The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software

Ricky W. Butler
George B. Finelli

NASA Langley Research Center
Mail Stop 130
Hampton, VA 23665–5225
(804)864-6198
Arpanet address: rwb@air12.larc.nasa.gov

Abstract

This paper affirms that the quantification of life-critical software reliability is infeasible using statistical methods whether applied to standard software or fault-tolerant software. The classical methods of estimating reliability are shown to lead to exhorbitant amounts of testing when applied to life-critical software. Reliability growth models are examined and also shown to be incapable of overcoming the need for excessive amounts of testing. The key assumption of software fault tolerance—separately programmed versions fail independently—is shown to be problematic. This assumption cannot be justified by experimentation in the ultrareliability region and subjective arguments in its favor are not sufficiently strong to justify it as an axiom. Also, the implications of the recent multiversion software experiments support this affirmation.

Index Terms—Life-Critical, Validation, Software Reliability, Design Error, Ultrareliability, Software Fault-Tolerance

1 Introduction

The potential of enhanced flexibility and functionality has led to an ever increasing use of digital computer systems in control applications. At first, the digital systems were designed to perform the same functions as their analog counterparts. However, the availability of enormous computing power at a low cost has led to expanded use of digital computers in current applications and their introduction into many new applications. Thus, larger and more complex systems are being designed. The result has been, as promised, increased performance at a minimal hardware cost; however, it has also resulted in software systems which contain more errors. Sometimes, the impact of a software bug is nothing more than an inconvenience. At other times a software bug leads to costly downtime. But what will be the impact of design flaws in software systems used in life-critical applications such as industrial-plant control, aircraft control, nuclear-reactor control, or nuclear-warhead arming? What will be the price of software failure as digital computers are applied more and more frequently to these and other life-critical functions? Already, the symptoms of using insufficiently reliable software for life-critical applications are appearing [1, 2, 3].

For many years, much research has focused on the quantification of software reliability. Research efforts started with reliability growth models in the early 1970's. In recent years, an emphasis on developing methods which enable reliability quantification of software used for life-critical functions has emerged. The common approach which is offered is the combination of software fault-tolerance and statistical models.

In this paper, we will investigate the software reliability problem from two perspectives. We will first explore the problems which arise when you test software as a black box, i.e. subject it to inputs and check the outputs without examination of internal structure. Then, we will examine the problems which arise when software is not treated as a black box, i.e. some internal structure is modeled. In either case, we argue that the associated problems are intractable—i.e., they inevitably lead to a need for testing beyond what is practical.

2 Software Reliability

For life-critical applications, the validation process must establish that system reliability is extremely high. Historically, this ultrahigh reliability requirement has been translated into a probability of failure on the order of 10^{-7} to 10^{-9} for 1 to 10 hour missions. Unfortunately, such probabilities create enormous problems for validation. For convenience, we will use the following terminology:

name	failure rate (per hour)
ultrareliability	$< 10^{-7}$
moderate reliability	10^{-3} to 10^{-7}
low reliability	$> 10^{-3}$

Software does not physically fail as hardware does. Physical failures (as opposed to hardware design flaws) occur when hardware wears out, breaks, or is adversely affected by environmental phenomena such as electromagnetic fields or alpha particles. Software is not subject to these problems. Software faults are present at the beginning of and throughout a system's lifetime. To such an extent, software reliability is meaningless—software is either correct or incorrect with respect to its specification. Nevertheless, software systems are embedded in stochastic environments. These environments subject the software program to a sequence of inputs over time. For each input, the program produces either a correct or an incorrect answer. Thus, in a systems context, the software system produces errors in a stochastic manner; the sequence of errors behaves like a stochastic point process.

In this paper, the inherent difficulty of accurately modeling software reliability will be explored. To facilitate the discussion, we will construct a simple model of the software failure process. The driver of the failure process is the external system that supplies inputs to the program. As a function of its inputs and internal state, the program produces an output. If the software were perfect, the internal state would be correct and the outputs produced would be correct. However, if there is a design flaw in the program, it can manifest itself either by production of an erroneous output or by corruption of the internal state (which may affect subsequent outputs).

In a real-time system, the software is periodically scheduled, i.e. the same program is repeatedly executed in response to inputs. It is not unusual to find "iteration rates" of 10 to 100 cycles per second. If the probability of software failure per input is constant, say p, we have a binomial process. The number of failures S_n after n inputs is given by the binomial distribution:

$$P(S_n = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

We wish to compute the probability of system failure for n inputs. System failure occurs for all $S_n > 0$. Thus,

$$P_{sys}(n) = P(S_n > 0) = 1 - P(S_n = 0) = 1 - (1-p)^n$$

This can be converted to a function of time with the transformation $n = Kt$ where $K =$ the number of inputs per unit time. The system failure probability at time t, $P_{sys}(t)$, is thus:

$$P_{sys}(t) = 1 - (1-p)^{Kt} \tag{1}$$

Of course, this calculation assumes that the probability of failure per input is constant over time.[1]

This binomial process can be accurately approximated by an exponential distribution since p is small and n is large:

$$P_{sys}(t) = 1 - e^{-Ktp} \tag{2}$$

This is easily derived using the Poisson approximation to the binomial. The discrete binomial process can thus be accurately modeled by a continuous exponential process. In the following discussion, we will frequently use the exponential process rather than the binomial process to simplify the discussion.

3 Analyzing Software as a Black Box

The traditional method of validating reliability is life testing. In life testing, a set of test specimens are operated under actual operating conditions for a predetermined amount of time. Over this period, failure times are recorded and subsequently used in reliability computation. The internal structure of the test specimens is not examined. The only observable is whether a specimen has failed or not.

