5

A functional approach to software reliability modeling

J. C. Munson
University of Idaho
Computer Science Department, Moscow, Idaho 83844-1010, USA.
Telephone: 208-885-7789. Fax: 208-885- 9052.
Email: jmunson@cs.uidaho.edu

Abstract

A new metaphor is presented for the measurement and modeling of the reliability of software systems that focuses on the functionality that the code is executing and not the software as a monolithic system. In computer software systems, it is the functionality that fails. Some functionalities may be virtually failure free while other functionalities will collapse with certainty whenever they are executed. The focus of this paper is on the notion that it is possible to measure the activities of a system as it executes its various functionalities and characterize the reliability of the system in terms of these functionalities. A software system is the sum of its functionalities. If we can know the reliability of the functionalities and how the system apportions its time among these functionalities, we can then know the reliability of the system. This view of reliability permits the dynamic estimation of the parameters of the underlying multinomial probability distribution representing the transition between program modules. This use of the multinomial probability distribution is particularly convenient in that it has a Dirichlet distribution as its natural conjugate family. Thus, a Bayesian approach may be employed so that each step or epoch in the dynamic operation of a system provides incremental information as to the evolving reliability assessment of the program.

Keywords

Software reliability, stochastic processes, Markov models, software functionality.

1 INTRODUCTION

The subject of this paper is measurement, specifically, the measurement of those software attributes that are associated with software failure events. Existing approaches to the understanding of software reliability patently assume that software failure events are observable. The truth is that the overwhelming majority of software failures go unnoticed when they occur. Only when these failures disrupt the system by second, third or fourth order

R. F. Boisvert (ed.), *Quality of Numerical Software*

effect do they provide enough disruption for outside observation. Consequently, only those dramatic events that lead to the immediate collapse of a system can been seen directly by an observer when they occur. The more insidious failures will lurk in the code for a long interval before their effects are observed. Failure events go unnoticed and undetected because the software has not been instrumented so that these failure events may be detected. In that software failures go largely unnoticed, it is presumptuous to attempt to model the reliability of software based on these inaccurate and erroneous data. If we cannot observe failure events directly, then we must seek another metaphor that will permit us to model and understand reliability in a context that we can measure.

Our current view of software reliability is colored by a philosophical approach that began with efforts to model hardware reliability (cf. Musa 1992). Inherent in this approach is the notion that it is possible to identify with some precision this failure event and measure the elapsed time to the failure event. For hardware systems this has real meaning. The case for software systems is not at all the same. Failure events are sometimes quite visible in terms of catastrophic collapses of a system. More often than not, the actual failure event will have occurred a considerable time before its effect is noted. In most cases it is simply not possible to determine with any certainty just when the actual failure occurred on a real time clock. This being the case, we then must look to new metaphors for software systems that will permit us to model the reliability of these systems based on things that we *can* measure with some accuracy.

Yet another problem with the hardware adaptive approach to software reliability modeling is that the failure of a computer software system is simply not time dependent. A system can operate without failure for years and then suddenly become very unreliable based on the changing functionalities that the system must execute. Many university computer centers experienced this phenomenon in the late 1960's and early 1970's when there was a sudden shift in computer science curricula from programming languages such as FORTRAN that had static run time environments to ALGOL derivatives such as Pascal and Modula that had dynamic run time environments. From an operating system perspective, there was a major shift in the functionality of the operating system exercised by these two different environments. As the shift was made to the ALGOL like languages, latent code in the operating system, specifically those routines that dealt with memory management, that had not been executed overly much in the past now became central to the new operating environment. This code was both fragile and untested. The operating systems that had been so reliable began to fail like cheap light bulbs.

As the discussion herein unfolds, we will see that the key to understanding program failure events is the direct association of these failures to execution events with a given functionality. A Markovian stochastic process will be used to describe the transition of program modules from one to another as a program expresses a functionality. The Markovian approach to software reliability modeling is not new. Laprie (1984), Littlewood (1979), and Sigcrist (1988a, 19988b) have described the transfer of control in a software system as a Markov process. What is novel about this investigation is that a method for determining the reliability of program functionality is presented together with a Bayesian approach for the dynamic determination of operational, functional, and system reliability. It will become fairly obvious just what data will be needed to describe accurately the reliability of the system. In essence, the system will be able to appraise us of its own health. The reliability modeling process is no

longer something that can be performed *ex post facto*. It may be accomplished dynamically while the program is executing. It is the goal of this paper to develop a methodology that will permit the modeling of the reliability of program functionality.

2 A FORMAL DESCRIPTION OF PROGRAM OPERATION

From a functional viewpoint, a program may be viewed as a set of program modules that are executing a set of mutually exclusive functionalities. If the program executes a functionality consisting of a subset of these modules that are fault free, it will never fail no matter how long it executes this functionality. If, on the other hand, the program is executing a functionality that contains fault laden modules, there is a very good likelihood that it will fail whenever that functionality is expressed (Munson, 1995). Further, it will fail with certainty when the right aspects of functionality are expressed.

