Applying Software Failure Modes and Effects Analysis to Interfaces

Nathaniel W. Ozarin, The Omnicon Group Inc.

Key Words: Interfaces, FMEA, Software FMEA

SUMMARY & CONCLUSIONS

Software failure modes and effects analysis (SFMEA) is sometimes applied to new mission-critical and safety-critical system development. This kind of analysis, like its older cousin Hardware FMEA (HFMEA), tries to determine all possible types of failure for each component, one by one, and attempts to predict system-level effects for each failure. While software variables and classes don't fail in the sense that hardware fails, variables do sometimes assume unexpected values [1] and class methods don't always perform as expected.

Every software system includes software and hardware interfaces. In large system developments, different groups of people usually develop different parts of the software, with developers depending on interface requirement specifications (IRSs) to guide their design. Unfortunately, IRSs are sometimes like brick walls that separate developers. Since software designers are naturally most concerned with their own designs rather than those on the other side of the wall, there is tremendous potential for unidentified failures across the interface.

Applying an SFMEA to both sides of the wall is an essential first step for understanding consequences of failures, but an analysis that fails to rigorously analyze the interface can lead to conclusions that are both incomplete and incorrect. Fortunately, SFMEA techniques can be expanded to include considerations that apply to any kind of interface. The idea is to apply a step-by-step analysis sequence to determine what could go wrong at an interface and the subsequent effects on the system software. SFMEA that includes thorough interface analyses provides a more complete picture of system robustness.

1 INTRODUCTION

SFMEA, like HFMEA, seeks to determine system-level effects when any single component fails in certain ways. In an SFMEA, the lowest level component is a variable, and a higher level component is typically a module or a class. The relationship between these two software items might correspond roughly to that between a resistor and a circuit board in the hardware world. We don't normally think of software as something that suddenly and unexpected fails after extensive testing and field trials in the sense that hardware component characteristics change over time or fail completely. Yet, software variables can and often do assume unexpected values whose system effects, like those in hardware, can range from benign to catastrophic.

SFMEA, unlike HFMEA, is not a reliability analysis and does not include failure rates because software components simply don't have them. Rather, an SFMEA looks for possibilities, not probabilities. The possibilities are based on the assumption that variables – considered one by one, like components in HFMEA – can assume any value at any time.

Interfaces between software modules present a new set of potential problems, such as a failure due to unexpected time delays among sets of incoming data, or a failure due to a bent connector pin jammed between two other pins that damages data flow of three signals simultaneously. However, understanding interface problems can also result in some very important and meaningful conclusions for improving system robustness. For example, two related interface signals might both be updated at 1-second intervals, but with intervals set by asynchronous clocks. An interface analysis may conclude that the resulting race condition could cause an occasional loss of data. With this conclusion, it is possible to change the design to eliminate the race condition. Traditional SFMEA, on the other hand, would simply hypothesize that the effected data has assumed an unexpected value of any magnitude at any time, which is an essential, but far less useful conclusion. Since both conclusions are needed, interface analysis becomes an important supplement to traditional SFMEA.

Moreover, traditional analysis does not require identification of a failure cause because there are so many ways a variable can assume an unexpected value. (Traditional SFMEA sometimes can list a failure cause – for example, "bad A/D converter," but relatively few software failures can be attributed to specific causes.) Interface analysis, on the other hand, always identifies the failure cause.

