Experience Report: Contributions of SFMEA to Requirements Analysis

Robyn R. Lutz* and Robert M. Woodhouse
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109

Abstract

This experience report describes the lessons learned from the use of Software Failure Modes and Effects Analysis (SFMEA) for requirements analysis of critical spacecraft software. The SFMEA process was found to be successful in identifying some ambiguous, inconsistent, and missing requirements. More importantly, the SFMEA process, followed by a backward analysis somewhat similar to Fault Tree Analysis (FTA), identified four significant, unresolved requirements issues. These issues involved complex system interfaces and unanticipated dependencies. Our results challenge some current views on the limitations of SFMEA and suggest that recent efforts by researchers to integrate SFMEA with a broader FTA approach have merit.

1. The Problem

There are software programs onboard spacecraft that must autonomously detect, identify, and oversee the recovery of the spacecraft from faults during flight. Since these faults can threaten the well-being of the spacecraft and the success of its scientific mission, the software that responds to such faults is considered to be critical by the development team. A fault is given the standard definition here of being either "a defect in a hardware device or component" or "an incorrect step, process, or data definition in a computer program" [4]. Those faults which can cause power loss, excessive temperature, propellant tank overpressure, interruption of uplink commandability, or loss of downlinked scientific and engineering telemetry are detected and handled by onboard software.

Requirements analysis of this critical software is difficult since the software is often both complex and highly coupled. The software that responds to faults is often dependent on other distributed software and hardware components (for example, a single hardware fault may affect multiple software processes) and subject to timing constraints (for example, the software must provide quick recovery of functionality). These properties make the correct and complete specification of requirements hard to determine and hard to validate.

In particular, inadequate software responses to extreme conditions and boundary cases are of concern. Appropriate software responses to anomalous hardware behavior, unanticipated states, invalid data, and signal saturation are robustness issues that should be resolved, if possible, during the requirements phase.

2. Our Approach

This experience report describes our use of Software Failure Modes and Effects Analysis (SFMEA), followed by a backward analysis somewhat similar to Fault Tree Analysis (FTA), to assist in analyzing the software requirements for critical portions of the spacecraft software. The approach was used on twenty-four software modules on two spacecraft systems, Cassini and Galileo. The goals were to help reduce the number of failure modes, minimize the effect of the remaining failure modes, and search for unanticipated failure modes. A failure mode is defined to be "the physical or functional manifestation of a failure." A failure is defined to be "the inability of a system or component to perform its required functions within specified performance requirements limits" [4].

Software Failure Modes and Effects Analysis is an extension of the hardware Failure Modes and Effects Analysis (FMEA). The procedure for performing hardware FMEA has been standardized [10]. There is no comparable standard for performing SFMEA, although its use is well-documented [3, 11]. For ex-

*To appear, ICRE '96, April 15-18, 1996, Colorado Springs, CO. First author's mailing address is Dept. of Computer Science, Iowa State University, Ames, IA 50011.

ample, a technique similar to SFMEA, called Software Error Effects Analysis (SEEA), was used in the development of the rendezvous and berthing software for the Columbus Free Flyer. For critical software, a SEEA was required [14]. The System Safety Analysis Handbook provides a brief, non-procedural description of SFMEA [13]. A more detailed description of the SFMEA process as applied to our project appears in Section 3.

We embedded the SFMEA in a two-step requirements analysis process (Fig. 1):

1. The SFMEA used forward searching to identify Cause/Effect relationships in which unexpected data or software behavior (causes) can result in failure modes (effects). For example, "outdated sensor data" (cause) can "prevent the software from commanding a needed hardware reconfiguration" (effect).

   Note that although the cause is often labeled a "fault" in descriptions of SFMEA, it is more useful to consider unexpected or anomalous data and behavior, as well as strictly incorrect data and behavior. This is especially true for SFMEA performed during requirements analysis, since a "fault" at this early stage often means nothing more concrete than a deviation from expectations.

2. A backward search technique was then used to examine the possibility of occurrence of each anomaly (cause) that produced a failure mode (effect). In the example above, the root node for the backward search was "outdated sensor data." In this case our backward search for circumstances that could lead to outdated sensor data found a situation in which failed hardware continued to provide (inaccurate) data to the software. This bad data, due to the voting logic in the software, could veto a needed recovery action. By demonstrating the possibility of a new failure mode (obsolete data preventing required actions), the requirements specifications were improved. The failure mode was eliminated by a change to the software requirements.

