Alternative Measures of Software Reliability

Herbert Hecht and Myron Hecht
SoHaR Incorporated, Culver City, California

Abstract

Software failure rate, long a primary reliability measure, is difficult to apply in a distributed computing environment with an uncontrolled number of active nodes and usage patterns. Since it is not useful as a measure, it is also not a good prediction tool during software development. Instead we assess reliability of software segments in terms of the severity of system level effects of their failure modes and the extent of the protection (fault detection and recovery) that is built into the program. Where the protection provisions cover all failure modes that can cause high severity failures, and where the effectiveness of the provisions has been established by test, the program may be considered reliable even in the absence of a quantitative failure rate assessment.

We describe UML based FMEA procedures as a means of focusing of V&V activities on coverage and effectiveness of protective measures, and recommend pursuit of these procedures as a means of reducing the cost of high reliability software.

Key Words: Software reliability, software robustness, UML tools, software FMEA

1. Introduction

We were successful in developing a computer-aided procedure for software FMEA except for the issue of software failure probability. Strictly speaking, failure probability is required for criticality analysis rather than FMEA in the narrower sense, but most hardware FMEA tools provide a column in the FMEA worksheet in which the part or assembly failure probability for a given mission phase is entered. To generate a system FMEA that includes hardware and software we need an expression for software failure probability for small code segments (equivalent to hardware parts). We decided to leave that task to some future geniuses and to focus on an alternative assessment of software reliability.

Our approach avoids two problems: (1) in a typical distributed computing environment the number of failures per time interval is much more determined by usage factors than by software reliability, making it necessary to find new approaches to the measurement of software reliability; and (2) software reliability cannot be predicted bottom-up in the way in which hardware reliability can be predicted from established parts failure rates.

Of course there are circumstances software and hardware failures can be subjected to the same statistical treatment as will be discussed in the next section. Also, it is recognized that the development and application environments affect failure probability but that does not help in estimating the failure probability of a given small code segment in the same sense as a schematic permits us to estimate the failure probability of a small electronic assembly. This dilemma is treated in Section 3 where we are concerned with reliability evaluation prior to coding. Once code is available for testing we should be in a better position to assess reliability, and the steps necessary for realizing this are discussed in Section 4. Conclusions and recommendations are presented in the final section of this paper.

2. Same Measure – Different Meaning

Some of the earliest quantitative work on software reliability concerned the decreasing failure rate during test[1, 2]. These models were soon followed by others, the most widely used one of which was that proposed by Musa[3]. All of these models shared the following assumptions:

- A large program being tested in a fixed computer environment
- Tests were being conducted by the developer
- Errors found during test were immediately corrected (or if they caused additional failures these were ignored)

The models differed from each other in addressing discrete (time to next failure) or continuous (failure rate) measures and in the failure reduction mechanism that was postulated. Since then several other software reliability growth models have been proposed that differ from those mentioned here primarily in the mathematical expression for the failure reduction process. The AIAA Recommended Practice for Software Reliability[4] lists these models as well as tools that facilitate their implementation.

A factor that contributed to the wide acceptance of the Musa model was the emphasis on execution time as the denominator of the failure rate expression. Other models had allowed calendar time or equivalent intervals to be used, and that had led to inconsistent results except where a very closely regulated test schedule was used (the same number of hours in each calendar period).

Quite another interpretation of software failure rate is encountered when software is in operation, say in a data processing center, where management is concerned about uninterrupted service. The following hypothetical failure statistics illustrate this issue.

