Qualitative Interpretation of Software Test Data

Herbert Hecht and Myron Hecht
SoHaR Incorporated
Beverly Hills, California

Abstract

During the final test phase of a high integrity software project the incidence of failures may be so low that conventional reliability growth models become unusable because the stochastics of the failure process mask any decreasing failure trend. While the low failure rate is of course desirable, it is frequently not sufficient by itself to demonstrate compliance with reliability or dependability requirements. Particularly for software used in safety systems, the maximum failure rate may be specified as low as 10^{-10} per hour, a value that cannot be verified in practice by conventional demonstration tests. We identify a qualitative approach to test data interpretation, particularly the examination of rare conditions as causes for failures, as a possible avenue for reliability assessment. This can be used as an alternative or as a supplement to redundancy for achieving the highest possible level of reliability for high integrity software. Further research in this area is recommended.

Introduction

Significant advances have been made in the development and testing of software for high integrity applications but none are sufficient for a technically rigorous certification of systems that require failure rates to be below 10^{-5} per hour. In this paper we identify the special quality assurance and test requirements of software for safety systems, explore the limitations of conventional approaches, and attempt to chart a direction for research that may lead to a more effective solution than is presently available.

The first problem encountered in this exploration is that the acceptable failure probability for such systems is far below the level that can be directly verified by test and is completely outside the domain of what can be achieved by reliance on the development process. This finding suggests that the safety system, or at least the software in the safety system, should be redundant. But redundancy is expensive, and for reasons discussed later, not completely dependable at the extremely low failure rates involved in safety systems. An adjunct, and ultimately a potential alternative, to redundancy is a qualitative assessment of failures in which specific development and test methodologies are aimed against the high risk failure types. The final thrust of this paper is an assessment of how far the combined qualitative-quantitative approach can go in verifying the failure rate requirements of typical safety system software.

What makes High Integrity Software so Unique?

The dependability requirements for some particularly demanding applications are embedded in regulatory documents, typically in non-mathematical terms, such as ACatastrophic failure conditions must be shown to be extremely improbable≅ [1], or ANo single failure [shall] result in loss of the protection function≅ [2]. The statutory requirements are usually interpreted as requiring aircraft flight control systems to have a catastrophic failure rate no greater than 10^{-10} per hour, and reactor safety systems to fail no more than once per 1 million demands. Since natural demands on the operation of a reactor safety system occur on the average about once per year, the interpretations arrive at almost the same reliability requirement in terms of hours.

These requirements are not directly verifiable by test for the system or software as a whole because the required test time greatly exceeds what can be achieved by conventional methodologies. To demonstrate that the failure rate of an item does not exceed x per hour requires approximately $1.5/x$ hours of test time under the most optimistic assumptions (no failures and a high risk test plan) [3] We are clearly talking about geological time scales. Even with test acceleration factors of 100, and assuming 100 test stations, such a test will take several decades. Unfortunately, the inadequacy of a test methodology for these programs is not only of academic interest, as the multiple crashes of Airbus 320 aircraft have shown.

As an alternative, we may want to assess reliability growth. This approach accepts that there will be initial test failures, but evaluates the reduction in failure frequency during successive intervals to estimate the reliability at the conclusion of test. Examples presented in a standard text on software reliability growth assessment show typical failure rates of 0.01 to 0.05 per hour (execution time) [4]. A program executing 8 hours per day with a failure rate of 0.03 per hour will fail almost 5 times per month. This failure experience permits statistical verification of reliability improvement and estimation of final failure rates by extrapolation of the reliability growth curve over q reasonable time interval. This approach cannot be scaled in practice to the extremely low failure rates specified for safety systems because the expected small number of observed failures does not support the estimation procedures.

