A Process for Failure Modes and Effects Analysis of Computer Software

Nathaniel Ozarin • The Omnicon Group • New York
Michael Siracusa • Massachusetts Institute of Technology

Key Words: FMEA, Software FMEA, Software failure, Mission critical software, Software fault tree

Summary and Conclusions

Software FMEA is a means to determine whether any single failure in computer software can cause catastrophic system effects, and additionally identifies other possible consequences of unexpected software behavior. The procedure described here was developed and used to analyze mission- and safety-critical software systems. The procedure includes using a structured approach to understanding the subject software, developing rules and tools for doing the analysis as a group effort with minimal data entry and human error, and generating a final report. Software FMEA is a kind of implementation analysis that is an intrinsically tedious process but database tools make the process reasonably painless, highly accurate, and very thorough. The main focus here is on development and use of these database tools.

Introduction

Aerospace system development sometimes includes a failure modes and effects analysis (FMEA) on computer software. Software FMEA does not predict software reliability, but aims to determine whether the failure of any single software variable can cause specific catastrophic events or other serious effects. At the same time, the analysis can identify possibilities of less serious consequence so that source code can be made more robust in specific areas before deployment. This paper describes a step-by-step approach for conducting a software FMEA and outlines development of a relational database to aid the process.

There are many published papers on software reliability but their approaches generally focus on the use of historical data – software bugs discovered over time – to predict future failures. Very few papers have addressed approaches to software FMEA. Some present simple examples of approaches to illustrate principles but they do not address complications of real-world operational flight programs.

This paper describes techniques developed for conducting software FMEA on code executed by an embedded microprocessor as part of a missile control system. Using these techniques, the analysis was completed on time and revealed several potential problems that were subsequently fixed by the software developers. In the subject analysis, the authors analyzed 3294 lines of assembly language code native to a Texas Instruments DSP. The same techniques can be applied to any other mission-, safety-, or revenue-critical system using high-level languages in different architectures. The products of this effort included a table summarizing the analysis and a detailed table listing analysis details variable by variable.

In a software FMEA, a failure is a software variable that is assigned an unintended value. This kind of failure can occur when a memory location is unintentionally overwritten, when internal processor or memory circuits fail, or when bad data is received from the outside world. The analysis seeks to determine observable system effects – usually manifest via system hardware and therefore dependent upon hardware analysis – when any one software failure occurs, and in particular to determine whether any single software fault can result in a catastrophic event.

The software FMEA looks for consequences of *all* potential software failures. It is independent of two essential but different kinds of analysis: (1) how the software design meets requirements, and (2) the adequacy of the requirements themselves. Testing cannot reveal weaknesses in these areas, nor can line-by-line analysis of the code. It is therefore essential that system analysts and software engineers do their homework very carefully in these areas. A subsequent analysis considers the implementation – that is, the detailed code and the interfaces with memory-mapped hardware. This analysis, the FMEA process, covers only the implementation part. The FMEA also does not consider correctness of algorithms or problems resulting from real-time design errors, but makes the assumption that every variable might fail without regard to cause.

Causes of Software Variable Failures

Software FMEA considers the things that can go wrong as a processor executes its code. The following are two causes of software failures and how they are treated in the FMEA analysis. There are obviously many more.

(1) Failures in Memory. Data in memory can become incorrect due to environment (EMI or EMP), hardware failures (often detectable by test routines), and unintentional overwrites (by far the most common case). Unintentional overwrites, in turn, can result from (a) system design errors, (b) programming errors (such as writing or reading beyond the limits of an array or table, or incorrectly computing an address), and (c) hardware failures that alter contents of a memory address. The altered address, if subsequently used to index memory for a write operation, will cause a write to the wrong location and thereby corrupt the value at that location. At the same

0-7803-7717-6/03/$17.00 © 2003 IEEE

time, the software fails to update the contents of the intended location. Consequences of unintentional overwrites are unpredictable. In the case of writing to the wrong location, we know that the value of the intended location will not be revised as expected, but we cannot predict the consequence of writing to the wrong location – results can be totally benign (with extraordinary luck and extraordinarily robust code) but often cause the software to crash. For the system analyzed by the authors, results could be catastrophic. The analysis therefore assumes the worst-case outcome.

(2) Software-related Failures. Programming errors can give variables incorrect values that can be considered failures. The analysis does not specifically address these issues but instead considers the implications of incorrect values they may cause. Examples include incorrect algorithms, scaling errors, use of stale data, and overflow.