Table 1. Data Center Monthly Failures

Month	Hardware	Software
January	18	11
February	14	12
March	18	14
April	15	16
May	12	20
June	11	24
July	13	23

The data center manager, Mike, is convinced that he is faced with a major software reliability problem. He calls a meeting where the software development supervisor, Sam, claims that the number of failures attributed to software has nothing to do with software reliability but is merely a reflection of the work environment. To support his position Sam produces the data shown in Table 2 and explains

- Corrective change requests, an indication of software faults encountered, and an index of reliability, have sharply decreased over the seven month period
- Total software change requests have gone up, increasing the workload of the department and causing slower response to corrective change requests; this increases the time that a fault remains in the operational software and the opportunity for it to cause failures

• The increasing number of adaptive and perfective change requests indicates that the program is being used much more often and thus a larger number of failures can be expected due a single fault.

Table 2. Monthly Software Change Request Summary

Month	Total	Corrective	Adaptive	Perfective
January	9	8	1	0
February	10	8	2	0
March	10	7	2	1
April	11	5	3	3
May	12	6	2	4
June	14	5	3	6
July	13	4	5	4

We don't want to adjudicate the conflict between Mike and Sam, but we can understand their positions. Is there any denying that the monthly number of software failures has roughly doubled since the beginning of the year? And isn't it true that the number of new faults found per month has been cut in half? The most striking observation is that the concept of software failure rate that is central to all the models mentioned in the beginning of this section is not easily applied in an environment where

- The program is running on multiple computers with no control on execution time
- Inputs represent random user requests
- Operation cannot be interrupted to diagnose failures and correct newly found faults

We should also ask the data center manager whether the number of failures is really the significant operating index. Perhaps the time lost due to the failures, or the effect of failures on the operation of the enterprise may be of more importance, and if they are, what software engineering measures are suitable?

This example has shown how difficult it is to obtain a meaningful measure of software reliability even in an environment where data are freely available. We now turn to the problem of predicting software reliability, first during the early stages of development and then after code is available for testing.

3. Software Reliability before Implementation

When the need for a new program arises in an environment where avoidance of failure is a top concern it is usual to formulate requirements for reliability based on the experience with predecessor systems. Typical expressions from different environments are

- Not more than one system-wide loss of service a year
- Availability of core services of at least 0.9999999 (0.9_7)
- Restoration time not to exceed 10 minutes

All of these expressions are meaningful at the system level and may affect the top level program design but they are very difficult to translate into requirements for a major program segment and they become irrelevant at the module level. How, then can we obtain some assurance that the requirements can be met as the program is being developed?.

We will describe a methodology for gaining insights into potential reliability problems during the early program development by use of artifacts generated by UML tools[5]. The approach, but not the tool that we developed, can also be used in non-UML environments. As an example we use the software for autonomously switching from an active component to a back-up (standby) component in case of failure of the active one. Such programs are frequently used in satellite systems, communication networks and process control. Figure 1 shows a *use case diagram*[6] for a plant control system where the two systems manage their roles autonomously, primarily with the aid of exchanges of heartbeat (HB) data. The use case diagram is normally the first formal document created for program development.

