1997 Hawaii Systems Conference, Maui, Hawaii, January, 1997

Test-Linked Specification For Safety-Critical Software

Herbert Hecht and Myron Hecht
SoHaR Incorporated
Beverly Hills, California
herb@sohar.com and myron@sohar.com

Abstract

The great difficulties that are encountered when reliability requirements for critical software have to be validated motivate an approach that facilitates testing for exceptional conditions that the software is expected to handle. It is shown that in several published studies, failures in previously tested critical programs occurred when rare events were encountered in the execution. Statement of requirements in the form of condition tables assures that all combinations of rare events that must be handled by the program are clearly recognized during development, and the condition tables can be translated directly into test cases that validate the conformance with these requirements.

Introduction

Safety-critical software is frequently required to have a failure probability of less than 10^{-6} per hour, and in some cases less than 10^{-9} per hour. That a program actually meets these requirements is impossible to demonstrate by conventional statistical interpretation of test results, even with perfect outcomes (no failures observed). In previous publications we have shown that failures in well-tested critical programs are almost exclusively due to rare conditions encountered during execution, and that in the most extensively tested programs failures are usually observed only when there is a coincidence for two or more rare conditions [HECH94]. Several authors have shown that weaknesses in the specification process make it easy for multiple versions to have the same faults, and that these are frequently associated with handling of rare conditions [KELL83, KNIG86].

Condition tables were introduced as a tool for test case selection by Goodenough and Gerhart [GOOD75]. David Parnas has pointed out that they provide a formal way of capturing complete requirements, including the response to rare conditions, and that such tables are more readily accepted and understood by engineers than the typical language-based formal specification methods [PARN91]. A typical condition table is shown in Figure 1, and for the time being just the format of the table will concern us; the content will be discussed later.

.95S"≤s1≤1.05S"	y	n	y	Y	y	n	n	n
.95S"≤s2≤1.05S"	y	y	n	Y	n	y	n	n
.95S"≤s3≤1.05S"	y	y	y	N	n	n	y	n
	S1=s1	S1=θ	S1=s1	S1=s1	S1=s1	S1=θ	S1=θ	S1=θ
	S2=s2	S2=s2	S2=θ	S2=s2	S2=θ	S2=s2	S2=θ	S2=θ
	S3=s3	S3=s3	S3=s3	S3=θ	S3=θ	S3=θ	S3=s3	S3=θ

Figure 1. Typical Condition Table

The upper left part of the table lists three conditions (in this case the range for signals s1, s2, and s3) and the subsequent entries in the rows pertaining to each signal signify whether the condition is met. The columns therefore specify a unique condition for the combination of the three signals, and it is quite apparent that this way of specifying the conditions can be directly translated into test cases. The bottom part of the table specifies the action to be taken when the three conditions prevail that are identified for that column above the heavy line. This is equivalent to specifying the outcome of the test case. In this table we are dealing only with binary conditions, and since there are three independent variables (signals) the number of combinations is $2^3 = 8$. Inspection shows that there are 8 columns, and thus the specification of the conditions is complete.

Application to a Feedwater Control System

The condition table shown in Figure 1 actually represents the specification for sensor validation for a digital feedwater control system for a power plant. Because this is an essential system, the water level is measured by three level sensors and the logic for accepting sensor signals for further processing was given in the textual specification (with some translation of local jargon into English) as:

The control system shall utilize the existing three level signals, which will be input and validated. Any level signal greater than ± 5% of the average shall be considered invalid.

The sentence contains a number of incorrect or ambiguous expressions, the worst one of which is (after some further translation) "Any level signal not within ± 5% of the average shall be considered invalid." Now assume that the sensor range extends from 0 to 100 and that the current correct reading is 60. One of the sensors fails and reads 0, while the other two continue to read 60. The instantaneous average reading is 120/3 = 40, and 5% range is 38 to 42. All signals are now invalid -- hardly the intended result. The explanation offered for this significant fault in the specification was that it was copied from the specification of the existing analog control system. The specification for the digital system had undergone several reviews, and the reviewers considered portions of the specification that had come from the existing system as "solid" and not in need of a thorough review. This is a fairly typical history of serious specification errors in critical systems.