To assist in the subsequent discussion of program functionality, it will be useful to make this description somewhat more precise by introducing some notation conveniences. Assume that a software system S was designed to implement a specific set of mutually exclusive functionalities F. Thus, if the system is executing a functionality $f \in F$ then it cannot be expressing elements of any other functionality in F. Each of these functionalities in F was designed to implement a set of software specifications based on a user's requirements. From a user's perspective, this software system will implement a specific set of operations, O. This mapping from the set of user perceived operations, O, to a set of specific program functionalities, F, is one of the major tasks in the software specification process.

A pilot astronaut on the Space Shuttle, for example, is not aware of the functionality of the Primary Avionics Software System (PASS) that governs the complete operation of the shuttle. A metaphor has been carefully constructed by system designers that permits the pilot to control the shuttle as if it were a standard airplane. The pilot, for example, has a control stick the controls two user perceived operations: roll and pitch. These operations are implemented in software functionalities that monitor the change in resistance in x and y coordinate rheostats in the base of the control stick. The pilot *operates* the spacecraft. The software *functionalities* monitor the change in resistance in the rheostats.

Each operation that a system may perform for a user may be thought of as having been implemented in a set of functional specifications. There may be a one-to-one mapping between the user's notion of an operation and a program functionality. In most cases, however, there may be several discrete functionalities that must be executed to express the user's concept of an operation. For each operation, o, that the system may perform, the range of functionalities, f, must be well known. Within each operation one or more of the system's functionalities will be expressed. For a given operation, o, these expressed functionalities are those with the property

$$F^{(o)} = \{f : F \mid \forall \text{ IMPLEMENTS}(o, f)\}.$$

It is possible, then, to define a relation IMPLEMENTS over $O \times F$ such that IMPLEMENTS(o, f) is true if functionality f is used in the specification of an operation, o

The software design process is basically a matter of assigning functionalities in F to specific program modules $m \in M$, the set of program modules. The design process may be thought of as the process of defining a set of relations, ASSIGNS over $F \times M$ such that ASSIGNS(*f, m*) is true if functionality f is expressed in module m. For a given software system, S, let M denote the set of all program modules for that system. For each functionality $f \in F$, there is a relation p over $F \times M$ such that $p(f,m)$ represents the probability of execution events of module m when the system is executing functionality f.

There is a relationship between program functionalities and the software modules that they will cause to be executed. These program modules will be assigned to one of two distinct sets of modules that, in turn, are subsets of M. One set of software modules may or may not execute when the system is running a particular functionality. These modules are said to be potentially involved modules. The set of potentially involved modules is.

$$M_p^{(f)} = \{m : M_F \mid \exists f \in F \bullet \text{ASSIGNS}(f,m) \wedge 0 < p(f,m) < 1\}.$$

In other program modules, there is extremely tight binding between a particular functionality and a set of program modules. That is, every time a particular functionality, f, is executed, a distinct set of software modules will always be invoked. These modules are said to be indispensably involved with the functionality f. This set of indispensably involved modules for a particular functionality, f, is the set of those modules that have the property that

$$M_i^{(f)} = \{m : M_F \mid \forall f \in F \bullet \text{ASSIGNS}(f,m) \Rightarrow p(f,m) = 1\}.$$

As a direct result of the design of the program, there will be a well defined set of program modules , M_F, that might be used to express all aspects of a given functionality, f. These are the modules that have the property that

$$m \in M_f = M_p^{(f)} \cup M_i^{(f)}.$$

From the standpoint of software design, the real problems in understanding the dynamic behavior of a system are not necessarily attributable to the set of modules, M_i, that are tightly bound to a functionality. The real problem is the set of potentially invoked modules, M_p. The greater the cardinality of this set of modules, the less certain we may be about the behavior of a system performing that functionality. For any one instance of execution of this functionality, a varying number of the modules in M_p may execute. This fact has strong implications for the design of testable and reliable software.

3 A STOCHASTIC DESCRIPTION OF PROGRAM OPERATION

The transition from one module to another may be described as a stochastic process. In which case we may define an indexed collection of random variables $\{X_t\}$, where the index t runs through a set of non-negative integers, $t = 0,1,2,\cdots$ representing the epochs of the process. At any particular epoch the software is found to be executing exactly one of its M modules. The

fact of the execution occurring in a particular module is a *state* of the system. For this software system, the system is found in exactly one of a finite number of mutually exclusive and exhaustive states that may be labeled $0,1,2,\cdots,M$, the set of program modules. In this representation of the system, there is a stochastic process $\{X_t\}$, where the random variables are observed at epochs $t = 0,1,2,\cdots$ and where each random variable may take on any one of the $(M+1)$ integers, from the state space $A = \{0,1,2,\cdots,M\}$.

A stochastic process $\{X_t\}$ is a Markov chain if it has the property that

$$\Pr[X_{t+1} = j | X_t = i_t, X_{t-1} = i_{t-1}, X_{t-2} = i_{t-2}, \cdots, X_0 = i_0] = \Pr[X_{t+1} = j | X_t = i_t]$$

for any epoch $t = 0,1,2,\cdots$. and all states $i_0, i_1, \cdots, i_t$ in the state space A. This is equivalent to saying that the conditional probability of executing any module at any future epoch is dependent only on the current state of the system. The conditional probabilities $\Pr[X_{t+1} = j | X_t = i_t]$ are called the transition probabilities. In that this nomenclature is somewhat cumbersome, let $p_{ij}^{(n)} = \Pr[X_n = j | X_{n-1} = i]$. Within the execution of a given functionality, the behavior of the system is static. That is, the transition probabilities do not change from one epoch to another. Thus, $\Pr[X_{t+1} = j | X_t = i_t] = \Pr[X_1 = j | X_0 = i_0]$ for i, j, in S, which is an additional condition of a Markov process.