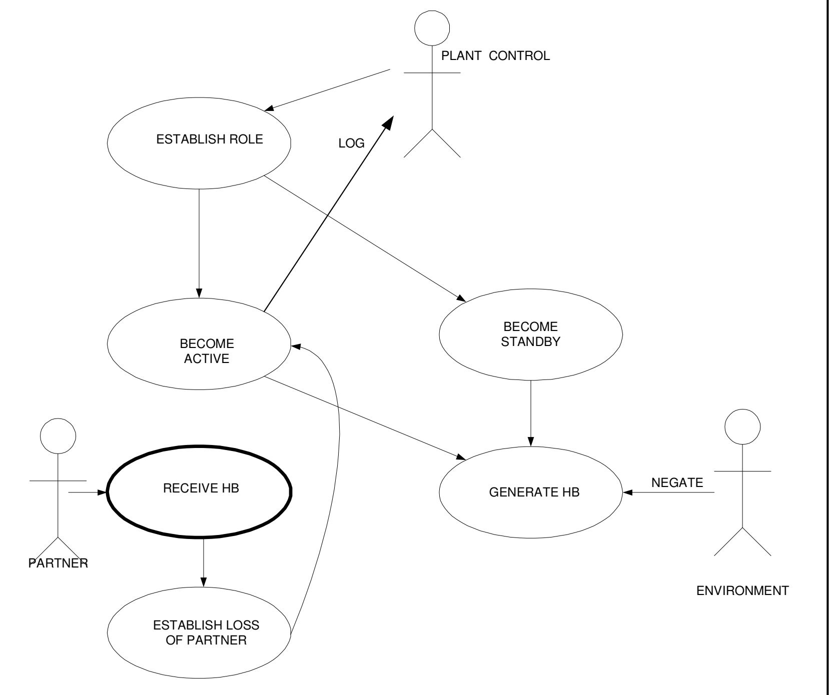

Figure 1. Active/ Standby Use Case Diagram

In the diagram the stick figures are called *actors* but they are not necessarily persons. In our example the environment and the partner (the corresponding computer program running on the partner system) are certainly not persons, and the plant control may or may not be one. The ovals are the specified *methods* and these become the elements for which failure modes are analyzed. The directed lines or arcs denote information flow and hence the paths through which failure effects propagate.

The following discussion concentrates on the *Receive HB* method (the heavy framed oval). The most common failure mode for this method is failure to report receipt of a heartbeat when in fact it was sent by the partner. If this happens when own program is already in the active role the only action is to log the failure for external analysis. If it happens when own program is in the standby role it will transition to the active role and notify plant control. Again, this is a low severity failure. But the *Receive HB* method *may* also have a failure mode in which it signals receipt of heartbeats when the partner does not generate them. That failure mode may disable the entire plant communication system in the following scenario: Partner fails while in active mode. Own program does not note absence of heartbeats and does not take over. To determine whether this failure mode is likely to happen we need to examine the details of the *Receive HB* method, shown in Figure 2.

A valid heartbeat consists of three evenly spaced pulses over a defined interval. The *Counter* method counts the pulses and when it receives three pulses sends a message to the

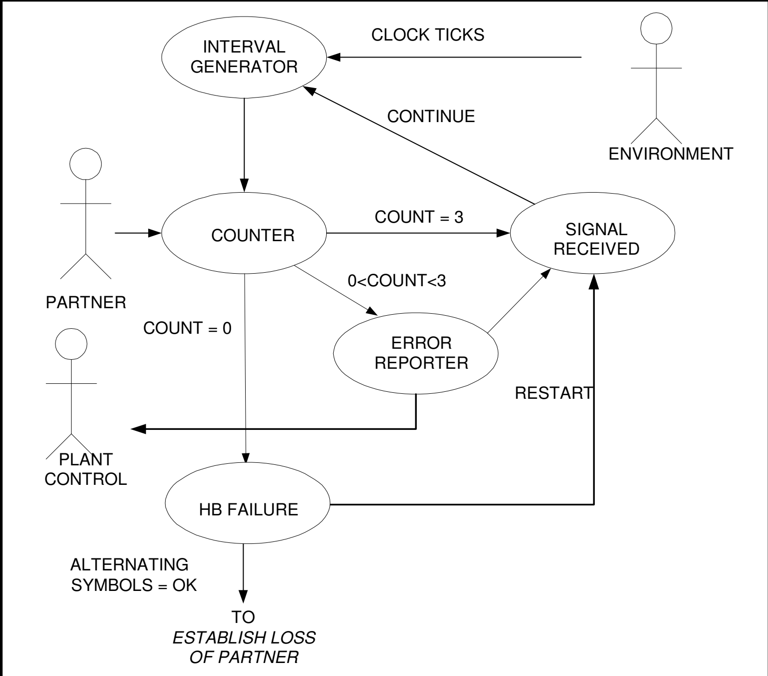

Figure 2 *Receive HB* Use Case Diagram

Signal Received method. If no pulses are received it sends a message to the *HB Failure* method. The latter method waits three HB cycles before initiating actions appropriate to loss of partner. Thus, the failure modes that prevent recognition of a loss of partner are (i) spurious generation of exactly three pulses per interval in *Counter* and (ii) spurious transmission of defined alternating symbols by *HB Failure.* Both of these conditions will have to exist for at least three HB cycles to affect the actions at the *Active/Standby* level.

