MORE EFFECTIVE V&V BY USE OF SOFTWARE FMEA[†]

Herbert Hecht, Xuegao An and Myron Hecht
SoHaR Incorporated, Culver City CA

1. Introduction

Verification and validation of embedded software for spacecraft, launch vehicles and aircraft is a very costly and lengthy activity, and one for which there are at present no clear completion criteria. An experienced V&V team will make use of its past experience and of the Lessons Learned material available from the NASA website[1]. But there always remains the question about the applicability of the experience to the environment of the new software and about the inherent subjectivity of the V&V process.

Safety and reliability review of hardware benefits from being organized around the FMEA (and other established analyses), and when a part level FMEA is prepared by a responsible organization there is reasonable assurance that all single failure modes have been addressed, and that high severity failure modes have been identified. Up to now no software equivalent for a part level FMEA has been possible. Software FMEA has been largely confined to functional blocks, the selection of which is subjective and for which no completion criterion exists.

Model-based software development, particularly when it utilizes UML tools, provides a discipline and artifacts that make programs more transparent. We use these capabilities to automate significant steps in the generation of software FMEA. Automation not only reduces the labor required but also makes the process repeatable and removes many subjective decisions that have previously impaired the credibility of software FMEAs. The computer-aided software FMEA discussed in this paper can be the central organizing element for the verification and validation (V&V) of embedded software for real-time systems. The adoption of this technique provides large economic benefits because V&V frequently consumes the majority of the development resources for embedded software.

The role of software FMEA in V&V and previous work in software FMEA generation are described in the next section, followed by an overview of the automation technique. Section 4 provides details of extraction of failure mode information from the UML development tool, and Section 5 describes the translation of failure modes to failure effects. The last section discusses the identification of detection and compensation provisions for the failure effects and the utilization of the Remarks column..

This research was conducted as part of the MoBIES[*] project, sponsored by DARPA/IXO. The authors want to acknowledge the encouragement received from Dr. John Bay of DARPA and Raymond Bortner of AFRL. The software used in the examples is part of the Open Control Program developed by Boeing St. Louis for the MoBIES project.

2. The Role of FMEA in V&V

A large part of the cost of developing critical, and particularly embedded, software is attributable to verification and validation activities intended to provide assurance that

- The software satisfies the specification
- Creditable failure modes that can cause serious consequences have been eliminated or compensated for (e. g., by alternate processing provisions)
- Required attributes have been provided (e. g., modifiability)

[†] Portions of the research leading to this paper were conducted under contract F33615-02-C-3253

[*] MoBIES is an acronym for Model-Based Integration of Embedded Systems.

The second bullet is usually the most difficult one and the one addressed in this paper. For the hardware portion of a system the Failure Modes and Effects Analysis (FMEA) has long been the organizing principle for design reviews as well as for testing to show the absence of, or satisfactory response to, critical failure modes. Because of the importance of the FMEA for control of failure modes there have been attempts to generate a procedure for software FMEA for at least twenty years[2]. At the present time there are fundamentally two approaches for partitioning software for a system FMEA: functional[3] or by output variables, considering one variable at a time[4]. Functional partitions of a program are subjective and different analysts can come up with different lists of functions for a given program. In particular, exception handling may not be recognized as a distinct function in spite of its great potential for causing failures[5] Generating a software FMEA based on failures of a single output variable misses conditions in which a programming error affects multiple variables.

The advent of model-based system and software development, and particularly of the Unified Modeling Language (UML)[6], has motivated us to take a fresh look at overcoming the previous difficulties in generating software FMEA. The features of UML that are particularly important to this process are

- Formatted specifications with controlled relations to the code
- A rich assortment of automatically generated development artifacts
- Allowed actions of code based on the class specification (since UML uses an object-oriented paradigm)
- Ability to tag artifacts for assessment as part of the FMEA effort
-

The exploitation of these capabilities for computer-aided software FMEA are described in the following sections of this paper.

3. Overview of UML-Based Software FMEA

The central portion of Figure 1 is an FMEA worksheet in a format derived from MIL-STD-1629[7]. The purpose of a worksheet is to list and classify failure modes so that decision makers can concentrate on those with the highest importance. One of the items that make a failure mode important is the severity, shown in a column somewhat right of the center. Following the severity column are the detection method and the compensation, both of which have important bearing on the management of a failure mode as will be discussed in the last section of this paper. The Remarks column is also discussed later.