The applications that are addressed by reliability growth assessment are by no means trivial or non-essential ones. At least the lower range of failure rates discussed there is acceptable in many important commercial applications dealing with inventory control, fund transfers, and even telephone switching. In most cases there will be hundreds of installations of a given program, so that a failure rate as low as 0.001 per hour will result in multiple failure reports each month. Statistical analysis of these reports permits evaluation of the effectiveness of current development, test, and maintenance processes. Analysis of the causes of failure can be used to determine the direction in which these processes must be further developed to reduce the failure rate.

Because of the lack of a reasonably large sample of naturally occurring software failures, none of these important feedback mechanisms are available in the high integrity system environment. The

current practice is to select development and test methodologies on the basis of experience in the general quality software field. But these methodologies, by themselves, are manifestly insufficient for assuring that the failure probability will not exceed the allowable thresholds. These considerations have led many researchers in this field to conclude that the systems must be made robust or fault tolerant so that the safety system mission can be accomplished even if there are residual flaws in the software [5, 6, 7]. This points to some form of redundancy, and the implementation problems that must then be dealt with are discussed in the next heading.

The Pros and Cons of Redundancy

Our society has long ago learned to live with imperfect hardware items by employing redundancy in forms such as multi-lamp light fixtures, multi-engine aircraft, and multi-channel electro-mechanical portions of safety systems. It seemed natural to carry this concept over to software. If we use two independent programs, the reliability requirement for the individual program is considerably eased and may approach a level at which the best of the existing test methodologies may become useful. Two problems arise: the Aindependence≅ of the individual programs, and the mechanism for selecting the correct output when there are differences.

It is very difficult to achieve truly independent failure mechanisms in multiple programs if they must accept the same inputs and furnish the same outputs (meet the same requirement), and carefully controlled experiments have shown that correlated failures must be expected [8, 9]. The output selection can use switching or voting. The former requires intelligence to choose the correct output when there are differences, and imperfections in the design or implementation of this intelligence will obviously detract from the failure reduction that can be achieved by redundancy. A significant problem associated with voting is that the outputs of the individual programs may not arrive at the same time. Practically every measure taken to achieve a high degree of independence of the programs (e. g., differences in programming language, program structure, development tools) will cause differences in execution time. The need to determine when a valid output has been received from a given program, and then to hold this value until valid outputs from the other versions are available, makes a voter a much more complicated (and therefore less reliable) structure than might be assumed from looking at a block diagram representation. Also, significant delays are introduced by the need to first hold until all versions have completed and then to furnish the consensus value to enable each program to proceed to the next computation.

Thus, the benefits of redundancy are appreciably reduced by the allowances that must be made for lack of independence and imperfections of the selection mechanism. The cost of developing and testing multiple versions, and the reduction in throughput for voting, are other disadvantages that must be considered. Redundancy can and has been used to improve software reliability for safety systems. But it carries a high cost, and the independence and selection issues make it difficult to determine by how much it reduces the failure probability. The following headings discuss an alternative or a possible supplement to redundancy for safety system software.

All Failures are Not Created Equal

The statutory language that was quoted earlier contained important clauses that limited the types of failures to which it applies. For aircraft systems it applies to *catastrophic* failures, and for nuclear reactor systems to those that cause *loss of the protection function.* It will now be investigated whether these restrictions permit an approach to quality assurance and test that has the potential of overcoming the difficulties discussed in previous headings.

In earlier publications we had called attention to the fact that many failures in well-tested systems were caused by rare events [10, 11]. The same data also shows that *multiple rare events* are almost the exclusive cause of the most critical failures in these systems, the class to which the statutory requirements apply. The data that will now be examined relate to failures in final (Level 8) testing of Space Shuttle Avionics software (SSA) over a period of about 18 months immediately following the Challenger accident. We classified an event as rare when it had not been experienced during the preceding 20 launches. Examples of rare events are engine failure, a large navigational error, or an incorrect and immediately harmful crew command. When there was any doubt whether the description related to a rare event, the classification defaulted to non-rare. When at least one rare event (RE) was responsible for the failure the corresponding failure report was classified as a rare event report (RR). Many failures due to rare events were associated with exception handling and redundancy management, indicating lack of a suitable test methodology for these software routines.

