Software reliability modelling and decision making: An approach*

Rajesh Goyal
Institute for Systems Studies and Analyses, Defence Research and Development Organisation, Metcalfe House Complex, Delhi 110 054, India
Received 17 June 1996; accepted 20 September 1996

The aim of this paper is to briefly discuss, the available models for evaluating the reliability of software, along with their assumptions/limitations. This assessment shall finally give a step-by-step procedure for fitting a model to software failure data in order to obtain estimates of performance measures like undetected errors, mean time between failures and software reliability. This approach will help in deciding, whether the system is ready for release, or how much more testing still needs to be done?

Software is essentially an instrument for transforming a discrete set of inputs into a discrete set of outputs. It comprises of a set of coded statements whose functions may be to evaluate an expression and store the result in a temporary or permanent locations, decide which statement to execute next, or to perform input/output operation.

Software development is a complex discipline requiring coordinated activities of multifarious inputs, phase wise analysis, evaluation and processing with inherent quality assurance as a continuous and parallel process. Quality must be built into the software design right from the conceptual stage and engineered through the life cycle, by using effective management and technical process. Since, to a large extent, software is produced by humans, the finished product is often imperfect in the sense that a discrepancy exists between what the software can do versus what the user or the computing environment wants it to do in an optimised manner. The computing environment refers to the physical machine, operating system, compiler and translator utilities. These discrepancies which may be due to affects of computing environments or introduced as a result of human interaction/activities are what we call software faults. Basically, software faults can be attributed to poor definition of user requirements, incomplete comprehension of these requirements by developers, lack of indepth technical knowledge of user activities, ignorance of the rules of computing environment, and to poor communication between the user and the programmer or poor interpretation of the software requirements when design document is undertaken. Even if we know that software contains faults, we generally do not known their exact identity.

Currently, there are two approaches available for indicating the existence of software faults, namely program proving (validation) and program testing (verification). Program proving is formal and mathematical while program testing is practical and heuristic. It is the symbolic or physical execution of a set of test cases with the intent of exposing embedded faults in the program. Like program proving, program testing remains an imperfect tool for assuring program correctness. A given testing strategy may be good for exposing certain kind of faults but not for all possible kinds of faults in the program. In practice neither proving nor testing can guarantee complete confidence in the correctness of a program. Each has its advantages and limitations. They are, in fact, complementary method for decreasing the likelihood of program failure.

Due to imperfection of these approaches in assuring a correct program, a measure is needed which reflects the degree of program correctness and which can be used in planning and controlling additional resources needed for enriching software quality. One such measure that is commonly used is software reliability. A commonly used approach for measuring software reliability is by an analytical model whose parameters are generally estimated from available data on software failures (number of faults). Reliability and other rele-

---

vant measures are then computed from the fitted model.

**Software Reliability**

Two approaches for measuring the software reliability can be adopted. The first measure could be binary in nature so that an imperfect program would have zero reliability while a perfect program would have reliability value one. Second, the software reliability should be defined as the relative frequency of times that the program works as intended by the user. This approach is similar to that taken in testing where a percentage of the successful cases is used as a measure of program quality. According to this, software reliability is a probabilistic measure and can be defined as the probability that software faults do not cause a failure during a specified exposure period in a specified user environment.

More precisely software reliability can be like this, let $F$ be a class of faults, defined arbitrarily, and $T$ be a measure of relative time, the unit of which is dictated by the application at hand. Then the reliability of a software package with respect to the class of faults $F$ and with respect to the measure $T$, is the probability that no fault of the class occurs during the execution of the program for a specified period of relevant time.

Software reliability is also a useful measure for giving the user confidence about software correctness. Planning and controlling of testing resources via software reliability measure can be done by balancing the additional cost of testing in terms of time and money, and the corresponding improvement in the software reliability. As more and more faults are exposed by the validation and verification process, the additional cost of exposing the remaining faults generally rises very quickly. Thus, there is a point beyond which continuation of testing to further improve the quality of the software can be justified only if such improvement is cost effective.