For systems that are designed to attain a probability of failure on the order of 10^{-7} to 10^{-9} for 1 hour missions or longer, life testing is prohibitively impractical. This can be shown by an illustrative example. For simplicity, we will assume that the time to failure distribution is exponential.[2] Using standard statistical methods [4], the time on test can be estimated for a specified system reliability. There are two basic approaches: (1) testing with replacement and (2) testing without replacement. In either case, one places n items on test. The test is finished when r failures have been observed. In the first case, when a device fails a new device is put on test in its place. In the second case, a failed device is not replaced. The tester chooses values of n and r to obtain the desired levels of the α and β errors (i.e., the probability of rejecting a good system and the probability of accepting a bad system respectively.) In general, the larger r and n are, the smaller the statistical estimation errors are. The expected time on test can be calculated as a function of r and n. The expected time on test, D_t, for the replacement case is:

$$D_t = \mu_o \frac{r}{n} \tag{3}$$

[1]If the probability of failure per input were not constant, then the reliability analysis problem is even harder. One would have to estimate p(t) rather than just p. A time-variant system would require even more testing than a time-invariant one, since the rate must be determined as a function of mission time. The system would have to be placed in a random state corresponding to a specific mission time and subjected to random inputs. This would have to be done for each time point of interest within the mission time. Thus, if the reliability analysis is intractable for systems with constant p, it is unrealistic to expect it to be tractable for systems with non-constant p(t).

[2]In the previous section the exponential process was shown to be an accurate approximation to the discrete binomial software failure process.

no. of replicates (n)	Expected Test Duration D_t
1	10^{10} hours = 1141550 years
10	10^{9} hours = 114155 years
100	10^{8} hours = 11415 years
10000	10^{6} hours = 114 years

Table 1: Expected Test Duration For r=1

where μ_o is the mean failure time of the test specimen [4]. The expected time on test for the non-replacement case is:

$$D_t = \mu_o \sum_{j=1}^{r} \frac{1}{n-j+1} \tag{4}$$

Even without specifying an α or β error, a good indication of the testing time can be determined. Clearly, the number of observed failures r must be greater than 0 and the total number of test specimens n must be greater than or equal to r. For example, suppose the system has a probability of failure of 10^{-9} for a 10 hour mission. Then the mean time to failure of the system (assuming exponentially distributed) μ_o is:

$$\mu_o = \frac{10}{-ln[1-10^{-9}]} \approx 10^{10}$$

Table 1 shows the expected test duration for this system as a function of the number of test replicates n for $r = 1$.[3] It should be noted that a value of r equal to 1 produces the shortest test time possible but at the price of extremely large α and β errors. To get satisfactory statistical significance, larger values of r are needed and consequently even more testing. Therefore, given that the economics of testing fault-tolerant systems (which are very expensive) rarely allow n to be greater than 10, life-testing is clearly out of the question for ultrareliable systems. The technique of statistical life-testing is discussed in more detail in the appendix.

4 Reliability Growth Models

The software design process involves a repetitive cycle of testing and repairing a program. A program is subjected to inputs until it fails. The cause of failure is determined; the program is repaired and is then subjected to a new sequence of inputs. The result is a sequence of programs $p_1, p_2, ..., p_n$ and a sequence of inter-failure times $T_1, T_2, ..., T_n$ (usually measured in number of inputs). The goal is to construct a mathematical technique (i.e. model) to predict the reliability of the final program p_n based on the observed interfailure data. Such a model enables one to estimate the probability of failure of the final "corrected" program without subjecting it to a sequence of inputs. This process is a form of prediction or extrapolation and has been studied in detail [5, 6, 7]. These models are called "Reliability Growth Models". If one resists the temptation to correct the program based on the last failure, the method is equivalent to black-box testing the final version. If one corrects the final version and estimates the reliability of the corrected version based on a reliability growth model, one hopefully has increased the efficiency of the testing process in doing so. The question we would like to examine is how much efficiency is gained by use of a reliability

[3]The expected time with or without replacement is almost the same in this case.

growth model and is it enough to get us into the ultrareliable region. Unfortunately, the answer is that the gain in efficiency is not anywhere near enough to get us into the ultrareliable region. This has been pointed out by several authors. Keiller and Miller write [8]:

> The reliability growth scenario would start with faulty software. Through execution of the software, bugs are discovered. The software is then modified to correct for the design flaws represented by the bugs. Gradually the software evolves into a state of higher reliability. There are at least two general reasons why this is an unreasonable approach to highly-reliable safety-critical software. The time required for reliability to grow to acceptable levels will tend to be extremely long. Extremely high levels of reliability cannot be guaranteed *a priori.*

Littlewood writes [9]:

> Clearly, the reliability growth techniques of §2 [*a survey of the leading reliability growth models*] are useless in the face of such ultra-high reliability requirements. It is easy to see that, even in the unlikely event that the system had achieved such a reliability, we could not assure ourselves of that achievement in an acceptable time.

The problem alluded to by these authors can be seen clearly by applying a reliability growth model to experimental data. The data of table 2 was taken from an experiment performed by Nagel and Skrivan [10]. The data in this table was obtained for program A1, one of six programs investigated.

Number of Bugs Removed	failure probability per input
1	0.9803
2	0.1068
3	0.002602
4	0.002104
5	0.001176
6	0.0007659

Table 2: Nagel Data From Program A1

The versions represent the successive stages of the program as bugs were removed. A log-linear growth model was postulated and found to fit all 6 programs analyzed in the report. A simple regression on the data of table 2 yields a slope and y-intercept of: -1.415 and 0.2358, respectively. The line is fitted to the log of the raw data as shown in figure 1. The correlation coefficient is -0.913. It is important to note that in the context of reliability growth models the failure rates are usually reported as failure rates per *input*, whereas the system requirements are given as failure rates per hour or as a probability of failure for a specified mission duration (e.g. 10 hours). However, equation 2 can be rearranged into a form which can be used to convert the system requirements into a required failure rate per input.

$$p = \frac{-ln(1 - P_{sys})}{Kt} \tag{5}$$

If the system requirement is a probability of failure of 10^{-9} for a 10-hour mission and the sample rate of the system (i.e., K) is $10/sec$, then the required failure rate per input p can be calculated

