Estimation and Enhancement of Real-Time Software Reliability through Mutation Analysis [*]

Robert M. Geist
A. Jefferson Offutt
Frederick C. Harris, Jr.

Department of Computer Science
Clemson University
Clemson, South Carolina
USA 29634-1906
phone: 803-656-2258
fax: 803-656-0145
email: rmg@cs.clemson.edu

Abstract

A simulation-based method for obtaining numerical estimates of the reliability of N-version, real-time software is proposed. An extended stochastic Petri net is used to represent the synchronization structure of N versions of the software, where dependencies among versions are modeled through correlated sampling of module execution times. The distributions of execution times are derived from automatically generated test cases that are based on mutation testing. Since these test cases are designed to reveal software faults, the associated execution times and reliability estimates are likely to be conservative. Experimental results using specifications for NASA's planetary lander control software suggest that mutation-based testing could hold greater potential for enhancing reliability than the desirable but perhaps unachievable goal of independence among N versions. Nevertheless, some support for N-version enhancement of high quality, mutation-tested code is also offered. Experimental results on data diversity, in which retry with a mutation-directed variation in input is attempted after system failure, suggest that mutation analysis could also be valuable in the design of fault-tolerant software systems.

keywords: software reliability, Petri nets, correlated sampling, mutation analysis, constraint-based test data generation, data diversity

[*]This work was partially supported by NASA Langley Research Center under grant NAG-1-1024.

1 Introduction

The use of multi-version software to improve computer system reliability remains a topic of vigorous debate [2, 15, 17, 18]. One cause for concern is easily seen in considering a simple model of majority voting: if each of three voters independently votes "yes" (meaning a *correct* vote) with probability p, then the probability of a majority "yes" decision is $3p^2 - 2p^3$, which is larger than p when p is larger than $1/2$. However, if the votes are perfectly correlated, then the probability of a majority "yes" decision is just p itself, and there is no improvement.

Thus the issue is readily identified as "version correlation," but the meaning of this phrase in the software development environment can be elusive. A substantial clarification was provided by Eckhardt and Lee [12] and by Littlewood and Miller [19]. Using Littlewood's notation [19], we let random variable X represent an input to any of a collection P of programs designed to perform the same task, and let $\Theta(x)$ be the probability that a randomly chosen program from P fails on specific input x. The expected value of the random variable $\Theta(X)$, $E[\Theta(X)]$, is then the probability that a randomly chosen program fails on a randomly chosen input. The key observation from [12, 19] is that *independently developed programs* do not necessarily fail independently. The probability that two independently chosen programs from P both fail on the same randomly chosen input is easily seen to be $E[\Theta(X)^2]$, but the probability of both failing on independently chosen inputs is $E[\Theta(X)]^2$, and these terms differ by the variance, $V[\Theta(X)]$. Similarly, if $\Theta_A(x)$ is the probability that a randomly selected program from development methodology A fails on input x, and $\Theta_B(x)$ the same for methodology B, then the probability that both independently selected programs fail on the same randomly chosen input is $E[\Theta_A(X)\Theta_B(X)]$, which differs from independent failures, $E[\Theta_A(X)]E[\Theta_B(X)]$, unless the covariance, $COV[\Theta_A(X), \Theta_B(X)] \equiv E[\Theta_A(X)\Theta_B(X)]$ - $E[\Theta_A(X)]E[\Theta_B(X)]$, is zero. A zero covariance is clearly equivalent to a zero correlation, $\rho[\Theta_A(X), \Theta_B(X)] \equiv COV[\Theta_A(X), \Theta_B(X)]/\sqrt{V[\Theta_A(X)]V[\Theta_B(X)]}$, and thus it is this correlation that deserves attention.

A second major concern, which precludes lending precise quantification to software reliability, is the unknown operational distribution on the input space. Testing based on random (uniform) sampling from the input space is often carried out, but reliability estimates based thereon are usually regarded as optimistic. A complementary technique for obtaining non-trivial, conservative estimates has been missing.

In this paper we address both concerns. We propose a method for estimating the reliability of multi-version, real-time software that incorporates non-zero correlation as an independent model parameter. In section 2 we develop an Extended Stochastic Petri Net [11] as a representation of the synchronization structure of N software versions, where dependencies among version performances are captured through correlated sampling of module execution times. The execution time distributions are estimated by executing the modules on test case sets that are automatically generated from the module source code. This automated test case generation is based on *mutation testing* and is described in section 3. Since the test cases generated in this way are designed to reveal faults, they can be regarded as "stressful input," and thus we suggest that the execution time profile and the derived reliability estimate should be regarded as conservative. In section 4 we describe the results (MTTF) of applying the proposed method to a 5-version implementation of the accelerometer sensor processing module in NASA's planetary lander control software.

We expect the real benefit of the optimistic and conservative pair of reliability estimates provided by random testing and our proposed method will be in gauging the relative merits of techniques designed to enhance reliability. One such technique, providing programmers with mutation-generated I/O pairs, is examined in section 5. A second technique, a mutation-directed variation on Ammann and Knight's *data diversity* [1], is discussed in section 6. Conclusions follow in section 7.

2 The Synchronization Model

We consider a program *module* to be a self-contained piece of software, typically a subroutine, function, or small program. Thus, our results impact *unit* testing, rather than *integration* testing.

The restriction to the real-time software environment allows us to focus reliability estimation on execution time profiles. A module fails if it fails to produce the correct output within a tightly specified time interval. Even well-tested programs may fail to meet timing constraints on some input, which has a major impact on system reliability. In our model, we combine *timing failures*, where the output is not produced in time, and *functional failures*, where the output is incorrect, by considering a functional failure to be an infinite-time response, which necessarily violates the timing constraint. We recognize that this is a simplification in the representation of functional failures, and that we are no longer able to distinguish among the different types of functional failures. However, as shown later, the two classes of failures, timing and functional, can be separated when necessary for analysis. We then define *module execution time* to be the time until a correct answer is produced. Thus a distribution of module execution times will have points at $+\infty$ that represent functional failures.

We represent the synchronization of our N-version software system in terms of these execution time distributions. This will allow us to represent correlation between modules as an independent model parameter.