The Software FMEA Process

The remainder of this discussion outlines a process for conducting a FMEA on computer software. It is summarized in Table 1.

Table 1. Summary of the Software FMEA Process

Step	Subject	Description
1	System and Software Familiarization	Using tools and guidelines to understand the system under analysis.
2	Database Tool Development	Development of linked tables to maintain information and guide analysts.
3	Developing Rules and Assumptions	Applying knowledge and experience to lay out clear rules for analysis.
4	Developing Descriptive Failure Modes	Defining the ways that variable values can be failures.
5	Determining System Effects of Individual Failures	Examining variables one by one in every usage while using data in previously developed tables to aid the analysis.
6	Generating the Report	Using the database tool to automate report generation.

Step 1: Familiarization

Understanding the software's operation, its safety requirements, and its relationship with hardware is the most important step in the FMEA process. Since FMEA should not be performed by people who developed the code, analysts are confronted with the difficult task of analyzing software they've never seen before. Typically, the software will have insufficient comments and misleading variable names. Worse, real-world design documents almost never describe source code with complete accuracy. The comments and design data can serve as important references for understanding, but they cannot be relied upon because the task is to analyze operations of the code, not the intent of its authors. Therefore, we use the executable code to develop flow charts, verbal descriptions, and other methods for this purpose, with a description prepared for each software method or function.

Timing and calling sequences among software functions must also be understood, but tying the functions together is more difficult. In the subject analysis, the authors developed calling diagrams to show relationships among all functions – who calls whom, and in what sequence. A calling diagram may be divided into several pages, with related functions on one page. The calling diagram provides an additional advantage: it helps the project manager divide the software under analysis into logical sections of related functions that can be assigned to individual staff members for greater efficiency. Each staff member then concentrates on assigned portions of the software and is best suited to fill in the associated database tables.

Other diagrams also aid human understanding. For example, dataflow diagrams are particularly useful for analyzing code designed for extensive data processing. Traditional diagrams such as class diagrams are the least useful to analysts because they represent interactions at an overly high level of abstraction.

Step 2: Database Tool Development

Database tools are the main focus of this discussion. The database tables help analysts understand the software, organize the FMEA process, and aid in its automation. The authors used Microsoft Access because it was available, but also because it allows data tables to be shared by multiple users and updated in real time – an essential feature for a group effort. The following discussion focuses on table development in a logical sequence that would be used in any software FMEA process.

The first table to be filled out defines the software modules (such as classes) which represent a collection of functions. Figure 1 is part of a typical module definition table, shown with sample data.

module_def : Table

	module	description	id
+	atod	Drivers to control Analog to Digital Converter.	1
+	humid	Contains algorithms for humidity control.	2
+	main	Controls main flow control of program.	3
+	temp	Contains algorithms for temperature control.	4
+	uart	Drivers to control UART.	5

Figure 1. Module Definition Table Example

The next table is a subroutine definition table that defines each function and what it is used for. Figure 2 is an example. Note that determining preconditions and post-conditions in this table as an aid to familiarization may not be cost-effective for a particular analysis.

subroutine_def : Table

	subroutine	description	module	scope	argumer	returns	calls	called_by	precon	postconditi
+	ADCready	Checks if ADC is read to start next conversion or get data.	atod	Local	None.	ZF = 0 if ready.	Nothing.	getTemp	None.	Returns whether or not ADC is ready.
+	controlTemp	Runs temperature Control algorithm	temp	Global	r0 = disired temp	Nothing.	startFan, heatOn, convertTemp	mainloop	None.	Control temperature to desired value by turning on fans or heaters.

Figure 2. Subroutine Definition Table Example

The next table is a variable definition table. Every variable and data structure is documented here for the FMEA analysis. It may also be helpful to create your own variables to reference memory that is indirectly used or to combine variables that are treated similarly or as a group. Figure 3 is an example of this table, in which the "hardware" field is used if the variable is directly associated with any hardware in the system. All memory-mapped variables will have this field filled out. It is a good idea to make a separate table to record all hardware-software interfaces and link them to this field.

Entering data in this table by hand is a tedious and error-prone task, and it is advantageous to write a small program that strips variable declarations and associated comments from the source code. One of the authors wrote a PERL script to do this. It is easily modified to retrieve similar information from different varieties of source code.

A variable usage table is also essential for the analysis. This table lists all functions that use (i.e., read) each variable and all functions that modify (or set) it. The variable usage table helps the analyst understand the code's operations and provides a means to track implications of failures. Figure 4 on the next page is a sample table. Here, "line" is the source code line number at which the variable is being used ("U") or set ("S"). The "description" field notes the meaning of the particular use or set.