The correct specification for a digital system would have referred to "± 5% of the average of the previous cycle". Now we return to Figure 1 and consider the content rather than the format of the condition table. The conditions in the upper left corner are stated in a simple mathematical notation which is defined below.

sn (n = 1, 2, 3): current cycle narrow range sensor signals
Sn (n = 1, 2, 3): current cycle validated sensor signals
S'n (n = 1, 2, 3): previous cycle validated sensor signals
S" = (S'1 + S'2 + S'3)/3 (average previous cycle signal)

In the first column of the table all three sensor signals can be validated. In the second column s1 falls outside the allowable range and can thus not be validated. These conditions are stated in unambiguous engineering terminology, and have a very much better chance of being correctly implemented than the textual format. The completeness of the requirements expressed in the table reveals that it is possible to transition from the normal (three valid signals) state to a state with only a single valid signal or possibly even to one with no valid signals. This, by itself, is a significant result of the use of this methodology. Admittedly, the probability of such a transition is low, but in an application as critical as the feedwater control system it should not be ignored.

The table can guide the construction of test cases that will provide a comprehensive validation that requirements have been met. The basic functionality will be tested by means of the eight combination of conditions that correspond to the columns in the table. Further test cases can be constructed to check for numerical accuracy at the transitions from the allowable to the non-allowable sensor signals.

Application to Realistic Specifications

The example cited above was kept to just three input conditions in order to permit an easy introduction to the concept of condition tables as a means of structuring software test. For a realistic software product a much greater number of input conditions must be expected, and the exponential relation between input rows and test condition columns may lead to an unreasonable number of test cases. The following paragraphs show how these difficulties can be overcome.

The most universally applicable technique is to adopt a hierarchical structure for the requirements (not the software structure that is generally unknown for non-developed programs). An example of such a requirements structure is shown in Figure 2. The requirements for feed water control are partitioned into three blocks for which requirements are expected to be independent. Thus, Sensor Monitoring is required to furnish validated sensor readings or no readings at all, Valve Actuation is required to furnish correct valve positions for all valid outputs of

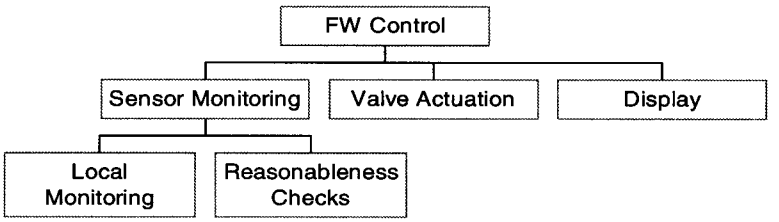

Figure 2 Requirements Structure

Sensor Monitoring, and Display is required to furnish a status indication of the system independent of the results of the other two requirement blocks. This structure permits separate condition tables to be generated for each of the lower level blocks and avoids the exponential expansion of required test cases. If the condition tables at that level are still too large, further decomposition of the requirements may be possible, as indicated here for sensor monitoring.

Where interactions of requirements cannot be avoided, the number of test cases can still be minimized by working with the action part of the individual condition tables (below the heavy line in Figure 1) rather than with the input conditions. Thus, if interactions between Sensor Monitoring and Valve Actuation cannot be ruled out, it still may be assumed that all actions that result in at least two validated signals affect Valve Actuation identically, that all actions that yield a single validated signal have an identical effect, and finally that a unique effect may prevail when there are no validated signals. Thus, the original eight input conditions have been condensed into three states for which tests of interactions with the Valve Actuation requirements may have to be conducted.

When the suspected interactions between requirements blocks cannot be grouped into a manageable number of output states, testing of random combinations of requirements from the individual blocks may be considered. This approaches the random test case generation which has been found to be an efficient means of obtaining test coverage in the Halden experiments that are discussed later in this paper.

Relation to Prior Research

When a program with a fixed number of faults is subjected to repeated testing under a random set of test cases, the faults tend to result in failures in a reasonably constant order. This has been known at least since 1982 and holds true whether faults during a given set of test cases are fixed as they are encountered or not [NAGL82]. Thus, some faults tend to be detected earlier while others tend to be detected much later. The principal cause for this order in the detection sequence is the place in the software structure in which the fault resides. The opening statements in a program, e. g., the ones that read in data or select the processing path based on the data values, tend to be accessed during every run, while specific processing steps are traversed only for certain data ranges. Thus, faults in the opening statements are much more likely to be detected and will tend to be detected earlier than faults in segments that will be executed only under a restricted set of data values. The faults least likely to be accessed during normal execution are those in the exception handlers or routines called by these, and in a typical test program these will be discovered last *or not at all.* Thus, in a previously tested program the most likely areas to still contain faults are those dealing with rarely encountered conditions and exception handling. This conclusion holds an obvious lesson for the design of test cases, but that lesson seems to have escaped a large number of software designers of high academic standing that participated in the project described in the next paragraph.