When pulses are received but their number is less than three a message is sent to the *Error Reporter* method. The first time this condition is encountered the *Error Reporter* sends a message to the *Signal Received* method and normal processing continues. If the counter continues to report one or two pulses, an error report is sent to an exception handler (shown as part of plant control) that has visibility of plant performance. If the performance is normal, the currently active module remains in control and the counter method in the standby controller is diagnosed as possibly defective. If there is any indication that the plant performance deviates from normal the standby module is commanded to become active.

An FMEA worksheet for the *Receive HB* method based on Figure 2 is shown in Table 1. The level of detail is comparable to a parts approach for hardware portions of the system but no failure probability is listed. The entries in the severity column range from II (loss of service) to IV (minor). Severity I is usually reserved for failure effects that can lead to fatalities. No entry in the last column means that the service is not affected.

The worksheet reveals both strengths and weaknesses of the *Receive HB* method. The important strength shown is that severity II failure effects will occur only under highly unlikely circumstances: spurious count of exactly 3, and spurious generation of a defined alternating sequence. A weakness is that internal fault detection (self-test) can detect only failure to furnish output. The method depends primarily on external detection (Plant Control) to recognize spurious outputs. But because the likelihood of these events occurring at all is very small (specific conditions have to be met to constitute an acceptable message) the dependence on external detection may be acceptable in most circumstances.

Table 1. FMEA for Figure 2

ID	Item/Function	Failure Mode & Causes	Local Fail. Effect	Failure Detection	Compen-sation	Seve-rity
1.1.1.1	Interval Generator	No interval started. Loss of clock ticks or internal failure	HB failure	Self-Test	Note 1	IV
1.1.1.2	Interval Generator	Long interval. Missing clock ticks or internal failure.	HB failure	External	Note 1	IV
1.1.2.1	Counter	No count. External or internal failure	HB failure	Self-Test	Note 1	IV
1.1.2.2	Counter	Spurious count exactly = 3. Internal failure	HB failure	External	Note 2	II
1.1.2.3	Counter	Spurious count 1 or 2. Internal failure	Spurious HB	Error Reporter	Note 3	IV
1.1.3.1	HB Failure	Does not send Restart. Internal Failure	None	Self-Test	Note 4	
1.1.3.2	HB Failure	Spurious Restart. Internal Failure	HB Failure	External	Note 1	IV
1.1.3.3	HB Failure	No or random output to *Loss of Partner*. Internal failure	HB Failure	External	Note 1	IV
1.1.3.4	HB Failure	Spurious defined alternating signals	Spurious HB	External	Note 1	II
1.1.4.1	Signal Received	No *Continue* output. External or internal failure.	HB Failure	Self-Test	Note 1	IV
1.1.4.2	Signal Received	Spurious *Continue* output. Error in input processing	None	External	Counter	
1.1.5.1	Error Reporter	No output to *Signal Received*. External or internal failure.	HB Failure	Self-Test	Notes 1, 5	IV
1.1.5.2	Error Reporter	No output to Plant Control. External or internal failure	None	External	Note 5	
1.1.5.3	Error Reporter	Spurious output(s). Internal failure	None	Note 6		

Note 1: Temporary failure effects are suppressed because the *Loss of Partner* method waits for three intervals to activate.
Note 2: No effect if active or if partner active and operational. Otherwise Severity II
Note 3: No effect if single occurrence.
Note 4: Will cause no effect when count >0.
Note 5: Will produce any effect only under the extremely unlikely condition of count=1 or 2 and another significant failure.
Note 6: Spurious outputs will be detected by *Signal Received* and Plant Control, respectively.

