Leading Our World In Motion

2005-01-0785

SAE TECHNICAL
PAPER SERIES

Effective Application of Software Safety Techniques for Automotive Embedded Control Systems

Barbara J. Czerny, Joseph G. D'Ambrosio, Brian T. Murray and Padma Sundaram
Delphi Corporation

Reprinted From: Occupant Safety, Safety-Critical Systems, and Crashworthiness (SP-1923)

ISBN 0-7680-1631-2

9 780768 016314

SAE International™

**2005 SAE World Congress
Detroit, Michigan
April 11-14, 2005**

400 Commonwealth Drive, Warrendale, PA 15096-0001 U.S.A. Tel: (724) 776-4841 Fax: (724) 776-5760 Web: www.sae.org

All SAE papers, standards, and selected books are abstracted and indexed in the Global Mobility Database.

ISSN 0148-7191

2005-01-0785

Effective Application of Software Safety Techniques for Automotive Embedded Control Systems

Barbara J. Czerny, Joseph G. D'Ambrosio, Brian T. Murray and Padma Sundaram
Delphi Corporation

ABSTRACT

Execution of a software safety program is an accepted best practice to help verify that potential software hazards are identified and their associated risks are mitigated. Successful execution of a software safety program involves selecting and applying effective analysis methods and tasks that are appropriate for the specific needs of the development project and that satisfy software safety program requirements. This paper describes the effective application of a set of software safety methods and tasks that satisfy software safety program requirements for many applications. A key element of this approach is a tightly coupled fault tree analysis and failure modes and effects analysis. The approach has been successfully applied to several automotive embedded control systems with positive results.

INTRODUCTION

The last decade has seen rapid growth of automotive safety-critical systems controlled by embedded software. Embedded processors are used to achieve enhancements in vehicle comfort, feel, fuel efficiency, and safety. In these new embedded systems, software is increasingly controlling essential vehicle functions such as steering and braking independently of the driver. Although many of these systems help provide significant improvements in vehicle safety, unexpected interactions among the software, the hardware, and the environment may lead to potentially hazardous situations. As part of an overall system safety program, system safety analysis techniques can be applied to help verify that potential system hazards are identified and mitigated.

During the execution of a system safety program, developers of embedded control systems recognize the need to protect against potential software failures. Unlike mechanical or electrical/electronic hardware, software does not wear out over time, and it can be argued that software does not fail. However, software is stored and executed by electronic hardware, and the intended system functionality that is specified by the software may not be provided by an embedded system if potential electronic hardware failures occur or if the software is incorrect.

In this paper, we define a *software failure* as any deviation from the intended behavior of the software of a system. There are three main categories of potential causes of software failure modes: hardware failures, software logic errors, and support software (e.g. compiler) errors.

Typical sources of potential hardware failures, which can be either internal or external to the controller the software executes on, include:

- Memory failures in either the code space or variable space,
- CPU failures (ALU, registers), and
- Peripheral failures (I/O ports, A/D, CAN, SPI, watchdog, interrupt manager, timers).

For example, memory cell failures can cause conditions where the software inadvertently jumps to the end of a routine or into the middle of another routine. Interrupt failure modes, such as return of incorrect priority or failure to return (thereby blocking lower priority interrupts), can also be caused by memory corruption.

Software logic errors may arise due to incomplete or inconsistent requirements, errors in software design, or errors in code implementation. Software logic errors can lead to failure conditions such as infinite loops, incorrect calculations, abrupt returns, taking a longer time to complete routine execution, etc. In addition, software stored in an embedded system may not be correct if the tools necessary to configure, compile and download the software do not function as expected.

Similar to the effective best-practice approach applied to help prevent potential system hazards due to hardware failures, embedded system developers can apply system safety engineering methods to protect against software failures. However, the unique potential failure modes and the overall complexity of software warrant that additional software-specific analysis methods and tasks be included in the overall system safety program. To address this need, the system safety

program should include a software safety component as well. A software safety program involves the execution of a number of software-related tasks intended to help identify and mitigate potential software failures.

Although requirements for an automotive software safety program can be derived from existing software safety guidelines and published sources [1,2,3,4], efficient methods and tasks for satisfying these requirements, that are appropriate for the automotive domain, are still needed. In this paper, we present a set of methods and tasks that we have effectively applied to several automotive embedded control systems to satisfy automotive software safety program requirements. First, we describe a generic software life cycle and its relation to a software safety process proposed by Delphi [5]. Next, we provide details on specific analysis methods and tasks that we applied for each of the major steps in the life cycle. Finally, we present our conclusions.