Current approach for measuring the software reliability is basically parallel to that used for hardware reliability assessment with appropriate modifications to account for the inherent differences between software and hardware. For example, hardware shows mixture of decreasing and increasing failure rate. The decreasing failure rate is due to the fact that, as test or use time on hardware system accumulates, failures, most likely due to design errors are encountered and their causes are removed. The increasing failure rate is primarily due to wearout or aging effect. A software may become obsolete because of changes in user’s requirement or because of new technological changes in hardware, but once it is modified to reflect these changes, we no longer talk of the same software but an enhanced or a modified version. Like hardware, software exhibits a decreasing failure rate as the time on the system accumulates and faults, say, due to design and coding, are corrected. However, there is no increase in fault pattern as in the case of hardware. The software may die its natural death due to obsolescence and not due to increased failures.

**Reliability Models**

A number of analytical models have been proposed for software reliability measurement. The approaches are based mainly on the failure data and number of faults in the software and can be classified accordingly.

In reliability theory, the distribution of time plays an important role. The probability that the software will fail in the interval from 0 to $t$ is given by

$$F(t) = \int_0^t f(x) \, dx$$  \hspace{1cm} (1)

where $f(x)$ is the probability density function for failure time, and the probability that the software operates without an error up to time $t$, called the reliability of the software (survival function), is given by:

$$R(t) = 1 - F(t)$$  \hspace{1cm} (2)

Thus, instantaneous error rate usually called as hazard rate or conditional failure rate is given by:

$$z(t) = \frac{F'(t)}{R(t)} = \frac{f(t)}{1 - F(t)}$$  \hspace{1cm} (3)

This gives the error rate as a function of the error time distribution. Now, since $F'(t) = -R'(t)$ it follows that

$$z(t) = \frac{-R'(t)}{R(t)}$$  \hspace{1cm} (4)

and by solving this differential equation, reliability may be given as

$$R(t) = \exp \left[ \int_0^t z(x) \, dx \right]$$  \hspace{1cm} (5)

If the software error detection process is beyond the initial debugging development phase, a constant rate is often assumed, and this leads to
the exponential model.

\[ R(t) = \exp[-at] \]  \hspace{1cm} (6)

and the error time distribution is given by

\[ f(t) = a \exp[-at] \hspace{1cm} t > 0, \hspace{0.5cm} a > 0 \]  \hspace{1cm} (7)

where \(1/a\) is referred to as the mean time between failures or errors. Software mean time between failure (MTBF) is, thus given by:

\[ \text{MTBF} = \int_{0}^{\infty} R(x) \, dx \]  \hspace{1cm} (8)

If a constant error rate can not be assumed, a function (or hazard function) used for the error rate is given by:

\[ z(t) = abt^{(b-1)} \hspace{1cm} t > 0, \hspace{0.5cm} a, b > 0 \]  \hspace{1cm} (9)

If \(b < 1\) the error rate decreases as \(t\) increases, for \(b = 0\) the error rate is constant and if \(b > 1\) the error rate increases. This gives a two parameter Weibull distribution

\[ f(t) = abt^{(b-1)} \exp[-at^b] \hspace{1cm} t > 0 \]  \hspace{1cm} (10)

A constant error rate for software is never found, therefore, a decreasing error rate model like Weibull \((b < 1)\) are generally appropriate.

Times Between Failures Models

In this class of models the process under study is the time between failures. The most common approach is to assume that the time between failure, say, time between \((i-1)\)th and \(i\)th failures, follows a distribution whose parameters depend on the number of faults remaining in the program during this interval. Estimates of the parameters are obtained from the observed values of the times between failures and estimates of the software reliability, mean time between failures are then obtained from the fitted model.

In these models it is expected that the successive failure time intervals will get longer as faults are removed from the software. For a given set of observed values, this may not be exactly so due to the fact that failure times are random variables and observed values are subjected to statistical fluctuations. Following assumption are made in developing these models such as independent time between failures, equal probability of the exposure of each fault, embedded faults are independent of each other, faults are removed after each occurrence and no new faults introduced during correction.