The SSA program had undergone intensive test prior to the phase reported on here. NASA classifies the consequences of failure (severity) on a scale of 1 to 5, where 1 represents safety critical and 2 mission critical failures with higher numbers indicating successively less mission impact. During most of this period test failures in the first two categories were analyzed and corrected even when the events leading to the failure were outside the contractual requirements (particularly more severe environments or equipment failures than the software was intended to handle); these categories were designated as 1N and 2N respectively. Results of the analysis are shown in Table 1.

Table 1. Analysis of SSA Data

Severity	No. Reports Analyzed (RA)	No.of Rare Reports (RR)	No.of Rare Events(RE)	Ratios		
				RR/RA	RE/RA	RE/RR
1	29	28	49	0.97	1.69	1.75
1N	41	33	71	0.80	1.83	2.15
2	19	12	23	0.63	1.32	1.92
2N	14	11	21	0.79	1.57	1.91
3	100	59	100	0.59	1.37	1.69
4	136	63	92	0.46	0.88	1.46
5	62	25	42	0.40	0.63	1.68
All	385	231	398	0.60	1.23	1.72

Rare events were the exclusive cause of identifiable failures in Category 1 (safety critical). The one report that was not classified as an RR contained a very sparse analysis that precluded a classification by our strict criteria. In the other critical categories (1N - 2N) rare events were clearly the leading cause of failure but not the only one. In all critical categories, where failures were due to rare events, the cause was overwhelmingly *multiple* rare events (see last column, RE/RR). For the combination of the first four severity categories, the RE/RR ratio exceeds 1.95, indicating that on the average the coincidence of two rare events was the cause of the failure.