We specify the concurrent operation of the N-versions, the timer, and the voting mechanism by a special type of Extended Stochastic Petri Net [11]. A Petri net is a directed bipartite graph whose two vertex sets are called *places* and *transitions*. Places are traditionally represented by circles and transitions by rectangles. Places may contain one or more *tokens*, represented by small discs. An example is shown in figure 1. The semantics attached to such nets are rules for simulation:

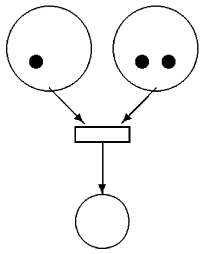

Figure 1: Simple Petri Net.

- If every input place to a transition contains one or more tokens, the transition is *enabled*; the transition of figure 1 is enabled.
- Enabled transitions may *fire*, that is, remove one token from each input place and add one token to each output place.
- If firing an enabled transition would disable a concurrently enabled transition (conflict), the firing transition is chosen at random.

Petri nets and their extensions [20, 21, 22] have been used by many authors in systems performance modeling [3, 10, 14] and reliability modeling [23, 26].

We augment Petri nets with two extensions. The first extension we need is non-zero firing time distributions. Specifically, we attach to each transition a distribution of firing times. When a transition is enabled, we randomly select a firing time from the attached distribution. After the selected time elapses, the transition fires if it is still enabled. In our model, the firing time distributions of transitions representing program versions will be empirical execution time profiles obtained by executing the programs on automatically selected test cases.

The second extension we require is *correlated firing.* Since the correlation of interest is the degree to which all programs perform relatively well or relatively poorly on a given input, rather than any linear relationship among their real execution times, we implement this correlated firing as a correlation of *firing time distribution percentiles.* If two programs are to be highly correlated, we wish to choose an execution time from the same relative location in each distribution. Since these distributions contain points at $+\infty$ representing functional failures, two programs with correlation 1 will then exhibit the same functional behavior during the same execution, i.e. both will succeed or both fail.

In this implementation transitions are grouped. (One group will contain N transitions representing the program versions.) When transition i becomes enabled, it locates all other concurrently enabled transitions in its group. If there are none, a random number $r_i \in [0,1]$ is used to select a firing time in the standard way, that is, *firing time* $= F_i^{-1}(r_i)$, where F_i is the distribution function for transition i. However, if there are other enabled transitions in i's group, one of these, call it j, is selected at random. The most recent value $r_j \in [0,1]$ that was used to select a time from j's distribution is used together with a group correlation factor, K, to determine the new selection value $r_i \in [0,1]$. With probability K we let $r_i = r_j$, and with probability $1-K$ we select r_i at random from $[0,1]$.

The relationship between the selection values r_i and r_j is easily expressed. If R_j is a random variable with uniform (0,1) density, f_{R_j}, and R_i is a random variable whose dependence on R_j is as described, then the conditional density of R_i is

$$f_{R_i|R_j}(r_i|r_j) = K\delta(r_i - r_j) + (1-K)f_{R_j}(r_i)$$

where δ denotes the unit impulse function (see [24]). We then have

$$\begin{aligned} E[R_i|R_j = r_j] &= Kr_j + (1-K)/2, \\ E[R_iR_j] &= K/3 + (1-K)/4, \\ V[R_i] &= 1/12, \\ V[R_j] &= 1/12, \\ COV[R_i, R_j] &= K/12, \end{aligned}$$

and thus the correlation of R_i and R_j is

$$\rho[R_i, R_j] = K.$$

A graphical Petri net simulation tool that includes arbitrary firing time distributions and correlated firing, called XPSC, has been developed at Clemson University and was used in this study.

Our multiversion software model is shown in figure 2 with its initial token assignment. The distribution attached to each transition represented by a thin rectangle is instantaneous, i.e. time 0 with probability 1. The firing time attached to the *timer* transition is a deterministic parameter to the model, T. Execution of the 5-version software is represented by transitions $exec_1$, ..., $exec_5$. Any $exec_i$ that completes (returns correct output) deposits a token in place *count*, which causes one of the *correct* transitions to fire. When the *timer* completes, a vote is taken and a token is either deposited in place *success* (if at least three $exec_i$s have fired) or in place *failure* (if two or fewer $exec_i$s have fired). Place *failure* represents system failure. A token in place *success* causes

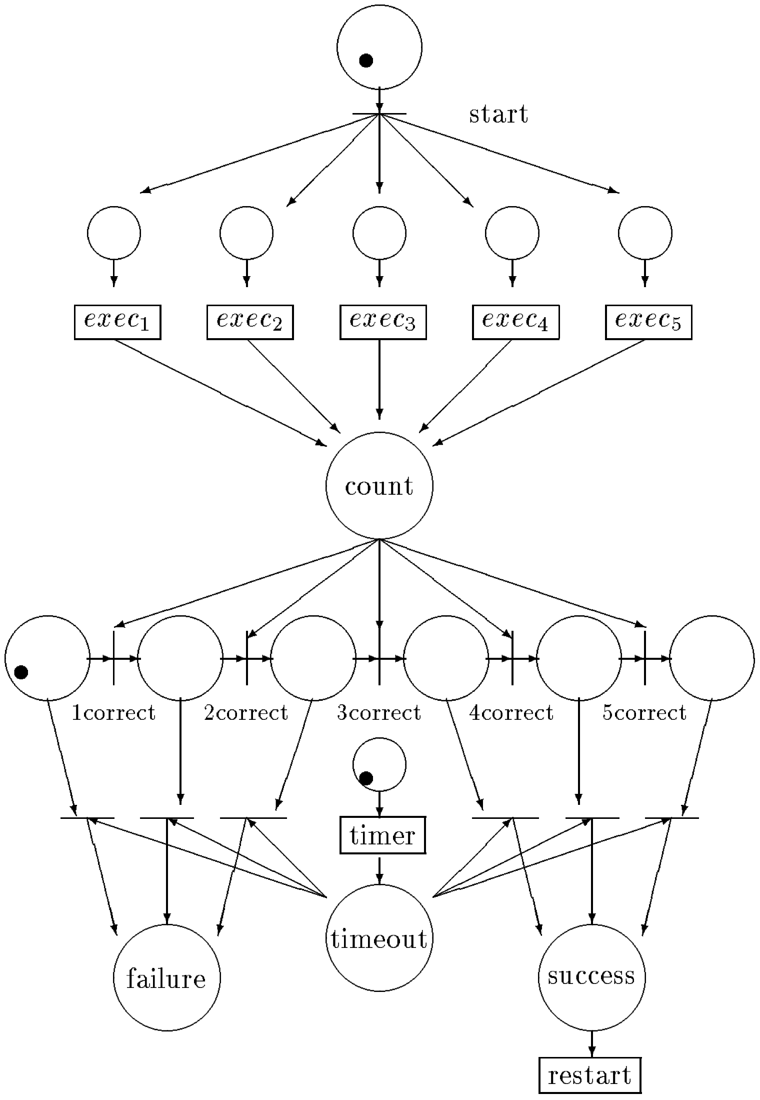

Figure 2: Multi-version Software Model.

