Realistic Assumptions For Software Reliability Models

David Zeitler
Smith's Industries Aerospace & Defense Systems, Inc.
4141 Eastern Avenue, S.E., Grand Rapids, MI 49518-8727
PHONE: (616)241-8168 / EMAIL: zeitler@si.com

Abstract

A definition of reliability appropriate for systems containing significant software that includes trustworthiness and is independent of requirements will be stated and argued for. The systems addressed will encompass the entire product development process as well as both product and its documentation. Cost incurred as a result of faults will be shown to be appropriate as a performance measurement for this definition. This and more realistic assumptions will then be shown to lead to the use of auto-regressive integrated moving average (ARIMA) mathematical models for the modeling of reliability growth.

Key Words: Reliability definition, trustworthiness, growth models, software system reliability, model assumptions.

1 GROUNDWORK

Most authors find the assumptions made in modeling reliability questionable for application to the real world of software development. To more firmly anchor theoretical development to the real world, I will start by reviewing the reasons for reliability models in general and specifically how considerations for the reliability of software based systems dictate that we both change the measurement of reliability considered and the assumptions upon which the model must be based.

1.1 Reliability Enhancement Framework

A good framework from which to start considering software reliability is presented in RADC-TR-87-171 [6]. It does a good job of identifying the tasks involved in statistical reliability improvement and relating them to the DOD-STD-2167A terminology. Too often, reliability discussion begins at the mathematical models for reliability growth and ignore the larger picture of the full reliability program. The RADC document is also the first attempt I've seen to pull together a specification for implementation of a software reliability program. I have added relationships to the framework to show explicitly the presence of the three major types of reliability models.

As shown in figure 1. There are four tasks in this framework with associated outputs: Goal Specification, Prediction, Estimation and Assessment. Goal Specification is a nearly independent process that provides targets for the development process based on an application level model. Predictions take a second model based on development metrics and provide feedback into the design process. Estimations take test measurements and a third model to provide feedback into the test and burn in processes. Finally the assessments take operational data and feed it back into all the models to help tune their predictive and estimation capabilities. The three models identified roughly correspond to component count predictions, stress analysis predictions and growth estimation models currently in use in hardware reliability work.

So we have three separate models, each with a different goal. The first two models are predictive in nature, while the third is estimating the reliability of an existing product. Most current work is aimed at the development of this third type of reliability model. Since the strongest inferences will be possible at this later stage of a program, this is also the most productive area to be addressing.

1.2 Software, Hardware & Systems Reliability

It is often stated that software reliability is very different from hardware reliability, but what exactly is this difference? Usually the statement is made in relation to the idea that software doesn't wear out, but

Reprinted from *Proc. Int'l Symp. Software Reliability Eng.*, IEEE CS Press, Los Alamitos, Calif., 1991, pp. 67–74.

the differences are more fundamental. Software is a set of instructions, like a blueprint or schematic, *it* cannot fail. It can however be wrong in a multitude of ways.

Analogous component levels for software and hardware are illustrated in table 1. The basis of this analogy is comparative equality in the level of functionality provided by the components on each side of the table. For example, both resistors and processor instructions provide a fundamental functionality for their respective disciplines. Note that on either side of this table the levels shown are hazy. The table's primary purpose is to illustrate relationships between software terminology and roughly equivalent hardware structures for comparison purposes.

Hardware reliability analysis almost exclusively addresses the first two levels of the hierarchy (discrete components), with more recent work attempting to extend the models to the higher levels. Since usage of parallel paths through the system is not uniform, this type of analysis quickly becomes extremely complex and highly dependent on the end users input distribution.

From this relationship between hardware and software we can see first that there are equivalent ways of thinking of the two seemingly very different disciplines. Software is most comparable to the blueprints, schematics and production process specifications of hardware. At best, if we compare software programs with hardware components, we then must think of development methods, standards, etc. as the blueprints or production specifications. Thus each program developed is analogous to an individual hardware unit from production. We can now see from this analogy that statistical analysis of software must be carried out across many software units to achieve the same effects as applying statistical analysis of hardware components to many parts.

Carrying this analogy just a bit farther, to parallel hardware reliability we should be looking at the reliability of individual CPU instructions much as hardware has determined the reliability of individual components. This is quite different from the current work in software reliability, which jumps in at the higher levels, treating the software as a black box, without attempting to address the low levels which hardware has built upon. Hardware has discrete components at these levels that can be analyzed independent of the particular use of the component. This examination of fundamental software components independent of their use will yield only that the components do not fail. Therefore, something different must be going on when we discuss reliability for software systems.