   The backward search is similar to a Fault Tree Analysis, except that the root node (the cause) is not necessarily a fault or even an event. A Fault Tree Analysis, on the other hand, takes a known fault or hazard as its root and works backward to determine the possible causes [1]. Another difference between our backward search and FTA is that Software FTA is usually applied to code, whereas the backward search here is applied to software requirements. Since Fault Tree Analysis has been previously documented in detail, no further description is provided here [6].

   Note also that the backward search in this requirements analysis evaluates only the "possibility" of occurrence, not the likelihood. At the requirements phase of development there is insufficient knowledge to provide any numerical measurement of the probability of occurrence.

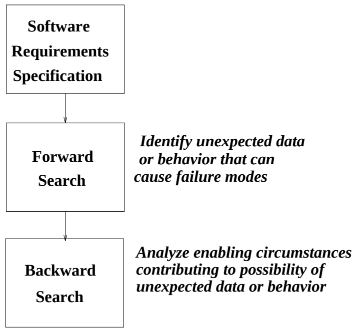

Figure 1. Overview of Analysis Process

2.1. Integrating SFMEA and Backward Search

In our experience, the strength of SFMEA (identifying previously unknown failure modes) and the strength of backward search (identifying combinations of events and circumstances that could cause the hypothesized fault to occur) were complementary. Thus, some current views regarding the limited effectiveness of SFMEA were not supported by the results of our integrated SFMEA and backward search approach.

For example, SFMEA is often described as only considering one discrepant event (fault) at a time, rather than combinations of events. We found, however, that when integrated with a backward analysis, the SFMEA often helped isolate combinations of events and circumstances that can lead to undesirable states.

It was interesting that in four cases the failure mode identified by the SFMEA was not a previously known failure mode. Thus, if a FTA had been performed starting from the known failures, these four requirement inadequacies would have remained hidden. Instead, the SFMEA isolated a cause (e.g., bad input) that led to an undesired effect (e.g, bad control decision). The backward search (e.g., "how could that bad input reach the software?") then identified a combination of events or unexpected interactions that could lead to the failure mode postulated in the SFMEA.

Our results indicate that recent work to integrate the forward search for effects (typical of SFMEA) and the backward search for contributing causes (typical of FTA) has merit [2, 8, 9]. For example, a recent paper by Maier describes the use of a fault-tree based hazard analysis to derive safety requirements for a robot's control software. FMECA (Failure Modes, Effect, and Criticality Analysis) is performed on the documented software requirements. Maier finds that the major benefit of the FMECA lies in its being a preparatory activity to fault tree construction. [8].

A recent paper by McDermid and Pumfrey describes a technique for software safety analysis based on a structured approach to the "imaginative anticipation of hazards" [9]. Based on the HAZOP approach [6], their work concentrates on information flows and develops sets of guide words to prompt consideration of hypothetical failures. Whereas we perform the SFMEA first and then the backward search, they (consistent with the HAZOP technique) first perform the backward search for causes and then consider the effects of each hypothetical failure.

It would be interesting to compare the effect of the ordering of the searches on the success of the analysis. From our limited experience, it is not clear whether the order of the steps is significant. For example, all four of our unanticipated failure modes might have been identified even if a backward search for contributing causes had preceded the SFMEA.

2.2. SFMEA During Requirements Analysis

Software Failure Modes and Effects Analysis is most commonly used during design analysis. We found that SFMEA was effective during requirements analysis when, as in our case, the requirements specification provided sufficient detail. The requirements document that we used contained over 200 pages of English text, data tables, and flowcharts describing 24 software modules. The requirements specification defined a new software system. There was no reuse of software components from previous systems.

For simple, stand-alone software where few details are documented at the requirements stage, SFMEA is not feasible until a design document exists. However, we found that for a complex, embedded application such as a spacecraft, the SFMEA improved the quality of the software requirements specification as well as the understanding of the software problem.

In particular, SFMEA made the following contributions to the requirements analysis.

1. Early understanding of requirements. Understanding what the software requirements are is a huge task in a complex, distributed system. SFMEA helped identify constraints that would be imposed on the design by other parts of the system or by the context in which the embedded software operated. These constraints and dependencies were thus able to be incorporated into the requirements specification.
2. Communication. The requirements specification document is written by a system engineer, and then handed off to a design development team. A clear, unambiguous, and complete document minimizes the possibility of misunderstanding at this juncture. The SFMEA assisted in this effort.
3. Error removal. Requirements errors, especially interface requirements errors, have historically been a source of persistent errors during spacecraft development (sometimes escaping detection until system testing) [7]. Often these requirements errors involve unanticipated failure modes or interface dependencies that are difficult to detect. The SFMEA was able to identify some such errors prior to design decisions being made, saving subsequent time and effort.