The table shown in Figure 3 has been constructed from published data on the multi-version software experiment that involved 5 programmer teams from each of four leading software engineering faculties in the U. S [ECKH91]. The objective of the experiment was to determine the likelihood of related faults in independently designed and coded programs. These findings could then be interpreted into the probability that voting among three versions (or, more generally, N-versions) of programs independently developed from a

No. prior anomalies	Observed failures	Total no. of tests	Failure fraction
0	1,268	134,135	0.01
1	12,921	101,151	0.13
2	83,022	143,509	0.58

Figure 3. Results of individual tests on the accelerometer selection software

common requirement will lead to correct results even if the individual programs do contain faults. The common requirement for each of the 20 programs was to compute an acceleration vector from an array of six non-orthogonal accelerometers. Since only three independent accelerometers outputs are required to compute the acceleration vector, the program should be able to give correct results as long as no more than three accelerometer failures were encountered. Each of the expert programming teams was responsible for testing their own program. The tests reported in the table were conducted on the delivered (and tested) programs and consisted of introducing one anomaly into a previously good accelerometer. The number of prior anomalies (that existed at the beginning of the run) is listed in the first column. The test reported in the last row started with two prior anomalies and concluded with three; thus still within the envelope of the requirements of the program.

It is seen that prior testing removed practically all faults associated with the handling of the first accelerometer failure (no prior anomalies), but was much less effective in detecting faults associated with second or third failures. The results shown in the last row suggest that very few of the tests conducted by the individual teams subjected the programs to runs in which a third failure was introduced on top of two prior failures. Yet, any analysis leads to the presumption that programming to overcome the effect of a third failure will be more difficult than the management of a lesser number of failures, and that the code should be carefully tested for performance under third failure conditions. As has been pointed out in the preceding section, the condition table methodology is very effective in emphasizing the need for testing for multiple failure conditions and its use would have reduced or even eliminated the disparity in the failure fraction between the first and last rows in the above table.

A further example of the need for testing for multiple failure conditions is shown in the following example from the final acceptance testing of the Space Shuttle Avionics (SSA) software.. The analysis concentrated on the severity of the consequences of the failures and on the conditions that initiated the failure. It was found that the majority of failures, and particularly in the highest severity categories, occurred under input conditions that contained at least one rare event (RE), a condition not likely to be encountered in routine operation [HECH94]. Examples of rare events in that environment include loss of main engine thrust, very unusual or unauthorized crew procedures, and computer hardware failures.

When at least one RE was responsible for the failure the corresponding failure report was classified as a rare event report (RR). The data were collected during the acceptance test for release 8B of the program, the first flight program immediately after the Challenger accident. The program had undergone intensive test prior to the period 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.

Rare events were clearly the leading cause of failures among the most severe failure categories (1 - 2N) and were an important cause among all reports in this population. In the less critical failure categories (which presumably came from code segments that had not undergone as thorough prior testing) a greater fraction of faults was due to causes not directly related to rare events. The number of rare events per rare report (RE/RR, the entries in the last column) remains relatively constant for all severity classes around the average of 1.72. This indicates that inability to handle more than one RE at a time is really at the root of the problem, and again highlights the merits of the condition table approach.
Classification by the number of rare events in the conditions that caused the failure clearly shows the merit of the condition table approach in assuring that rare events and multiple rare events are adequately covered in both development and testing. At this point it is appropriate to ask "How often have we traced a thread