The four columns that we have just discussed can be considered the output of the FMEA worksheet. They are significant for the management of failure modes, whether they are acceptable or whether they need to be eliminated or mitigated. We will now discuss the columns to the left of these, that represent the input to the worksheet, information derived from the program file and from system data.

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 partitions. 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.

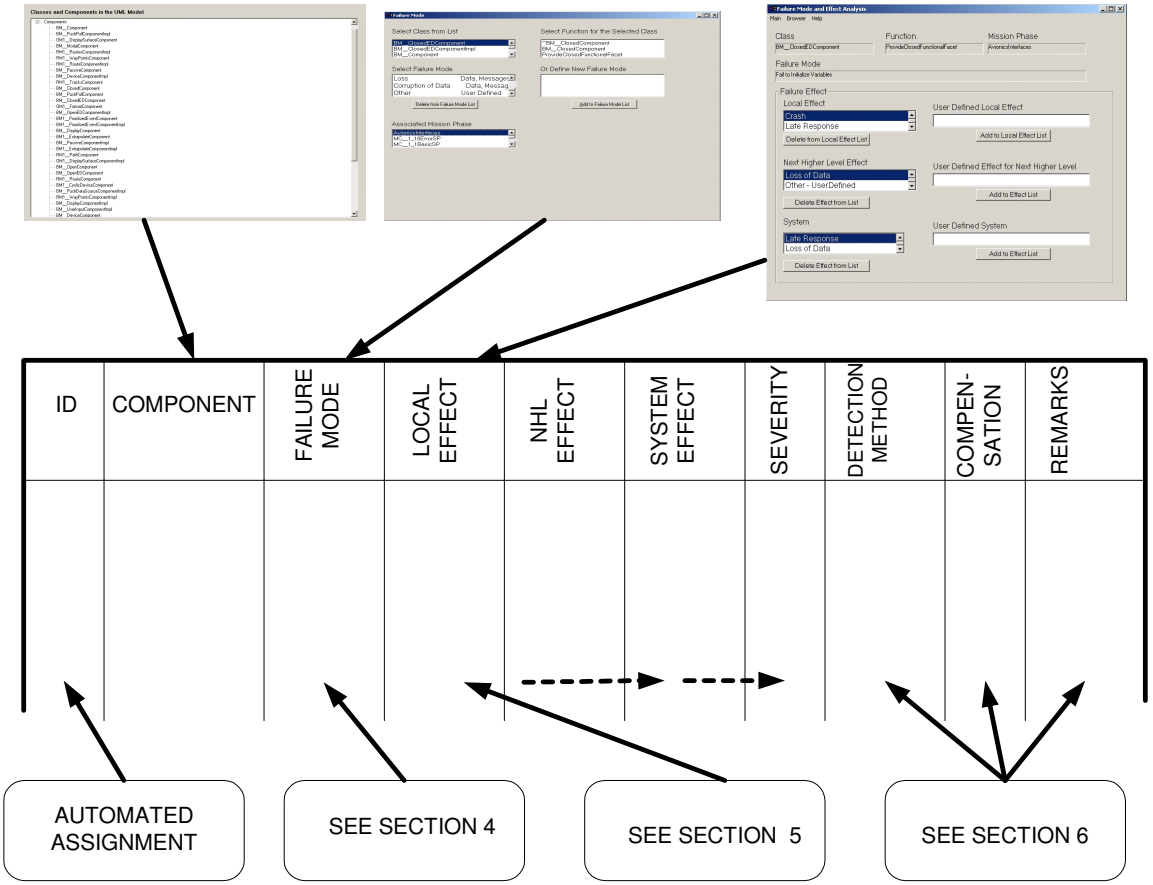

Figure 1. Overview of FMEA Generation

The component name is obtained from the UML listing 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. Failure modes are assigned in a computer-aided rather than fully automated manner but this can change (at least for 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 in the next section. The severity is directly associated with system level effects and can be assigned automatically in a given project context.

The entries in the detection method, compensation and Remarks columns are described in Section 6. These entries depend heavily on knowledge of the software and system design.

Figure 1 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 on-orbit 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.