a special *restart* transition to fire, which reinitializes the net but not the system clock (a process regeneration point). Restart transitions are also available in XPSC.

The group of $exec_i$ transitions is a correlated group with correlation factor K, another parameter of the model. Thus a long (or short) firing time for an $exec_i$, representing difficult (or trivial) input, can cause all others to select relatively long (or short) firing times if the correlation is high.

Our measure of interest is the mean time to reach system failure state. We note that if all correlations are zero, we can compute MTTF analytically. If $F_i(T)$, $i = 1, 2, ..., 5$, is the distribution function of $exec_i$ evaluated at the fixed timeout interval T, then the probability that we reach place *success* on the first pass is

$$p = \sum \prod_{i=1}^{5} F_i(T)^{n_i}(1 - F_i(T))^{1-n_i}$$

where the sum is taken over $\{(n_1, ..., n_5) | n_i \in \{0, 1\}, \sum n_i \geq 3\}$. The number of passes before failure is geometrically distributed with parameter p, so that MTTF $= T/(1 - p)$. This provides a convenient check on the XPSC simulation results, which should converge to this value as the correlation K approaches 0.

It is important to observe that the case $K = 1$, where we sample from precisely the same relative performance levels, corresponds to the "natural" synchronization of the $exec_i$ versions on identical inputs only if these $exec_i$ versions exhibit a natural high correlation of performance on all the inputs giving rise to the $exec_i$ distributions. Such structure allows us to use whatever natural correlation exists among versions and still treat correlation as an independent model parameter.

The distributions used for the $exec_i$ transitions in the study were measured execution times of five independently developed implementations of the accelerometer sensor processing module of NASA's planetary lander control software, described in section 4. The input cases on which each module was executed were generated by an automated system that is based on mutation testing.

3 Mutation Testing

Generating test cases that are effective at finding faults is a technically difficult task. One important criterion for generating test data is *relative adequacy* as defined by DeMillo et al [7]:

Definition. If P is a program to implement function F and Π is a collection of programs, then test set T is adequate for P relative to Π if P(t)=F(t) $\forall$ t$\in$T, and $\forall$ Q $\in \Pi$, Q $\neq$ F $\Rightarrow \exists$ t$\in$T such that Q(t)$\neq$F(t).

In other words, a test set is adequate if it distinguishes the program being tested from a set of incorrect programs. *Mutation testing* [7] is a testing technique based on relative adequacy. It can be regarded as a software analogue of the hardware fault-injection experiment. Mutation testing systems apply a collection of *mutation operators* to the test program, each of which produces a set of executable variations, called *mutants*, of the original program.

Test cases are used to cause the mutants to generate incorrect output. Mutant programs that have been shown to be incorrect by a test case are considered "dead" and are not executed against subsequent test cases. Some mutants are functionally equivalent to the original program and cannot be killed. The mutation score of a test set is then the percentage of non-equivalent mutants that are dead. If the total number of mutants is M, the number of dead mutants is D, and the number of equivalent mutants is E, the mutation score is calculated as:

$$MS(P,T) = \frac{D}{(M-E)}.$$

This mutation score is a close approximation of the adequacy of a set of test data; a test set is *relative adequate* if its score is 100% (all mutants were killed). The goal of mutation testing is to find test data to kill all mutants. The assumption is that such test data will provide a strong test of the original program.

The effectiveness of this approach is based upon a fundamental premise: if the software contains a fault, it is likely that there is a mutant that can only be killed by a test case that also reveals the fault. Mutation-adequate tests have been shown experimentally [4, 13] and analytically [5] to be high quality tests.

The most recent mutation system is Mothra [6], which allows a tester to examine remaining live mutants and design tests that kill them. The mutation operators used by Mothra [16] represent more than 10 years of refinement through several mutation systems. These operators explicitly require that the test data meet statement and (extended) branch coverage criteria, extremal values criteria, and domain perturbation; the mutation operators also directly model many types of faults.

Unfortunately, generating mutation-adequate tests can be a labor-intensive task. To solve this problem, Offutt [9] devised an adequacy-based scheme for automatically generating test data through a Constraint-Based Testing (CBT) system. In constraint-based testing, we represent the conditions under which each mutant will die as mathematical constraints on the inputs, and then generate a test case that satisfies the constraint system. An implementation of this technique, Godzilla, has been integrated with Mothra.

The Godzilla system develops test data to detect the same classes of faults that the Mothra software testing system models. Godzilla generates test data by posing the question, "What properties must the program inputs have to kill each mutant?" The inputs must cause the mutant to have an incorrect program state after some execution of the mutated statement. Godzilla uses the same syntactic information that Mothra uses to force the syntactic change represented by the mutation into making a semantic difference during execution. Since faults are modeled as simple faults on single statements, an initial condition is that the mutated statement must be reached (reachability). A further condition is that once the mutated statement is executed, the test case must cause the mutant to behave erroneously, i.e. the fault that is being injected must result in a failure in the program's output.

Godzilla describes these conditions on the test cases as mathematical systems of constraints. The reachability condition is described by a system of constraints called a *path expression.* For each statement in the program, the path expression contains a constraint system that describes each execution path through the program that will reach that statement.