SOFTWARE SAFETY LIFE CYCLE OVERVIEW

Table 1 shows the typical software development life cycle phases and corresponding software safety tasks performed during each phase. The tasks shown satisfy the requirements of a proposed Delphi software safety program procedure [5]. Note that the Conceptual Design phase is actually part of the system development process, but is included here for completeness. In general, there may be more than one set of methods that can be applied to satisfy the required tasks. The specific set of methods selected depends on the target product's stage of development and on any unique aspects of the product.

Table 1: Relation Between Software Development Phases and Software Safety Tasks.

Software Development Phase	Typical Software Safety Tasks
Conceptual Design	Preliminary Hazard Analysis and SW Safety Planning
SW Requirements Analysis	SW Safety Requirements Analysis
SW Architecture Design	SW Safety Architecture Design Analysis
SW Detailed Design and Coding	SW Safety Detailed Analysis and SW Safety Code Analysis
SW Verification and Validation	SW Safety Testing, SW Safety Test Analysis, SW Safety Case

Figure 1 shows the six primary software life-cycle phases, where each phase has associated detailed software safety inputs, outputs, and tasks. These inputs, outputs, and tasks satisfy Delphi's proposed software safety program requirements for advanced automotive safety-critical systems and are consistent with part 3 of the IEC 61508 [2] standard that addresses software safety. The methods and tasks shown represent a tailored subset of those suggested by the Delphi software safety program requirements and by IEC 61508. Given that individual projects have unique aspects to them, the selected set of methods and tasks described in this paper may not be appropriate for all projects. In the following sections, we provide details of the specific software safety methods that we applied during the different software development phases of several of our automotive embedded control systems.

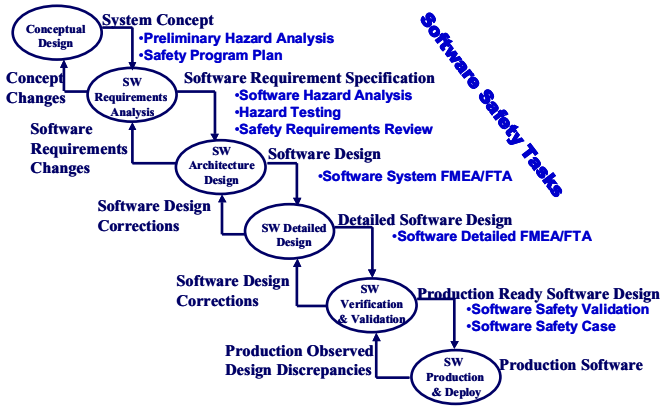

Figure 1: Software Life Cycle with Associated Software Safety Tasks.

CONCEPT DESIGN PHASE

During this phase of system and software development, project leaders must determine if a system safety program is required for the product concept. This decision is typically made based on past product knowledge or based on the results of a preliminary hazard analysis (PHA). Regardless of how the decision is made, a preliminary hazard analysis and system safety program plan are typically completed if a system safety program is required. If the preliminary hazard analysis identifies any potential hazards that may arise due to potential software failures, then a software safety program plan is developed as well.

PRELIMINARY HAZARD ANALYSIS

The goal of PHA is to identify potential high-level system hazards and to determine the criticality of potential mishaps that may arise. PHA is performed in the early stages of system development so that safety requirements for controlling the identified hazards can be determined and incorporated into the design early on. The PHA tends to quickly focus the design team's attention on the true potential safety issues of a product concept. The basic steps for performing a PHA are:

1. Perform brainstorming or review existing potential hazard lists to identify potential hazards associated with the system,
2. Provide a description of the potential hazards and potential mishap scenarios associated with them,
3. Identify potential causes of the potential hazards,
4. Determine the risk of the potential hazards and mishap scenarios, and

5. Determine if system hazard-avoidance requirements need to be added to the system specification to eliminate or mitigate the potential risks.

If time is a factor for a potential hazard or potential mishap occurrence, then the timing constraints that the potential hazard places on the design may be investigated as well.

Consider the hypothetical control system shown in Figure 2. A sensor provides the needed input signal to the system ECU. The system ECU then computes the actuator command satisfying the system function.

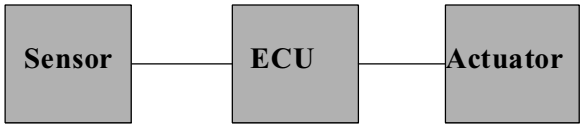

Figure 2: Example Control System.