2 A CLOSER LOOK AT SOFTWARE INTERFACE RISKS

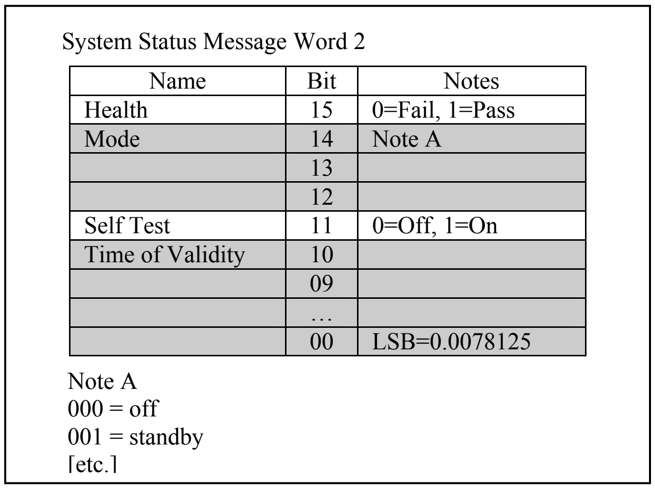
There are many kinds of software interfaces, some of which involve hardware. A complex nonstandard interface might require several pages in an IRS, and a relatively simple interface such as a software queue that passes data between software modules may be defined within the executable code itself (for example, where the code packs data to be sent into a class known by both sides of the queue). Yet virtually all interfaces involve development risks because (1) they are defined and implemented by humans [2], and (2) the development and final operating environments aren't the same. Even where common, off-the-shelf interface hardware and associated software drivers perform operations based on international standards, system software problems can still occur because higher-level considerations effect operations. A simple example is a bus controller that seems to work perfectly when parts of system are tested, but very occasionally runs out of real time when it must handle all of the system's bus traffic, resulting in lost data at rare and unpredictable times. A less common but real example is a commercial MIL-STD-1553 driver that occasionally caused a received data message to appear twice due to a bug in the manufacturer's software. If you are fortunate, these problems become known during integration and test, but it is well known that some problems don't show up until the system has been deployed, sometimes with catastrophic consequences.

Causes of software interface problems can be split into two categories: (1) defective interface definitions, and (2) unexpected operational performance. The distinction is important because each has its own set of potential effects, and a SMFEA that includes a thorough investigation of interfaces must look for different things in each. For example, an interface specification that fails to define a valid range for a data item can result in one kind of system failure, whereas delays in data reception may cause a different kind of failure.

Not surprisingly, human experience with interface problems is essential for performing an interface analysis. Your experience – including what you've read and heard about – tells you what to look for. The following sections present examples, based on the author's experience, of what to look for in the two categories of causes of software interface problems.

3 DEFECTIVE INTERFACE DEFINITIONS

There are many kinds of potential problems with interface definitions. Figure 1 is an example of a software interface design document for one message word at the bit level.



System Status Message Word 2

Name	Bit	Notes
Health	15	0=Fail, 1=Pass
Mode	14	Note A
	13	
	12	
Self Test	11	0=Off, 1=On
Time of Validity	10	
	09	
	…	
	00	LSB=0.0078125

Note A
000 = off
001 = standby
[etc.]

Figure 1 – Example of Software IDD Specification

First, measurement units or measurement zero points may be confusing or ambiguous. Some examples: (1) Aircraft altitude is often measured and conveyed in units of 100 feet, but the unit of measurement is still feet. (2) Rotation may be specified in radians, but rotation may range from $-\pi$ to $+\pi$ rather than from 0 to 2π. (3) Units of measurement may be assumed, with one team assuming meters and another assuming centimeters, or feet. (4) Different schemes may be used to represent negative quantities, such as 2s complement, a sign bit with a positive mantissa, or a fixed offset (as "excess 128"). (5) Time can be measured from different starting points – time 0 could be last midnight (typical in navigation equipment), the start of the current year (typical in financial projects), or some other event. A related risk: If a time measurement is used in more than one subproject, are requirements for simultaneity specified? (6) Parameter values typically change over time. There will be a problem if one team assumes that a received parameter value is updated twice a second, but the team supplying the parameter updates it just once a second.

Second, interface specifications may contain potentially confusing verbal descriptions. For example, in a system where parameters are to be echoed back after reception to confirm system integrity, the interface document specifies that a parameter at the interface "shall be ignored" in certain conditions. One team may reasonably assume that "ignored" means don't look at it and don't return it, but another team may reasonably assume that "ignore" means don't act upon it but echo it back anyway.

Third, separate design teams sometimes assume that the other teams will provide established "good practices" in their designs such as checking invalid parameter values that are passed across interfaces. This is sometimes a source of major rework at integration time. For example, one team may assume that all parameters it receives are checked for validity by the software that provides them. If that assumption is even partially incorrect, the result is an unreliable design that could be catastrophic some day despite extensive system tests. The opposite sometimes happens: both teams provide data checking, a redundant and usually wasteful effort that can add unnecessary complexity to a development. A related item is the definition of "invalid" for a particular parameter. Is a parameter with continuous values invalid if its value is outside a specified range, or is it also invalid if it shows an unrealistic rate of change within its specified range? Suppose the parameter is time. Is its value invalid if it fails to increment, if it jumps, if it overflows, or if it goes backwards?