The condition that the test case must cause an erroneous state is described by a constraint that is specific to the mutation operator. In general, this *necessity constraint* requires that the computation performed by the mutated statement create an incorrect intermediate program state. Although an incorrect intermediate program state is necessary for fault detection, it is clearly not sufficient to guarantee detection. In order to detect the fault, the test case must cause the mutant to produce incorrect output, in which case the final state of the mutant differs from that of the original program. Although deriving a test case that meets this *sufficiency condition* is certainly desirable, it is impractical. Determining the sufficiency condition implies knowing the complete path that the program will take, which is intractable. Thus, the partial solution of relying on the necessity condition is used.

To generate the test cases, Godzilla solves the conjunction of the path expression constraint with the necessity constraint to create a test case consisting of values for the input variables that will make the constraints true. Godzilla consistently generates test cases that kill over 95% of the mutants [8].

4 A Correlated Sampling Experiment

NASA's planetary lander control software is designed as an N-version voting system. Five implementations $(M_1, M_2, ..., M_5)$ of the accelerometer sensor processing module for this system were independently constructed and tested by programmers. A separate version, M_0, was written and tested using Mothra. This version had 55 lines and 3778 non-equivalent mutants. Godzilla generated 30 test cases to kill all 3778 mutants. The correctness of M_0 on the 30 cases was verified by hand so that M_0 was a correct, or "oracle" version of the modules. The average execution time of each module M_i on each test case was measured. When a module returned incorrect output on a test case, its execution time was considered to be infinite. These empirical execution time distributions were attached to the $exec_i$ transitions of the Petri net model described in section 2. All $exec_i$ transitions were placed in a single group with correlation factor K.

Since the test data was generated using the Godzilla generator, the empirical execution time distributions likely differ from those expected under normal conditions. These input sets attempt to maximize the likelihood of the module failing to produce correct output. Thus a conservative reliability estimate is suggested. Additionally, we do not allow for multiple correct answers (for example, that differ by an inconsequential value). This is a limitation of our implementation (specifically, of the Mothra system), not of the reliability model itself. Since this means that we may label some correct answers incorrect, this limitation contributes to the conservativeness of the reliability estimate.

The software used for this experiment was based on specifications provided by NASA [25]. These specifications are quite thorough with regard to what must be accomplished in each module and what the parameters must be. The module selected has tight timing requirements and reasonable computational complexity. The module specifications are summarized in figure 3. The module takes input from the accelerometer sensor (A_Counter), removes its natural electrical and temperature bias (steps 1 and 2), and produces accelerations in all three physical body directions.

Transforming accelerometer data (A_COUNTER) into vehicle accelerations (A_ACCELERATION) is a time-dependent operation that must execute as quickly as possible. The function requires the following steps:

1. A_GAIN := A_GAIN_0 + (G1 * ATMOSPHERIC_TEMP) + (G2 * ATMOSPHERIC_TEMP^2)
2. A_ACCELERATION_M := A_BIAS + A_GAIN * A_COUNTER
3. Shift A_STATUS and A_ACCELERATION right by 1 column.
4. A_ACCELERATION := ALPHA_MATRIX * A_ACCELERATION_M
5. For each row of A_STATUS and A_ACCELERATION: If any of the previous three values of A_STATUS is *unhealthy*, set A_STATUS(i,0) to *healthy*. Otherwise, for each of the three dimensions in A_ACCELERATION calculate the mean (μ) and the standard deviation (σ). If $|\mu$ - A_ACCELERATION(i,0)$|$ $>$ A_SCALE*σ, then set A_ACCELERATION(i,0) to μ and set A_STATUS(i,0) to *unhealthy*.

Figure 3: ASP Specification Summary.

The 5 programmers were given input data types and ranges, but not specific test data. They were also told that the module must satisfy very tight timing constraints. The 5 Fortran-77 versions of the module ranged from 42 to 55 lines of code, and were run on Sun 3/50s under SunOS version 4.1. Although we did monitor programmer debugging, we did not perform any acceptance testing.

Execution times for correct output of the 5 versions ranged from 89 to 151 milliseconds. As stated before, incorrect answers were penalized with an effective time of $+\infty$, yet, of the 30 test cases for each of the 5 versions (150 total responses), only 20 responses were incorrect. The number of functional failures per version was 0, 1, 1, 5, and 13.

The only parameters to the synchronization model were the value of the timing constraint, T, and the correlation among $exec_i$ modules, K. In figure 4 we show the mean time to failure (MTTF) as a function of the timing constraint T for a range of correlation factors. Each curve is a spline of 63 datapoints evenly spaced across the T range [89, 151] ms. Since the longest correct execution time was 151 ms, the right-hand endpoints of the curves in figure 4 show the reliability estimates when only functional errors are considered. The figure also shows that when the timing constraint is very tight all versions of the software will fail immediately, i.e. after one timer interval. We note extreme sensitivity of MTTF to correlation in the range $0 < K \leq 0.3$, where we see a precipitous drop in MTTF over a wide range of timing constraints. If this trend holds for software in general, then this is a cause for concern, since this study causes us to expect programs in a natural environment to correlate to at least this degree. For example, if we use a specific value of