Jelinski-Moranda Model

The model\(^1\) assumes failure rate is a Poisson process and proportional to the number of remaining defects. The hazard rate, constant between failure is given by:

\[ Z(t_i) = k[N-(i-1)] \]  \hspace{1cm} (11)

for the time between the \(i-1\) and \(i\)th failure, where,

\(N\) total number of initial errors, \(k\) constant of proportionality which keeps the area under probability curve equal to unity, \(t_i\) time between \(i\)th and \((i-1)\)th error discovered and \(i\) total number of errors found during testing time of an interval.

The reliability function is given by

\[ R(t_i) = \exp[k(N - n) t_i] \]  \hspace{1cm} (12)

where, \(n\) is the number of faults removed by \((i-1)\)th interval, and

\[ \text{MTTF} = \frac{1}{k(N - n)} \]  \hspace{1cm} (13)

\(N\) and \(k\) can be estimated by maximum likelihood estimate with the following equations:

\[ \sum_{i=1}^{n} \frac{1}{\bar{N} - (i-1)} = \frac{n}{\bar{N} - S/T} \]  \hspace{1cm} (14)

and,

\[ k = \frac{1}{\bar{NT} - S} \]  \hspace{1cm} (15)

where,

\[ S = \sum_{i=1}^{n} (i-1)X_i; \hspace{1cm} T = \sum_{i=1}^{n} X_i \]

and \(X_1, X_2, \ldots, X_n\) are elapsed time between successive failure.

Schick-Wolverton Model

This is the best known model\(^1\), and is based on the following hazard function or error rate:

\[ Z(t_i) = k[N-(i-1)] t_i \]  \hspace{1cm} (16)

where,

\(N\) total number of initial errors, \(k\) constant of proportionality which keeps the area under probability curve equal to unity, \(t_i\) time between \(i\)th and \((i-1)\)th error discovered and \(i\) total number of errors found by testing of an interval.

The hazard rate describes well known Rayleigh distribution for which survival function, i.e., reliability given by:

\[ R(t_i) = \exp[-k(N-(i-1)) \frac{t_i^2}{2}] \]  \hspace{1cm} (17)
Equations to estimate the total time \( T \) required to find all remaining errors and standard deviation \( \sigma \) for this estimate are:

\[
\hat{T} = \frac{1}{k} \sum_{i=1}^{N-n} 1/i \quad \ldots (18)
\]

\[
\hat{\sigma} = \frac{1}{k} \left[ \sum_{i=1}^{N-n} 1/i^2 \right]^{1/2} \quad \ldots (19)
\]

The maximum likelihood estimate for \( N \) and \( k \) are:

\[
\hat{N} = \left[ \frac{2n}{k} + \sum_{i=1}^{n} (i-1) t_i^2 \right] \frac{1}{\sum t_i^2} \quad \ldots (20)
\]

and

\[
k = \left[ \sum_{i=1}^{n} \frac{2}{N-(i-1)} \right] \frac{1}{\sum t_i^2} \quad \ldots (21)
\]

Lipow Modified Jelinski-Moranda Model

This model makes one modification to the Jelinski-Moranda model. It assumes that errors may not be corrected immediately when discovered. This modification was made for what usually happens during a real time development project. The parameters of this model are:

\[
MTBF(t) = \frac{1}{k[N-EC(p)]]} \quad \ldots (22)
\]

\[
R(t) = \exp(-k[N-EC(p)]) \quad \ldots (23)
\]

where,

\( N \) total number of inherent errors in the software. This number is assumed to be fixed and finite, \( k \) constant of proportionality which keeps the area under probability curve equal to unity and \( EC(p) \) is number of errors corrected up through the \( p \)th testing period.