There are many other sources of risk in interface definitions. Examples: (1) Bit order (is bit 00 the MSB or the LSB?). (2) Bit packing (bit endian vs. little endian machine assignments. (3) Use of different enumerations with the same name by different teams that exchange data – for example, in an enum called Color, red might have a value of 3 in one team's code but 5 in another team's code. Or, one team's enum might have included yellow, but the other team's enum doesn't. (4) Use of parameter names with apparently unambiguous meanings that two teams confidently interpret differently. (5) Contradictions in a document or contradictions across two or more documents. A requirement should appear exactly once and it should be referenced from all other places, but that is seldom followed in practice. It is common for two development teams that exchange data to have two different interpretations of a requirement. The interface SFMEA can help identify these errors.

4 UNEXPECTED OPERATIONAL PERFORMANCE

There are a number of typical problems with interface operations that sometimes don't become obvious until late in the development. This is particularly true for large or complex systems where many different data exchanges occur and data in some messages affects processing of data in other messages.

- Data sometimes arrives late due to asynchronism between sides of an interface.
- Data sometimes stops arriving due to withholding on the sending side, an overflowing queue, physical damage, or other reasons.
- Data is sometimes out of range or changes more quickly than expected due to problems with sensors or inability to process data as quickly as it arrives.
- Data is sometimes repeated or lost due to race conditions.
- Data is accepted and processed when it should be ignored due to system circumstances.
- Data values are occasionally out of range when such behavior should be expected – due to noise or sensor behavior – but the receiving software rejects *all* data as bad rather than just the occasional bad values.
- Groups of data arrive out of sequence due to processor overloading or prioritization of data handling.
- Incorrect message lengths, sometimes due to incorrect message construction.
- Failure to correctly return data as status to sender, sometimes due to different formats between received data and the returned status data.
- Failure to take advantage of redundant data for integrity checking, or misapplying such checks that sometimes lead to false error detection.
- Interface hardware failures of innumerable sorts, ranging from failed circuit boards (which may have their own software problems) to broken or shorted wires. If an entire connector has been inadvertently left unmated, or an entire circuit board died or was missing, what is the effect on system software?

5 WHAT TO LOOK FOR AND IMPLICATIONS

When examining interface-related software as part of an SFMEA, understanding the kinds of problems in sections 4 and 5 is a good guide to what to look for. For example, if you know that out-of-sequence data sometimes happens in the real world and you can establish that this kind of failure (1) is possible and (2) has consequences in the system under consideration, then you might look for sequence numbers appended to the data and interface software that checks them. Remember that an FMEA looks for only one failure at a time, so if a failure occurs that could result in out-of-sequence data, then the checking software is assumed to be operating without failure correctly. What the checking software does with this information is another matter – you obviously must examine how the checking software works and subsequent system behavior when it detects the failure.

Software developers apply many standard techniques for detection of data failures, many of which can be applied to interfaces [3]. These are beyond the scope of this paper but should be understood by SFMEA analysts when examining interfaces.

What makes interface an SFMEA different from a traditional SFMEA – and one of its key values – is the fact that a "single point failure" in an interface can result in multiple failures at the software variable level. For example, a dropped message at an interface may result in loss of a lot of data that in turn affects lot of variables concurrently, whereas a traditional SFMEA may consider just one variable "failure" at a time.