SFMEA has some well-known limitations and disadvantages that were confirmed by our experience. Like most failure analysis methods, SFMEA is time-consuming; much of it is tedious; and it depends on the domain knowledge of the analyst and the accuracy of the documentation. In addition, unlike hardware, a complete list of failure modes for software cannot be assembled. SFMEA is also a manual rather than an automatic method.

The SFMEA approach was chosen as part of the requirements analysis process on this project largely because it contributes to a systems approach to requirements validation. It focuses on the ways in which software can contribute to the system's reaching an undesirable state. SFMEA analyzes the software's response to hardware faults (e.g., malfunctioning sensors) and to operator errors that result in bad input data (e.g., inappropriate commands). SFMEA also analyzes the effect of incorrect software actions (e.g, a software process issuing erroneous reconfiguration commands) on the hardware components. SFMEA pays particular attention to hidden dependencies or interactions that could cause the propagation of erroneous data to other software modules. In this way the requirements analysis process exploits the available domain expertise.

SFMEA differs from a causal analysis such as FTA in that SFMEA postulates the existence of bad data or unexpected behavior and then investigates the effects of that anomaly on the correct functioning of the software module and the system. Whether the data or behavior could actually be corrupted in that manner (e.g., the arrival of outdated sensor data or abnormal termination of the software module) is not the primary concern at this point of development. The focus in SFMEA is instead on the consequences of incorrect data or inappropriate software activity. This is especially appropriate for requirements analysis since judgments as to whether a particular failure scenario is credible very often shift as development progresses.

If the effects of the bad data or unexpected behavior are shown to be acceptable, then confidence in the requirements is enhanced. Examples of acceptable effects are that bad data are rejected by the software or that premature termination of the software module still leaves the system in a consistent state.

If the effects of the bad data or unexpected behavior are shown to be unacceptable and a backward search confirms the possibility that the situation could occur, then the information is fed back into the requirements development process. Examples of unacceptable effects are that the bad data are used in a control decision resulting in erroneous issuance of commands, or that an abnormal termination of the software module results in a global variable being updated while the status variable still indicates that no change has been made.

3. The SFMEA Process

This section describes the process by which the SFMEA (the "Forward Search" in Fig. 1) was performed on the spacecraft software modules. Additional description is available in [12]. Detailed descriptions of backward search are available in [6].

In a message-passing model of a distributed system, two kinds of failures are generally represented: communication failures and process failures [5]. In accordance with this model, two kinds of failures are analyzed in a SFMEA for each software process. To assist in the analysis of any possible failures of the software, two tables are constructed: a Data Table and an Events Table. A Data Table involves communication failures. It provides the information needed to analyze data dependencies and software interface errors. An Events Table involves software process failures. The Events Table provides the information needed to analyze the effects of failures possibly caused by software that fails to function correctly. The investigation of faults in the two tables is consistent with current classifications of defects in software

The first type of table is the Data Table (Table 1). This table evaluates both the effect of *receiving* bad or unexpected input data on the behavior of the process being analyzed, and the effect of *producing* bad or unexpected output data on the behavior of the processes that use this data.

For each input (data item read or received by the software process) and each output (data item written or output by the software process (including, in our application, commands to spacecraft subsystems), each of the following four faults is postulated:

1. Absent Data: Lost or missing messages, absence of sensor input data, lack of input or output, failure to receive needed data, missing commands, missing updates of data values, data loss due to hardware failures, failure of a software process or sensor to send the data needed for correct functioning of this software module.
2. Incorrect Data: Bad data, flags or variables set to values that don't accurately describe the spacecraft's state or the operating environment, erroneous triggers, limits, deadbands, delay timers; erroneous parameters, wrong commands output, or wrong parameters to the right commands; spurious or unexpected signals.
3. Timing of Data Wrong: Data arrive too late to be used or be accurate, or too early to be used or be accurate; obsolete data are used in control decisions (data age); inadvertent, spurious (unexpected), or transient data.
4. Duplicate Data: Redundant copies of data, data overflow, data saturation.