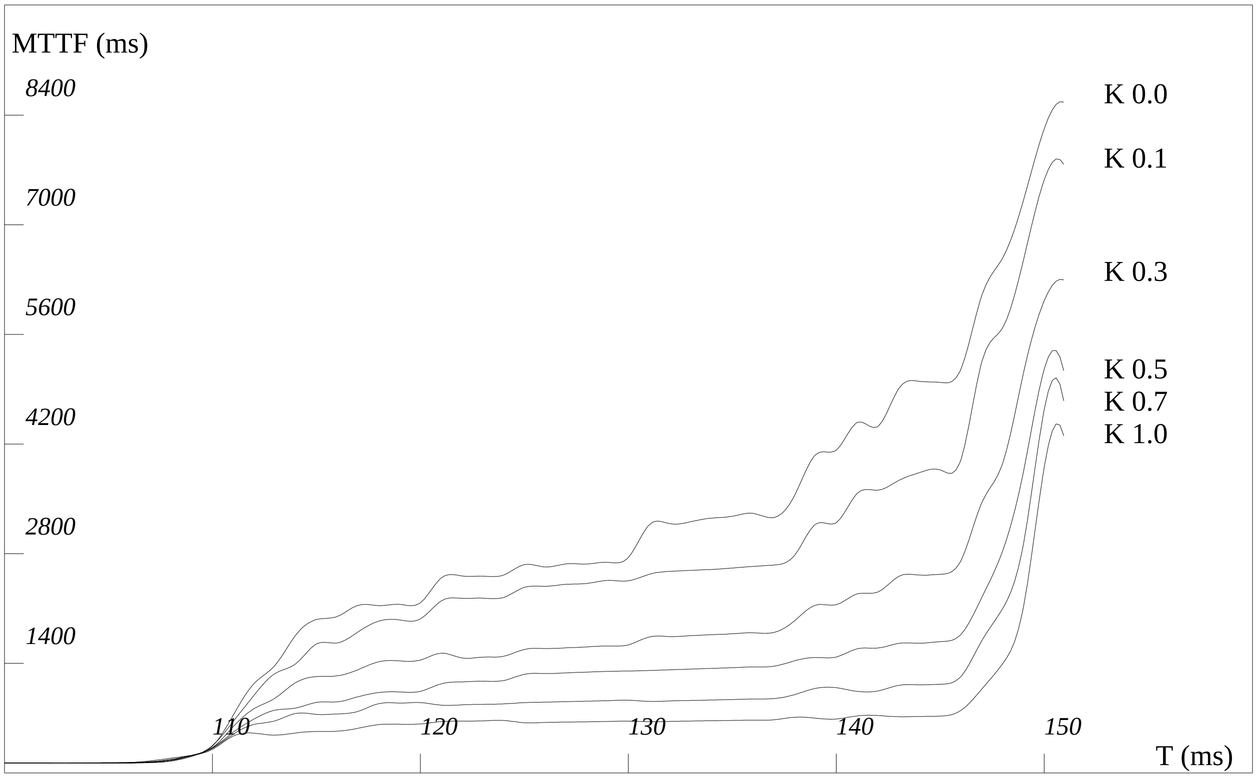

Figure 4: MTTF of Correlated Samples with Mutation Test Data.

the timing constraint, T, to transform each execution time random variable to binary (success is correct execution in time $\leq T$), then we can measure sample correlation of success/failure for the 10 possible pairings of programs on the sequence of automatically generated test data. A choice of $T = 110\ ms$ yielded 92 successes, 58 failures (30 test cases $\times$ 5 versions), and an average pairwise correlation of 0.200498.

To lend evidence to our conjecture that these reliability estimates are conservative, we ran the experiment again using random test cases in place of the Godzilla-generated test cases. We assumed a uniform distribution over the legal input space and selected 100 test cases at random. All five versions were executed on each of the 100 test cases, and an execution time (time to correct output) distribution for each was recorded. The synchronization model was run again using these new execution time distributions. The results are shown in figure 5, where we see a dramatic increase in estimated MTTF for all correlation and timing constraint values, indicating that random testing did not find as many faults as mutation testing. Average pairwise sample correlation of success/failure at $T = 110\ ms$ in this case was 0.157524, which was derived from 284 successes and 216 failures (100 test cases $\times$ 5 versions). The two sets of curves in figures 4 and 5 serve as estimated bounds on the reliability of our 5-version voting system. They also provide a gauge by which we might judge the effectiveness of reliability enhancement techniques.

5 An Enhancement

Programmers typically respond to test results by modifying their code to correct the faults found during testing, which can naturally be expected to improve the reliability. In a second experiment, each of the five programmers was provided with the 30 automatically generated test cases, including

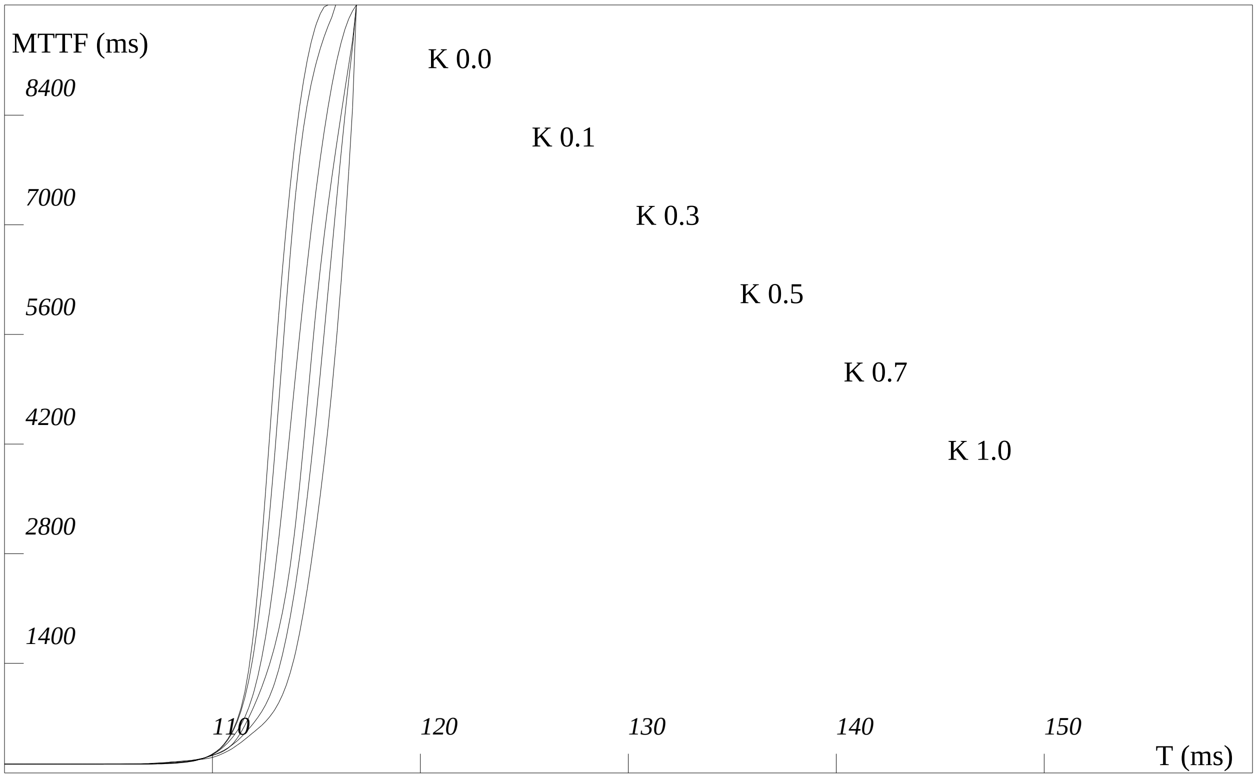

Figure 5: MTTF of Correlated Samples with Random Test Data.

the correct outputs, and given the opportunity to improve module code. The modules were changed, sometimes extensively, until they worked correctly on all 30 test cases. The automated test case generator was then run again to provide an alternative set of test cases with 100% mutation kill score. The new set also contained 30 test cases. Although this new set satisfied the same mutation criteria, the randomness inherent in the constraint satisfaction mechanism ensured different values.