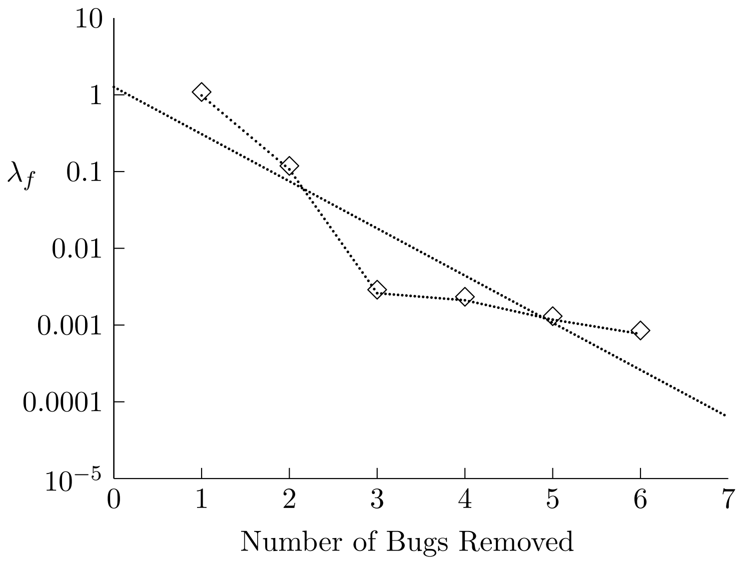

Figure 1: Loglinear Fit To Program A1 Failure Data

as follows:

$$\begin{aligned} p &= \frac{-ln(1-P_{sys})}{Kt} \\ &= \frac{-ln(1-10^{-9})}{(10/sec)(3600\ secs/hour)(10\ hours)} \\ &= 2.78 \times 10^{-15} \end{aligned}$$

The purpose of a reliability growth model is to estimate the failure rate of a program after the removal of the last discovered bug. The loglinear growth model plotted in figure 1 can be used to predict that the arrival rate of the next bug will be 6.34×10^{-5}. The key question, of course, is how long will it take before enough bugs are removed so that the reliability growth model will predict a failure rate per input of 2.78×10^{-15} or less. Using the loglinear model we can find the place where the probability drops below 2.78×10^{-15} as illustrated in figure 2. Based upon the model, the *24th* bug will arrive at a rate of 2.28×10^{-15}, which is less than the goal. Thus, according to the loglinear growth model, 23 bugs will have to removed before the model will yield an acceptable failure rate. But how long will it take to remove these 23 bugs? The growth model predicts that bug 23 will have a failure rate of about 9.38×10^{-15}. The expected number of test cases until observing a binomial event of probability 9.38×10^{-15} is 1.07×10^{14}. If the test time is the same as the real time execution time, then each test case would require 0.10 secs. Thus, the expected time to discover bug 23 alone would be 1.07×10^{13} secs or 3.4×10^{5} years. In table 3, the above calculations are given for all of the programs in reference [10].[4] These examples illustrate why the use of a reliability growth model does not alleviate the testing problem even if one assumes that the model applies universally to the ultrareliable region.

[4]Table 5 assumes a perfect fit with the log-linear model in the ultrareliable region.

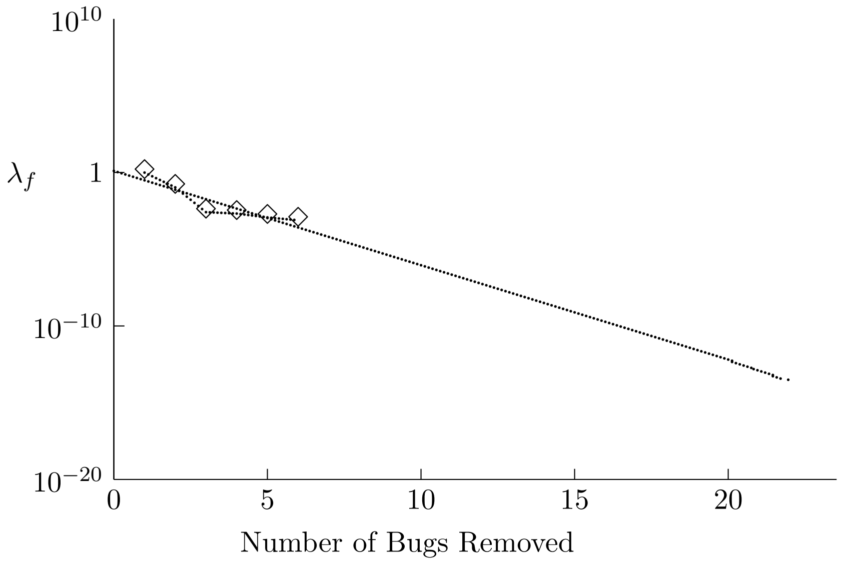

Figure 2: Extrapolation To Predict When Ultrareliability Will Be Reached

program	slope	y-intercept	last bug	test time
A1	-1.415	0.2358	23	3.4×10^5 years
B1	-1.3358	1.1049	25	3.3×10^5 years
A2	-1.998	2.4572	17	1.5×10^5 years
B2	-3.623	2.3296	9	4.5×10^4 years
A3	-.54526	-1.3735	58	6.8×10^5 years
B3	-1.3138	0.0912	25	5.3×10^5 years

Table 3: Test Time To Remove the Last Bug to Obtain Ultrareliability

4.1 Low Sample Rate Systems and Accelerated Testing

In this section, the feasibility of quantifying low-sample rate systems (i.e. systems where the time between inputs is long) in the ultrareliable region will be briefly explored. Also, the potential of accelerated testing will be discussed.

Suppose that the testing rate is faster than real time and let R = test time per input. Since each test is an independent trial, the time to the appearance of the next bug is given by the geometric distribution. Thus, the expected number of inputs until the next bug appears is $1/p$ and the expected test time, D_t, is given by:

$$D_t = R/p$$

Using equation 5, D_t becomes:

$$D_t = \frac{RKt}{-ln(1 - P_{sys})} \approx \frac{RKt}{P_{sys}} \tag{6}$$

From equation 6 it can be seen that a low sample rate system (i.e. a system with small K) requires less test time than a high sample rate system (assuming that R remains constant). Suppose that the system requirement is a probability of failure of 10^{-9} for a 10 hour mission (i.e. $P_{sys} = 10^9, t = 10$). If the system has a fast sample rate (e.g. K = 10 inputs/sec) and the time required to test an input is the same as the real-time execution time (i.e. R = 0.10 secs), then the expected test time is 10^{10} hours = 1.14×10^6 years. Now suppose that R remains constant but K is reduced. (Note that this usually implies an accelerated testing process. The execution time per input is usually greater for slow systems than fast systems. Since R is not increased as K is decreased the net result is equivalent to an accelerated test process.) The impact of decreasing K while holding R constant can be seen in table 4 which reports the expected test time as a function of K. Thus, theoretically, a