What we would like to ascertain is the unconditional probability of being in a particular module at a particular epoch. In order to find this conditional probability let us first observe that

$$\Pr[X_{t+1} = j | X_t = i_t] = \Pr[X_1 = j | X_0 = i_0].$$

It is clear that the unconditional probability of executing a module j at epoch n, then, is dependent only on the initial state of the system. Thus,

$$\Pr[X_n = j] = \sum_{i=0} p_{ij} \quad \Pr[X_0 = i]$$

Interestingly enough, for all software systems there is a distinguished module, the main program module that will always receive execution control from the operating system. If we denote this main program as module 0 then

$$\Pr[X_0 = 0] = 1 \text{ and } \Pr[X_0 = i] = 0 \text{ for } i = 1,2,\cdots,M.$$

We can see, then, that the unconditional probability of executing in a particular module j is

$$\Pr[X_n = j] = p_{ij}^{(n)} \Pr[X_0 = 0] = p_{ij}^{(n)}$$

We simply cannot measure time in any meaningful way for execution events of computer software. We can, however, observe the transition from one module to another. These transition intervals define the notion of the software **epoch**. An epoch begins with the onset of execution in a particular module and ends when control is passed to another module. The measurable event for modeling purposes is this transition among the program modules. We will count the number of calls from a module and the number of returns to that module. Each

of these transitions to a different program module from the one currently executing will represent an incremental change in the epoch number. Computer programs executing in their normal mode will make state transitions between program modules rather rapidly. In terms of real clock time, many epochs may elapse in a relatively short period.

We now wish to examine the potential for modeling the failure of a software system. When a program module fails, we can imagine that the module has made a transition to an absorbing state, a failure state, in the Markov transition matrix. Thus, every program may be thought to have a virtual module representing the failed state of that program. When this virtual module receives control, it will not relinquish it. The transition matrix for this new model is augmented by an additional row and a new column. For a program with M modules, let the error state be represented by a new state, $T = M+1$. For this new state,

$$p_{Tj}^{(n)} \begin{cases} = 0 & \text{for all } j = 1,2,\ldots,M \\ = 1 & \text{for } j = T \end{cases}, \quad n = 0,1,2,\cdots.$$

This represents the augmented row of the new transition matrix. Each row in the transition matrix will be augmented by a new column entry $p_{iT}^{(n)}$ for $i = 1,2,\cdots,M$, where $p_{iT}^{(n)}$ represents the probability of the failure of the i^{th} module in the n^{th} epoch. When a program dies, it is the result of a fault in one or more of its modules. Not all modules are equally likely to lead to the failure event. The fault proneness of the module is distinctly related to measurable software attributes (Munson and Khoshgoftaar, 1992). When program modules are executed that are fault prone, they are much more likely to fail than those that are not fault prone. We seek a forecasting or prediction mechanism that will capitalize on this understanding.

4 THE PROFILES OF SOFTWARE DYNAMICS

Each user of a software system will exercise the operations of this system in a particular manner. This is a characteristic of the user. Each of the independent and mutually exclusive operations of the system will be exercised by the user with a certain probability. This is the characteristic **operational profile** for the user. This operational profile is multinomial probability distribution for each of the operations in the set O.

When a software system is constructed by the software developer, it is designed to fulfill a set of specific functional requirements. The user will run the software to perform a set of perceived operations. In this process, the operations will typically not use all of the functionalities with the same probability. The **functional profile** of the software system is the set of unconditional probabilities of each of the functionalities F being executed by the user. Let Y be a random variable defined on the indices of the set of elements of F. Then, $o_k = \Pr[Y = k]$, $k = 1,2,\ldots,\#\{F\}$ is the probability that the user is executing program functionality k as specified in the functional requirements of the program and $\#\{F\}$ is the cardinality of the set of functionalities (Musa, 1992). A program executing on a serial machine can only be executing one functionality at a time. The distribution of o, then, is multinomial for programs designed to fulfill more than two specific functionalities. The prior knowledge of this distribution of functionalities should guide the software design process (Munson, 1996).

When a program is executing a given functionality, say f_k, it will distribute its activity across the set of modules, M_{f_k}. At any arbitrary epoch, n, the program will be executing a module $m_i \in M_{f_k}$ with a probability, $u_{ik} = \Pr[X_n = i | Y = k]$. The set of conditional probabilities u_{*k} where $k = 1,2,\ldots,\#\{F\}$ constitute the **execution profile** for functionality f_k. As was the case with the functional profile, the distribution of the execution profile is also multinomial for a software system consisting of more than two modules. As a matter of the design of a program, there may be a non-empty set $M_p^{(f)}$ of modules that may or may not be executed when a particular functionality is exercised. This will, of course, cause the cardinality of the set M_f to vary. A particular execution may not invoke any of the modules of $M_p^{(f)}$. On the other hand, all of the modules may participate in the execution of that functionality. This variation in the cardinality of M_f within the execution of a single functionality will contribute significantly to the amount of test effort that will be necessary to test such a functionality.