One potential hazard of such a system is an unintended system function. Unintended system function can result in undesirable system behavior that could potentially be hazardous. Some of the causes of unintended system function would include potential ECU failures or sensor failures.

Table 2: Example Control System PHA.

Pot. Hazard	Pot. Hazard Risk	Pot. Causes	Safety Strategy	Revised Pot. Hazard Risk
Unintended System function	High	Sensor Fault; ECU Fault; Motor driver fault; Actuator fault;	High integrity sensor signals High Integrity ECU Operation; High Integrity Mechanical Actuator	Low

Table 2 shows a portion of the PHA for this example. For the system without a safety strategy implemented, the potential risk is high, because an unintended event may occur. However, once appropriate safety features are incorporated as specified by the safety strategy, the revised potential risk is low. In this example, high integrity sensor, ECU, and actuator design strategies will be implemented to help ensure potential failures are detected and handled appropriately. For example, one method to provide a high integrity-sensor signal value is to use two sensors and compare the output of the sensors for consistency. A sensor fault is detected if the values from the sensors do not agree within some tolerance, and when this occurs, the system transitions to a fail-safe state (e.g., controlled shutdown of the system).

To achieve high integrity ECU operation, the design team must consider the potential for unintended system function due to software failures. As previously described, software failures may occur if hardware faults exist.

SOFTWARE SAFETY PROGRAM PLAN

A software safety program plan is the plan for carrying out the software safety program for a project. This plan typically includes the software safety activities deemed necessary for the project and the resources and timing associated with the activities. In effect, this plan defines the software safety life cycle for the project. The plan typically evolves during the software safety life cycle to reflect newly identified needs.

SOFTWARE REQUIREMENTS ANALYSIS PHASE

In this phase of software development, the goals of the software safety program include identifying software safety requirements to eliminate, mitigate, or control potential hazards related to potential software failures. Software safety requirements may also stem from government regulations, applicable national/international standards, customer requirements, or internal corporate requirements. A matrix identifying software safety requirements may be initiated to track the requirements throughout the development process.

Methods used to satisfy the software safety goals include:

1. Software Hazard Analysis,
2. Hazard Testing, and
3. Software Safety Requirements Review.

Software hazard analysis identifies possible software states that may lead to the potential hazards identified during the PHA. Using the link established between software states and potential hazards, software *hazard-avoidance requirements* are developed and included in the software safety requirements specification. To help quantify these hazard-avoidance requirements, hazard testing identifies specific *fault response times* that must be provided by the software functionality to help ensure that potential hazards are avoided. In general, all of these activities are tightly coupled, with interim results from one activity feeding into the others. Finally, software safety requirements review helps ensure that safety requirements are complete and consistent. The following sections provide more detailed descriptions of the software safety analysis methods that may be applied to satisfy the goals of the software safety program during this software development phase.

SOFTWARE HAZARD ANALYSIS

Software hazard analysis consists of identifying the potential software failures that may lead to potential system hazards. For each potential system hazard, possible software states leading to the potential hazard are identified. Based on the link established between the potential hazards of the system and the potential software causes, any identified system hazard-avoidance requirements are translated into corresponding software hazard-avoidance requirements.

The most common technique applied to accomplish this task is fault tree analysis, which is a top-down (deductive) analysis method that identifies potential causes for some top-level undesired event. The immediate causes for the top-level event are identified, and the process is repeated, such that the causes are considered events, and their associated causes are identified. The analysis continues until a base set of causes is identified. For system-level software hazard analysis, these base causes are software states. It is important to note that at this point a software architecture or detailed design does not exist, so the software states identified in the FTA are anticipated. As described later, the analysis must be updated to reflect the actual software architecture and detailed design.