6 UNDERSTANDING INTERFACE DESIGNS

Analysts faced with SFMEA across an interface between two or more pieces of software must understand (1) the nature or structure of the interface and (2) what the interface software does to prevent unexpected events or unexpected data values from causing an unacceptable system failure. The following paragraphs describe some common interfaces (see Figure 2).

a. *Serial bus message.* This interface is usually applicable to physically remote parts of the system. The interface passes serial data messages whose format and protocol are generally defined by a standard, such as Ethernet or MIL-STD-1553. The standards define criteria for correct data exchanges and handling of "bad" data. Such standard interfaces offer established means that help assure data integrity. Data is generally transmitted via commercial hardware.

b. *Shared memory.* This interface is applicable to isolated software functions on circuit boards that physically reside on the same hardware bus. High-reliability systems typically assign different blocks of memory to different software functions. The operating system normally checks for sharing violations automatically. However, system designers can designate a block of memory that is available to two or more software modules for the purpose of fast data exchange. This technique requires careful system design.

c. *Queue message.* This interface is applicable to systems like those that can use shared memory. The technique involves slower data exchanges but usually with less risk because the protocol is well defined, the operating system moves the data for you, and it typically reports delivery errors (which can be ignored at one's risk). One piece of software can send one or more blocks of data in rapid succession to another without concern about whether the recipients are busy. The sender typically specifies the data start location, the size of the data block, and a queue number. A recipient who has been designated to receive data on that queue number receives the data block when it has time to check whether any incoming data is available.

d. *API (Application Programmer Interface).* This interface is applicable to independent pieces of software executed by the same processor. Here, one developer produces a module or class that performs specific functions or services for client software typically developed by another person who may be located in a different part of the world. The client calls the API like a function call, usually passing some parameters as input data. Output data is sometimes returned in the opposite direction. This kind of interface is usually simple to set up and use, but there can be risk if the client doesn't fully understand the nature of the inputs and outputs, the limitations of the called service, and its timing.