Let us analyze whether the current implementation for spurious heartbeats (ID 1.1.2.2) meets the first of the bulleted requirements at the beginning of this section - not more than one system-wide loss of service a year. We assume that a component failure that causes cessation of heartbeats occurs once a month and that it is repaired in one hour. The component unavailability is therefore 1/720 = 0.0014. One half of these failures will occur in the active component where spurious heartbeats do not cause a failure effect. Thus, the fraction of vulnerability is 0.0007. Several hypothetical levels of probability of undetected spurious heartbeats and the corresponding annual loss of service probability are shown in Table 2. The *Receive HB* method may not be the only one that can cause loss of service, but even if there are several dozen of equally contributory methods the risk of violating the loss of service requirement is acceptably low. We have reached this conclusion without quantifying software reliability at the module, method, or line of code level. The component failure probability (used in the fraction of vulnerability calculation) is governed by hardware reliability and thus it can be expected that quantitative data will be available.

During the pre-implementation phase the primary purpose of software reliability assessment is to establish the general feasibility of a design and to identify areas where improvement may be necessary, such as the detection of some failure modes in our example. Once code becomes available the emphasis shifts to identification of all methods that can give rise to high severity failures and thorough exploration of these by review and test. A methodology for this is outlined in the following section.

Table 2. Probability of Loss of Service
(Calculated for 8600 hours of service)

Probability of Spurious Heartbeats	Probability of Loss of Service
0.01	0.06
0.001	0.006
0.0001	0.0006

4. Software Reliability during Testing

We organize the reliability assessment during the implementation phase with the aid of an expanded FMEA worksheet that is adapted from the former hardware-oriented MIL-STD-1629[7]. An example of this worksheet together with the UML artifacts that supply the pertinent software information is shown in Figure 3. Note the absence of a failure rate or probability column. Although such data are customarily provided in hardware FMEA worksheets they were not required in the standard, where only order-of-magnitude failure probabilities were used in criticality assessment. Our approach adapts this guidance to software.

