COMPUTER-AIDED SOFTWARE FMEA[†]

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

1. Introduction

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)

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.

The contributions of FMEA to verification of critical systems include identification of

- End (system level) effects for each failure mode
- Severity classification of these effects
- Means of detection of the failure mode
- Compensation or mitigation provisions
- Effects of additional or associated failures (in the Remarks column)

The system level effects and the associated severity classification are used to prioritize the verification effort, including the testing. The detection means is a further clue to the potential threat posed by a failure mode. Local detection, such as an alarm when a power supply fails, provides the best protection because it indicates directly what has failed and permitting immediate repair and recovery actions. Detection by an

[†] 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.

intermediate effect, e. g., inability to access a disk, is less effective because this could be due to failure in the disk mechanism, in the control electronics, in the power supply, or in the cables connecting these units. Least effective is detection at the system level, such as no output or no response, because this could be caused by many failure modes. Thus failure modes that can only be detected at the system level warrant a high level of V&V. This is recognized in the risk priority number (RPN) of a recent SAE standard[1] that classifies risk as a product of severity, frequency and difficulty of detection. Compensation (e. g., redundancy) or mitigation (load shedding in a power system) reduce the severity of the end effects. However, the dependence of the reduction on the functioning of the compensation should be noted in the Remarks column (e. g., for a severity IV failure mode the Remarks may read "Severity II if redundant element fails").

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[2,3]. At the present time there are fundamentally two approaches for partitioning software for a system FMEA: functional[4] or by output variables, considering one variable at a time[5]. 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[6]. 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)[7], 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 implementation of these concepts for computer-aided generation of software FMEA is diagrammed in Figure 1 and details are discussed in later sections of this paper.The central portion of Figure 1 is an FMEA worksheet in a format derived from MIL-STD-1629[8]. 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 has been mentioned before.

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, i. e., to accept them or to decide that they need to be eliminated or mitigated. We will now turn to the columns on the left of the worksheet, data input that represents 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.

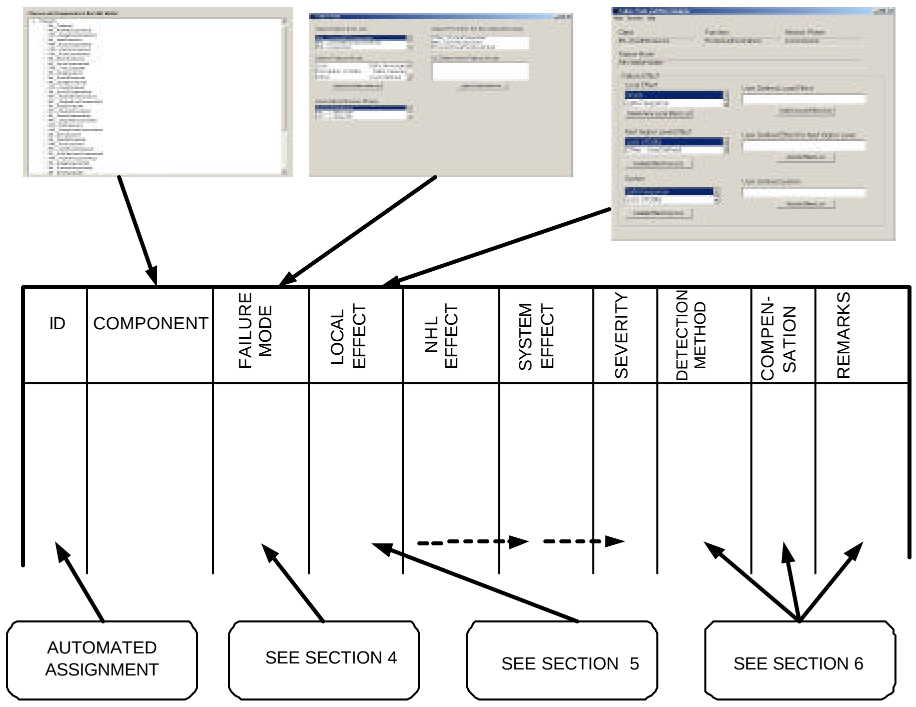

Figure 1. Generation of Software FMEA

The component name is obtained from the UML listing as shown at the top of the figure, and the ID will be automatically assigned based on the indenture levels of the listing. The ability to identify methods automatically from the UML tool listing and to enter them into the worksheet is a very significant advance over the previous "functional" decomposition. The software FMEA thus becomes comparable to a hardware FMEA that is generated from a bill of materials. It makes the FMEA and V&V activities based on it "complete".