Each operation will be implemented by a subset of functionalities, i.e. $F_e^{(o)} \subset F$. As each operation is run to completion it will generate an execution profile. This execution profile may represent the results of the execution of one or more functionalities. Most operations, though, do not exercise precisely one functionality. Rather, they may apportion time across a number of functionalities. For a given operation, let l be a proportionality constant. Then, $0 \le l_k \le 1$ will represent the proportion of epochs that will be spent executing the k^{th} functionality in $F_e^{(o)}$. Thus an operational profile of a set of modules will represent a linear combination of the conditional probabilities, u_{ik} as follows:

$$p_i = \sum_{f_k \in F_e^{(o)}} l_k u_{ik}.$$

The manner in which a program will exercise its many modules as the user chooses to execute the functionalities of the program is determined directly by the design of the program. Indeed, this mapping of functionality onto program modules is the overall objective of the design process. The **module profile**, q, is the unconditional probability that a particular module will be executed based on the design of the program. It is derived through the application of Bayes' rule. First, the joint probability that a given module is executing and the program is exercising a particular functionality is given by

$$\Pr[X_n = j \cap Y = k] = \Pr[Y = k]\Pr[X_n = j | Y = k] = o_k u_{ik}$$

where j and k are defined as before. Thus, the unconditional probability, q_i of executing module j under a particular design is

$$\begin{aligned} q_i &= \Pr[X_n = i] \\ &= \sum_k \Pr[X_n = i \cap Y = k] \\ &= \sum_k o_k u_{ik} \end{aligned}$$

As was the case for the functional profile and the execution profile, only one module can be executing at any one time. Hence, the distribution of q is also multinomial for a system consisting of more than two modules.

The final profile consideration will be the determination of the transition probabilities $p_{ij}^{(0)} = \Pr[X_1 = j \mid X_0 = i]$ of $\boldsymbol{P}^0$. Each row i of $\boldsymbol{P}$ represents the probability of the transition to a new state j given that the program is currently in state i. As such, this row represents the

transition profile for each module. These transitions are mutually exclusive events. The program may only transfer control to exactly one other program module. Under this assumption, the conditional probabilities that are the rows of $\boldsymbol{P}^0$, also have the property that they are distributed multinomially. They profile the transitions from one state to another.

5 ESTIMATES FOR THE PROFILES

The focus will now shift to the problem of understanding the nature of the distribution of the probabilities for various profiles. We have so far come to recognize these profiles in terms of their multinomial nature. The multinomial distribution is useful for representing the outcome of an experiment involving a set of mutually exclusive events. Each of these events would correspond to a program executing a particular module in the total set of program modules. Let $\Pr(Y = i) = w_i$ represent the event that the random variable has taken the value i from the earlier defined state space and $w_T = 1 - w_1 - w_2 - \cdots - w_M$, under the condition that $T = M + 1$, as defined earlier. In which case w_i is the probability that the outcome of a random experiment is an integer from the state space. If this experiment is conducted over a period of n trials then the random variable X_i will represent the frequency of S_i outcomes. In this case, the value, n, represents the number of transitions from one program module to the next. Note that $X_T = n - X_1 - X_2 - \cdots - X_M$. This particular distribution will be useful in the modeling of a program with a set of k modules. During a set of n program steps, each of the modules may be executed. These, of course, are mutually exclusive events. If module i is executing then module j cannot be executing.

The multinomial distribution function with parameters n and $\boldsymbol{w} = (w_1, w_2, \ldots, w_T)$ is given by

$$f(\boldsymbol{x} \mid n, \boldsymbol{w}) = \begin{cases} \dfrac{n!}{\prod_{i=1}^{k-1} x_i!} w_1^{x_1} w_2^{x_2} \cdots w_M^{x_M}, & (x_1, x_2, \cdots, x_M) \in S \\ 0 \text{ elsewhere} \end{cases}$$

where x_i represents the frequency of execution of the i^{th} program module. The expected values for the x_i are given by $E(x_i) = \bar{x}_i = nw_i$, $i = 1, 2, \ldots, k$.

We would like to come to understand, for example, the multinomial distribution of a program's execution profile while it is executing a particular functionality. The problem, here, is that every time a program is run we will observe that there is some variation in the profile from one execution sample to the next. It will be difficult to estimate the parameters $\boldsymbol{w} = (w_1, w_2, \ldots, w_T)$ for the multinomial distribution of the execution profile. Rather than estimating these parameters statically, it would be far more useful to us to get estimates of these parameters dynamically as the program is actually in operation, hence the utility of the Bayesian approach.