4. Failure Mode Identification

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. We concluded that the distinctions could be collapsed without detracting from the usefulness of the FMEA for directing the V&V effort into the areas of greatest risk.

Responsible software developers recognize the possibility of these failure modes and protect against them by assertions and tests. It is thus only necessary to postulate typical failure modes to serve as initiators in checking for the presence of defensive programming constructs.

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 two basic failure modes are automatically entered for it. If the method provides output the *incorrect output* failure mode is added. The recognition of a method that provides output is at present up to the analyst but is capable of being automated. The invocation of the other failure modes is up to the analyst as shown in Figure 2.

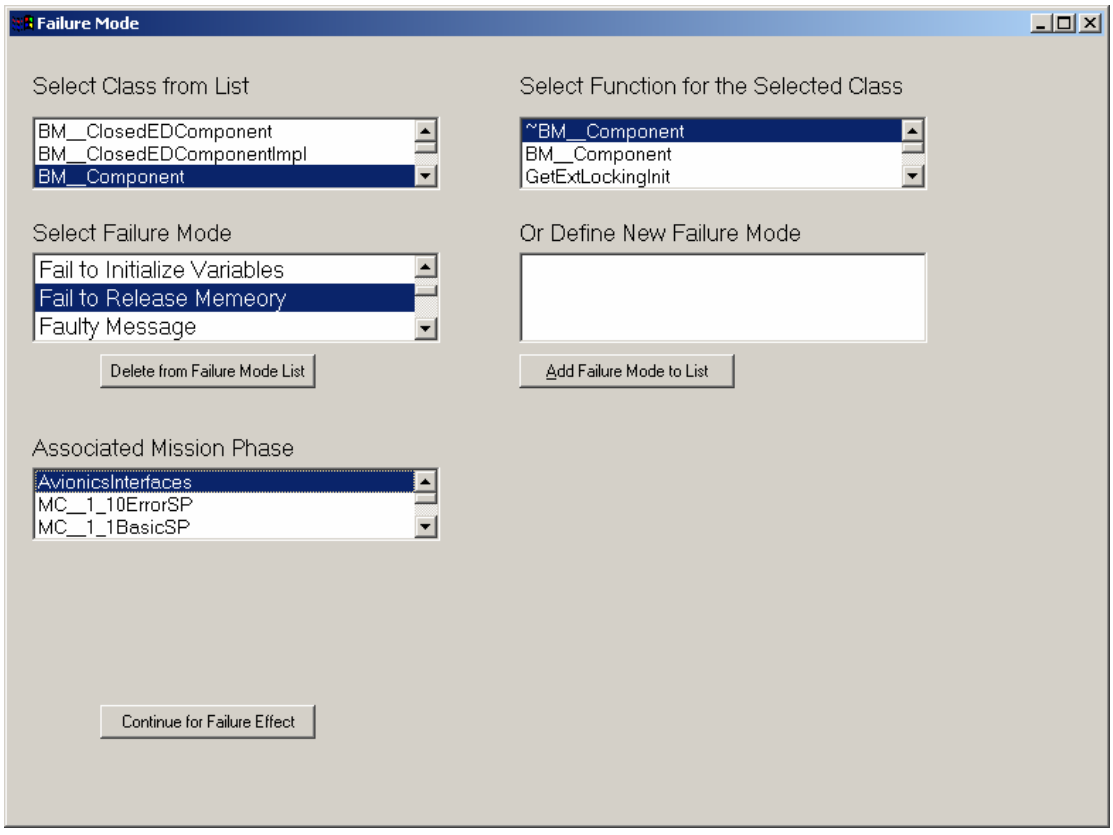

Figure 2 Failure Mode Screen

5. Local Effects and transition to NHL and System Level and Phase

The local effect of a function/method failure of a class is summarized in the following figure. Again, a user can add to this list if necessary.

Local Effect	
Local_Effect_ID	**Local_Effect_Desc**
1	Crash
2	Stop
3	Late Response
4	Loss of Data
5	Corruption of Data
0	Other - User Defined

Figure 3 Local Effect Selection

At each of the higher levels we need to distinguish between failures due to *native failure modes* and those caused by incorrect input. An important source of native failure modes are methods that are added to a parent class to create a child class. The effects of native failure modes are analyzed as described above for local failure modes.

