tailored for the situation whose data fits the model. They are not the purpose of this paper.

- Jelinski-Moranda
- Schick and Wolverton Model
- Goel and Okumoto Imperfect debugging model
- Littlewood-Verrall Bayesian model

So What?

Michael Lyu has this to say about models and their usefulness. The chapter numbers mentioned below refer to the following book, edited by him;

- *http://www.cse.cuhk.edu.hk/~lyu/book/reliability/*

'For the *'so what*' question, the answer is two-fold. First, if there is no software reliability measurement, there is no evidence of a quantifiable software development process which can be engineered and improved. The application of software reliability models indicates the maturity of an underlying software development process from an industry viewpoint. Secondly, even though we cannot guarantee the accuracy of an SRGM for a particular project in advance, we can still use the SRGM prediction results to confirm the readiness (or otherwise) of the software project in terms of its reliability. Chapter 9, for example, takes this point further for industry practice. Besides, whether an SRGM is applicable can also be tested by a trend analysis, which is handled in Chapter 10'.

Software vendors are interested in this kind of 'crystal-ball' gazing to assess the quality of new outgoing software and resource planning for the maintenance stage of the software's life, that is, between ship and EOL.

Software Development Defect Plot

The number of defects in the life cycle of software development has been found to conform to a numerical distribution represented by the Rayleigh equation, a subset of the ubiquitous Weibull distribution well known in reliability circles (Rayleigh = Weibull for (m=2) for those in the know).

The linear curve represents the observed data and the curve is a fit of the Rayleigh distribution to these results.

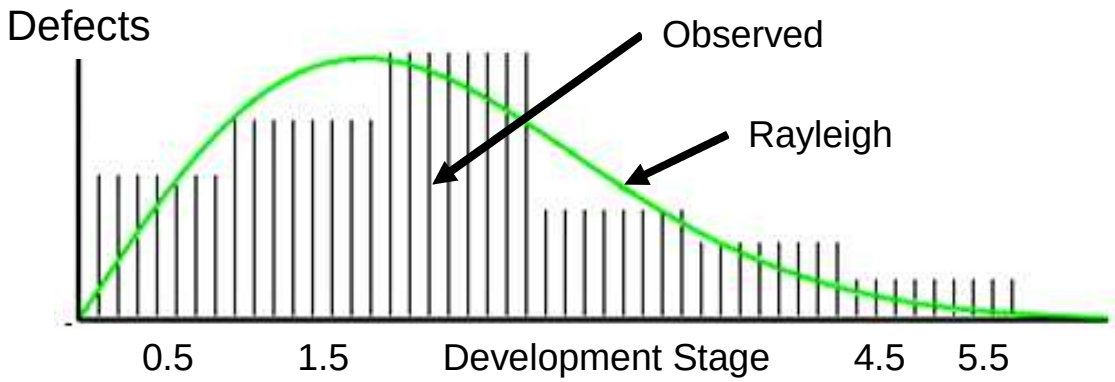

Figure 3 Rayleigh Curve Showing Defects vs Development Stage

A Rayleigh curve and time 'slots' for the various stages of software development are shown below;

Software Stage	Timescale (relative)
HLD - High Level Design	0.5
LLD - Low Level Design	1.5
Coding - Implementation	2.5
UT - Unit Testing	3.5
IT - Integration Testing	4.5
ST - System testing	5.5

Table 3 Timescale of Development Software Cycle

The Rayleigh curve model shows the number of defects detected in the six stages of development outlined in column 1 in Table 2 above. Other model distributions cover the testing/deployment phase, the exponential and S-curve models.

NOTE: The Rayleigh *pdf* and *cdf* functions are shown below;

$$f(x,\sigma) = \frac{x}{\sigma^2} e^{-\frac{x^2}{2\sigma^2}} \quad \text{and} \quad F(x) = (1 - e^{-\frac{x^2}{2\sigma^2}})$$

Reliability Prediction Software

There are many different companies that sell reliability prediction software packages and there are many different reliability prediction methodologies, handbooks and guidelines (MIL-HDBK-217F, G, H, 217Plus, Telcordia SR332, etc). As an example, see the URL below for a description of the SREPT (Software Reliability Estimation and Prediction Tool): *http://people.ee.duke.edu/~kst/srept.html*

The author would like to thank Dr. Bill Highleyman (editor of www.availabilitydigest.com) and Michael Lyu for their reviews and inputs.

Dr. Terry Critchley is a retired IT consultant living near Manchester in the UK. Terry studied Physics at Manchester University (using some of Rutherford's original equipment!) and gained an Honours degree in Physics, followed five years later with a PhD in Nuclear Physics. He then joined IBM as a Systems Engineer and spent 24 years there in a variety of accounts and specialisations, latterly joining Oracle for 3 years. Terry joined his last company, Sun Microsystems in 1996 and left there in 2001. In 1993 he co-authored a book on Open Systems for the British Computer Society and is currently writing a book on high availability, tentatively entitled "Availability Management." He is also 'mining' tons of his old material for his next book.