Littlewood-Verrall Bayesian Model

It uses a different approach to the development of a model for times between failures. In this it is considered that software reliability should not be specified in terms of number of errors in the program, instead the time between failures are considered and are assumed to follow an exponential distribution. The parameter of this distribution, i.e., failure rate is treated as a random variable following gamma distribution, viz.

\[
f(k_i \mid m, g(i)) = [g(i)]^m k_i^{m-1} \exp(-g(i)k_i)/\Gamma m \quad \ldots (25)
\]

In this, \( g(i) \) describes the quality of the programmer and the difficulty of the programming task. It is claimed that the failure phenomena in different environments can be explained by this model by taking different forms for \( g(i) \).

The other well known models of this category is Goel and Okumoto Imperfect Debugging Model.

Fault Count Models

This class of models is concerned with modelling of number of failures seen or faults detected in given testing interval. As faults are removed from the system, it is expected that the observed number of failures per unit time will decrease. Time in this can be calender time, CPU time, the number of test cases run or some other relevant measure. In this method time intervals may be fixed in advance and number of errors in each interval is treated as a random variable.

Poisson distribution has been found to be an excellent model in many fields of application where interest is in the number of occurrences. The basic approach of these models is that of Poisson distribution whose parameter is a random variable and can take different forms for different models. Key assumptions for these models are that the testing intervals are independent of each other, testing during intervals is reasonably homogeneous and the number of faults detected during non-overlapping interval are independent of each other.

Musa Execution Time Model

This model is based on the fact that reliability estimation in the time domain can be based only on actual execution time as opposed to elapsed or calender time. The key assumptions are that the failure intervals are independent and Poisson distributed. The execution time between failure is exponentially distributed, and the failure rate is proportional to number of remaining errors. The hazard function for this model is given by

\[
Z(t) = kf(N-n_c) \quad \ldots (26)
\]

where, \( t \) is the execution time utilized in executing the program up to the present, \( f \) is the linear execution frequency (average instruction execution rate divided by the number of instructions in the program), \( k \) is a proportionality constant, which is a fault exposure ratio that relates fault exposure frequency to the linear execution frequency, \( N \) is the initial number of errors and \( n_c \) is the number of errors corrected during \((0,t)\).
Shooman Exponential Model

This model is similar to the Schick-Wolverton Model. For this model, the hazard function is of the following form

\[ Z(t) = k(N/I - n_c(x)) \]  \hspace{1cm} (27)

where \( t \) is the operating time of the system measured from its initial activation, \( I \) is the total number of instruction in the program, \( x \) is the debugging time since the start of the system integration, \( n_c(x) \) is the total number of faults corrected during \( x \), normalized with respect to \( I \), and \( k \) is a proportionality constant.

Goel-Okumoto Nonhomogeneous Poisson Process Model

In this model, it is assumed that a software system is subjected to failures at random times caused by faults present in the system. Let \( N(t) \) be the cumulative number of failures observed by time \( t \), then \( N(t) \) can be modelled as a nonhomogeneous Poisson process, i.e., as a Poisson process with a time dependent failure rate. Based upon the study of actual failure data from software systems the following form of the model is proposed

\[ P[N(t) = y] = \frac{[m(t)]^y}{y!} \exp[-m(t)], \quad y = 0, 1, 2, \ldots \]  \hspace{1cm} (28)

where, \( m(t) = a[1 - \exp(-bt)] \),

here \( m(t) \) is the expected number of failures observed by time \( t \), 'a' is the expected number of failures to be observed eventually and 'b' is the fault detection rate per fault.

Fault Seeding Models

The basic approach in this class of models is to 'seed' a known number of faults in a program which is assumed to have an unknown number of faults. The program is tested and the observed number of seeded and indigenous faults are counted. From these, an estimate of the fault content of the program prior to seeding is obtained and used to assess software reliability and other measures.

Assumptions made for the development of such models are that the seeded faults are randomly distributed in the program and indigenous and seeded faults have equal probability of being detected. Popular models of this class are Mill Seeding Model and Nelson Model.