A different method of analysis is applicable to failures due to incorrect input received at a higher level from a lower one or from any other source. The effect at the next higher level (NHL) is shown in Table 1.

Table 1. Effects due to Incorrect Input at NHL.

Detection and Compensation	NHL Effect
None	Crash
Detection only	Stop
Detection and re-try	Delayed output
Detection and default value	Degraded output
Can call alternate method	None

At the system level there may also be native failure modes for which failure effects are assigned as at the local level. Failure modes that originated at NHL are translated at the system level in accordance with Table 1. Failure modes that were detected at NHL cause system level failure effects as shown in Table 2.

Table 2. System Level Effects due to NHL Detected Failure Modes

System Level Protection	NHL crash	NHL stop	NHL delayed output	NHL degraded output
None	Crash	Crash	Crash	Degraded output
Detection only	Crash	Stop	Stop	Degraded output
Detection and re-try	Crash	Stop	Degraded output	Degraded output
Detection and default value	Degraded output	Degraded output	Degraded output	Degraded output
Can call alternate method	None	None	None	None

Perusal of Table 2 shows the distinguishing feature for reducing the severity of all software failures at the system level is the protection mechanism (detection and compensation). This suggests that effective V&V should focus on these provisions – their presence and quality.

6. Detection, Compensation and Remarks

Hardware failure modes, such as an open resistor or a shorted capacitor usually disable the affected function permanently. Software failures, particularly in well-tested programs, are frequently transitory. The program response to typical inputs and computer states had been found satisfactory, and thus the failure is likely to be due to unusual events. For this reason re-try of the affected program segment frequently permits resumption of normal execution.

To facilitate re-try the failure must be detected, and thus insertion of failure detection provisions is a standard practice in software developed by responsible organizations. The most effective detection methods are those inserted immediately after an error-prone programming step, such as accepting data, non-trivial mathematical or logical operations, and formatting output. In many cases the detection is in the form of an assertion "$x = A$" which, if not true, causes the program to enter an exception handling routine (the = sign denotes any logical operator and A any suitable expression). Other detection provisions, typically found in system software (schedulers, operating systems, middleware) protect against incorrect message passing, exceeding time limits, and anomalous event sequences.

Detection means close to the source of the failure mode are more effective in preventing higher level effects than those that are more remote because there is a lower probability of contamination of other processes and because they can invoke the most appropriate exception handling. Our approach is therefore particularly aimed at identification of assertions. The current mechanism is to let the programmer tag these, but more automated methods will also be explored. The detection tag permits automated entry into the detection column and also directs (together with the compensation provisions) assignment of NHL and system level effects in the above tables.

The software designer is responsible for providing appropriate compensating provisions, some of which have already been mentioned: roll-back and retry, program restart, use of default values, and alternate execution paths. At present these parts of the program must be manually tagged, and this will lead to automatic entry into the Compensation columns in the FMEA worksheet and direct the selection of the NHL and system level effects.

Entries into the Remarks column also depend on tags to be provided by the analyst for

- Conditions that lead to more severe failure effects in the presence of other anomalies (e. g. failures of monitoring software that normally causes no effect but very severe effects if the monitored function fails)
- Effects that can be overcome by automated or manual measures at the system level
- Failure modes for which the higher level effects and their severity must be assessed by probabilistic methods (by convention, the entries for these reflect the most severe possibility)

It will have become apparent that the generation of the FMEA, even with the computer support, requires much insight into the software and system design. It is a purpose and benefit of our approach that much less FMEA expertise is required, and that the analysis can therefore be generated by software designers and system engineers.

References

[1] Llis.nasa.gov

[2] Hall, F. M., Paul, R.A and Snow, W. E., "Hardware/Software FMECA", *Proc. of the 1983 Reliability and Maintainability Symposium,* Orlando, FL, January 1983, pp. 320-327

[3] Bowles, J. B. and Chi Wan, " Software Failure Modes and Effects Analysis for a Small Embedded Control System**"**, *Proc. of the 2001 Reliability and Maintainability Symposium,* Philadelphia PA, January 2001, pp. 1 – 6.

[4] Goddard, P. L. "Software FMEA Techniques", *Proc. of the 2000 Reliability and Maintainability Symposium,* Los Angeles CA, January 2000, pp. 118 – 122.

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

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

[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 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