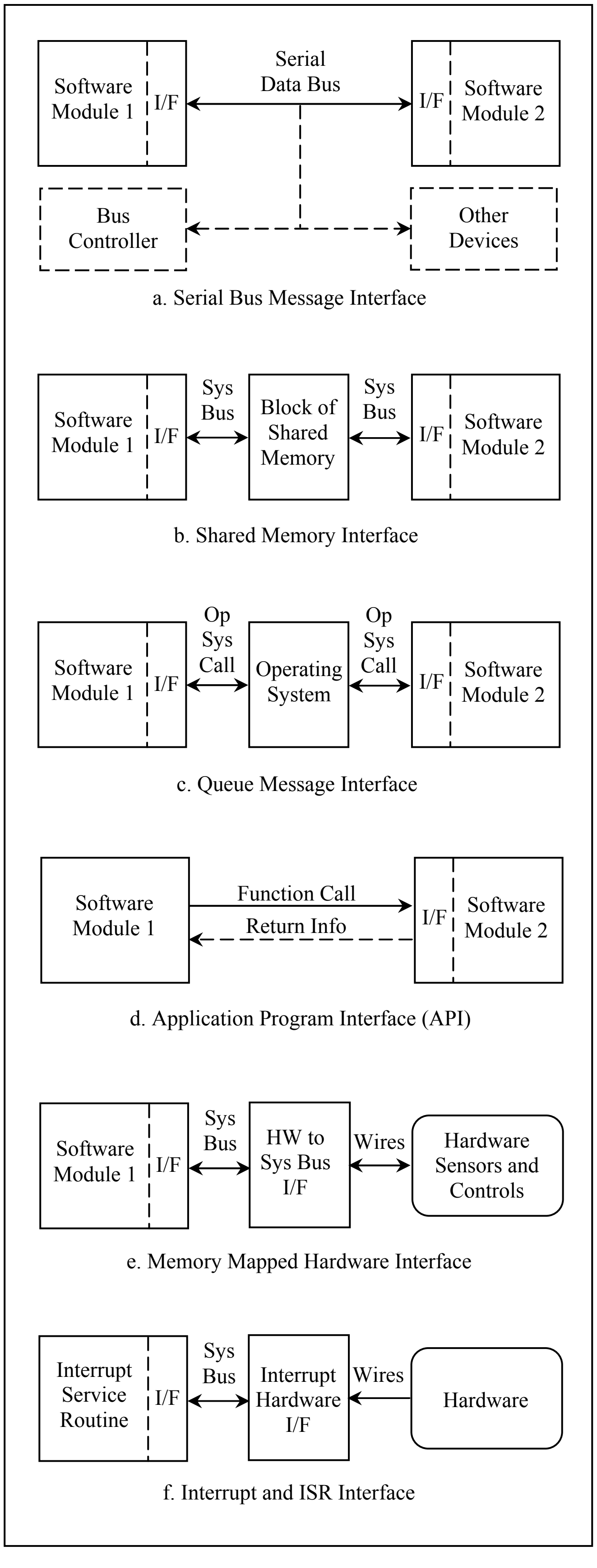


Figure 2 – Common Types of Software Interfaces

e. *Memory-mapped hardware interface.* This interface is applicable where hardware sensors or controls are accessible to software as memory addresses. An SFMEA doesn't consider hardware failures but it does analyze the effects of bad data from hardware sensors on the software that processes it, as well as bad data from software that operates hardware controls. This kind of interface can be very straightforward but the analysis may become more complex if the interfacing software involves activities such as statistical processing, crosscheck validations among different signals, and critical timing.

f. *Interrupt and ISR (Interrupt Service Routines).* This interface is applicable to hardware devices that cause the software to stop what it's doing and jump to a specific task (the ISR) associated with the device that caused the interrupt. Upon completion of the ISR, software execution should return to the previous task. An inherent danger is that the previous task, upon resumption, may combine intermediate results it had computed prior to the interruption with results computed *after* the interruption. This can produce catastrophic results. Use of interrupts often becomes complex because systems typically have many interrupts assigned with different priorities – meaning that some interrupts will interrupt the activities of others asynchronously and unpredictably – making this kind of interface very risky and very hard to analyze exhaustively. A number of excellent papers have been written on the subject [4] and analysts who perform an SFMEA should read and understand them.

Figure 3 summarizes typical activities of these common software interfaces. Not all activities will apply to any one interface, but the SFMEA must address them as part of the overall analysis.

7 SFMEA AT INTERFACES, STEP BY STEP

How should an SFMEA address failure modes at interfaces, particularly for interfaces between software modules from independent groups of developers? First, for the SFMEA to be effective, it must cross the interface by considering each side of the interface as though it were part of a single module. To do so, you must obtain required information from each development team. Next, you must look for failure possibilities that are not normally considered in a hardware FMEA, which typically looks at each component, one at a time, and considers the ways it might fail and the system consequences of each. If the goal of an SFMEA is to look at all possibilities of software failure, then it must examine interface interactions across the interfaces. A general process for each particular interface follows (Note: SFMEA process must be tailored to each individual system under analysis). The steps suggested here will overlap (see Table 1).

Step 1 is to identify the interfaces that should be in the SFMEA. This is not a trivial task. First, as with any SFMEA, only critical parts of a software system should be subject to the analysis because the effort can involve considerable time and expense. Unfortunately, identifying critical software and associated interfaces is difficult because items that are apparently noncritical can sometimes affect critical items. A sobering but easily understood example from the hardware world is Swissair Flight 111, on which a short circuit in the entertainment system – not normally considered safety critical – caused loss of the aircraft. The safest approach is to have the analysis team and the development team jointly determine which parts of a system are safety-critical or mission-critical.