Data Item	*Data Fault Type*	*Description*	*Effect*
Critical mode flag	Incorrect value	Flag set to true during non-critical mode	Unnecessary reconfiguration commanded

Table 1. Data Table Example

For each of these four fault types the Data Table includes the description of the fault as applied to the relevant data item and the effect, both locally and more globally on the subsystem and system.

The second type of table is the Events Table (Table 2). For each event that occurs as the process executes, four event fault types are postulated. What constitutes an event depends on the level of detail of the documentation provided, but is usually considered to be a single action (e.g., perform a calculation, sample a sensor value, command an antenna to slew to another position). Although some requirements will not be broken down into events until the design stage of development, many of the critical requirements are already specified in terms of the actions, or transformations of inputs into outputs, that must occur.

There are four types of event faults:

1. Halt/Abnormal Termination: Open, stuck, hung, deadlocked at this point (event) in the process.
2. Omission: Event fails to occur but process continues execution; jumps, skips, short.
3. Incorrect Logic/Event: Behavior is wrong, logic is wrong, branch logic is reversed, wrong assumptions about state, preconditions, "don't cares" aren't truly so; event (e.g., command issued) is wrong to implement the intent or requirement.
4. Timing/Order: Event occurs at wrong time or in wrong order, event occurs too early (premature; system not in proper mode to receive or process it), too late; the sequence of events is incorrect, an event that must precede another event doesn't occur as it should; iterative events occur intermittently rather than regularly; events that should occur only once instead occur iteratively.

For each of these four fault types, the Events Table includes a description of the fault and its local and global effect.

Event	*Event Fault Type*
Calculate pointer into a table	Incorrect logic
Description	*Effect*
Pointer is miscalculated	Points past end of table, preventing reconfiguration

Table 2. Events Table Example

Using the SFMEA tables, concerns and possibly vulnerable areas were identified. These were documented in sufficient detail so that a reader could determine whether the requirements needed to be changed. Most of the effort of performing a SFMEA was expended here, in reviewing the SFMEA tables to see if the software lacked robustness against failures. During this step, contact with the project personnel was important to distinguish ambiguous/incomplete documentation from actual requirements flaws. Including copies of all relevant documentation as well as explicit references to memos, expert opinions, etc., in the final reports encouraged rapid feedback of the results into the development process.

4. Results

Forty-eight issues were identified in the first seven SFMEA completed for the Cassini spacecraft software currently under development. The detection and resolution of these requirements issues are described below in some detail to illustrate the successes and limitations of the forward and backward search techniques we used.

In addition, results from four more recently performed SFMEA and thirteen SFMEA performed on a prior spacecraft (Galileo) are described to provide supplementary context. Since information is unavailable regarding which issues resulted in changes to requirements in these early or very recent SFMEAs, their results are only summarized.

Of the forty-eight issues found in the first seven SFMEA performed on the current spacecraft, twenty-five resulted in changes to the requirements specifications. Of greatest interest were the four of these twenty-five involving unresolved requirements issues identified during the SFMEA. All four of these major issues involved interface requirements (between software modules), historically a difficult area for requirements validation. In each case, further requirements analysis was undertaken and led to a change in the requirements.

The four issues were failure modes involving previously unanticipated scenarios inadequately handled by existing requirements.

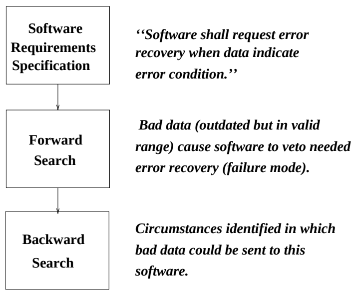

Figure 2. Example of Use of SFMEA

For example (Fig. 2), one SFMEA identified the following cause/effect relationship: Cause: inaccurate data from a sensor; Effect: prevention of needed error recovery. More specifically, it was found that inaccurate input data could in effect veto the execution of a software response that was needed. This could occur when the input data appeared to be healthy, but was actually reflecting an obsolete state.

Finding this cause/effect relationship in the SFMEA led to a backward speculative inquiry as to whether any set of circumstances existed in which the software would actually receive seemingly healthy but obsolete data. The backward search from the cause (obsolete data in healthy range) found that the current interface requirements allowed obsolete data from a failed sensor to continue to be sent to the software. If a sensor failed with healthy values, then that data would continue to be used indefinitely, allowing the failure mode identified in the SFMEA.