It is important for information in individual fields to be consistent among tables so that fields in various tables can be linked to create detailed reports. To maintain database relationships among tables – and minimize typing – the database provides tables with pull-down menus that list names or phrases in other tables. In this way, analysts simply click on the desired selection to enter it. There is no need to type a piece of information more than once.

Step 3: Developing Rules and Assumptions

In Step 1, analysts develop a good understanding of the code to be analyzed. In Step 2, they develop a database tool suitable for the FMEA requirements and enter descriptive information in the database. In Step 3, analysts develop rules for the analysis and then apply them.

The FMEA is based on stated assumptions that serve as analysis rules. Rules for analyzing assembly language will be different from rules for analyzing C code, and each set of rules for a particular analysis will evolve as the staff does its work and faces new situations. It is naturally important to maintain the set of rules and be sure that each analyst follows them. Some rules developed by the authors are listed below as examples.

1. *Input variables*. The FMEA table lists only variables that are used as inputs. If an input variable appears more than once in a subroutine, then it may appear more than once in the table, but it can be listed just once if it serves the same purpose at each appearance.
2. *Outputs to hardware.* Some variables are outputs only, such as those used to write to memory-mapped hardware peripherals. These variables do not fail in a software analysis because the locations they represent are non-memory hardware. However, the input variables that affect these output variables are subject to failure in the analysis process.
3. *Output variables.* Every output variable (other than memory-mapped outputs to hardware) is an input variable

variable_def : Table

	variable	hardwa	type	scope	modul	function
+	#Humidities	None	alias	Global	main	inside_humid, outside_humid
+	#Temperatures	None	int	Global	main	inside_temp, outside_temp
+	adc_channel_table_adr	None	int	Local	atod	Address of ADC channel conversion table
+	adc_control_reg	ADC	int	Local	atod	Memory mapped ADC control Register.
+	adc_data_reg	ADC	int	Local	atod	Memory-mapped ADC data Register.
+	desired_humid	None	int	Global	main	Desired humidity.
+	desired_temp	None	int	Local	main	Desired temperature.
+	inside_humid	None	int	Local	main	Last inside humditiy reading.
+	inside_temp	None	int	Local	main	Last inside temperature reading.
+	outside_humid	None	int	Local	main	Last outside humidity reading.
+	outside_temp	None	int	Local	main	Last outside temperature reading.
+	uart_control_reg	None	int	Local	uart	Memory-mapped UART control Register.
+	uart_rx_fifo	UART	int	Local	uart	Memory-mapped UART Receive FIFO.
+	uart_tx_fifo	UART	int	Local	uart	Memory-mapped UART Transmit FIFO.

Figure 3. Variable Definition Table Example

variable_usage : Table

variable	u_s	subroutine	line	description
inside_temp	S	mainloop	124	Set to inside temperature obtained from calling getTemp(0).
outside_temp	S	mainloop	136	Set to outside temperature obtained from calling getTemp(1).
inside_temp	U	controlTemp	101	Used to determine whether or not to turn on the fans. Compared to outside_temp.
outside_temp	U	controlTemp	102	Used to determine whether or not to turn on the fans. Compared to

Figure 4. Variable Usage Table Example

to at least one piece of software. Therefore variables are only considered as inputs for failure analysis, and the affected output variables are determined.

4. *Numeric values.* Numeric values or literal values assigned by the assembler do not involve memory and therefore do not fail. However, the variables that use them are subject to failure.

5. *Variables affecting decision logic.* A variable with an incorrect value that is used in decision logic may cause unintended execution of code that is supposed to change values of other variables (directly or through subroutine calls), or it may cause the program flow to incorrectly skip over a section of code that changes values of other variables (again, directly or through subroutine calls). All variables affected in this way are considered in the FMEA.

Step 4: Developing Descriptive Failure Modes

After a preliminary set of rules is developed, failure modes are defined in a failure mode table. A failure mode specifies how a variable can fail in a way that affects the routines that use it. A variable fails when it is assigned an unintended value, and the most common failure mode for a variable is "Incorrect Value." This label indicates the variable has a value other than what is expected or intended. For example, system problems may occur when a flag is set to the opposite value of the one intended, or when a variable representing an analog value is set to any value other than the correct one – regardless of whether the unintended value is 'too low' or 'too high.' However, this label is often too vague. In some situations, only specific values or values within specific ranges will cause a problem. For such situations, the table should show the range of values that can cause a problem. Typical failure modes are True, False, Set, Clear, High Value, Low Value, Incorrect Address, Incorrect Value, Value equal to, Value greater than, Value greater than or equal to, Value in range, Value less than, Value less than or equal to, Value not equal to, Value not in range, Bit Stuck High, and Bit Stuck Low. Note that some failure modes such as "Value equal to" need an actual value. This special case is listed in a separate field elsewhere (in the FMEA table). In all situations, an unintended value can (1) cause an unintended effect, (2) can prevent an intended effect, or (3) both.