To aid in the process of characterizing the nature of the true underlying multinomial distribution, let us observe that the family of Dirichlet distributions is a conjugate family for observations that have a multinomial distribution (Wilks, 1962). The p.d.f. for a Dirichlet

distribution, $D(\alpha,\alpha_T)$, with a parametric vector $\alpha=(\alpha_1,\alpha_2,\ldots,\alpha_M)$ where $(\alpha_i>0;\ i=1,2,\ldots,M)$ is

$$f(w|\alpha)=\frac{\Gamma(\alpha_1+\alpha_2+\cdots+\alpha_M)}{\prod_{i=1}^{M}\Gamma(\alpha_i)}w_1^{\alpha_1-1}w_2^{\alpha_2-1}\cdots w_M^{\alpha_M-1}$$

where $(w_i>0;\ i=1,2,\ldots,M)$. The expected values of the w_i are given by

$$E(w_i)=\mu_i=\frac{\alpha_i}{\alpha_0}\ \text{where}\ \alpha_0=\sum_{i=1}^{T}\alpha_i.$$

In this context, α_0 represents the total number of observed epochs and α_i represents the total number of transitions to the module *I* from all other modules

Within the set of expected values $\mu_i, i=1,2,\ldots,T$, not all of the values are of equal interest. We are interested, in particular, in the value of μ_T. This will represent the probability of a transition to the terminal failure state from a particular program module. So that we might use this value for our succeeding reliability prediction activities, it will be useful to know how good this estimate is. To this end, we would like to set 100(1-α)% confidence limits on the estimate. For the Dirichlet distribution, this is not clean. To simplify the process of setting these confidence limits, let us observe that if $w=(w_1,w_2,\ldots,w_M)$ is a random vector having the *M*-variate Dirichlet distribution, $D(\alpha,\alpha_T)$, then the sum $z=w_1+\cdots+w_M$ has the beta distribution, $f_\beta(z|\gamma,\alpha_T)$ or alternately $f_\beta=(w_T|\gamma,\alpha_T)$, where $\gamma=\alpha_1+\alpha_2+\cdots+\alpha_M$. Thus, we may obtain $100(1-\alpha)\%$ confidence limits for $\mu_T-a\le\mu_T\le\mu_T+b$ from

$$F_\beta(\mu_T-a|\gamma,\alpha_T)=\int_0^{\mu_T-a}f_\beta(w_T|\gamma,\alpha_T)dw=\frac{\alpha}{2}$$

and

$$F_\beta(\mu_T+b|\gamma,\alpha_T)=\int_0^{\mu_T+b}f_\beta(w_T|\gamma,\alpha_T)dw=1-\frac{\alpha}{2}.$$

The value of the use of the Dirichlet conjugate family for estimation purposes is twofold. First, it permits us to estimate the probabilities of the module transitions directly from the observed transitions. Secondly, we are able to obtain revised estimates for these probabilities as the observation process progresses. Let us now suppose that we wish to model the behavior of a software system whose execution profile has a multinomial distribution with parameters *n* and $W=(w_1,w_2,\ldots,w_M)$ where *n* is the total number of observed module transitions and the values of the w_i are unknown. Let us assume that the prior distribution of *W* is a Dirichlet distribution with a parametric vector $\alpha=(\alpha_1,\alpha_2,\ldots,\alpha_M)$ where $(\alpha_i>0;\ i=1,2,\ldots,M)$. Then the posterior distribution of *W* for the behavioral observation $X=(x_1,x_2,\ldots,x_M)$ is a Dirichlet distribution with parametric vector $\alpha^*=(\alpha_1+x_1,\alpha_2+x_2,\ldots,\alpha_M+x_M)$ (c.f. DeGroot, 1970). As an example, suppose that we now wish to model the behavior of a large software system with such a parametric vector. As the system makes sequential transitions from one module to another, the posterior distribution of *W* at each transition will be a Dirichlet distribution.

Further, for $i = 1,2,\ldots,T$ the i^{th} component of the augmented parametric vector α will be increased by 1 unit each time module m_i is executed.

6 RELIABILITY ESTIMATION

In order that the essence of the specification and design process be captured, two matrices will be defined to preserve the mapping of $O \times F$ and also the mapping of $F \times M$. A matrix $\boldsymbol{G}$ that will be used to describe the mapping $O \times F$ of the operations to the program functionalities. Each element of this matrix will have the property that

$$g_{ij} = \begin{cases} 1 \text{ if IMPLEMENTS } (o_i, f_j) \text{ is TRUE} \\ 0 \text{ if IMPLEMENTS } (o_i, f_j) \text{ is FALSE} \end{cases}.$$

The matrix $\boldsymbol{S}$ that will be employed to describe the mapping $F \times M$ of the functionality to the program modules as a result of the program design process. Each element of this matrix will have the property that

$$s_{jk} = \begin{cases} 1 \text{ if ASSIGNS } (f_j, m_k) \text{ is TRUE} \\ 0 \text{ if ASSIGNS } (f_j, m_k) \text{ is FALSE} \end{cases}.$$

These two matrices $\boldsymbol{S}$ and $\boldsymbol{G}$ will permit the subsequent partitioning of reliability estimates into functional, operational, and system reliability. Functional reliability is the reliability of each of the many system functions. Operational reliability is the user's perception of the reliability of those operations performed by him. Finally, system reliability is the reliability of the system as it will be used.

6.1 Functional Reliability

To this point we have created a mechanism for modeling the transition of each program module to the failure state. In this sense the reliability of each module m_i may be directly determined by the elements, p_{iT}^0, of $\boldsymbol{P}^0$. With the Bayesian approach, we have also established a mechanism for refining our estimates of these reliabilites and establishing a measure of confidence in each of these estimates. This information will now be used to establish the reliability of functionalities that employ each of the modules in varying degrees. The successive powers of $\boldsymbol{P}^0$ will show the failure likelihood for each of the modules. This will permit us to postulate on the probability of a failure at some future epoch n based on the current estimates of failure probability.