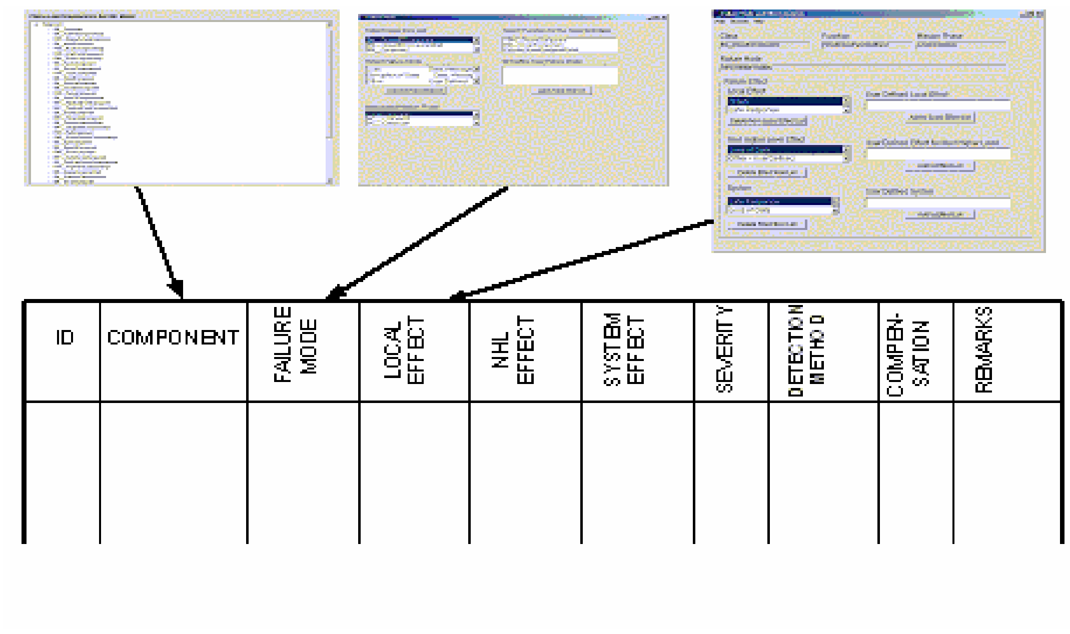

Figure 3. Example of FMEA Generation

The identification number (ID) for failure modes is usually a hierarchical construct of type *aa.bb.cc.dd* where *aa* is the index of a major software component (e. g., configuration item), and the other indices refer to successively lower indent levels. There is no limit to the level of indentation that can be used. Once the lowest level is reached, usually a *method* in UML nomenclature, the failure modes can be identified by letters that have mnemonic significance (e. g., *s* for stop, *i* for incorrect result) or by numeric suffixes.

The component name is obtained from the UML class chart as shown at the top of the page, and the ID will be automatically assigned based on the indenture levels of the listing. The extraction of the component name from the listing is a very significant step, and the one that permits a claim of objectivity and completeness for this technique. The UML class chart can be considered equivalent to the parts listing in a bill of materials for hardware FMEA.

Failure modes are assigned in a computer-aided rather than fully automated manner but automation may become possible in a given programming environment as experience is gained. Details of the failure modes assignment and of the association of failure effects with a given failure mode are discussed later. The severity is directly associated with system level effects and can be assigned automatically in a given project context.

Figure 3 shows a worksheet for only a single operational mode whereas most practical systems have multiple modes or phases that can result in significantly different failure effects. An example is a spacecraft that has test, launch and on-orbit modes. Failure modes of the guidance software that have very severe effects in the launch mode have only marginal effects in the other two modes. Another column can be added to the FMEA worksheet to denote the operational phase or mode for which failure effects are evaluated.

In 2002 Haapanen and Helminen[8] published a survey of the literature on software FMEA. It listed over 20 different failure modes, including hang, stop, missing data, incorrect data and wrong timing of data. Many of the distinctions are not necessary when the emphasis is on evaluation of failure effects and identifying the areas of greatest risk.

All program classes and methods have the potential of causing a *crash*, cessation of processing with possible impairment of computer resources, and a *stop*, cessation of processing without impairment of computer resources and usually with a diagnostic on the location of the stop. Individual methods that output data have a further failure mode of *faulty message*. For other cases *failure to initialize* and *failure to release memory* may also have to be considered. To handle conditions that do not fit these predefined failure modes our taxonomy lets the user define other failure modes. When a new program method is accessed by the analyst, the basic failure modes are automatically entered for it and others can be added by the analyst.

Responsible software developers recognize the possibility of critical failure modes and protect against them by assertions, checksums and in-line tests. It is thus only necessary to postulate typical failure modes to check for the presence of defensive programming constructs, the key to our reliability assessment. The FMEA worksheet recognizes two aspects of defensive programming: detection and compensation (recovery). Detection by itself can at least convert a crash into a stop (where environment data can be furnished for diagnosis). But compensation in the form of re-try, restart, assignment of default values, or invocation of alternate routines can provide a much higher degree of protection. Table 3 shows how the translation of lower level failures is affected by protection at the higher level (or at the output of the lower level). The capture of fault detection and compensation provisions for entry into the FMEA is made possible by the "tag" construct of UML.