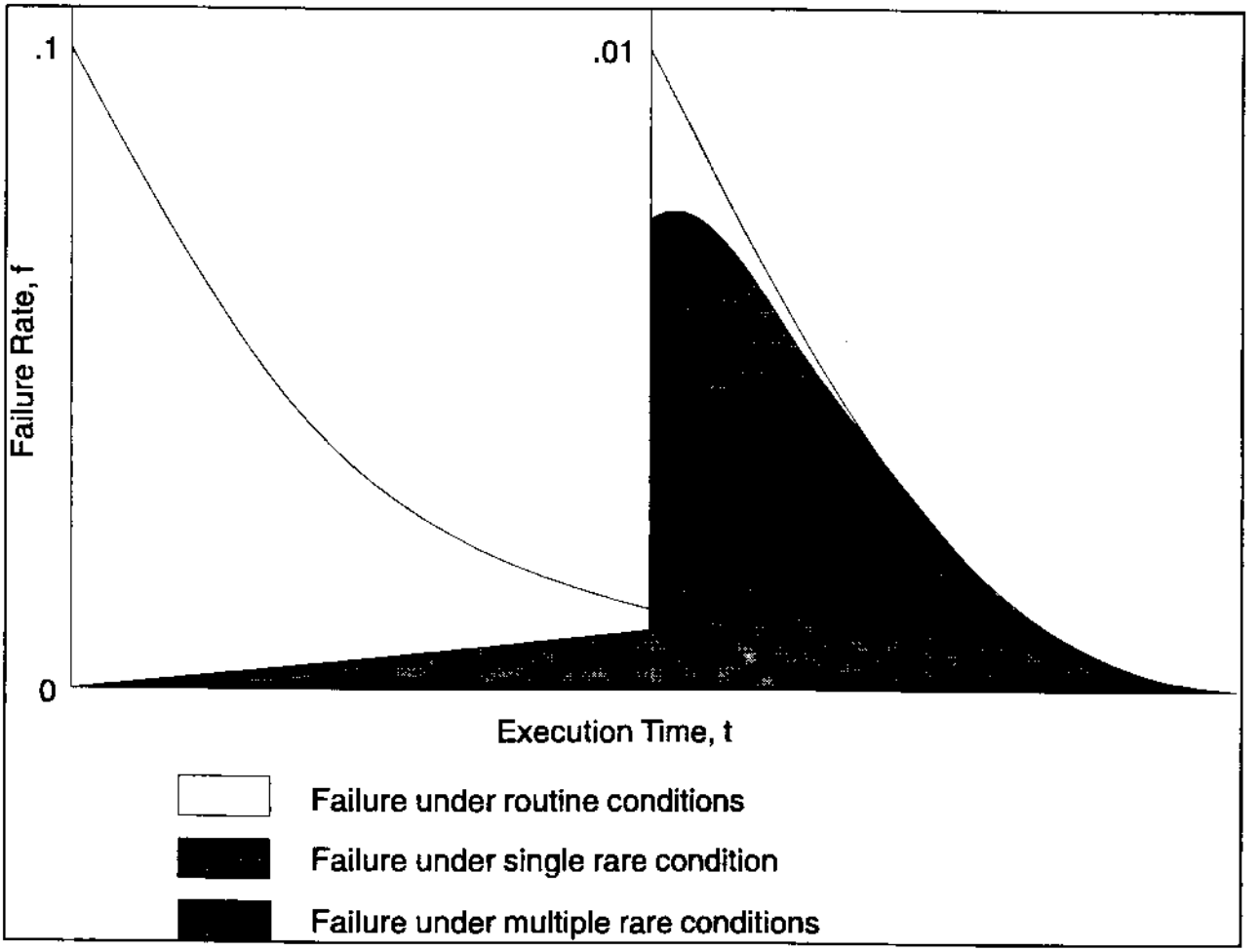
This shows that inability to handle more than one RE at a time is really at the root of the problem. At this point it is appropriate to ask "How often have we traced a thread involving more than one rare event?" and "How often have we concentrated on test cases that involved more than one rare event at a time?" The thoroughness of final testing in the shuttle program (in the post-Challenger environment) surfaced these weaknesses which probably would have been detected in most other situations only after they caused operational failures. Table 1 also shows that failures in Categories 4 and 5 are not primarily due to rare events, and that in Category 3 the importance of rare events is much less than in the more critical categories. One explanation for that is that program segments which cannot cause critical failures are tested less thoroughly and therefore arrive at the final test still containing some of the faults that cause failures under non-rare conditions. These findings, together with other data about the role of rare events in causing software failures [11], have led to the formulation of a hypothesis that is discussed in the next heading.

Qualitative Interpretation of Test Data

The hypothesis advanced here deals with using qualitative test data, such as the number of rare events necessary to produce a failure, to assess the probability of a catastrophic failures of the software in operational use. An overview of the test process that supports this hypothesis is shown in Figure 1 on the following page.

The overall test process is divided into two phases, not necessarily of equal duration: reliability growth and reliability assessment. At first reliability growth is relatively easy to assess from quantitative data, such as the fraction of failures per test run or per test hour. Initially, practically all failures occur under routine conditions (in frequently executed portions of the program). As the faults responsible for these initial failures are corrected, the failure intensity decreases. Subsequent test phases will identify faults in less frequently executed sections of the code where there is greater opportunity to encounter failures under rare conditions, such as initialization, exception handling, redundancy management, calibration and other maintenance support. As fewer failures occur during routine execution the rate of fault removal, and hence the reliability growth, decreases. When the failure rate data lack a consistent trend, we transition from the reliability growth phase to a reliability assessment phase where the failure rate fluctuates about a seemingly constant value. It will now be shown how valuable information about the effectiveness of test can be obtained from qualitative analysis of the observed failures during that phase.