Development Phases and Model Selection

In this section, various phases of software development process and the applicability of software reliability models in respective phases has been discussed. Major phases of software development activities are:

(a) Requirement Analysis Phase
(b) Structured Analysis and Design Phase
(c) Coding, Unit Testing and Debugging Phase
(d) Integration Phase

Requirement and design phases—During requirement analysis phase and design phase errors may be detected by conducting walk-through or by some other formal procedures. Existing software reliability models are not applicable during these phases.

Coding, unit testing and debugging phase—The typical environment during module coding and unit testing phase is such that the test cases generated from the module input domain do not form a representative sample of the operational usage distribution. Further, times between exposures of modular fault are not random since the test strategy employed may not be random testing since the test cases are executed in a deterministic fashion.

The time dependent models, especially the time between failure models are not applicable during these phases as the assumptions of these models are not applicable. Only fault seeding models can be applied with their assumptions.

Integration phase—In this phase modules are integrated into sub-system or whole system and test cases are generated to verify the correctness of the integrated system. Test cases for this purpose can be generated randomly from an input domain or can be generated deterministically using a reliable test strategy, which is more effective. The exposed faults are corrected and there is a high possibility that removal of these faults may introduce new faults. Under such testing conditions, fault seeding models are theoretically applicable. If deterministic testing, i.e., boundary values are used, time between failure models are not applicable as the assumption of independence of inter-failure time is not fulfilled. Fault count models can also be applied, if sets of test cases are independent of each other. If random testing is used according to an assumed input profile distribution, i.e., generating input through simulation, then most of the existing models are applicable, and time between failures and failure count models are most effective.

Other phases in which software reliability and
other performance measures are obtained, are ‘Acceptance Testing Phase’ and ‘Operational Phase’. In these phases models used for integration testing phase are applicable.

Model Selection Procedure

Due to different underlying assumptions of software reliability models, there is a significant distinction among them. In this section a method has been discussed for choosing a specific model for measuring reliability and other performance measures for decision making. The selection of model is to be based on development method consideration. A step-by-step procedure for fitting a model to software failure data is given below. The various steps and decision making process are shown in Fig. 1.

Step 1 Collection and study of failure data—The model discussed in different development phases requires the failure data. In most of these models data should be in the form of either times between failures or as failure counts. The first step in developing a model is to study such data for distribution of errors count. A plot of such data as function of time (calendar time, CPU/ex-ecution time or number of test cases executed) is made. The objective of this plot is to try to determine the appropriate variable to be used in the model. Data normalization technique can be used due to change in size of the software during testing.

Step 2 Choose a reliability model—At this step choose an appropriate model based upon the assumptions and testing procedure. Data and plots from step-1 are used for this purpose.

Step 3 Obtain estimates of model parameters—Use common statistical method like method of moment and maximum likelihood method to estimate the parameters of the model depending upon the nature of the data available.

Step 4 Obtain the fitted model—By substituting the estimated values of the parameters in the chosen model, the fitted model is obtained. Thus a fitted model based upon the failure data and chosen model form is available.

Step 5 Perform goodness of fit test—to check the model Chi-square or Kolmogorov-Simirnov tests are performed. If the model fits, i.e., the statistic $(\text{Chi-Square}/D_{\text{max}})$ is not significant move to step-6. If it is significant, i.e., model does not fit, collect additional data or take another appropriate model and go to Step 3.

Step 6 Obtain estimate of performance measures—at this step, compute various quantitative measure to assess the performance of software system. Confidence intervals of these estimates are also obtained at this step.

Step 7 Decision making—The final objective of developing a model is to use it for making decisions, e.g., whether to release the software system or continue testing. Such decisions can be taken on the basis of results obtained at Step 6.

Conclusion

In this paper various software reliability models and their assumptions are discussed by defining software reliability and other performance measures. Model selection at various stages of software development life-cycle has also been touched upon. Finally, a method for fitting a developed model from failure data is discussed. The methodology and models reviewed in this paper can be used in large software development activities.

Acknowledgements

The author is indebted to Dr N K Jaiswal, Director ISSA, and Col B Chandrashekar, Project Director for their encouragement and support. Thanks are also due to Col P S Cheema and Dr Arun Kumar for their valuable suggestions.
References