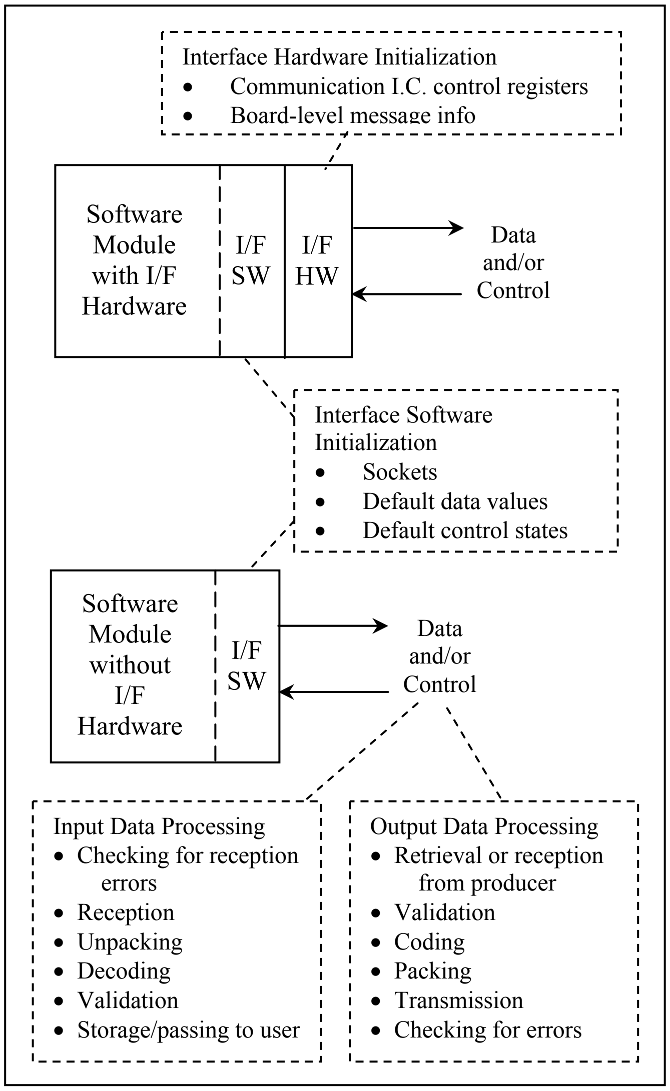

Interface Hardware Initialization
• Communication I.C. control registers
• Board-level message info
Software Module with I/F Hardware
I/F SW
I/F HW
Data and/or Control
Interface Software Initialization
• Sockets
• Default data values
• Default control states
Software Module without I/F Hardware
I/F SW
Data and/or Control
Input Data Processing
• Checking for reception errors
• Reception
• Unpacking
• Decoding
• Validation
• Storage/passing to user
Output Data Processing
• Retrieval or reception from producer
• Validation
• Coding
• Packing
• Transmission
• Checking for errors

Figure 3 -- Typical Software Interface Activities

Step 2 is to examine each interface design to determine whether all possibilities of data failure – as applicable to the particular interface under analysis – are even detected. Each kind off failure listed in Section 4, and many others, can be detected and handled by common software techniques, but software designers don't always think of (or understand) all reasonable possibilities of failure. Since no one can foresee all thing that can go wrong, the analysis team should jointly develop an applicable list of failure modes for each kind of interface being considered. The lists are living documents and will be revised as the analysis proceeds. If certain failure modes are not apparently detected, the analysis team should alert the development team well before the final report is prepared. If you think the software developers might have missed something, pick up the phone!

Step 3 is to examine the handling technique used for each kind of applicable failure in this interface, and determine the consequences. Loss of one data value (or message or data block) may be perfectly benign, but an ongoing loss will probably affect system performance. (Note that adequate redundancy may also make the consequence of such failures "no effect.") Partial loss of data or late data usually causes some system consequences and the SFMEA must determine what they are. Unexpected repeat of data messages (as seen by the author, due to problems with commercial driver software for a MIL-STD-1553 interface) may be very unlikely, but the analysis should address the possibility to determine system consequences. If the analysis is at the class or module level, then the analysis team must depend on information provided by the development team. As noted above, the analysis will be no more accurate than the information provided by the developers.

Step 4 is to examine relationships between this interface and other interfaces, particularly timing relationships. A module that expects data from two interfaces may fail if the data exchanges are not correctly synchronized or if one provides data that is older than the other. Sometimes data is sent across two or more different interfaces to provide redundancy. (This is commonly done in critical aircraft systems.) Assuming that only one failure at a time occurs, how does receiving software determine a failure and what does it do with that information? Conversely, and perhaps more importantly, what happens if the receiving software fails and unexpectedly declares an error with good data? A well-designed system with redundant interfaces might also have redundant software modules to provide greater robustness, but not all do.

Step	Subject	Description
1	Identification of Critical Interfaces	Identify interfaces relevant to safety-critical and mission-critical software.
2	Identification of Failure Modes for Each Interface	Develop lists of failure modes applicable to each interface under analysis.
3	Determine Software Error Handling Techniques	Determine whether and how software detects and handles errors.
4	Determine Relationships Among Interfaces	Examine dependencies among data conveyed by separate interfaces.
5	Examine Interface Document Requirements	Look for errors, ambiguities, inadequate details, and contradictions that can mislead designers.
6	Generating the Report	Develop a worksheet to summarize findings in a meaningful way.