In hardware we are looking at reliability hazard functions as the probability that an individual component will fail. In software, we are looking at the probability that a given component will be used in an inappropriate manner. This implies that we are looking not at the probability of failure of a concrete replicate of a design, as in hardware, but rather at the probability that the designer incorrectly uses a component in the design. For example, choosing the correct packaging of a resistance component for the target environmental conditions and reliability requirements, but using the wrong resistance value in the circuit for the full range of the circuits intended function (i.e., incorrect functional design as opposed to incorrect environmental design).

Reliability analysis now recommends reduced part counts and greater operational margins to improve reliability. In the context of the software system reliability improvement process I'm suggesting, reliability might also be suggesting that the use of complex components, which have a high probability of misuse, be minimized. This would reduce the probability of design errors.

An example of the kind of design criteria that might come out of this type of analysis would be 'Reduce use of complex non-programmable components'. Or perhaps, the development of software modules unique to the system should be minimized, since they probably would not have the maturity and/or level of testing of common software. As a confirmation that we're on the right track, note that this agrees with common practice for quality software development today. So we can see from this analogy that software reliability *is* significantly different from hardware reliability.

1.3 Separation of Software from System Reliability

We are at the point where the functionality of systems are primarily resident in the software. Special purpose or single purpose hardware is nearly a thing of the past. Since we have shown above that software reliability is more related to functionality than to physical components, addressing software reliability is equivalent to addressing system reliability. It is also equivalent to addressing the reliability of the software/system development process, since the reliability of the end product is as much a result of the process that developed it as of the product itself.

This then suggests that a system level approach to the overall reliability problem will be more effective than attempting to isolate either hardware or software.

Figure 1: Software Reliability Framework

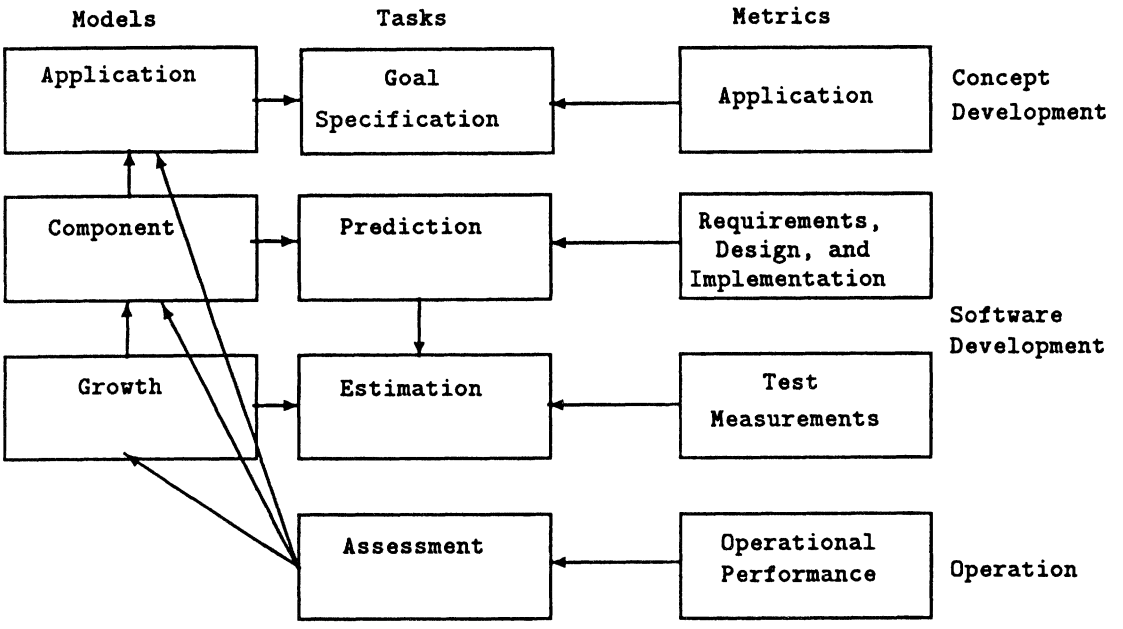

Table 1: analogous Hardware/Software Structures

Hardware	Software
Primitive Components Resistors,Capacitors, etc.	Primitive Instructions move, shift, add, etc.
Integrated Chips CPU, UART, MMU, etc.	Units Procedures, Modules, etc.
Sub-circuits RS422, ARINC, etc.	CPC's Packages, Objects, etc.
Boards Serial I/O, Memory, etc.	TLCSC's Exec, I/O, etc.
Sub-systems (HWCI's) Display, Keyboard, etc.	Programs (CSCI's) Operational Flight Program
Systems Navigation System, Weapon System, Fuel Savings System, etc.	