K	Expected Test Time, D_t
10/sec	1.14×10^6 years
1/sec	1.14×10^5 years
1/minute	1.9×10^3 years
1/hour	31.7 years
1/day	1.32 years
1/month	16 days

Table 4: Expected Test Time as a function of K for R = 0.1 secs

very slow system that can be tested very quickly (i.e. much faster than real-time) can be quantified in the ultrareliable region. However, this is not as promising as it may look at first. The value of K is fixed for a given system and the experimenter has no control over it. For example, the sample rate for a digital flight control system is on the order of 10 inputs per second or faster and little can be done to slow it down. Thus, the above theoretical result does nothing to alleviate the testing problem here. Furthermore, real time systems typically are developed to exploit the full capability of a computing system. Consequently, although a slower system's sample rate is less, its execution time per input is usually higher and so R is much greater than the 0.10 secs used in table 4. In fact one would expect to see R grow in proportion to $1/K$. Thus, the results in table 4 are optimistic. Also, it should be noted that during the testing process, one must also determine whether the program's answer is correct or incorrect. Consequently, the test time per input is often much greater than the real-time execution time rather than being shorter. In conclusion, if one is fortunate enough to have a very slow system that can exploit an accelerated testing process, one can obtain ultra-reliable estimates of reliability with feasible amounts of test times. However, such systems are usually not classified as real-time systems and thus, are out of the scope of this paper.

4.2 Reliability Growth Models and Accelerated Testing

Now lets revisit the reliability growth model in the context of a slow system which can be quickly tested. Suppose the system under test is a slow real-time system with a sample rate of 1 input per minute. Then, the failure rate per input must be less than $10^{-9}/60 = 1.67 \times 10^{-11}$ in order for the program to have a failure rate of $10^{-9}/hour$. Using the regression results, it can be seen that approximately 17 bugs must be removed:

bug	failure rate per input
16	1.87332×10^{-10}
17	4.55249×10^{-11}
18	1.10633×10^{-11}

Thus one could test until 17 bugs have been removed, remove the last bug and use the reliability growth model to predict a failure rate per input of 1.106×10^{-11}. But, how long would it take to remove the 17 bugs? Well, the removal of the last bug alone would on average require approximately 2.2×10^{10} test cases. Even if the testing process were 1000 times faster than the operational time per input (i.e. R = 60/1000 secs), this would require 42 years of testing. Thus, we see why Littlewood, Keiller and Miller see little hope of using reliability growth models for ultrareliable software. This problem is not restricted to the program above but is universal. Table 5 repeats the above calculations for the rest of the programs in reference [10]. At even the most optimistic

program	slope	y-intercept	last bug	test time
A1	-1.415	0.2358	17	42 years
B1	-1.3358	1.1049	19	66 years
A2	-1.998	2.4572	13	31 years
B2	-3.623	2.3296	7	19 years
A3	-.54526	-1.3735	42	66 years
B3	-1.3138	0.0912	19	32 years

Table 5: Test Time To Remove the Last Bug to Obtain Ultrareliability

improvement rates, it is obvious that reliability growth models are impractical for ultrareliable software.

5 Software Fault Tolerance

Since fault tolerance has been successfully used to protect against hardware physical failures, it seems natural to apply the same strategy against software bugs. It is easy to construct a reliability model of a system designed to mask physical failures using redundant hardware and voting. The key assumption which enables both the design of ultrareliable systems from less reliable components and the estimation of 10^{-9} probabilities of failure is that the separate redundant components fail independently or nearly so. The independence assumption has been used in hardware fault tolerance modelling for many years. If the redundant components are located in separate chassis, powered by separate power supplies, electrically isolated from each other and sufficiently shielded from the environment it is not unreasonable to assume failure independence of physical hardware faults.

The basic strategy of the software fault-tolerance approach is to design several versions of a program from the same specification and to employ a voter of some kind to protect the system from bugs. The voter can be an acceptance test (i.e., recovery blocks) or a comparator (i.e., N-version programming). Each version is programmed by a separate programming team.[5] Since the versions are developed by separate programming teams, it is hoped that the redundant programs will fail independently or nearly so [11, 12]. From the version reliability estimates and the independence assumption, system reliability estimates could be calculated. However, unlike hardware physical failures which are governed by the laws of physics, programming errors are the products of human reasoning (i.e., actually improper reasoning). The question thus becomes one of the reasonableness of assuming independence based on little or no practical or theoretical foundations. Subjective arguments have been offered on both sides of this question. Unfortunately, the subjective arguments for multiple versions being independent are not compelling enough to qualify it as an axiom. The reasons why experimental justification of independence is infeasible and why ultrareliable quantification is infeasible despite software fault tolerance are discussed in the next section.

5.1 Models of Software Fault Tolerance

Many reliability models of fault-tolerant software have been developed based on the independence assumption. To accept such a model, this assumption must be accepted. In this section, it will be shown how the independence assumption enables quantification in the ultrareliable region, why quantification of fault-tolerant software reliability is unlikely without the independence assumption, and why this assumption cannot be experimentally justified for the ultrareliable region.