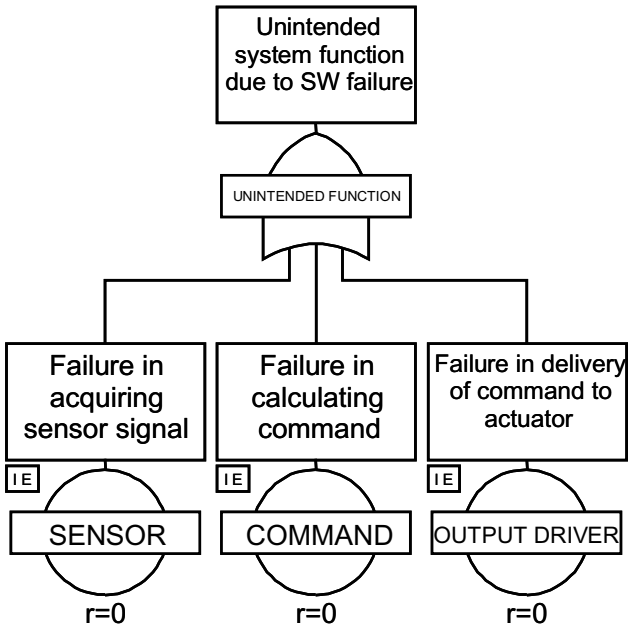

Figure 3: System-Level Fault Tree for the Example Control System.

For the example control system, the unintended system function fault tree in Figure 3 shows the identified potential software failures. For each of these potential software failures, high-level software safety requirements are specified (Table 3).

HAZARD TESTING

During hazard testing, requirements to test the actual behavior of the system under the potential hazard conditions are developed. The results of this testing provide the fault response times and signal deviation levels required by the system to avoid a potential hazard before it occurs. The fault response times drive the design of the software diagnostics. This timing is very critical in designing the software tasks schedule and may also give some insight into whether the chosen controller has enough processing power and throughput to handle the various tasks within the given time. Although these tests may initially be performed using simulation models or bench setups, the identified fault response times should be confirmed by testing the actual system in a vehicle if possible.

Table 3: Example Software Safety Requirements.

Req. No.	Software Safety Requirement
SW-SAFETY-1	Software sensor diagnostics shall detect deviations of actual vs. measured sensor signal.
SW-SAFETY-2	Software shall detect deviations of computed actuator command.
SW-SAFETY-3	Software shall detect actuator control errors resulting in a deviation of delivered vs. computed command

For the example control system, hazard testing using simulation and vehicle testing might lead to the hypothetical requirement that the undesired behavior produced in the vehicle due to failures shall not exceed a specific amount within a specific amount of time. Although the software does not yet exist, it may be possible to use this quantitative vehicle level requirement to quantify the software safety requirements based on the vehicle and system simulation model. In this paper, we assume that the vehicle requirement corresponds to the following ECU requirement: *the ECU output command delivered to the actuator shall not deviate from the desired value by X amount for more than Y ms*.

SOFTWARE SAFETY REQUIREMENTS REVIEW

Software safety requirements review examines the software safety requirements identified by software hazard analysis, to help ensure they are complete and consistent. Early identification of missing, incorrect or inconsistent software safety requirements allows the requirements to be modified with little or no impact to program schedule or cost. Late identification of software safety requirements deficiencies can result in expensive, schedule impacting changes to the overall

design. The software safety requirements analysis process also evaluates the software functional requirements for their impact on safety. The end product of this task is a set of software safety requirements for the software design. These requirements will be based on the earlier developed system level safety requirements and the results of the hazard analyses and hazard tests. The requirements also may include general software safety coding guidelines and industry, government, or international coding standards that must be followed by the software development team.

Table 4: Revised Software Safety Requirements.

Req. No.	Requirement
SW-SAFETY-1	Software sensor diagnostics shall detect deviations of actual vs. measured sensor signals of TBD amount within TBD ms.
SW-SAFETY-2	Software command diagnostics shall detect deviations of computed Actuator command of X amount within Y ms.
SW-SAFETY-3	Software communication/driver diagnostics shall detect actuator communication errors within Y ms
SW-SAFETY-4	Software failure management routine shall initiate controlled shutdown of the system immediately after a diagnostic detects a failure.
SW-SAFETY-5	All software shall conform to the MISRA C coding guidelines

For the example control system, the safety analysis results & requirements shown in Table 2, Figure 3, and Table 3 are reviewed for consistency and completeness. Table 4 above shows the updated requirements. The existing SW-SAFETY-2 requirement is revised based on the ECU integrity requirements obtained from hazard testing, with the specific limit values being directly assigned. The SW-SAFETY-1 requirement is revised to reflect that a TBD level and TBD detection time will be specified once the relationship between the sensor signal and command output is better defined. Since a communication error could result in a bad command being delivered to the actuator, SW-SAFETY-3 requirement is revised to reflect the detection time determined by hazard testing. At this point, there are no requirements on what should happen after a fault is detected. To address this, another requirement, SW-SAFETY-4 is added to specify system behavior once a fault is detected. It is also common to identify existing external or internal corporate standards that will be followed. Finally, a fifth requirement, SW-SAFETY-5, is added indicating that the software shall adhere to the MISRA coding guidelines [3] to help ensure best practice coding techniques are followed.