What's more, if we solve the reliability problem as discussed here, the solution will be equally applicable to areas that hardware reliability has traditionally considered out of scope. A solution to software reliability then should cover not only the software aspects of the reliability question, but the wider system aspects simultaneously.

In [7] Fabio Schreiber concludes that the time is ripe for the unification of several research areas into a unified investigation of system reliability. I am taking his suggestion a step further here in including not only the system performance, but also in effect the performance of the development process as well. This is not only desirable from the relationship of reliability to functionality, but is necessitated by the inadequacy of requirement specification techniques for complex systems.

1.4 Definition of Software Reliability

Most authors that address the definition of systems reliability, address it in terms of adherence to requirements. With the present state of the art in requirements specification for complex systems, this hardly seems reasonable. We cannot yet determine if a system adheres to requirements when these requirements are incomplete and/or ambiguous as they invariably are.

According to Webster's New Collegiate Dictionary, two possible meanings of reliability seem to apply. The first is consistency in response, in which case software (or any other deterministic process) cannot be unreliable. Another definition has reliability meaning 'to be dependable, or trustworthy'. The first definition doesn't seem to fit at all. We can be sure that software does not have perfect reliability. So it appears reliability means to perform with little probability of unexpected behavior, (i.e., we can depend on or trust it). This is beyond just repeatable behavior and does seem to get at the issue here. So if the system always does what is expected of it, it is perfectly reliable. Thus we see that trustworthiness is a major component of reliability.

Reliability is then more closely associated with meeting user expectations, regardless of stated requirements. (Note: this is not in addition to meeting requirements. Any given requirement, regardless of source, may not meet user expectations and meeting them will still be considered a failure!) Reliability is then related to perceived failures as opposed to what may be considered actual failures. It is well known that, although we're making progress, the precise specification of a complex system is not at this time possible. The result is that there is room for user interpretation, which can lead to perceived failures. These failures are as expensive (or more expensive) than 'actual' failures. In either case, the user is not satisfied with the operation of the system and it is considered unreliable.

So I am defining reliability as that which minimizes the users perceived failures.

1.5 Measurement of Reliability

Perceived failures can be measured in terms of cost due to user complaints, so I'm suggesting we use cost as a measure of reliability. These costs consist of the lost productivity or production due to down time, management costs for handling the return of faulty product, costs associated with negotiation of variances against requirements, etc. The cost does not however stop at the customer. We also will need to consider the cost to the developer. This cost is in terms of the lost prestige (and therefore presumably potential market share in the future) as well as the immediate cost to analyze and fix the product returned. This can be modeled as a constant times the cumulative cost incurred by the customer.

Measurement of perceived failures at the user is too late to help us estimate the growth curve of the system reliability during development. It will provide adequate feedback into future reliability analyses, but a measure of perceived failures during the development process is needed. This can be obtained through the early application of operational testing using actual users. Early testing can be focused on the user interface, with gradual incorporation of functionality as the integration of the system progresses.

In [5] Ragnar Huslende lays out an extension of standard reliability concepts for degradable systems or any partially operable system. As Huslende states, any system can be viewed as degradable. Using cost as a measure of the system performance we can see that zero cost is equivalent to the no-fault state of Huslende's H(t). So I will be looking at cost as a measure of system performance.

$$c(t) = c_u(t) + c_p \cdot c_u(t) + c_f(t)$$

where:

c_u	=	cost to the user with respect to time
c_p	=	impact of customers costs on developers
c_f	=	developers cost to fix the product

This gives us a value that can be measured tangibly (although not necessarily precisely) regardless of the

specifications, development process, or other variables (controllable or not). Using the cost as our performance measure also gives us a handle on the varying degree of severity that failures tend to cause.

Measuring reliability performance as cost also allows us to focus on those problems that are important to us and the customer. We all know of little problems that are annoyances in software based systems, but are certainly not worth spending massive effort to correct. Likewise, we all would consider any effort expended to eliminate risk of life to be spent well. With cost as our measure of reliability, little problems that are soon worked around have a negligible impact on reliability, while loss of life has a correspondingly large impact.