Each of the five "improved" modules was executed on all 30 of the new test cases, and an execution time (time to correct output) distribution for each was recorded. The synchronization model was executed again using these new distributions. The results are shown in figure 6, where we see a dramatic improvement in reliability (MTTF) over that exhibited in figure 4.

As a control, the five original modules were also executed on these new test cases, and the synchronization model was run with the resulting execution time distributions. The results are similar to those in figure 4, but report slightly lower MTTF. This indicates that the new test cases represent at least as stringent a test as the original set, as well as indicating that the MTTF improvement in figure 6 over figure 4 genuinely indicates a substantial improvement in reliability.

We note that although the relative effects of correlation are still visible in figure 6, the correlation $K = 1.0$ curve of figure 6 lies entirely above the correlation $K = 0.0$ curve of figure 4. Since an N-version system with $K = 1.0$ is equivalent to a single version, an interpretation is that any single version of code for which specification includes matching mutation-generated I/O pairs may be more reliable than a 5-version voting system in which testing is under the complete control of the individual unit developers. The results also suggest that constraint-based testing could hold greater potential for reliability enhancement than the desirable, but perhaps unachievable, goal of independence among N versions. Nevertheless, an alternative interpretation would lend support to N-version programming. It may be unfair to compare the higher quality code represented by figure 6 with that represented by figure 4, since testing in the latter case was uncontrolled. If we

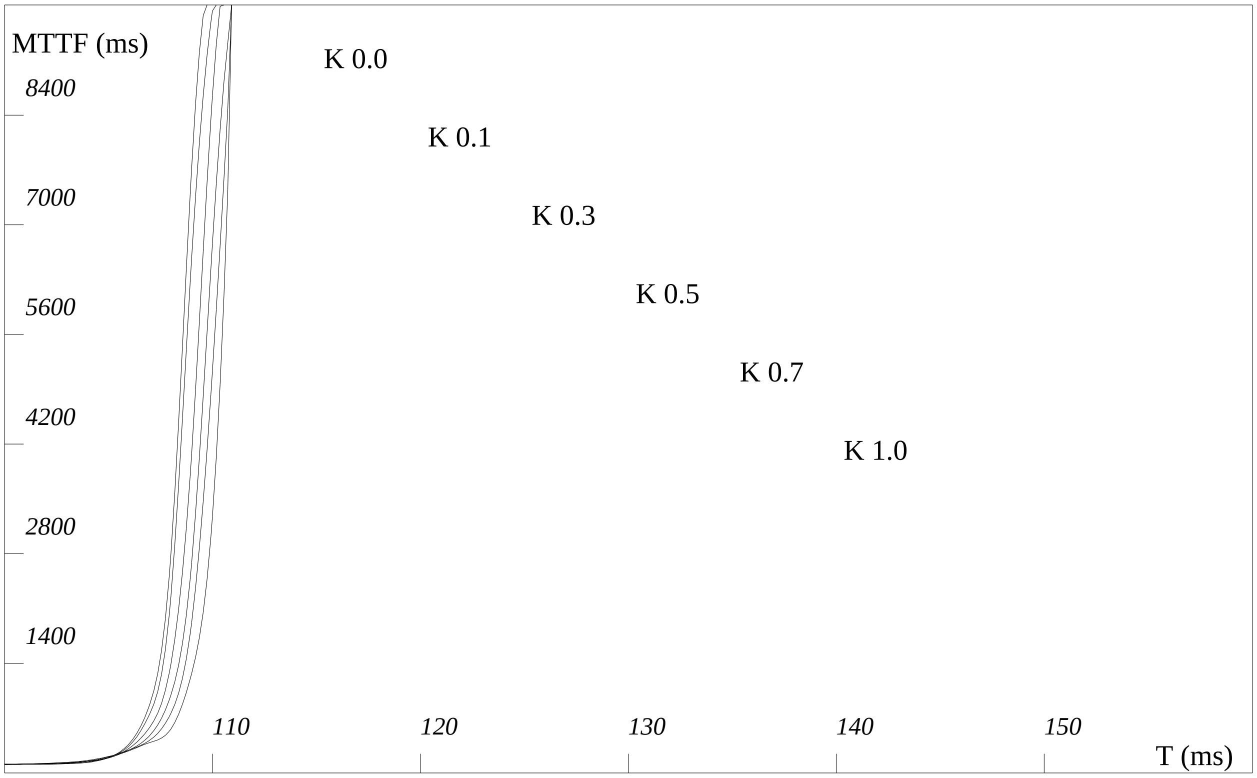

Figure 6: MTTF of Improved Code with New Mutation Test Data.

focus only on the higher quality code represented by figure 6, we see that for any specific timing constraint T within the range [105, 110] ms successful N-version programming (K small) *does* offer substantial improvement over the single version case ($K = 1.0$).

We emphasize that the present experiment is a very simple one, and a complete case study will be needed before we can attempt to draw definitive conclusions. For example, since random testing did not find as many faults as mutation testing (see figure 5), we can conclude that random testing is unlikely to hold as much potential for reliability enhancement. However, we cannot conclude that presenting programmers with 30 test cases generated by an alternative, non-random scheme (e.g. data flow testing or functional testing) would have been less effective than using the mutation-generated cases.