SOFTWARE ARCHITECTURE DESIGN PHASE

In this phase of software development, the goals of the software safety program include identifying the safety-critical software components and functions, and applying appropriate high-level analysis methods to these components and functions to help ensure potential hazards are avoided or mitigated. The software development team specifies the software components and functions that are needed to create a functional system that satisfy identified software requirements (including software safety requirements). From the existing software hazard analysis, an *integrity* or *criticality level* can be assessed for each software component or function. The criticality level depends on the potential hazards that could arise from a malfunction of the software component or function. The higher the criticality, the greater the level of analysis required. There are various schemes for quantifying criticality or integrity, with the simplest being to label software components or functions as either safety critical (if they can lead to a potential hazard) or non-safety critical (if they cannot lead to a potential hazard).

To satisfy these goals, the existing fault tree analysis is extended to identify the specific software components or functions that produce software states that may lead to potential hazards, and a system-level software Failure Modes and Effects Analysis (FMEA) is performed to provide broad coverage of potential failures. Software component or function criticality is assigned based on the highest risk potential hazard that is linked to potential software causes in the developed fault trees.

```
INIT:
        PowerUpTest();
MAIN_LOOP:
        DetermineSystemMode();
        AcquireSensorInput();
        DiagnoseSensorInput();
        ComputeOutput();
        Check&SendOutput();
BACKGROUND_LOOP:
        ECUDiagnostics();
SHUTDOWN:
        Shutdown();
```

Figure 4: Example Control System Software Architecture.

To help understand the analysis methods presented in this section, a software architecture for the example control system is shown in Figure 4. This software architecture must accommodate identified safety requirements (Table 4), and in some cases specific software modules need to be included (e.g., DiagnoseSensorInput()). The architecture includes an initialization task, which is run at power up, a main loop and a low priority background loop, both of which are run during normal execution (after the initialization task is complete), and a shutdown task that is executed

based on the results of the DetermineSystemMode() function.

FAULT TREE ANALYSIS

At this stage of development, the existing fault tree is revised such that specific software modules are included in the fault tree. This typically involves replacing the existing software portion of the fault tree, which to this point has been developed based on knowledge of the necessary software function but not on the software structure, with a new software sub-tree based on a structured analysis of the software architecture. The newly developed software sub-tree is compared to the old sub-tree to be sure no knowledge is lost.

For the example control system, the software portion of the fault tree shown in Figure 3, is replaced with a tree developed by identifying the immediate causes of the quantified top software event. The tree is created by stepping through the software architecture shown in Figure 4 to identify relevant software failures of the software components. Event descriptions in the tree are quantified based on the requirements in Table 4. A portion of the revised tree is given in Figure 5

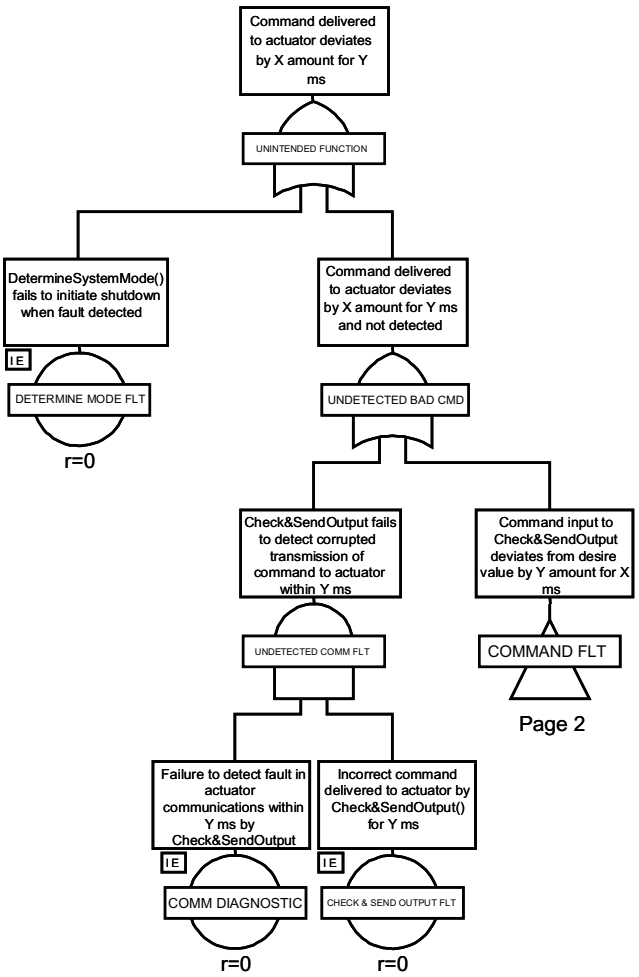

Figure 5: Revised Software System-Level Fault Tree.