Not all states that a system of M program modules can get into hold equal fascination for us. In the augmented program model above, we postulated the existence of a virtual program module m_K, a program module representing an error state. This module may or may not be invoked depending on the particular functionality being executed. If a functionality executes a set of unreliable modules with a high probability then the functionality will not be reliable. If, on the other hand, the functionality executes only highly reliable modules, then the

functionality will be perceived to be reliable. It is a characteristic of each functionality that it exhibits an execution profile u_{*k}. Each module has an associated reliability. Let us define the reliability of the j^{th} module to be $r_j = 1 - p_{jT}^{(0)}$. The expected value for the reliability of the functionality, then is

$$\mu_f = E(r_f) = \sum_{j=1}^{m} u_{\bullet j} r_j ,$$

where $u_{\bullet j}$ is the execution profile for the functionality as defined earlier and m is the number of modules. This reliability estimate, however, is only for a particular functionality. It is derived from the execution profile of a given functionality. Thus, each functionality has its own independent reliability assessment. That was the original intent of this investigation, to demonstrate a mechanism for the determination of the reliability of program functionality. It is program functionalities that fail. Some functionalities are more reliable than others. We can measure reliability at the functional level.

The reliability of the individual functionalities is dependent on the distribution of $u_{\bullet j}$ for the functionality. As was indicated earlier, the underlying distribution for $u_{\bullet j}$ is multinomial. We may derive estimates for these probabilities in precisely the same manner that we used for developing the estimates for the conditional probabilities of the transition matrix. We simply need to count the frequency that each module is executed when a particular functionality is being executed. Computation for the estimates for the $u_{\bullet j}$ will proceed as above as will the determination for the confidence intervals for these estimates.

Now we arrive at the real problem in the estimation of the reliability of a functionality. Our long term ability to understand and/or estimate the execution profile of a system is clouded by the set of modules $M_p^{(f)}$ that may or may not execute when a particular functionality is expressed. It would be unreasonable to assume that the execution profiles for all functionalities were stable and knowable. (The worse the design, the less stable will be these profiles). The Bayesian approach to reliability determination is of value in that we may use the information currently at our disposal to provide the best estimate as to the future behavior of the system. If the set $M_p^{(f)}$ is empty, then the behavior of the system is quite tractable.

The very best way to create a system whose reliability may never be understood is to design the system with a large set of modules $M_p^{(f)}$. The obverse of this coin is a tractable system with little or no variability in its functionality execution profiles. The reliability of such a system may be assessed quite accurately. This is not to imply that a reliable system is one whose reliability may be measured accurately: quite the contrary. We may conceive of a very unreliable system whose behavior may be well understood. A reliable system is one that *by design* spends a high proportion of its execution time in modules that are not likely to fail. In other words, there is a strong positive correlation between the measures of complexity of a module and its design functional profile. Such fault prone modules may, in fact, be identified by their intrinsic attributes during the design stage (Munson and Ravenel, 1993; Munson and Khoshgoftaar, 1992). If a system is carefully designed with these criteria controlling the design process, the resulting software will be reliable. It may, however, not be robust.

A design is robust if it does not suffer a diminution in its functional reliability if the face of departures from its design functional profile. Not all reliable systems are robust. The overall robustness of a system is dependent on how the specific functionalities implement users'

operations. A robust system is one that remains reliable in the face of departures from the design operational profile of the system. Systems may be designed for both reliable and robust operation.

6.2 Operational Reliability

The users of computer software do not perceive the functional granularity of the software. They interact with it on an operational level. Thus, the user will assess the software based on the reliability of the operations that he/she will perform. Just as it was possible to measure the reliability of functionality, each operation has an associated reliability. This reliability may be determined directly from the mapping between operations and functionalities. The concept of operational reliability enables the reliability assessment to be made on those activities that are most likely to be used in practice. Chen, et.al. (1995) conducted a series of experiments along this line.

They found that faults present in the code have only a marginal effect on the reliability on the operational reliability of a system. The overall objective of the software system architect is to make the software reliable in the set of operations that a user will perform. Just as it was possible to determine the reliability of a functionality, so too is it possible to establish the notion of the reliability of an operation. This may be determined from the mapping of operations to functionalities as follows:

$$\mu_{o_i} = \sum_j g_{ij} \mu_{f_j}$$

Each operation performed by a user will cause a given set of functionalities to be expressed. We have established the feasibility of computing the reliability of the functionalities of a program. Thus, each operation has a given reliability. The operational reliability of a system is the expected value of the reliabilites of the individual operations under a given operational profile.

A user's departure from an operation profile is a source of great concern. Each operation has an associated reliability. If the design is such that the user will spend most of his/her time in performing operations that are reliability, then the system will be perceived to be reliable by this individual. A robust design is one that will cause users with differing operational profiles to experience the same degree of system reliability.

There is a down side to the estimation of the operational reliability of a system. There may be substantial variability in the functional reliability for any given functionality. This is due to the variation in the cardinality of the sets of potentially invoked modules M_p^f for each functionality. In the extreme case, consider that one of the elements of M_p^f has a disproportionally large number of faults. Sometimes functionality f may invoke this module and other times it will not be invoked. Those execution instances in which f is used will be far more likely to fail that those in which f is not used. Necessarily, as we begin to understand the relationship between program architecture and reliability we may ultimately learn to limit the cardinality of the sets M_p^f in the design process.