5.1.1 INdependence Enables Quantification Of Ultrareliability

The following example will show how independence enables ultrareliability quantification. Suppose three different versions of a program control a life-critical system using some software fault tolerance scheme. Let $E_{i,k}$ be the event that the ith version fails on its kth execution. Suppose the probability that version i fails during the kth execution is $p_{i,k}$. As discussed in section 2, we will assume that the failure rate is constant. Since the versions are voted, the system does not fail unless there is a coincident error, i.e., two or more versions produce erroneous outputs in response to the same input. The probability that two or more versions fail on the kth execution causing system failure is:

$$P_{sys,k} \quad = \quad P(\ (E_{1,k} \wedge E_{2,k}) \text{ or } (E_{1,k} \wedge E_{3,k}) \text{ or } (E_{2,k} \wedge E_{3,k}) \text{ or } (E_{1,k} \wedge E_{2,k} \wedge E_{3,k})\ ) \qquad (7)$$

Using the additive law of probability, this can be written as:

$$P_{sys,k} \quad = \quad P(E_{1,k} \wedge E_{2,k}) + P(E_{1,k} \wedge E_{3,k}) + P(E_{2,k} \wedge E_{3,k}) - 2P(E_{1,k} \wedge E_{2,k} \wedge E_{3,k}) \qquad (8)$$

If independence of the versions is assumed, this can be rewritten as:

$$P_{sys,k} \quad = \quad P(E_{1,k})P(E_{2,k}) + P(E_{1,k})P(E_{3,k}) + P(E_{2,k})P(E_{3,k}) - 2P(E_{1,k})P(E_{2,k})P(E_{3,k}) \qquad (9)$$

[5] Often these separate programming teams are called "independent programming" teams. The phrase "independent programming" does not mean the same thing as "independent manifestation of errors."

The reason why independence is usually assumed is obvious from the above formula—if each $P(E_{i,k})$ can be estimated to be approximately 10^{-6}, then the probability of system failure due to two or more coincident failures is approximately 3×10^{-12}.

Equation (9) can be used to calculate the probability of failure for a T hour mission. Suppose $P(E_{i,k}) = p$ for all i and k. Then

$$P_{sys,k} = 3p^2 - 2p^3 \approx 3p^2$$

and the probability that the system fails during a mission of T hours can be calculated using equation (1):

$$P_{sys}(T) = 1 - (1 - P_{sys,k})^{KT} \approx 1 - (1 - 3p^2)^{KT}$$

where K = the number of executions of the program in an hour. For small p_i the following approximation is accurate:

$$P_{sys}(T) \approx 1 - e^{(-3p^2KT)} \approx 3p^2KT$$

For the following typical values of $T = 1$ and $K = 3600$ (i.e., 1 execution per second), we have

$$P_{sys}(T) \approx 3p^2KT = 3(10^{-6})(10^{-6})(3600) = 1.08 \times 10^{-8}$$

Thus, an ultrareliability quantification has been made. But, this depended critically on the independence assumption. If the different versions do not fail independently, then equation (7) must be used to compute failure probabilities and the above calculation is meaningless. In fact, the probability of failure could be anywhere from 0 to about 10^{-2} (i.e., 0 to $3pKT$[6]).

5.1.2 Ultrareliable Quantification Is Infeasible Without Independence

Now consider the impact of not being able to assume independence. The following argument was adapted from Miller [13]. To simplify the notation, the last subscript will be dropped when referring to the kth execution only. Thus,

$$P_{sys} = P(E_1 \wedge E_2) + P(E_1 \wedge E_3) + P(E_2 \wedge E_3) - 2P(E_1 \wedge E_2 \wedge E_3) \tag{10}$$

Using the identity $P(A \wedge B) = P(A)P(B) + [P(A \wedge B) - P(A)P(B)]$, this can be rewritten as:

$$\begin{aligned} P_{sys} = \; & P(E_1)P(E_2) + P(E_1)P(E_3) + P(E_2)P(E_3) - 2P(E_1)P(E_2)P(E_3) \\ & +[P(E_2 \wedge E_1) - P(E_1)P(E_2)] \\ & +[P(E_3 \wedge E_1) - P(E_3)P(E_1)] \\ & +[P(E_3 \wedge E_2) - P(E_3)P(E_2)] \\ & -2[P(E_1 \wedge E_2 \wedge E_3) - P(E_1)P(E_2)P(E_3)] \end{aligned} \tag{11}$$

This rewrite of the formula reveals two components of the system failure probability: (1) the first line of equation 11 and (2) the last 4 lines of equation 11. If the multiple versions manifest errors independently, then the last four lines (i.e. the second component) will be equal to zero. Consequently, to establish independence experimentally, these terms must be shown to be 0. Realistically, to establish "adequate" independence, these terms must be shown to have negligible effect on the probability of system failure. Thus, the first component represents the "non-correlated" contribution to P_{sys} and the second component represents the "correlated" contribution to P_{sys}. Note that the terms in the first component of P_{sys} are all products of the individual version probabilities.

[6] 3pKT is a first-order approximation to the probability that the system fails whenever any one of the 3 versions fail.

If we cannot assume independence, we are back to the original equation (10). Since $P(E_1 \wedge E_2 \wedge E_3) \leq P(E_i \wedge E_j)$ for all i and j, we have

$$P(E_i \wedge Ej) \leq P_{sys} \text{ for all } i, j.$$

Clearly, if $P_{sys} < 10^{-9}$ then $P(E_i \wedge E_j) < 10^{-9}$. In other words, in order for P_{sys} to be in the ultrareliable region, the interaction terms (i.e. $P(E_i \wedge E_j)$) must also be in the ultrareliable region. To establish that the system is ultrareliable, the validation must either demonstrate that these terms are very small or establish that P_{sys} is small by some other means (from which we could indirectly deduce that these terms are small.) Thus, we are back to the original life-testing problem again.

From the above discussion, it is tempting to conclude that it is necessary to demonstrate that each of the interaction terms is very small in order to establish that P_{sys} is very small. However, this is not a legitimate argument. Although the interaction terms will always be small when P_{sys} is small, one cannot argue that the *only* way of establishing that P_{sys} is small is by showing that the interaction terms are small. However, the likelihood of establishing that P_{sys} is very small without directly establishing that all of the interaction terms are small appears to be extremely remote. This follows from the observation that without further assumptions, there is little more that can be done with equation (10). It seems inescapable that no matter how (10) is manipulated, the terms $P(E_i \wedge E_j)$ will enter in linearly. Unless, a form can be found where these terms are eliminated altogether or appear in a non-linear form where they become negligible (e.g. all multiplied by other parameters), the need to estimate them directly will remain. Furthermore, the information contained in these terms must appear somewhere. The dependency of P_{sys} on some formulation of interaction cannot be eliminated.