Two immediate causes of delivery of a bad command to the actuator are identified:

1. Command delivered to actuator deviates by X amount for Y ms and is not detected, and

2. DetermineSystemMode() fails to initiate shutdown when fault detected.

The branch of the fault tree for the first cause includes potential failures of each of the software modules needed to produce and deliver the command to the actuator, and potential failures in the associated diagnostics intended to detect deviations in the desired command. Including the diagnostics in the tree results in the introduction of AND gates in this branch. The branch of the fault tree for the second cause includes the failure of the software module that initiates system shutdown if a fault is detected. This branch does not contain any AND gates because there are no identified diagnostics as of yet to detect this type of failure.

Potential failures of all of the software modules in the MAIN LOOP in Figure 4 appear in the fault tree, so all of these modules can be considered safety critical software modules. However, the only single point failure that may cause the top software event is failure of the DetermineSystemMode() module, so this module is assigned a higher criticality level than the others. During the detailed design phase, this software module will be analyzed in more detail due to its higher criticality.

SYSTEM LEVEL SOFTWARE FMEA

Software FMEA aids in identifying structural weaknesses in the software design and also helps reveal weak or missing requirements and latent software non-conformances. A software FMEA can be performed at different design phases to match the system design process. The goal of the software FMEA performed during the software safety architecture analysis is to examine the structure and basic protection design of the system. The PHA and the hazard testing results are key inputs to the system-level software FMEA. The FMEA techniques described in this paper are consistent with the recommendations of SAE ARP 5580 [6]. In contrast to SAE J-1739 [7], SAE ARP 5580 provides specific guidance for software FMEAs.

Analysis of the software components and functions assumes that a high-level design description of the software architecture is available. The analyst performing the software FMEA needs to have a complete understanding of the software design, the underlying hardware structure, interfaces between the software and hardware elements, the software language to be used and specifics of the software tools being used. If possible, the system development program should use compilers that are certified to a standard for the language to be used. Thus, early involvement in the software design FMEA will allow needed compiler and

language restrictions to be imposed on the design process at a cost-effective time [8].

System-level software FMEA uses inductive reasoning to examine the effect on the system of a software component or function failing to perform its intended behavior in a particular mode. Generic failure modes (guide words) are applied to the top-level software components and the impacts are analyzed. In an approach consistent with SAE ARP 5580 there are four failure modes for all components and two additional failure modes for interrupt service routines (ISRs). The four common failure modes are:

- Failure to execute,
- Executes incompletely,
- Executes with incorrect timing which includes incorrect activation and execution time (including endless loop), and
- Erroneous execution.

The two additional software failure modes for ISRs are:

- Failure to return, thus blocking lower priority interrupts from executing, and
- Returns incorrect priority.

The failure to return failure mode for an ISR also includes the condition where an ISR fails to complete, and thus goes into an endless loop.

System-level software FMEA is performed by assessing the effects of the relevant failure modes for each functional subroutine. The effects of the failure modes on the software outputs are analyzed to identify any potentially hazardous outcomes. If potentially hazardous software failure events are identified then either a software safety requirement was not provided or a safety requirement was not adequately translated into the software design. In these cases, a software safety requirement is added and the software design is modified to accommodate this change. In order to assess the changes made to the software, the system-level software FMEA is updated when changes are made.

For each component or function, failure mode guidewords are applied and the local and system-level impacts are analyzed, including assigning a severity. This is documented in a tabular form. Recommendations to improve the safety of the software design are documented and passed on to the software design team. Table 7 shows a portion of the system-level software FMEA documented in a tabular form for the example control system. Safety-related software requirements identified by the FMEA are added to the software safety requirements for the design. To maintain consistency between the FMEA and FTA, specific software failure modes and new diagnostics identified by the FMEA can be included in the system fault tree.