Subsequently, the software requirements for the piece of software that passed the sensor data to this module were changed. The new requirement states that the software will ensure data freshness by only distributing data from a sensor that has passed a test.

It is interesting to note that the new requirement indicated by this SFMEA was not actually for the software on which the SFMEA was performed but for software on the other side of the data interface. This provides an example of how integrating SFMEA with a backward search for enabling circumstances can detect hidden interface errors. In this case, the combination of SFMEA and backward search uncovered a latent requirement which could then be readily resolved prior to design.

A second example of a requirements concern found by the use of SFMEA and backward search illustrates how subtle the intertwining requirements that lead to a robust design sometimes are. This failure mode involved a race condition between two different requests for error-handling. If an overpressure in the fuel tank occurred and the “wrong” software request won the race, then the software would skip a milder response using non-consumable resources (latch valves) and proceed to a more drastic response using consumable resources (pyro valves). This failure mode was detected by noting in the SFMEA that the persistence limits (timers from the time the overpressure was detected until the request for fault-handling was triggered) were identical for the two software modules. In normal operations this is correct, since the timers will start counting at different pressures. However, if both software modules were to start counting concurrently (SFMEA cause), a race condition would ensue (SFMEA effect).

Taking the SFMEA cause (concurrent countdown) as the root node, a backward search found that this hypothetical occurrence could happen only when both software modules detected an error at the same time. This is possible only in the following combination of circumstances: a large overpressure in a propellant tank occurs just prior to the simultaneous enabling of two software programs, which occurs only when the spacecraft has just achieved insertion into orbit around the planet.

Finding this unanticipated failure mode in the SFMEA led to a requirements change (staggered enabling of the software programs) to prevent the consequences of this remote but undesirable state. This provides a good example of how SFMEA combined with backward search can successfully investigate the effects of multiple or coincident events (anomalies) during requirements analysis.

Twenty-one additional issues identified in the SFMEA prompted easier changes to the requirements specification. Each of these documentation issues resulted in an update to the requirements specification. However, none involved an underlying inadequacy in the software requirements nor was additional requirements analysis needed. It is probable that any sufficiently close reading of the requirements document or other means of informed static analysis, combined with early analysis of operational scenarios, would raise the same issues. The numbers in parentheses indicate how many issues in each category were found.

- Correct/Clarify documentation (9) This included adding rationales for requirements, documenting assumptions, and removing ambiguous statements. An example is that the conditions under which a message is to be re-sent were unclear.
- Resolve inconsistencies (7). An example is inconsistent naming of flags.
- Add rules specifying proper operational usage (2). An example is that flight maneuvers sometimes must take into account whether error recovery software has executed yet.
- Add missing requirements (3). An example is that a requirement to disable a portion of the software following initial execution was missing.

The remaining twenty-three of the forty-eight issues raised by the SFMEA did not result in changes to the requirements specification. The main reasons for this were:

- Cost/benefit tradeoff decisions by the developers (e.g., some multiple-failure scenarios were too unlikely to merit the cost of change);
- SFMEA analyst error (where all later agreed that the existing requirements sufficed);

- Issues that disappeared as a side-effect of other updates to the requirements;
- Requirements that were cited as missing but were, in fact, documented elsewhere (in which case, only a cross-reference was needed); and
- Stylistic disagreements (e.g., in what section of the document a requirement belonged).

Similar results to those found on the Cassini spacecraft were found in earlier SFMEA performed on portions of the software of Galileo, another spacecraft. In general, since SFMEA emphasizes a systems approach to software analysis (software responses to hardware faults; effects on hardware of software actions), the SFMEA is sometimes able to uncover hidden dependencies among the components. Examples of the issues found by these earlier SFMEA include:

- Unexpected interactions among distributed software processes could occur. In one case additional bus commands were required, but this was not recognized until the SFMEA postulated a new failure mode.
- Erroneous invocation of software programs was found in several instances. For example, a software requirement to ignore transient faults was missing in one case.
- Unexpected interaction among redundant components, i.e., between the (nearly) identical copies of the software resident on the prime and backup processors, led to a failure mode. For example, required hardware reconfigurations were omitted when a service routine was invoked by both redundant components within a certain time interval.
- Unexpected propagation of results could occur in one scenario, given the current requirements. This meant that during a programmed delay, certain commands to a remote unit were able to be reissued, contrary to the intended behavior.
- Unstated assumptions required for correct behavior were not always documented in the specifications. As an example, a precondition that was not checked but assumed to be true (the settings of some switches) could occasionally be false.