Although the possibility that a method may be discovered for the validation of software fault-tolerance remains, it is prudent to recognize where this opportunity lies. It does not lie in the realm of controlled experimentation. The only hope is that a reformulation of equation (10) can be discovered that enables the estimation of P_{sys} from a set of parameters which can be estimated using moderate amounts of testing. The efficacy of such a reformulation could be assessed analytically before any experimentation.

5.1.3 Danger Of Extrapolation to the Ultrareliability Region

To see the danger in extrapolating from a feasible amount of testing that the different versions are independent, we will consider some possible scenarios for coincident failure processes. Suppose that the probability of failure of a single version during a 1 hour interval is 10^{-5}. If the versions fail independently, then the probability of a coincident error is on the order of 10^{-10}. However, suppose in actuality the arrival rate of a coincident error is $10^{-7}/hour$. One could test for 100 years and most likely not see a coincident error. From such experiments it would be tempting to conclude that the different versions are independent. After all, we have tested the system for 100 years and not seen even one coincident error! If we make the independence assumption, the system reliability is $(1 - 3 \times 10^{-10})$. But actually the system reliability is approximately $(1 - 10^{-7})$. Likewise, if the failure rate for a single version were $10^{-4}/hour$ and the arrival rate of coincident errors were $10^{-5}/hour$, testing for one year would most likely result in no coincident errors. The erroneous assumption of independence would allow the assignment of a 3×10^{-8} probability of failure to the system when in reality the system is no better than 10^{-5}.

In conclusion, if independence cannot be assumed, it seems inescapable that the intersection of the events E_1, E_2, and E_3 (i.e. $P(E_i \wedge E_j)$) must be directly measured. As shown above, these occur in the system failure formula not as products, but alone, and thus must be less than 10^{-12} per input in order for the system probability of failure to be less than 10^{-9} at 1 hour. Unfortunately, testing to this level is infeasible and extrapolation from feasible amounts of testing is dangerous.

Since ultrareliability has been established as a requirement for many systems, there is great incentive to create models which enable an estimate in the ultrareliable region. Thus, there are many examples of software reliability models for operational ultrareliable systems. Given the ramifications of independence on fault-tolerant software reliability quantification, unjustifiable assumptions must not be overlooked.

5.2 Feasibility of a General Model For Coincident Errors

Given the limitations imposed by non-independence, one possible approach to the ultrareliability quantification problem is to develop a general fault-tolerant software reliability model that accounts for coincident errors. Two possibilities exist:

1. The model includes terms which cannot be measured within feasible amounts of time.
2. The model includes only parameters which can be measured within feasible amounts of time.

It is possible to construct elaborate probability models which fall into the first class. Unfortunately since they depend upon unmeasurable parameters, they are useless for the quantification of ultrareliability. The second case is the only realistic approach.[7] The independence model is an example of the second case. Models belonging to the second case must explicitly or implicitly express the interaction terms in equation (10) as "known" functions of parameters which can be measured in feasible amounts of time:

$$P_I = f(p_1, p_2, p_3, ..., p_n)$$

The known function f in the independence model is the zero function, i.e., the interaction terms P_I are zero identically irrespective of any other measurable parameters.

A more general model must provide a mechanism that makes these interaction terms negligibly small in order to produce a number in the ultrareliable region. These known functions must be applicable to all cases of multi-version software for which the model is intended. Clearly, any estimation based on such a model would be strongly dependent upon correct knowledge of these functions. But how can these functions be determined? There is little hope of deriving them from fundamental laws, since the error process occurs in the human mind. The only possibility is to derive them from experimentation, but experimentation can only derive functions appropriate for low or moderate reliability software. Therefore, the correctness of these functions in the ultrareliable region can not be established experimentally. Justifying the correctness of the known functions requires far more testing than quantifying the reliability of a single ultrareliable system. The model must be shown to be applicable to a specified sample space of multi-version programs. Thus, there must be extensive sampling from the space of multi-version programs, each of which must undergo life-testing for over 100,000 years in order to demonstrate the universal applicability of the functions. Thus, in either case, the situation appears to be hopeless—the development of a credible coincident error model which can be used to estimate system reliability within feasible amounts of time is not possible.

[7]The first case is included for completeness and because such models have been proposed in the past.

5.3 The Coincident-Error Experiments

Experiments have been performed by several researchers to investigate the coincident error process. The first and perhaps most famous experiment was performed by Knight and Leveson [14]. In this experiment 27 versions of a program were produced and subjected to 1,000,000 input cases. The observed average failure rate per input was 0.0007. The major conclusion of the experiment was that the independence model was rejected at the 99% confidence level. The quantity of coincident errors was much greater than that predicted by the independence model. Experiments produced by other researchers have confirmed the Knight-Leveson conclusion [12, 15]. A excellent discussion of the experimental results is given in [16].

Some debate [16] has occurred over the credibility of these experiments. Rather than describe the details of this debate, we would prefer to make a few general observations about the scope and limitations of such experiments. First, the N-version *systems* used in these experiments must have reliabilities in the low to moderate reliability region. Otherwise, no data would be obtained which would be relevant to the independence question.[8] It is not sufficient (to get data) that the individual versions are in this reliability region. The coincident error rate must be observable, so the reliability of "voted" outputs must be in the low to moderate reliability region. To see this consider the following. Suppose that we have a 3-version system where each replicate's failure rate is $10^{-4}/hour$. If they fail independently, the coincident error rate should be $3 \times 10^{-8}/hour$. The versions are in the moderate reliability region, but the system is potentially (i.e. if independent) in the ultrareliable region. In order to test for independence, "coincident" errors must be observed. If the experiment is performed for one year and no coincident errors are observed, then one can be confident that the coincident error rate (and consequently the system failure rate) is less than 1.14×10^{-4}. If coincident errors are observed then the coincident error rate is probably even higher. If the coincident error rate is actually $10^{-7}/hour$, then the independence assumption is invalid, but one would have to test for over 1000 years in order to have a reasonable chance to observe them! Thus, future experiments will have one of the following results depending on the actual reliability of the test specimens:

1. demonstration that the independence assumption does *not* hold for the low reliability system.
2. demonstration that the independence assumption does hold for systems for the low reliability system.
3. no coincident errors were seen but the test time was insufficient to demonstrate independence for the potentially ultrareliable system.