SOFTWARE DETAILED DESIGN AND CODING PHASE

In this phase of software development, the goals of the software safety program include analyzing the detailed software design, and analyzing the implemented software to help ensure software safety requirements are satisfied. Subsystem interfaces may be analyzed to identify potential hazards related to subsystems. The analysis may check for potentially unsafe states that may be caused by I/O timing, out-of-sequence events, adverse environments, etc. Two methods that may be used to achieve the software safety program goals are detailed software FTA and detailed software FMEA.

These activities can be performed in a coordinated manner. The software hazard analysis that was performed at a high-level using FTA during the requirements and architecture phases, can be further extended to decompose the identified potential hazards into software variables and states. A detailed FMEA can be applied to all or higher risk software modules by tracing potential failures in input variables and processing logic through the software to determine the effect of the failure. These effects are then compared against those that cause each of the potential hazards to occur to determine if the individual potential failure can lead to a potential hazard. Potential failures, which can lead to one of the potential hazards, are identified along with appropriate software design corrective actions.

Typically the level of analysis performed depends on the criticality level (or potential risk) of individual software modules, and the product design's overall stage of development (e.g., prototype vs. production). In the following sections, the process for performing a complete FTA and FMEA at the detailed design and code level is described. In cases where less analysis is required, a subset of the methods described in this paper can be applied.

```
DetermineSystemMode(Boolean Flag1, Boolean Flag2)
        {
        Enumerated SystemState = (NORMAL,
                        FAILED);

        SystemState = LookUpState(Flag1, Flag2);
        If (SystemState == FAILED) Then
                CallShutDownTask();
        }
```

Figure 6: Example Code for Determining System Mode.

To help understand the analysis methods presented in this section, a hypothetical coded procedure for the example control system is provided in Figure 6. This software code for the DetermineSystemMode() procedure looks up a system state based on diagnostic flags that have been set by other routines. If a critical

failure has occurred, the system transitions to a safe state (in our example system, the controller shuts down).

DETAILED SOFTWARE FAULT TREE ANALYSIS

From the software architecture phase, the existing fault tree links top-level software components and functions to the potential hazards. With the software detailed design and code now available, the fault tree can be extended to identify lower-level software components that directly assign the output of the top-level components already in the fault tree. These lower-level software components can be tagged as safety critical and any additional software hazard avoidance requirements that are needed can be specified. As the results from the detailed software FMEA technique become available, the FTA and FMEA results can be compared for consistency and completeness.

DETAILED SOFTWARE FMEA TECHNIQUE

Detailed software FMEA is a systematic examination of the real time software of a product to determine the effects of the potential failures in the individual variables implemented in the software. The detailed FMEA allows a more thorough assessment of the degree to which the system design remains vulnerable to potential individual failures, such as single point memory failures. In addition, a detailed FMEA may be used to assess the protection provided by the diagnostic approach to potential dormant software non-conformances. This detailed analysis is time consuming, and is typically only applied to high criticality software components. For distributed systems with redundant controllers, the need for detailed software FMEA is reduced, because by design, potential software failures due to hardware faults typically do not lead to potential hazards.

Table 5: Example Variable Mapping.

Variable	Routines				
	Acquire-Sensor-Input	Diagnose Sensor-Input	Compute-Output	Check & Send Output	Deter. System Mode
Variable-1	Output	Input			
Varaible-2	Local				
Variable-3			Output	Input	
…	…	…	…	…	…
Variable-n				Output	Input

To support the FMEA, a variable mapping is developed to map all the input, output, local, and global variables of the software to their corresponding software routines. Thus each variable, which is either an input or an output, has a mapping. Each input variable is either a hardware input or is an output of another routine. Table 5 shows a variable mapping for a portion of the example control system.

Once the mapping is in place, failure modes are developed both for the variables used and for the software processing logic. The variable failure modes for input variables and the failure effects for output variables are based on the variable type.

Variable Failure Modes

Three basic variable types are recognized: Analog, Enumerated, and Boolean. An analog type variable is any variable that measures quantity in a continuous manner. Enumerated variables are those, which can have a limited number of discrete values, each with a unique meaning. All variables with only two possible values are treated as Boolean variables. Variables are stored in memory locations, and if the memory locations, buses, and data registers do not contain data integrity protection (e.g. parity), any variable may be corrupted during operation. Thus, the potential failure modes for each variable type, shown in Table 6 below, must be considered as possible input failure modes to every routine that uses the variable. The following list contains an example of potential variable failure modes for a portion of a software routine "DetermineSystemMode()" shown in Figure 6:

- Flag1 set to TRUE when it should be FALSE,
- Flag1 set to FALSE when it should be TRUE,
- Flag2 set to TRUE when it should be FALSE,
- Flag2 set to FALSE when it should be TRUE,
- SystemState set to FAILED when it should be NORMAL, and
- SystemState set to NORMAL when it should be FAILED.