6.3 System Reliability

A central thrust of this investigation is to establish a framework for establishing the reliability of program modules, the basic building blocks of a computer program. Each program module has an associated reliability. The total system is a composite of all of the program modules. As a direct result of design decision made in the implementation of program functionality, a module profile emerged for the design. The module profile q_i is the unconditional probability that a module will be in execution at any epoch. The expected value for the system reliability r_S then is simply

$$r_S = \sum_{j=1}^{m} q_j r_j .$$

A lower bound on this value may be obtained from

$$r_S^l = \sum_{j=1}^{m} q_j r_j^l$$

where

$$r_j^l = 1 - (p_{jT}^{(0)} + b)$$

is derived from the upper $\alpha / 2$ confidence limit for each of the transition failure estimates.

The system reliability is a function of the user's operational profile. It is interesting to note that each user brings to the setting a distinct operational profile. Thus, there is clearly a different system reliability for each user of a given system. One user's reliable system might well be the bane of another user's existence.

7 MEASUREMENT FOR RELIABILITY ESTIMATION

The basic premise of this paper is that we really cannot measure temporal aspects of program failure. There are, however, certain aspects of program behavior that we can measure and also measure with accuracy. We can measure transitions into and out of program modules, for example. We can measure the frequency of executions of each functionality, if the program is suitably instrumented. We can also measure the frequency of executions of program operations, again supposing that the program has been instrumented to do this.

Let us now turn to the measurement scenario for the modeling process described above. Consider a system whose requirements specify a set, A, of user operations. These operations, again specified by a set of functional requirements, will be mapped into a set, B, of elementary program functionalities. The functionalities, in turn, will be mapped by the design process into the set of M program modules.

We will need a mechanism for tracking the actual behavior of the user of the system. To this end we will require a vector, $\boldsymbol{O}$, in which the program will count the frequency of each of the operations. That is, an element o_i of this vector will be incremented every time the program initiates the i^{th} user operation. Each of the operations is distinct and they are

mutually exclusive. Thus we may use the Bayesian estimation process to compute estimates for the actual posterior operational profile for the software. If the design of the software system is not robust in regards to departures from the prior assumptions about the operational profile, then the posterior distribution of the operational profile will serve as a leading indicator of potential reliability problems with the software.

A guiding principle of the design process will be the prior execution profile for each functionality. That is, we will use our prior knowledge of the complexity of each of the program modules to minimize our exposure to the modules that are likely to be fault prone. Again, if the design is not robust in regards to departures from these prior distributions, then we may anticipate severe reliability problems for the software system. A current assessment of the frequency with which functionalities are executed may be maintained in a matrix $\boldsymbol{S}$ as defined above. As was the case with the operational profile, an element o_j of this vector will be incremented every time the program initiates the j^{th} functionality.

Provision must be made to record the behavior of the total system as it transitions from one program module to another. If there are a total of m modules, then we will need an $n \times n$ $(n = m+1)$ matrix $\boldsymbol{T}$ to record these transitions. Whenever the program transfers control from module m_i to module m_j the element t_{ij} of $\boldsymbol{T}$ will increase by one. Every time that a failure is caused by a fault in module m_i then t_{in} of $\boldsymbol{T}$ will increase by one representing a transition from that module to the failure state. Necessarily, if the program is maintaining the transition matrix, it will not be able to update the vector for the virtual module that represents the failure state of the program. When the program transitions to this hypothetical state, it will be dead. If $\boldsymbol{T}$ is being maintained by the operating system, or even better, in the hardware, then we will also be able to capture the program's transition into the virtual failure module.

The posterior probabilities for the transition matrix $\boldsymbol{P}_{f_j}^{(0)}$ may be obtained from a reduced matrix $\boldsymbol{T}^j$ obtained from $\boldsymbol{T}$ in the following manner. The reduced matrix $\boldsymbol{T}^j$ will have non-zero entries for just those modules that are elements of the set modules M_{f_j} that are directly involved in the activities of the functionality f_j. Thus, the elements of $\boldsymbol{T}^j$ may be created by

$$t_{ik}^{f_j} = s_{ji} s_{jk} t_{ik} .$$

The execution profile $u_{\bullet j}$ for each functionality may be derived from either the row or column marginals of $\boldsymbol{T}^j$ to wit:

$$u_{\bullet j} = \sum_{i=1}^{m} t_{ij}^{f_j} \Big/ \sum_{j=1}^{m}\sum_{i=1}^{m} t_{ij}^{f_j} = \sum_{i=1}^{m} t_{ji}^{f_j} \Big/ \sum_{j=1}^{m}\sum_{i=1}^{m} t_{ij}^{f_j}$$

The very nature of the milieu in which programs operate dictates that they will modify the behavior of the individuals that are using them. The result of this is that the user's initial use of the system as characterized by the operational profile will not necessarily reflect his/her future use of the software. There may be a dramatic shift in the operational profile of the software user based directly on the impact of the software or due to the fact the users' needs have changed over time. A design that has be established to be robust under one operational profile may become less than satisfactory under new profiles. We must come to understand that some systems may become less reliable as they mature due to circumstances external to the program environment.