Another table, shown in Figure 5, lists failure mode causes to represent hypotheses of what causes a variable to fail. Again, the list of causes will differ among analyses. However, the table will prevent different people from devising new ways to describe the same thing.

A system effects table also prevents different people from devising new ways to describe the same thing. This table is set up here in Step 4 for use in the Step 5. Typical system effects might include Unpredictable, Loss of data error logging, Uncommanded motion, and Improper steering control. An "Unpredictable" system effect is one whose consequences cannot be determined. For example, an incorrect pointer may cause a memory overwrite and corrupt a random set of variables. "Unpredictable" means an unknown system failure is certain (or is expected) to occur. "Unpredictable" does not include the possibility that system behavior might be normal. In addition, "Unpredictable" does not include the possibility of a critical hazard.

failure_mode_cause_def : Table

cause	description
ADC Failure	Analog to Digital converter failed
Hardware / Software Failure	Failure Due to Hardware or Software
Hardware Failure	Failure Due to Hardware
Software Failure	Memory was overwritten
UART Failure	UART not operating properly

Figure 5. Failure Mode Cause Definition Table Example

Figure 6 shows a sample system effects table. On any row, the first column identifies the failure effect. Analysts fill in this column as the work proceeds into and during Step 5. The database tool fills in the other columns of Table 6 automatically after completion of the FMEA table (the work of Step 5), but a preview here is worthwhile. The second column states the number of software failures whose likelihood of causing the effect is "possible" (depending on unpredictable circumstances), and the third column identities these exact failures by FMEA table identification numbers. The forth and fifth columns list similar information for software failures whose likelihood of causing the effect is "definite" (predictable under all circumstances).

system_effect_def : Table

effect	poss	possible_fm	defin	definite_fmea_ids
Improper Temperature Control	3	3.6, 3.70, 1.32	10	1.4,1.6,1.7,1.8,2.1,2.3,4 ,4.6,4.7,6.1
Uncommanded Temperature Control	5	1.12,1.13,1.14 ,2.16, 2.13	2	1.8,1.10
Unpredicatable	0		5	4.2, 4.8, 5.6, 5.8, 5.9

Figure 6. System Effects Table

Step 5: Determining System Effects of Individual Failures

The tables described so far aid in the analysis but (except for the database-generated parts of Figure 6) don't represent results. The FMEA table, described now, represents the heart of the analysis because it documents all single point (software variable) failures in the system. The tabular worksheet form is familiar to reliability engineers and is thus the best way to represent the analysis. The FMEA table is partially automated by the database tool in the sense that it provides information created in other tables during the analysis. Analysts typically click on a blank cell and the database tool provides pull-down list with all recognized possibilities. There is no need to retype information already in another table. However, development of tables is an iterative process. For example, if none of the existing failure modes appearing in a pull-down list fits the failure under consideration, you must add it to the failure mode table so that it becomes available to all other analysts working on the FMEA table.

Table 2 lists information that typically appears across the page for one input variable in a software FMEA table. This information is summarized below vertically due to space limitations. Each piece of information in the "Item" column is typically a separate column in the deliverable FMEA report.

In Table 2, the Input Variable field contains the variable being analyzed. Every time a variable is used in the system there should be a separate entry of that variable in the FMEA table. However, it sometimes may be sufficient to have a single entry for multiple uses of a variable if the system and local effects are identical. Some variables have different purposes in different modules and will result in different effects locally in the code as well as in the system. Certain variables may also have more than one failure mode and thus may have multiple entries for a single use in the code. For example, a variable used as a condition in a series of branches may have a different effect depending on what its unintended value has become when the variable was corrupted.

Software failure probabilities cannot be assigned to a particular failure as in a hardware FMEA, but a workable assessment of a system failure possibility is whether (1) a failed variable *might* cause a particular effect, depending on unpredictable but valid states of the system and other variables, or (2) *will* cause the effect regardless of other circumstances. The authors used the terms Possible and Definite to represent these possibilities in the FMEA table. A related rule of analysis was that a failed variable whose system effects could be corrected in subsequent iterations is described as "Definite" because (based on yet another rule) the analysis precludes self-healing of failures.