Table 3. Translation of Lower Level Effects

Higher Level Protection	Lower Level Effect			
	Crash	Stop	Delayed Output	Degraded Output
None	Crash	Crash	Crash	Degraded output
Detection only	Crash	Stop	Stop	Degraded output
Detection and re-try	Crash	Stop	Delayed output	Degraded output
Detection and default value	Degraded output	Degraded output	Degraded output	Degraded output
Alternate method	None	None	None	None

Earlier in this paper we discussed the difficulty of evaluating the reliability of low level software segments, and probably few practitioners needed our exposition of this issue. The visibility into protection and compensation provisions afforded by the FMEA, and the effect of these on operation at the higher level shown in Table 3 permits us to base the reliability assessment primarily on the presence and nature of the protection.

This also has important implications on the formulation of test plans. Of course all functional requirements need to be tested, but testing for reliability should concentrate on the verification of the protective provisions. Because the protective measures are largely similar, if not identical, within broad classes of programs, the generation of test cases can be automated or at least greatly simplified. The essential questions for reliability assessment thus become

- Are methods and higher level constructs that can cause high severity failures covered by detection and compensation provisions?
- Has the effectiveness of these provisions been proven by test?

If the answers to these questions are in the affirmative the need for a numerical estimate of the software failure rate may be much less urgent but there is still the problem of being consistent with the hardware practice when a system level FMEA is to be generated. We suggest that the software reliability assessment described here may in many instances also be applicable to hardware, and that a system reliability assessment format may evolve in this direction. The bottom-up calculation of failure probability was historically convenient and could be automated. But it has long been recognized that system failures can arise from causes other than parts failures, and thus alternatives to the parts failure rate summation may be desirable for reasons other than consistency with software practice.

The preceding discussion has exclusively dealt with UML based software development but the general principles can also be applied where UML tools are not used. The benefit of the automation will of course be lost.

5. Conclusions and Recommendations

The concept of software failure rate that is the basis of most software reliability models is appropriate for programs running on a fixed population of computers and in a stable computer usage environment. In a distributed computing environment, and particularly when new users are constantly being added, the number of failures per unit time is affected by events (e. g., number of times a given segment is accessed) that are not properties of the software. Therefore the number of failures per time interval does not translate directly into a measure of software reliability.

Given this problem in measuring software failure probability, it becomes very difficult to formulate a meaningful failure probability prediction during program development. Instead we assess reliability of software segments in terms of the severity of system level effects of their failure modes and the extent of the protection (fault detection and recovery) that is built into the program. Where the protection provisions cover all failure modes that can cause high severity failures, and where the effectiveness of the provisions has been established by test, the program may be considered reliable even in the absence of a quantitative failure probability assessment.

We recommend that the application of the FMEA procedures described above, and focusing of V&V activities on coverage and effectiveness of protective measures, be pursued as a means of reducing the cost of software development for applications requiring high reliability.

References

[1] Jelinski, Z., and P. B. Moranda, "Software Reliability Research" in *Statistical Computer Performance Evaluation,* W. Freiberger, ed., pp. 465 – 484, Academic Press, New York, 1972

[2] Shooman, M. L., "Probabilistic Models for Software Reliability Prediction" in *Statistical Computer Performance Evaluation,* W. Freiberger, ed., pp. 485 - 502, Academic Press, New York, 1972

[3] Musa, J. D., "A Theory of Software Reliability and its Application", *IEEE Transactions on Software Engineering,* SE-1 (3), pp.312-327, 1975

[4] ANSI.AIAA R-013-1992, American National Standard, *Recommended Practice for Software Reliability,* AIAA, Washington DC, 1993

[5] Boggs, Wendy and Michael, *Mastering UML with Rational Rose*, Sybex, 2002

[6] Rosenberg, D., *Use Case Driven Object Modeling with UML,* Addison-Wesley, 1999

[7] Department of Defense, "Procedures for Performing a Failure Modes, Effects and Criticality Analysis", AMSC N3074, 24 Nov 1980 (the standard is no longer active but is still widely used)

[8] Haapanen Pentti and Atte Helminen, "Failure Mode and Effects Analysis of Software-Based Automation Systems", *STUK-YTO-TR 190*, August 2002