If the system under test is a low reliability system, the independence assumption may be contradicted or vindicated. Either way, the results will not apply to ultrareliable systems except by way of extrapolation. If the system under test were actually ultrareliable, the third conclusion would result. Thus, experiments can reveal problems with a model such as the independence model when the inaccuracies are so severe that they manifest themselves in the low or moderate reliability region. However, software reliability experiments can only demonstrate that an interaction model is inaccurate, never that a model is accurate for ultrareliable software. Thus, negative results are possible, but never positive results.

The experiments performed by Knight and Leveson and others have been useful to alerting the world to a formerly unnoticed critical assumption. However, it is important to realize that

[8]that is, unless one was willing to carry out a "Smithsonian" experiment, i.e. one which requires centuries to complete.

these experiments cannot accomplish what is really needed—namely, to establish with scientific rigor that a particular design is ultrareliable or that a particular design methodology produces ultrareliable systems. This leaves us in a terrible bind. We want to use digital processors in life-critical applications, but we have no feasible way of establishing that they meet their ultrareliability requirements. We must either change the reliability requirements to a level which is in the low to moderate reliability region or give up the notion of experimental quantification. Neither option is very appealing.

6 Conclusions

In recent years, computer systems have been introduced into life-critical situations where previously caution had precluded their use. Despite alarming incidents of disaster already occurring with increasing frequency, industry in the United States and abroad continues to expand the use of digital computers to monitor and control complex real-time physical processes and mechanical devices. The potential performance advantages of using computers over their analog predecessors have created an atmosphere where serious safety concerns about digital hardware and software are not adequately addressed. Although fault-tolerance research has discovered effective techniques to protect systems from physical component failure, practical methods to prevent design errors have not been found. Without a major change in the design and verification methods used for life-critical systems, major disasters are almost certain to occur with increasing frequency.

Since life-testing of ultrareliable software is infeasible (i.e., to quantify $10^{-8}/hour$ failure rate requires more than 10^8 hours of testing), reliability models of fault-tolerant software have been developed from which ultrareliable-system estimates can be obtained. The key assumption which enables an ultrareliability prediction for hardware failures is that the electrically isolated processors fail independently. This assumption is reasonable for hardware component failures, but not provable or testable. This assumption is not reasonable for software or hardware design flaws. Furthermore, any model which tries to include some level of non-independent interaction between the multiple versions can not be justified experimentally. It would take more than 10^8 hours of testing to make sure there are not coincident errors in two or more versions which appear rarely but frequently enough to degrade the system reliability below $(1 - 10^{-8})$.

Some significant conclusions can be drawn from the observations of this paper. Since digital computers will inevitably be used in life-critical applications, it is necessary that "credible" methods be developed for generating reliable software. Nevertheless, what constitutes a "credible" method must be carefully reconsidered. A pervasive view is that software validation must be accomplished by probabilistic and statistical methods. The shortcomings and pitfalls of this view have been expounded in this paper. Based on intuitive merits, it is likely that software fault tolerance will be used in life-critical applications. Nevertheless, the ability of this approach to generate ultrareliable software cannot be demonstrated by research experiments. The question of whether software fault tolerance is more effective than other design methodologies such as formal verification or vice versa can only be answered for low or moderate reliability systems, not for ultrareliable applications. The choice between software fault tolerance and formal verification must necessarily be based on either extrapolation or nonexperimental reasoning.

Similarly, experiments designed to compare the accuracy of different types of software reliability models can only be accomplished in the low to moderate reliability regions. There is little reason to believe that a model which is accurate in the moderate region is accurate in the ultrareliable region. It is possible that models which are inferior to other models in the moderate region are superior in the ultrareliable region—again, this cannot be demonstrated.

Appendix

In this section, the statistics of life testing will be briefly reviewed. A more detailed presentation can be found in a standard statistics text book such as Mann-Schafer-Singpurwalla [4]. This section presents a statistical test based on the maximum likelihood ratio[9] and was produced using reference [4] extensively. The mathematical relationship between the number of test specimens, specimen reliability, and expected time on test is explored.

$$\begin{aligned} \text{Let } n &= \text{the number of test specimens} \\ r &= \text{observed number of specimen failures} \\ X_1 < X_2 < ... < X_r &= \text{the ordered failure times} \end{aligned}$$

A hypothesis test is constructed to test the reliability of the system against an alternative.

$$\begin{aligned} H_o: \text{ Reliability } &= R_0 \\ H_1: \text{ Reliability } &< R_0 \end{aligned}$$

The null hypothesis covers the case where the system is ultrareliable. The alternative covers the case where the system fails to meet the reliability requirement. The α error is the probability of rejecting the null hypothesis when it is true (i.e. producer's risk). The β error is the probability of accepting the null hypothesis when it is false (i.e. consumer's risk).

There are two basic experimental approaches—(1) testing with replacement and (2) testing without replacement. In either case, one places n items on test. The test is finished when r failures have been observed. In the first case, when a device fails a new device is put on test in its place. In the second case, a failed device is not replaced. The tester chooses values of n and r to obtain the desired levels of the α and β errors. In general, the larger r and n are, the smaller the statistical testing errors are.

It is necessary to assume some distribution for the time-to-failure of the test specimen. For simplicity, we will assume that the distribution is exponential.[10] The test then can be reduced to a test on exponential means, using the transformation:

$$\mu = \frac{t}{-ln[R(t)]}$$

The expected time on test can then be calculated as a function of r and n. The expected time on test, D_t, for the replacement case is:

$$D_t = \mu_o \frac{r}{n} \tag{12}$$

where μ_o is the mean time to failure of the test specimen. The expected time on test for the non-replacement case is:

$$D_t = \mu_o \sum_{j=1}^{r} \frac{1}{n-j+1} \tag{13}$$

In order to calculate the α and β errors, a specific value of the alternative mean must be selected. Thus, the hypothesis test becomes:

$$\begin{aligned} H_o : \mu &= \mu_o \\ H_1 : \mu &= \mu_a \end{aligned}$$

[9]The maximum likelihood ratio test is the test which provides the "best" critical region for a given α error.