Although a backward search was not explicitly performed in connection with these earlier SFMEAs, several of the failure modes resulted from both the analysis of the effects of the hypothesized anomaly (the SFMEA) and the analysis of the possibility of that anomaly ever occurring (the backward search). It would be interesting to see if other applications of FMEA to software similarly contain elements of both forward and backward searches.

5. Conclusion

The lessons learned from our application of SFMEA followed by backward analysis to the spacecraft software requirements were the following.

1. Although SFMEA is usually employed for design validation, we found that SFMEA was feasible and useful for requirements analysis in a large, well-documented system.
2. Integrating the forward search approach of SFMEA with a backward search for contributing causes enhances the effectiveness of SFMEA. In particular, we found previously unknown failure modes, multiple and coincident anomalies, and hidden dependencies between software processes. These results challenge the current view of SFMEA as limited to investigating known failure modes, single failures, and simple deviations from expected behavior.
3. Our experience tends to validate recent efforts to combine forward and backward search techniques. Whether the order of analysis matters is currently not clear and merits experimental study. Our results from first performing a forward search for effects, then a backward search for contributing causes have not resolved this issue, nor do they answer the question of which order is most cost effective.
4. The biggest benefit of combining SFMEA with a backward search for contributing causes was in the discovery of unknown failure modes during the requirements analysis. We found four such issues that needed further requirements analysis to understand and resolve. In addition, we found twenty-one other requirements issues significant enough to change the content of the requirements specifications (e.g., missing, ambiguous, or inconsistent requirements). These twenty-one requirements issues could probably have been found by some combinations of other static analysis methods (e.g., formal specification, formal inspections, and construction of operational scenarios). However, it is probable that the four new failure modes could not have been readily found with other requirements analysis methods.

For large, well-documented projects, we recommend SFMEA in combination with backward search as an effective way to remove requirements errors and add robustness to the requirements specification.

Acknowledgments

The authors thank Steven Tyler and Robert Keston for their early definition of the SFMEA process at JPL. The authors thank Anna Bruhn, Sarah Gavit, Yoko Ampo, Ching Chen-Tsai, John Day, Merlin Dorfman, and the reviewers for their valuable suggestions.

The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

[1] S. S. Cha, N. G. Leveson, and T. J. Shimeall, "Safety Verification in Murphy Using Fault Tree Analysis," *Proc 10th Int Conf on S/W Eng*, Apr 1988, Singapore, 377-386.

[2] P. Fenelon and J. A. McDermid, "An Integrated Tool Set for Software Safety Analysis," *JSS*, 1993; 21: 279-290.

[3] J. R. Fragola and J. F. Spahn, "The Software Error Effects Analyis; A Qualitative Design Tool," *IEEE Symp on Comp S/W Rel*, 1973, 90-93.

[4] *IEEE Std Glossary of S/W Eng Terminology*, IEEE Std 610.12-1990.

[5] L. Lamport and N. Lynch, "Distributed Computing Models and Methods," *Formal Models and Semantics, Vol. B, Handbook of Theoretical Computer Science*, Elsevier, 1990.

[6] N. Leveson, *Safeware, System Safety and Computers*, Addison-Wesley, 1995.

[7] R. Lutz, "Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems," *JSS*, to appear.

[8] T. Maier, "FMEA and FTA To Support Safe Design of Embedded Software in Safety-Critical Systems," *CSR 12th Annual Wkshp on Safety and Rel of S/W Based Sys*, 1995, Bruge, Belgium.

[9] J. A. McDermid and D. J. Pumfrey, "A Development of Hazard Analysis To Aid Software Design," *Proc of COMPASS '94*, Jun 27-30, 1994, Gaithersburg, MD, pp. 17-25.

[10] *Procedures for Performing a Failure Mode, Effects and Criticality Analysis*, MIL-STD-1629A, 24 Nov 1980.

[11] D. J. Reifer, "Software Failure Modes and Effects Analysis," *IEEE Trans on Rel*, R-28, 3, Aug 1979, 247-249.

[12] "Software Failure Modes & Effects Analysis," *Software Product Assurance Handbook* . JPL D-8642, 1995.

[13] System Safety Society, *System Safety Analysis Handbook*, July 1993.

[14] J. Wunram, "A Strategy for Identification and Development of Safety Critical Software Embedded in Complex Space Systems," IAA 90-557, pp. 35-51.