The Affected Variable field in Table 2 is used to identify variables that are directly affected by a failed input variable. Since the failed variable may affect countless other variables as execution proceeds, where does the list of affected variables stop? The authors developed a rule of analysis to draw the line. A "directly affected" variable is a downstream (subsequent in the program execution) variable whose value does not depend on another downstream variable. For example, if the input variable fails in such a way that it causes a branch to take place when it should not, it may skip calling a subroutine or setting a variable. The variables normally modified in the skipped subroutine as well as the skipped variable should all show up in the Affected Variable fields. However, only variables that are directly affected should be added to these fields.

Table 2. Elements of the FMEA Table

Item	Description	How Entered	Report
ID	Unique identifier for a particular type of failure caused by a each variable.	Automatically	Yes
Input Variable	The variable under analysis.	Pull-down list	Yes
Failure Mode	Examples in Step 4.	Pull-down list	Yes
Fail Range Low	Lower acceptable limit, if applicable.	Analyst	Yes
Fail Range High	Upper acceptable limit, if applicable.	Analyst	Yes
Fail Mode Cause	Examples in Figure 5.	Pull-down list	Yes
Local Effect	Text for analysts' benefit.	Analyst	No
System Effect (several)	Effect of failure on system. Will generally be one of several, with each identified as "possible" or "definite."	Pull-down list	Yes
Notes	Suggestions or notes.	Analyst	Yes
Subroutine	Where this variable fails.	Pull-down list	Yes
Line Number	Location in source code listing.	Analyst	Yes
Affected Variables	All directly affected variables.	Pull-down list	No
Analysts Notes	Whatever is helpful for analysis.	Analyst	No

To illustrate this rule, consider the code fragment below. If effects of input variable A are under consideration, then the directly affected variables are B, C, J, E and G. Variables A and F are not directly affected by A because their assigned values depend on directly affected variables B and C. When effects of B and C are analyzed, then A and F will be considered directly affected.

```
void sub1(void){
    A = B + C
    D = B * C

    if(A < 0){
        B = 0
        C = 0
        Call sub2()
    }
    else {
       E = (C * D) + F
    }
}

void sub2(void){
   J = K - B
   if(B == 0 || C == 0){
       A = 0
       F = 2
   }
   G = J * F
}
```

The Affected Variable fields do not show up in the final report but they are essential for tracing single variable failure effects through the entire system. Additional Affected Variable fields may be added as needed.

Step 6: Generating the Report

An analysis report is generally prepared to summarize key FMEA findings, explain ground rules, and elaborate on the system under analysis. Fault trees may also be included. Software fault trees are similar to those prepared for hardware but can be far more complex because the effects of a failed variable generally depend on states of other variables and on related hardware. Finally, the report may include recommendations for improving software reliability for the specific system under analysis.

The great advantage of the database tool is that it can create any number of tables for the report in virtually any format. The tool can also combine information from tables to provide an organized summary of the analysis.

References

1. Goddard, Peter L., "Software FMEA Techniques," *Proceedings of the Annual Reliability and Maintainability Symposium*, January 2000.

2. Bowles, John B., "Software Failure Modes and Effects Analysis For a Small Embedded Control System," *Proceedings of the Annual Reliability and Maintainability Symposium*, January 2000.

3. Lutz, Robin R., "Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems," *Proceedings of the IEEE International Symposium on Requirements Engineering,* January 1993.

Biographies

Nathaniel W. Ozarin
The Omnicon Group Inc.
40 Arkay Drive
Hauppauge, NY 11788 USA
nozarin@omnicongroup.com

Nat Ozarin is a senior engineering consultant at The Omnicon Group Inc., a company specializing in reliability and safety analysis for the military, medical, industrial, and transportation industries. His background includes hardware engineering, software engineering, systems engineering, programming, and reliability engineering. He received a BSEE from Lehigh University, an MSEE from Polytechnic University of New York, and an MBA from Long Island University. He is an IEEE member.

Michael Siracusa
The Omnicon Group Inc.
40 Arkay Drive
Hauppauge, NY 11788 USA
msiracusa@omnicongroup.com

Michael Siracusa is an associate consulting engineer at The Omnicon Group Inc., where he has specialized in database development. He received his BSCE from Stony Brook University and divides his time between Omnicon and MIT, where he is enrolled in an MS/PhD program.