Failure modes are assigned in a computer-aided mode but this operation can change to a more automated manner (at least for a given programing 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 an airborne missile that has test, carry and free-flight modes. Failure modes of the guidance software that have very severe effects in the free-flight 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[9] 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 important for teaching and for improvement of programming practices but they can be collapsed for purposes of the FMEA if they lead to identical effects. Responsible software developers recognize the possibility of these failure modes and protect against them by assertions and testing. 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. We refer to stop and crash as basic failure modes. 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.

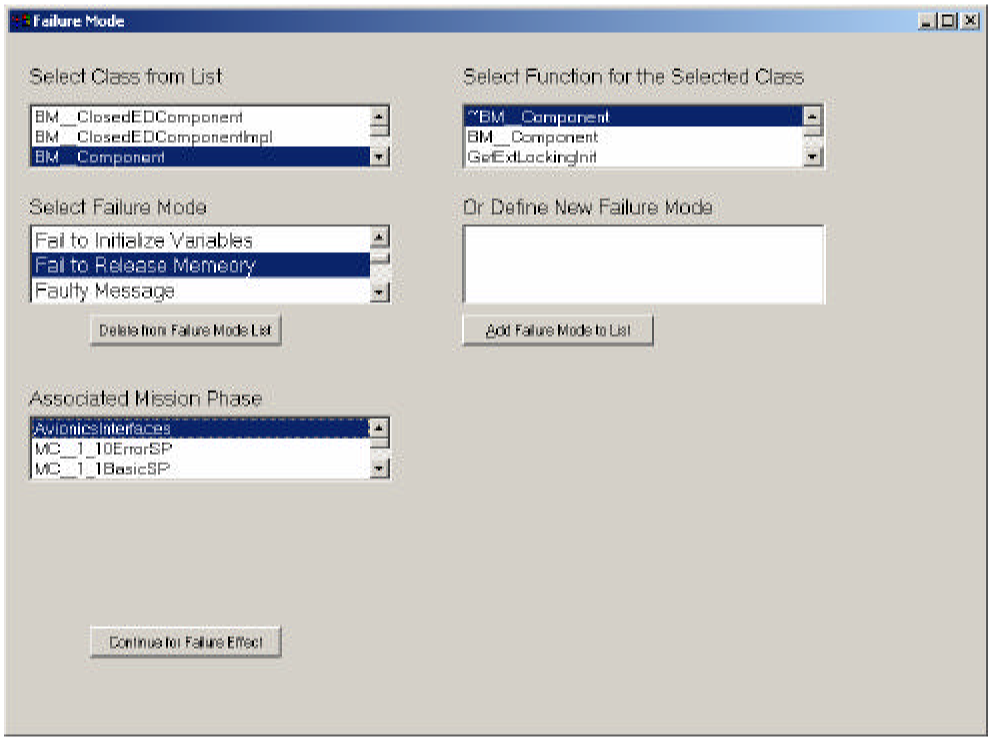

Figure 2 Failure Mode Screen

5. Local Failure Effects and Transition to NHL and System Level

The local effect of a function/method failure of a class is usually assessed where a communication with another function takes place. The local effect can be a pass-through of a failure mode, such as a corrupted message where no detection provisions were in place, or the failure mode may be modified by local detection or compensation provisions as shown in the remarks column of the following table.

Table 1. Local Failure Effects

Local Failure Effect	Remarks
None	Failure mode is insignificant or compensated for
Late response	Result of a successful retry
Corrupted message	Failure of error detection
Loss of data	Unsuccessful error correction
Stop	Locally detected crash of a method
Crash	

.

In addition to the listed failure effects the user can add any number of user-defined effects. 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 2. Again, the user can add effects if necessary.

Table 2. 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 2. Failure modes that were detected at NHL cause system level failure effects as shown in Table 3.

Table 3. 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

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 is tested many times during development and had been found satisfactory; 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 software designers and system engineers can therefore assume a more active role in FMEA generation.

References

1 SAE J 1739: Potential Failure Mode and Effects Analysis in Design (Design FMEA) and Potential Failure Mode and Effects Analysis in Manufacturing and Assembly Processes (Process FMEA) Reference Manual, SAE, Warrendale PA, June 2000

2 Reifer, Donald J.,"Software Failure Mode and Effects Analysis", *IEEE Transactions on Reliability*, vol.28, no.3, August 79

3 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

4 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.

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

6 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

7 Boggs, Wendy and Michael, *Mastering UML with Rational Rose*, Sybex, 2002

8 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)

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