The vertical line in the middle of the figure represents the transition from the reliability growth phase to reliability assessment. There is also a change of scale for the vertical axis, because beyond that point the failure rate is so low that it is difficult to show what happens at the original scale. Because the failure rate is already very low it is difficult to evaluate reliability growth by the conventional means in the presence of the normal stochastic characteristics of software failures. Instead, the suggested approach depends on the qualitative change in the failure modes, and particularly on entering a region in which the predominant failure type is due to at least two rare conditions.



Rare conditions cause the program to enter code that had not previously been executed, and where there is therefore a much higher probability of uncovering a fault than in previously executed segments. Rare conditions can be caused by:

- hardware failures: computer, voter, bus, mass memory, sensors, I/O processing
- environmental effects: high data error rates (e. g., due to lightning), excessive workloads, failures in the controlled plant
- operator actions: sequences of illegal commands that saturate the error handling, non-sensical command sequences (not necessarily illegal), logical or physical removal of a required component.

Whether data actually follow this graph will depend a lot on the test case generation strategy. The test cases should provide a population that is rich in individual rare conditions and the randomization among the stimuli should make it likely that multiple rare conditions will be encountered. As an example, in the nuclear environment test cases may be comprised of four

independent stimuli that represent success (routine operation) or failure (a rare event) for temperature sensor processing, radiation sensor processing, computer channel redundancy management, and software self-test, respectively. Assume that four random numbers are generated to represent the individual stimuli, and that the boundaries for routine and failure outcomes are selected so that for each individual stimulus there is 0.8 probability of success (routine operation). The probability of encountering rare events in a test case under these conditions is shown in Table 2. It is seen that this distribution will yield multiple rare event test cases with a probability of slightly over 0.18.

Table 2. Probability of Rare Events
Four simultaneous stimuli, each 0.8 probability of success

No. of Rare Events	Probability
0	0.4096
1	0.4096
2	0.1536
3	0.0257
4	0.0016

A criterion for declaring the reliability assessment phase successful is that the *most recent X failures* that have been observed *all involve multiple rare conditions*, and that the joint probability of encountering the multiple rare conditions is less than the allowable failure probability of the software. The key to computing the joint probability is that the probability of the individual events, though low, will be known with much greater certainty than that of the joint event. Data for the probability of the individual events can be (and actually is) collected from a much larger population than that for which software failure data can be collected. There are hundreds of sensors of a given type in use in the nuclear industry, so that failure rates as low as 0.05 per year will result in a sufficiently large number of failures to permit a good estimate of the mean time between failures.

The following example will show how this reasoning works for a given test case and how the X parameter can be selected. The simulated conditions that caused a specific failure are: (a) a faulty temperature sensor and (b) failure of a computer I/O channel. In operation the temperature sensor failure is estimated to be encountered no more often than once in 20 years and computer channel failures have occurred at a rate of one in 10 years. Since these conditions occurred on different components it is accepted that they are independent. Replacement of the temperature sensor takes 1 hour (0.0001 years), while replacement of the computer channel can be effected in 15 minutes (0.00003 years). The leading causes of the joint event that produced the failure are:

a. the temperature sensor fails while the computer is in repair -- the probability of this is

$$P[s|rep\text{-}c] = P[s]\ P[c]\ T[c] = 0.05 \times 0.1 \times 0.000\ 03 = 0.15 \times 10^{-6} \text{ per year}$$

b. the computer channel fails while the temperature sensor is in repair

$$P[c|rep\text{-}s] = P[c]\ P[s]\ T[s] = 0.1 \times 0.05 \times 0.0001 = 0.5 \times 10^{-6} \text{ per year}$$

where P[z] = probability of device z failing during 1 year (z = channel, sensor)
T[z] = time to repair device z in units of years (z = c, s).

The total probability of the joint event that produced the failure is therefore 0.65×10^{-6} per year. It is thus seen that at least for some failures caused by multiple rare events the probability of occurrence can be computed even though the probability of observing the actual failure may be negligibly small.