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

Figure 4. Analysis of Space Shuttle Avionics Software

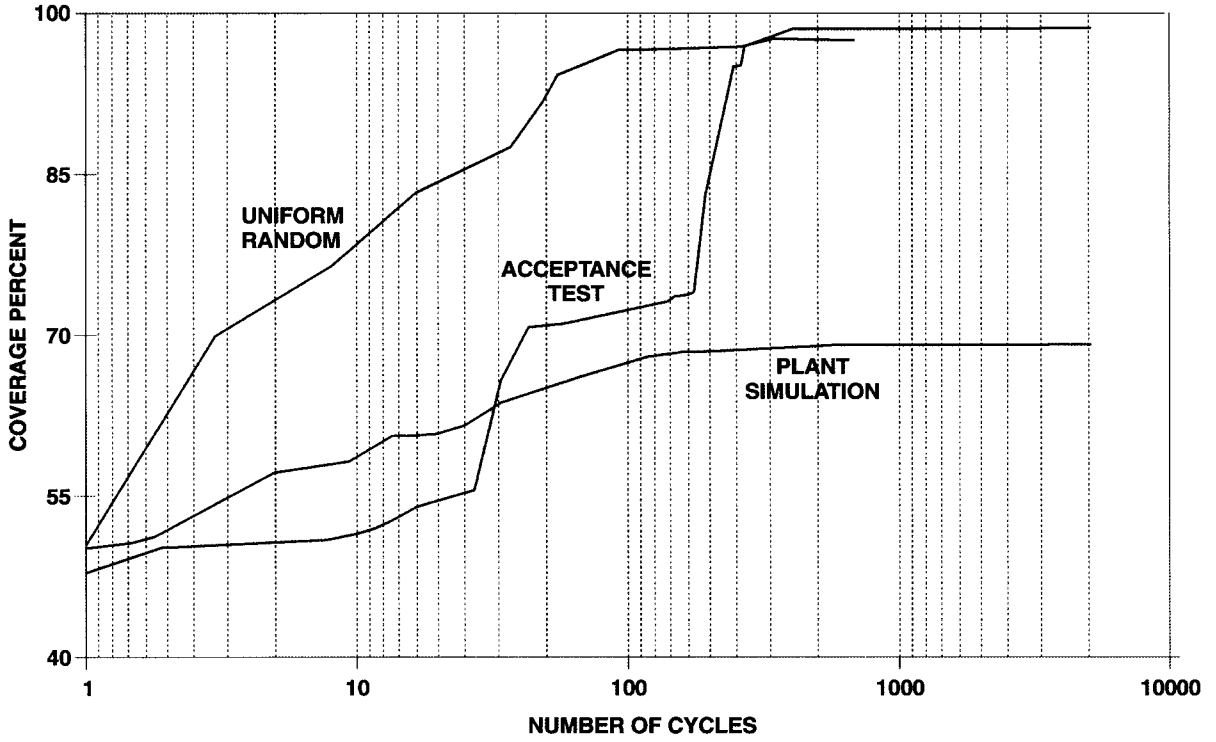

Figure 5. Effect of Test Strategies

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 usual answers to these questions indicate that it is quite likely that programs will fail when multiple rare events are encountered. The thoroughness of final testing in the shuttle program surfaced weaknesses which probably would have been detected in many other situations only after they caused operational failures.

A final example from prior research that confirms the benefits of the approach suggested above comes from the testing of nuclear power plant safety systems at the European nuclear test facility at Halden, Norway [DAHL90]. A program was tested by means of several strategies of which Figure 5 shows the three most significant ones. The ordinate represents branch coverage achieved by test, and a separate section in the referenced document indicates that the number of faults found was roughly proportional to branch coverage. Testing by means of the plant simulator subjected the program to a large number of plant conditions but did not include any failures in the computer or data systems. The test never entered the branches of the program that dealt with those failures. Testing by the acceptance test procedure(essentially, requirements based testing) and uniform random testing both ultimately reached about 98% branch coverage, but the uniform random testing reached a high level of coverage with considerable fewer test cycles and this was interpreted as an advantage of that methodology. Examination of the acceptance test procedure shows two abrupt steps, where coverage increased rapidly in very few test cycles. In a discussion with the authors of the paper is was found that the acceptance test procedure first dealt with all normal functional requirements, then with mildly disturbed conditions, and finally with really rare conditions. If the order of testing had been reversed the acceptance test procedure would probably have surpassed the efficiency of the random test generation. This demonstrates how important it is to have a requirements format that emphasizes the exceptional states that a program will have to handle.

REFERENCES

[DAHL90] G. Dahll, M. Barnes, and P. Bishop, "Software Diversity -- A way to Enhance Safety?", *Proc. Second European Conference on Software Quality Assurance,* Oslo, May 1990

[ECKH91] 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

[GOOD75] J.B. Goodenough and S.L. Gerhart, "Toward a Theory of Test Data Selection", *IEEE Transactions on Software Engineering*, Vol. SE-1, No.2, pp.156-173, June 1975

[HECH94] 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

[KELL83] J. P. J. Kelly and A. Avizienis, "A Specification-Oriented Multi-Version Fault-Tolerant Software Experiment", *Proc. FTCS-13,* PP. 120-126, Milan, Italy, June 1983

[KNIG86] J. C. Knight and N. G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming", *IEEE Transactions on Software Engineering,* Vol. SE-12, No. 1, 1986

[NAGE82] Phyllis M. Nagel and James A. Skrivan, "Software Reliability: Repetitive Run Experimentation and Modeling", NASA CR-165839, Februrary 1082

[PARN91] D. L. Parnas, G. J. K. Asmis, and J. Madey, "Assessment of Safety-Critical Soft ware", *Proc. 9th Annual Softw. Reliability Symp.,* Colorado Springs CO, May 1991