6 Mutation Gradient Retry

Mutation analysis may also prove effective in the design of fault-tolerant software. In a real-time environment, sensor data often contains noise. Ammann and Knight [1] have proposed retrying on failures with forced minor variations in input data. This technique, called *data diversity*, could be an effective means of providing software fault tolerance. Missing from their approach is any guidance on the direction of data variation. If we find that a minor variation caused us to enter a domain region of lesser mutation kill potential, as identified by the constraints generated during testing, then we could reasonably expect less data sensitivity and hence a greater probability of success on that retry.

To explore this possibility within the present testing framework, we isolated a particularly troublesome mutation-generated test case on which 4 of 5 original versions exhibited functional failure. (No single test case caused all versions to fail.) We fixed all input values at this point except for the values of $G1$ and $G2$ (see specifications, figure 3). By randomly varying $G1$ and $G2$ across their legal ranges (-5.0, 5.0) we obtained 15,000 system failure points, shown in figure 7. These are points where 3 or more versions exhibited functional failure. This diagram is strikingly

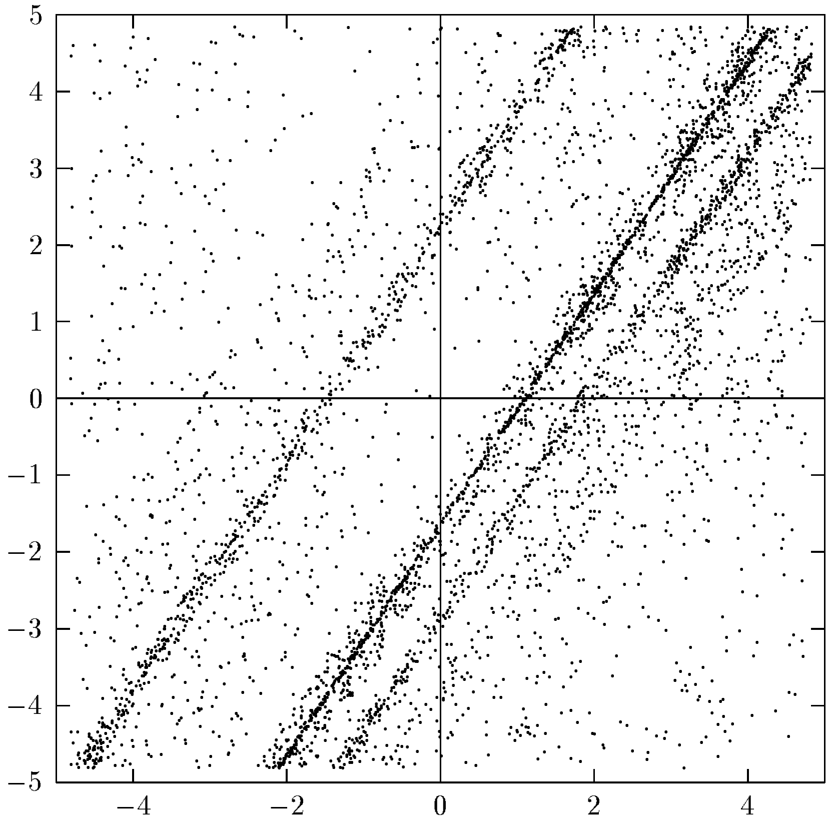

Figure 7: Failure Points in (G1,G2) Space.

similar to those of [1] in suggesting that high density failure regions form hyperplanes in the input space.

We subjected each of the 15,000 failure points to a system retry with forced variation in $G1$ and $G2$. We varied the point $(G1, G2)$ in a random direction, as per [1], and in the direction of the downward mutation gradient, determined from mutation kill scores over a lattice of points in $(G1, G2)$ space. This process was automated, but computationally expensive. The number of failures surviving retry, as a function of distance moved, is shown in figure 8. We see random retry offered an order of magnitude reduction in failures (from 15,000 to <1,500), but that mutation-gradient retry fared significantly better. We have also plotted in figure 8 results of retry in the direction of the upward mutation gradient. If the mutation gradient generally points orthogonally to failure hyperplanes, then upward gradient movement should serve as well as downward, which is certainly supported by the results of this test.

7 Conclusions

We have presented a method for obtaining numerical estimates of the reliability of N-version, real-time software. The method uses an extended stochastic Petri net to represent the synchronization

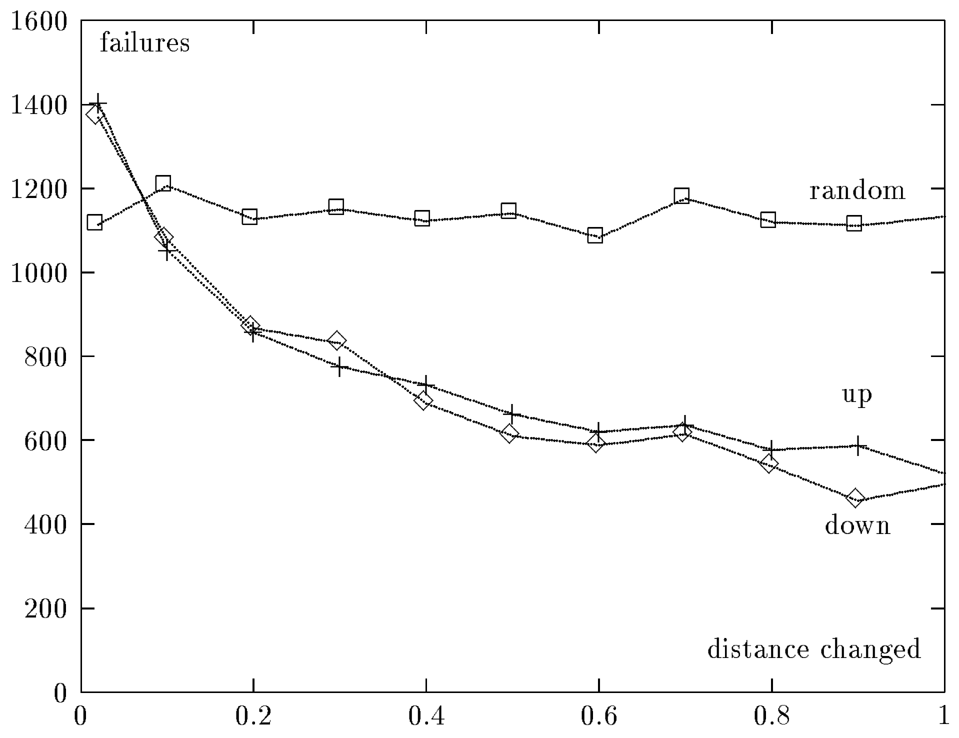

Figure 8: Failures Surviving Retry.

structure of the N versions (software modules). Net simulation is used to estimate system reliability under different correlation assumptions.