Table 1 – Summary of the Software FMEA Process

Step 5 is to examine the interface definitions established for this interface in the interface documentation. The idea here is look for possible misunderstandings in definitions such as measurement units, update rates, situations where data may be irrelevant or ignored, and other possibilities such as those listed in Section 3. Apparent problems with the interface documents are another reason to pick up the phone. All documentation problems should be fixed well before the final SFMEA report. If the analyst is not satisfied with an explanation, or believes that an explanation is incorrect, then the given explanation (and the explainer's name and date) should be included in the final report in the "basis of analysis" section.

Step 6 is to record findings on the worksheet and generate the final report. A typical hardware FMEA worksheet is a table of components, and a typical software FMEA worksheet is a table of variables, methods, or classes. An SFMEA across a software interface will include entries for components such as messages, input/output data associated with hardware, and APIs. Each of these components typically has several failure modes (for example, "invalid data value(s)," "lateness," and "loss of synchronism").

8 AVOIDING DESCRIPTIVE OVERKILL

Human analysts who produce any kind of FMEA can easily supply far more detailed failure result descriptions than necessary for others to understand the system consequences. This is particularly true for an interface SFMEA, when a single failure can produce a great number of variables with unexpected values. For example, an interface problem causing lost or unreliable control data to a transmitter could cause many problems when considered individually, but an adequate description may simply be "inability to control transmitter" or "transmitter operates autonomously." If either description includes the possibility that the transmitter could radiate, then the failure might be considered a critical failure as well.

An excellent approach for controlling the details of failure effect descriptions, and limiting the number of them, is to maintain all such descriptions in a table or in a database. There are a number of advantages to this approach.

- Starting with an initial table of system effect descriptions, analysts can see the desired level of detail and style.
- Descriptions can be categorized and arranged to make it easier to find the one that's needed at the moment, or to ascertain that a new description must be added.
- Putting descriptions in the same place prevents different people from inventing different ways to describe the same thing.
- A senior analyst can review new descriptive text as it is being supplied to assure that the level of detail is no greater than necessary.

There is merit using the phrase "same as …" because it limits the number of consequences that systems engineers must review, and also because it may mean that an error in a certain type of failure description may need to be corrected in only one place.

9 WIRING CONSIDERATONS

An SFMEA involving an interface with copper paths should consider software consequences of short (and open) circuits because a typical HFMEA doesn't. In this situation, the SFMEA should take advantage of a bent pin analysis for the interface, if a bent pin analysis exists [5]. (The bent pin analysis should include open wires as failure modes.) The SFMEA should also consider how a single failure mode of any electronic component such as a multiplexer or gate array might affect multiple software variables. A separate worksheet line should address each such hardware failure mode, the affected software components, and system consequences. The general idea, of course, is to consider all possibilities of failure without viewing a hardware/software boundary as point at which a particular analysis stops.

REFERENCES

1. N. Ozarin, M. Siracusa, "A Process for Failure Modes and Effects Analysis of Computer Software," *Proc. Ann. Reliability & Maintainability Symp*, (Jan.) 2003.
2. N. Ozarin, "Lessons Learned on Five Large-Scale System Developments," *Proc. of the IEEE International Systems Conference,* (Apr.) 2007.
3. G. J. Holzmann, "Conquering Complexity," *Computer*, (Dec.) 2007
4. S. M. Beatty, "Improving Software Safety: Finding the Defects that Testing and Inspection Miss," *Proc. of the 22nd International System Safety Conference,* 2004
5. N. Ozarin, "What's Wrong with Bent Pin Analysis, and What to Do About It," *Proc. Ann. Reliability & Maintainability Symposium*, (Jan.) 2008

BIOGRAPHY

Nathaniel W. Ozarin
The Omnicon Group Inc.
40 Arkay Drive
Hauppauge, New York 11788 USA

e-mail: ozarin@ieee.org

Nat Ozarin is a senior engineering consultant at The Omnicon Group Inc. (www.omnicongroup.com), 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.