These costs can be measured through existing or augmentation of existing accounting systems for tracking product that has been fielded. For systems under development, we will need to make use of the user interaction with early prototypes to measure relation to expectations. This also implies that we need to make the actual users an integral part of the standard testing process in order to improve reliability of systems.

2 MODELING

Essentially all proposed reliability models are growth models, an exception being the linear univariate predictive model(s) attempted by SAIC for RADC-TR-87-171. The intent of the reliability growth modeling process is to be able to predict, from actual failure data collected during early development or testing, both the expected end product reliability and the magnitude of effort necessary to achieve the target reliability, and hence the date when a system release can be achieved. Reliability growth models have been applied to several programs for these purposes. These applications have shown considerable promise. Many programs now also use an intuitive form of this by monitoring a problem reporting system during the late stages of a program with an eye toward determining when the software is ready for release.

Martin Trachtenberg in [8] provides a general formulation of software reliability with the relationship of his general model to major existing models. This work is a good foundation upon which to base further modeling work. In particular, my measure of reliability performance as cost can be easily incorporated into the model. Modification of the general model to incorporate dynamics is somewhat more complex however.

In Trachtenberg's model, failure rate is a function of software errors encountered. He considers f=f(e) and e=e(x) where f is the number of failures, e is the number of errors encountered and x is the number of executed instructions. To determine a failure rate, he differentiates f(e(x(t)) with respect to time to obtain a form in terms of current number of failures per encountered error (s), apparent error density or number of encountered errors per executed instruction (d), and software workload in terms of instructions per time unit (w). Thus arriving at failure rate $\lambda(t)$ as below:

$$\lambda(t) = \frac{df}{dt} = \frac{df}{de} \cdot \frac{de}{dx} \cdot \frac{dx}{dt} = s \cdot d \cdot w$$

This model is intuitively attractive and does provide a general structure from which other models can be viewed. In the following paragraphs I will consider each basic assumption and show the modification necessary for a realistic model. When possible, I will be using the notation from Trachtenburg's paper.

2.1 A Semi-structural Approach

As Schreiber stated in [7], cost based modeling can be extremely complex. This due to the complexities of incorporating variable failure impact into structural equations for the process. In addition, adding the development process into the reliability equations adds human systems into the equations. I am suggesting that an intermediate approach be taken. From a few basic assumptions about the relationships expected, I will be proposing a high level empirical model capable of capturing the necessary characteristics of the process, thereby avoiding the complexities of attempting to specify the micro level structural model under which the system actually operates.

2.2 Assumptions

Recent work (one specifically is Ehrlich, et al [3]) has shown that in some applications the current growth models proposed for software reliability provide reasonable estimates of test effort needed. The work discussed was specifically designed to meet the given assumptions of the models. Necessary characteristics for using the current available models vary, but generally include independence in the interarrival times of failures, uniform magnitude of impact of failures (making failure rate a reasonable measure of reliability), and uniform system load during test and/or operation (i.e., random test execution). None of these assumptions hold for many real-time systems and avionic systems in particular.

A clear indication that we have autocorrelation even in systems with uniformity in testing can be seen in the

telephony system test data analyzed by Ehrlich et.al. The plots show a good match between the homogeneous Poisson model being fitted and the actual data, but there is visual evidence of positive autocorrelation in the data sets. This suggests the presence of auto regressive factors needed in the model even for a process which fits the assumptions well.

In systems that cannot be easily tested in a random fashion, these auto regressive factors are likely to be sufficiently significant to trigger an early release of software or an over intensification of test effort by excursions from the too simplistic model that are merely based on random variation. At the very least, they will reinforce the positive feedback reactive effect seen where increased failures during test causes increased test effort (lagged by management response time constants of course).

2.2.1 Time base for real time systems

Real time systems have a more-or-less uniform instructions (or cycles) per unit time pattern. This makes measurement of instructions executed per unit time a poor measure for system load. Instead, we need to be measuring the number of instructions outside the operating system per time unit. This number will increase proportionately with system load.

No changes in the equations are necessary here, since we're just modifying the interpretation. This modification transfers us from the time domain to the computation domain, as discussed in [1]. More specifically, we are working in the application computation domain.