[10]If the failure times follow a Weibull distribution with known shape parameter, the data can be transformed into variables having exponential distributions before the test is applied.

A reasonable alternative hypothesis is that the reliability at 10 hours is $1 - 10^{-8}$ or that $\mu_a = 10^9$. The test statistic T_r is given by

$$T_r = (n-r)X_r + \sum_{i=1}^{r} X_i$$

for the non-replacement case and

$$T_r = nX_r$$

for the "replacement case". The critical value T_c (for which the null hypothesis should be rejected whenever $T_r \leq T_c$) can be determined as a function of α and r:

$$T_c = \mu_o \frac{\chi^2_{2r,\alpha}}{2}$$

where $\chi^2_{\nu,\alpha}$ is the α percentile of the chi-square distribution with ν degrees of freedom. Given a choice of r and α the value of the "best" critical region is determined by this formula. The β error can be calculated from

$$T_c = \mu_1 \frac{\chi^2_{2r,1-\beta}}{2}$$

Neither of the above equations can be solved until r is determined. However, the following formula can be derived from them:

$$\frac{\chi^2_{2r,\alpha}}{\chi^2_{2r,1-\beta}} = \frac{\mu_a}{\mu_o} \tag{14}$$

Given the desired α and β errors, one chooses the smallest r which satisfies this equation.

Example 1

Suppose that we wish to test:

$$\begin{aligned} H_o : \mu_o &= 10^{10} \\ H_1 : \mu_a &= 10^9 \end{aligned}$$

For $\alpha = 0.05$ and $\beta = 0.01$, the smallest r satisfying equation (14) is 3 (using a chi-square table). Thus, the critical region is $\mu_o \frac{\chi^2_{2r,\alpha}}{2} = 10^{10}(1.635)/2 = 8.18 \times 10^9$. The experimenter can choose any value of n greater than r. The larger n is, the shorter the expected time on test is. For the replacement case, the expected time on test is $\mu_o \frac{r}{n} = \frac{3 \times 10^{10}}{n}$:

no. of replicates (n)	Expected Test Duration D_t
10	3×10^9 hours
100	3×10^8 hours
10000	3×10^6 hours

Even with 10000 test specimens, the expected test time is 342 years.

Example 2

Suppose that we wish to test:

$$\begin{aligned} H_o : \mu_o &= 10^{10} \\ H_1 : \mu_a &= 10^9 \end{aligned}$$

Given $\alpha = 0.05$ and $r = 1$, the β error can be calculated. First the critical region is $\mu_o \frac{\chi^2_{2r,\alpha}}{2} = 10^{10}[0.1026]/2 = 5.13 \times 10^8$. From a chi-square table, the β error can be seen to be greater than 0.50.

Illustrative Table

For $\mu_o = 10^{10}$ and $\mu_a = 10^9$,

$$\frac{\mu_a}{\mu_o} \approx \frac{10^{-9}}{10^{-8}} = 0.1$$

The following relationship exists between α, r, and β:

α	r	β
.01	5	$\approx .005$
.01	3	$\approx .20$
.01	2	$\approx .50$
.05	3	$\approx .02$
.05	2	$\approx .10$
.05	1	$\approx .50$
.10	3	$\approx .005$
.10	2	$\approx .03$
.10	1	$\approx .25$

The power of the test $1 - \beta$ changes drastically with changes in r. Clearly r must be at least 2 to have a reasonable value for the beta error.

Acknowledgements

The authors are grateful to Dr. Alan White for his many helpful comments and to the anonymous reviewers for their careful reviews and many helpful suggestions.

References

[1] N. G. Leveson, "Software safety: What, why, and how," *Computing Surveys*, vol. 18, June 1986.

[2] I. Peterson, "A digital matter of life and death," *Science News*, Mar. 1988.

[3] E. Joyce, "Software bugs: A matter of life and liability," *Datamation*, May 1987.

[4] N. R. Mann, R. E. Schafer, and N. D. Singpurwalla, *Methods for Statistical Analysis of Reliability and Life Data*. New York: John Wiley & Sons, 1974.

[5] A. A. Abdalla-Ghaly and a. B. L. P. Y. Chan, "Evaluation of competing reliability predictions," *IEEE Transactions on Software Engineering*, pp. 950–967, 1986.

[6] B. Littlewood and P. A. Keiller, "Adaptive software reliability modeling," in *14th International Symposium on Fault-Tolerant Computing*, pp. 108–113, IEEE Computer Society Press, 1984.

[7] B. Littlewood, "Stochastic reliability-growth: A model for fault-removal in computer programs and hardware designs," *IEEE Transactions on Reliability*, pp. 313–320, 1981.

[8] P. A. Keiller and D. R. Miller, "On the use and the performance of software reliability growth models," *Reliability Engineering and System Safety*, pp. 95–117, 1991.

[9] B. Littlewood, "Predicting software reliability," *Philosophical Transactions of the Royal Society (London)*, pp. 513–526, 1989.

[10] P. M. Nagel and J. A. Skrivan, "Software reliability: Repetitive run experimentation and modeling," NASA Contractor Report 165836, Feb. 1982.

[11] A. Avizienis, "The n-version approach to fault-tolerant software," *IEEE Transactions on Software Engineering*, pp. 1491–1501, Dec. 1985.

[12] R. K. Scott, J. W. Gault, and D. F. McAllister, "Fault-tolerant software reliability modeling," *IEEE Transactions on Software Engineering*, May 1987.

[13] D. Miller, "Making statistical inferences about software reliability," NASA Contractor Report 4197, Nov. 1988.

[14] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumptions of independence in multiversion programming," *IEEE Transactions on Software Engineering*, vol. SE-12, pp. 96–109, Jan. 1986.

[15] T. J. Shimeall and N. G. Leveson, "An empirical comparison of software fault-tolerance and fault elimination," *IEEE Transactions on Software Engineering*, pp. 173–183, Feb. 1991.

[16] J. C. Knight and N. G. Leveson, "A reply to the criticisms of the Knight & Leveson experiment," *ACM SIGSOFT Software Engineering Notes*, Jan. 1990.