Module execution time distributions are derived by executing the modules on test cases that are generated by a mutation testing system. Since these test cases are designed to reveal faults, and hence represent stressful input, we contend that such execution time distributions are likely to deliver a conservative estimate of system reliability. A major advantage of this approach is that it allows us to directly incorporate results from software testing into the reliability measure.

We conducted a simple empirical investigation of this approach using five independently developed implementations of the accelerometer sensor processing module from NASA's planetary lander control software. We observed a rapid decrease in estimated reliability of the 5-version system as the correlation increased over the narrow range of $0.0 < K \leq 0.3$. This is disturbing, in that it suggests that even relatively low correlation among versions could prevent significant reliability enhancement through N-way voting alone.

We also considered a technique for reliability enhancement in which programmers were given mutation-generated I/O pairs and required to match them. The resulting dramatic improvement in estimated system reliability suggests that constraint-based testing may hold greater potential for reliability enhancement than achieving independence among N versions. Nevertheless, we find that successful N-version programming (K small), when applied to the improved modules, does offer additional significant reliability enhancement for a range of tight timing constraints.

We emphasize that our coding experiment was a very simple one, and a complete case study will be needed before we can attempt to draw definitive conclusions.

Finally, we considered an application of mutation analysis to the design of fault-tolerant software. We modified Ammann and Knight's data diversity technique, which calls for retry on system failure with forced minor random variation in input data, to incorporate a specific variation direction determined by the mutation kill gradient. Results show a significant reduction in system failures. However, this study considered only a two-dimensional cross section of the input space and did not consider effects of multiple retries. An extended study that incorporates the entire input domain as well as the effects of timing constraints on limiting multiple retries is warranted.

References

[1] P. Ammann and J. Knight. Data diversity: An approach to software fault-tolerance. *Proc. 17th Int. Symp. on Fault-Tolerant Computing (FTCS-17)*, pages 122–126, Pittsburgh, PA, July, 1987.

[2] A. Avižienis. The n-version approach to fault-tolerant software. *IEEE Trans. Soft. Engr.*, SE-11(12):1491–1501, December 1985.

[3] G. Balbo, S. Bruell, and S. Ghanta. Combining queueing networks and generalized stochastic petri nets for the solution of complex models of system behavior. *IEEE Trans. on Comp.*, **37**, 1988.

[4] T. A. Budd. *Mutation Analysis of Program Test Data.* PhD thesis, Yale University, New Haven CT, 1980.

[5] T.A. Budd and D. Angluin. Two notions of corrections and their relation to testing. *Acta Informatica*, 18(1):31–45, November 1982.

[6] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An extended overview of the Mothra software testing environment. In *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, Banff Alberta, July 1988. IEEE Computer Society Press.

[7] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4), April 1978.

[8] R. A. DeMillo and A. J. Offutt. Experimental results of automatically generated adequate test sets. In *Proceedings of the Sixth Annual Pacific Northwest Software Quality Conference*, pages 209–151, Portland OR, September 1988. Lawrence and Craig.

[9] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Trans. Soft. Engr.*, 17(9):900–910, September 1991.

[10] J. Dugan, A. Bobbio, G. Ciardo, and K. Trivedi. The design of a unified package for the solution of stochastic petri net models. *Proc. Int. Workshop on Timed Petri Nets*, Torino, 1985.

[11] J.B. Dugan, K.S. Trivedi, R.M. Geist, and V.F. Nicola. Extended stochastic petri nets: Applications and analysis. *Proc. 10th Int. Symp. on Computer Performance (PERFORMANCE 84)*, pages 507–520, December, 1984.

[12] D. Eckhardt and L. Lee. A theoretical basis for the analysis of multi-version software subject to coincident errors. *IEEE Trans. Soft. Engr.*, **SE-11**, 1985.

[13] M.R. Girgis and M.R. Woodward. An experimental comparison of the error exposing ability of program testing criteria. In *Proceedings of the Workshop on Software Testing*, pages 64–73. IEEE Computer Society Press, July 1986.

[14] M. A. Holliday and M. K. Vernon. A generalized timed Petri net model for performance analysis. *IEEE Transactions on Software Engineering*, pages 1297–1310, December 1987.

[15] J. Kelly and A. Avižienis. A specification-oriented multi-version software experiment. *Proc. FTCS-13*, Milan, 1983.

[16] K. N. King and A. J. Offutt. A Fortran language system for mutation-based software testing. *Software Practice and Experience*, 21(7):686–718, July 1991.

[17] J. Knight and N. Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Trans. Soft. Engr.*, **SE-12**, 1986.

[18] J. Knight and N. Leveson. An empirical study of failure probabilities in multi-version software. *Proc. of FTCS-16*, Vienna, 1986.

[19] B. Littlewood and D. Miller. A conceptual model of multi-version software. *Proc. FTCS-17*, Pittsburgh, 1987.

[20] M. Marsan, G. Conte, and G. Balbo. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Trans. on Comp. Sys.*, **2**:93–122, 1984.

[21] Michael K. Molloy. Performance analysis using stochastic Petri nets. *IEEE Trans. Comput.*, C-31(9):913–917, September 1982.

[22] J.L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[23] Y-B. Shieh, D. Ghosal, and S. Tripathi. Modeling of fault-tolerant techniques in hierarchical systems. *Proc. FTCS-19*, Chicago, 1989.

[24] K.S. Trivedi. *Probability and Statistics with Reliability, Queueing, and Computer Science Applications.* Prentice-Hall, Englewood Cliffs, NJ, 1983.

[25] B. Withers, D. Rich, D. Lowman, and R. Buckland. Software requirements: Guidance and control software development specification. *NASA Contractor Report 182058*, NASA Langley Research Center, June, 1990.

[26] T. Yoneda, K. Nakade, and Y. Tohma. A fast timing verification method based on the independence of units. *Proc. FTCS-19*, Chicago, 1989.