2.2.2 Cost of Failures

Including a non-constant cost for failures will remove the uniform failure impact assumptions made in the standard models, this puts our emphasis on cost per unit time. The cost of a failure is both a function of the fault that causes it and of the situation in which the failure occurs.

$c(f) = c(f,b) \cdot p_b(t)$

where:

$c_f(a,b)$ = cost of failure a occurring in situation b

$p_b(t)$ = probability of situation b at time t

Since situations (or test cases) cannot occur randomly in the types of systems we're looking at, but occur in the context of an operational profile, cost due to the occurrence of any particular failure is a time based function related to the operational profiles. These operational profiles will produce a 'fine grain' correlation structure in the occurrences of failures.

Our early measurements for growth estimation are taken during the integration of the system. This integration process limits the nature of the profiles to functions currently integrated and operational. Thus we will also see a 'coarse grain' correlation structure to the occurrences of failures.

For our cost based reliability measurement we replace failure rate, with cost rate ($\gamma(t)$) based measures. These cost based measures will be more directly usable by management and give planners a better handle on appropriate test effort feedback into the development process. We can also see that this process level feedback into the product will impact failure occurrences.

$$\begin{aligned} \gamma(t) &= \frac{dc}{dt} \\ &= \frac{dc}{df} \cdot \frac{df}{de} \cdot \frac{de}{dx} \cdot \frac{dx}{dt} \\ &= c \cdot s \cdot d \cdot w \end{aligned}$$

2.2.3 Failure inter-arrival time

Software can be viewed as a transfer function. Its input must be taken as the combination of the values present at its inputs and the state of its memory at the beginning of execution. Output is the combination of values presented to its outputs and its internal state upon completion of execution. In many software systems, this transfer function is applied to essentially independent inputs and the system state is reset before each execution. In real-time avionic software, and most other real-time software systems, the inputs are sequences of highly correlated values applied to a system that retains its internal state from the previous input set.

Clearly then, the assumption of random distribution of inputs is not feasible here. Therefore a primary assumption of our growth models cannot be applied. We must then prepare our models for the likely autocorrelation of failure occurrence. This will imply some form of time-series model for the occurrence of software failures.

It also can be argued that a piece of code with many faults being found is likely to have more faults remaining in it, since the same factors that produced the faulty code affect the remainder of the code as well. Combined with this is the operational profile nature necessary for most real time systems testing. The operational profile ensures that the conditions that caused a failure must persist for some period of time, thus increasing

the probability of recurrence or related failure detection.

So we see that software systems failure occurrences are not independent, and in fact, will exhibit correlation structure that is dependent on the correlation structure of the inputs coming from an operational profile as well as on the system itself. This is not easily modelable in general.

2.2.4 Fault correction process

When a fault is corrected, the same process as that which created the fault is used to fix it. The fix then has a probability of introducing new faults (or not correcting the original fault) that is clearly not zero. If we look at the magnitude of the change in changed or added instructions, schematic symbols, drawing symbols, etc., (M_e) and the probability of introducing faults as the proportion of faults (e_0) to instructions at the beginning of the program (I), $p_0 = \left(\frac{e_0}{I}\right)$ and let $p_e(t) =$ the probability of fixing a fault at time t, we can view the remaining faults in the code at time t as

$e(t) = e(t-1) + p_e(t) \cdot (M_e \cdot p_0 - 1)$

rather than

$e(t) = e(t-1) - p_e(t)$

Using B as the linear lag operator, we have

$$(1 - B)e(t) = p_e(t) \cdot (M_e \cdot p_0 - 1)$$

The right hand side of the previous equation represents the random process of finding and fixing faults in the system. This random process is the combination of the above processes of failure inter-arrivals, fault recurrence and fault removal. Since together these represent a dynamic process driven by random noise, we can let the rhs $= \theta_0 + \theta(B)\epsilon_t$, $\epsilon(t)$ is a white noise driver for the process. We can see then that the fault content of the system is modelable as a standard autoregressive integrated moving average (ARIMA) [2] time-series model.

Note that this relationship implies that the fault content of the system is not strictly decreasing and in fact may exceed the initial fault content.

2.2.5 Recurrent faults & fault removal

Faults are not removed immediately in the real world. Often minor faults are determined to be not worth the effort to fix at all. Thus a realistic model of software maintenance must be a cost prioritized queue. The result is a clear dependence of the number of failures per fault on the cost of removing the fault when compared with the cost of encountering the fault and the expected number of occurrences of the fault if its not fixed.