The cause of the particular failure described above will be corrected once it is has been identified, and thus the probability of the failure is no longer of practical concern. However, if the most recent failures that are being experienced during a period of random testing as described above are all due to multiple rare events with a joint probability of at most p per year, then it can be argued that the total failure probability of the software is of the order of p, as is explained below.

Assume that the random test case generation produces five test cases with routine or single rare conditions for each test case with multiple rare conditions (approximately the distribution shown in Table 2). If the probability of a failure due to a test case with multiple rare conditions is assumed to be equal to that of test case with routine or single rare conditions, this can be represented by drawing black (single) or white (multiple) balls from an urn that contains five black balls and one white ball. The probability of drawing a white ball at the first drawing is 1/6 or 0.17, of successive white balls in two drawings (with replacement) is 0.0277, and for three successive white balls it is less than 0.005. If three successive failures due to multiple rare events have been observed, it can then be concluded that the probability of failure under single and multiple rare events is not equal, and there is a basis for assigning a chance of less than 1 in 200 that the next failure will be due to a routine or single rare event. For four successive failures due to multiple rare events the probability that this is due to random causes is less than 1 in 1000.

Conclusions and Recommendations

Conventional quality assurance and test methodologies cannot be used to demonstrate that software for safety systems meets the typical regulatory requirements. Redundancy can be helpful but is still insufficient to meet the rigorous requirements. We have outlined a semi-qualitative technique that may supplement, or in some applications replace, redundancy as the

qualitative technique that may supplement, or in some applications replace, redundancy as the methodology of choice. The statistical reasoning used here is not claimed to be rigorous, and the choice of a test termination criterion will involve subjective factors. But the approach offers significant practical advantages as a software test methodology for safety systems. Further research and experimentation in this area can provide substantial benefits for arriving at an objective validation technique. The criterion is self-adjusting to the allowable failure rate. An extremely low allowable failure rate will require more testing because it will require that the multiple rare events encountered in test failures have a low joint failure probability.

References

[1] FAA Advisory Circular 25.1309-1A, U. S. Department of Transportation, 6/21/88

[2] Code of Federal Regulations, Vol. 10, Part 50, Appendix A, *General Design Criteria for Nuclear Power Plants,* Criterion 21 -- Protection System Reliability and Testability (as of January 1986).

[3] Reliability Test Methods, Plans, and Environments for Engineering Development, Qualification and Production, MIL-HDBK-781, Department of Defense, 14 July 1987/

[4] John Musa, Iannino,A. and Okumoto, K., *Software Reliability Measurement Prediction, Application*, McGraw-Hill Co. 1987

[5] N. G. Leveson, *Safeware*, Addison-Wesley, 1995

[6] S. Seth, W. Bail, D. Cleaves et al., High Integrity Software for Nuclear Power Plants, NUREG/CR-6263, U. S. Nuclear Regulatory Commission, June 1995

[7] H. Hecht, M. Hecht, G. Dinsmore et al., Verification and Validation Guidelines for High Integrity Systems, NUREG/CR-6293, U. S. Nuclear Regulatory Commission, March 1995

[8] D. E. Eckhardt and L. D. Lee, "An Analysis of the Effects of Coincident Errors on Multi-Version Software", *Proc. Computers in Aerospace V Conference*, pp. 370-373, October 1985

[9] D. E. Eckhardt, A. K. Caglayan, J. C. Knight, et al., "An experimental evaluation of software redundancy as a strategy for improving reliability", *IEEE Trans. Software Engineering,* vol 17 no 7, July 1991, pp. 692 - 702

[10] Herbert Hecht, "Rare Conditions - An Important Cause of Failures", *Proc. COMPASS'93,* Gaithersburg MD, June 1993

[11] Herbert Hecht and Patrick Crane, "Rare Conditions and their Effect on Software Failures", *Proceedings of the 1994 Reliability and Maintainability Symposium,* pp. 334 - 337, January 1994