The continuing evaluation of the execution, function, and module profiles over the life of a system can provide substantial information as to the changing nature of the program's execution environment. This, in turn, will foster the notion that software reliability assessment is as dynamic as the operating environment of the program. That a system has functioned reliably in its past is **not** a clear indication that it will function reliably in the future.

8 SUMMARY AND CONCLUSIONS

The failure of a software system is dependent only on what the software is currently doing: its functionality. If a program is currently executing a functionality that is expressed in terms of a set of fault free modules, this functionality will certainly execute indefinitely without any likelihood of failure. A program may execute a sequence of fault prone modules and still not fail. In this particular case, the faults may lie in a region of the code that is not likely to be expressed during the execution of a functionality. A failure event can only occur when the software system executes a module that contains faults. If a functionality is never selected that drives the program into a module that contains faults, then the program will never fail. Alternatively, a program may well execute successfully in a module that contains faults just as long as the faults are in code subsets that are not executed.

Some of the problems that have arisen in past attempts at software reliability determination all relate to the fact that their perspective has been distorted. Programs do not wear out over time. If they are not modified, they will certainly not improve over time. Nor will they get less reliable over time. The only thing that really impacts the reliability of a software system is its functionality. A program may work very well for a number of years based on the functionalities that it is asked to execute. This same program may suddenly become quite unreliable if the user's mission (operational profile) suddenly changes.

By keeping track of the state transitions from module to module and functionality to functionality we may learn exactly where a system is fragile. This information coupled with the functional profile will tell us just how reliable the system will be when we use it as specified. Programs make transitions from module to module as they execute. These transitions may be observed. Transitions to program modules that are fault laden will result in an increased probability of failure. We can model these transitions as a stochastic process. Ultimately, by developing a mathematical description for the behavior of the software as it transitions from one module to another driven by the functionalities that it is performing, we can describe the reliability of the functionality. The software system is the sum of its functionalities. If we can know the reliability of the functionalities and how the system apportions its time among these functionalities, we can then know the reliability of the system.

Ongoing investigations into the etiology of software failures in the Primary Avionics Software System of the Space Shuttle has provided substantial insight into the measurement of the reliability of this system. This has led to the conclusion that it is not the software system that fails. It is the software system executing a particular functionality that fails. From this new perspective, the sequential execution of program functionalities may be modeled as a stochastic process. In particular, the program functionalities are physically expressed within a program as subtrees of modules in a program call tree hierarchy. The transitions between the program modules in a pairwise fashion may be represented in a transition matrix of a Markov

process. Program failures are represented by an absorbing state in the transition matrix. This view of reliability permits the dynamic estimation of the parameters of the underlying multinomial probability distribution representing the transition between program modules. This use of the multinomial probability distribution is particularly convenient in that it has a Dirichlet distribution as its natural conjugate family. Thus, a Bayesian approach may be employed so that each step or epoch in the dynamic operation of a system provides incremental information as to the evolving reliability assessment of the program.

9 REFERENCES

Chen, M., Mathur, A.P. and Rego, V. J. (1995) Effect of testing techniques on software reliability estimates obtained using a time-domain model. *IEEE Transactions on Reliability*, **44** (1) 97-103.

DeGroot, M. H. (1970) *Optimal Statistical Decision.* McGraw-Hill Book Company, New York. .

Laprie, J. C. (1984) Dependability evaluation of software systems in operation. *IEEE Transactions on Software Engineering,* **SE-10** (6) 701-714.

Littlewood, B. (1979) Software reliability model for modular program structure. *IEEE Transactions on Reliability*, **R-28** (3) 241-246.

Mazzuchi, T. A. and Soyer, R. (1993) A Bayes method for assessing product-reliability during development testing. *IEEE Transactions on Reliability*, **42**(3) 503-510.

Munson, J. C. (1996) A Software Blackbox Recorder. *Proceedings of the 1996 IEEE Aerospace Applications Conference*, IEEE Computer Society Press, Los Alamitos, CA, November, 309-320.

Munson, J. C. (1995) Software measurement: problems and practice. *Annals of Software Engineering,* **1**(1) 255-285.

Munson, J. C. and Khoshgoftaar,T. M. (1992) The detection of fault-prone programs. *IEEE Transactions on Software Engineering*, **SE-18** (5)423-433.

Munson, J. C. and Ravenel, R. H. (1993) Designing reliable software. *Proceedings of the 1993 IEEE International Symposium of Software Reliability Engineering*, IEEE Computer Society Press, Los Alamitos, CA, November, 45-54.

Musa, J. D. (1992) The operational profile in software reliability engineering: an overview. *Proceedings of the IEEE International Symposium on Software Reliability Engineering*, IEEE Computer Society Press, Los Alamitos, CA, 140-154.

Siegrist, K (1988a) Reliability of systems with Markov transfer of control. *IEEE Transactions on Software Engineering,* **SE-14** (7) 1049-1053.

Siegrist, K (1988b) Reliability of systems with Markov transfer of control, II. *IEEE Transactions on Software Engineering,* **SE-14** (10) 1478-1480.

Wilks, S.S. (1962) *Mathematical Statistics.* John Wiley and Sons, Inc., New York.