For our context, the queueing process can be expressed as a finite difference equation using the sum of the expected cost of not fixing the faults as the state variable. State changes occur whenever there is a fault with an expected cost for not fixing greater than the cost of repairing. The fault fixed will be the one with the highest return (not necessarily the first or last).

Thus we can see that known faults may never reach zero (as one would expect) and that the expected number of recurrences of a failure due to a given fault is a function of both the cost to fix it and its expected impact on lifetime cost of the system. Since the cost to fix is known to be increasing with time during development and the expected impact is dependent on the operational use of the system, we will have the number of failures occurring for each fault being dynamic also.

3 Summary

I have shown here that we can expect the occurrences of failures and the magnitude of their impact to vary dynamically in time and that they are all interdependent. Thus our model is clearly dynamic and an a priori determination of its order (let alone the exact structure) will be difficult. This is identically the case for work in social, economic and other human systems. Not really a surprise since we can see from the necessary definition of reliability in terms of customer expectations that products containing significant software components are deeply embedded in human systems. The approach taken to modeling and forecasting in human based systems is fitting autoregressive integrated moving average models to the real world data. I'm suggesting here that we need to do the same for estimating reliability growth.

In the September '90 QualityTIME section of IEEE Software [4], Richard Hamlet has an excellent discussion of quality and reliability. He clearly delineates for us the distinction between failures and faults. I suspect that Mr. Hamlet will disagree with my definition of reliability, as will Parnas. If however, we separate trustworthiness from reliability as suggested, we end up with the conclusion that software systems are perfectly reliable. If this is the case, we then should drop this thrashing over software reliability, and attack the problem of software trustworthiness. Webster however includes trustworthiness as a part of reliability. I will

then stand by my arguments. Reliability of software systems is a measure of how much trust the user can put in the software performing as expected.

More realistic assumptions are necessary for application of reliability growth modeling to software systems. The assumptions are:

- Failures do not have equal impact.
- Failure rates have dynamic structure.
- The fault removal process is a cost prioritized queue depending on the ratio of expected cost to cost to fix.
- Fault content has a dynamic structure and is not strictly decreasing.

I have shown that, because of my definition and these less restrictive assumptions on our model, software reliability must be modeled as an ARIMA time series. The intent here is not to replace existing models. They have already shown their value in some real world situations. My intent is to provide an avenue for further application of reliability analysis in software system development applications where the restrictive assumptions of current models will make them of little use.

3.1 Further research.

Considerable empirical work is needed to validate my claims. Costs need to be collected to perform this empirical work. Initial efforts could however make use of estimates from existing information and/or apply simulation techniques to initially validate the approach by showing that ARIMA models could produce the kinds of behavior known to be present in the software system development process. Eventually, it is hoped that information gained from this more empirical approach will lead to greater understanding of the process and structural models.

4 Acknowledgements

Thanks to Gerry Vossler and Derek Hatley for their review of this work and to Smiths Industries for their support. I would also like to thank the reviewers who pointed me toward related work in systems reliability which was quite helpful.

References

[1] Beaudry, M. Danielle, *Performance-Related Reliability Measures for Computing Systems*, IEEE Transaction on Computers, Vol c-27. No. 6, June 1978, pp 540-547.

[2] Box, George E. P. & Gwilym M. Jenkins, *Time Series Forecasting and Control*, Holden-Day 1976.

[3] Ehrlich, Willa K., S. Keith Lee, and Rex H. Molisani. *Applying Reliability Measurement: A Case Study*, IEEE Software, March 1990.

[4] Hamlet, Richard *New answers to old questions*, IEEE Software, September 1990.

[5] Huslende, Ragnar, *A Combined Evaluation of Performance and Reliability for Degradable Systems*, ACM-SIGMETRICS Conf. on Measurement and Modeling of Comput. Syst., 1981, pp. 157-163.

[6] *Methodology for Software Reliability Prediction*, RADC-TR-87-181, Science Applications International Corporation for Rome Air Development Center 1987.

[7] Schreiber, Fabio A., *Information Systems: A Challenge for Computers and Communications Reliability*, IEEE Journal on Selected Areas in Communications, Vol. SAC-4, No. 6, October 1986, pp 157-164.

[8] Trachtenberg, Martin, *A General Theory of Software Reliability Modeling*. IEEE Transactions on Reliability, Vol. 39, No. 1, 1990 April, pp 92-96.