Table 6: Failure Modes for Different Variable Type.

Variable Type	Failure Modes
Analog	High
	Low
Boolean	True when False
	False when True
Enumerated Example Values: A, B, C	A when it should be B
	A when it should be C
	B when it should be C
	B when it should be A
	C when it should be A
	C when it should be B

Software Processing Logic Failure Modes

In addition to potential variable failure modes, potential software processing logic failure modes may be considered. This type of analysis involves examining the operators (e.g., addition, subtraction, comparison) in the code to determine possible negative effects that must be addressed.

Integrating Results

Once an FMEA has been performed on each of the software modules, the output variables are used to provide a mapping between the modules. The failure effect on an output of one module is traced to the

corresponding input variable failure modes at the succeeding module. This variable failure mode/effect tracing is repeated until the top level routines are reached. To help support this activity, software threads that link software modules and variables from data acquisition to final output may be created. Once the set of effects of a failure have been traced to the top-level routines, the mapping of the failure to the hazards is determined.

The detailed software FMEA is analogous to the component level hardware FMEA process except that variables are substituted for signals and signal paths of the electronic hardware [8]. A portion of the detailed software FMEA is given in Table 8 for the example control system.

Finally, when the detailed FMEA is completed, a mapping will exist from the top-level potential hazards to the top-level critical variables. The top-level critical variables are those variables that are necessary and sufficient to enable a potentially hazardous software state. Figure 7 provides an example of a set of top-level critical variables identified by a detailed hazard analysis.

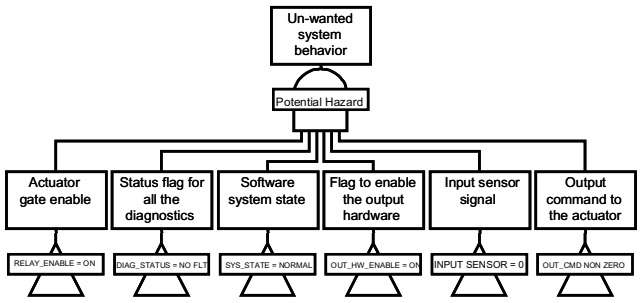

Figure 7: Example Fault Tree.

If the detailed FMEA identifies potential failure modes that trace to the identified hazards, then missing or incorrectly implemented software safety requirements are identified and corrected. Similar to the system-level FMEA, the software design deficiencies must be identified and the requirements documentation updated. The safety test plan document is updated with additional software safety testing requirements during the detailed design and coding phase.

DEFENSIVE PROGRAMMING

In addition to FTA and FMEA methods applied during this phase of software development, adopted or developed coding guidelines (e.g., [3]) often recommend that developers implement defensive programming techniques. Critical functions may be separated from non-critical functions in the code to reduce the likelihood that non-critical potential faults can lead to potential hazards. Using a logic “1” and “0” to denote states or decision results for safety-critical functions is not recommended due to bit-flip concerns. Software engineers should consider implementing reasonableness checks and sanity checks for critical signals.

SOFTWARE VERIFICATION AND VALIDATION PHASE

In this phase of software development, the goal of the software safety program is to execute safety test plans to help ensure that the software satisfies all software safety requirements. This typically involves performing unit testing and integration testing in any of the following environments: simulation, bench, and in-vehicle. The developed safety test plans demonstrate that fault detection and fault handling capabilities (e.g., see Table 4) are functioning as expected. In addition, software stress testing may be applied to help ensure the software is robust to changing inputs. Finally, compliance with any applicable government and international standards or relevant guidelines is assessed in this phase.

Although FTA and FMEA are primarily performed before the verification phase of product development, the detailed examination of requirements, design, and code they afford can be a significant help in verifying that the software satisfies specified requirements. FTA and FMEA results should be compared to those of actual testing during the verification phase to help ensure any assumptions or conclusions made during these analyses were correct.

SUMMARY AND DISCUSSION

In this paper, we have presented software safety methods and techniques that we have successfully applied to several advanced automotive systems. These methods and techniques satisfy the task requirements of a proposed Delphi software safety program procedure. A key component of this methodology is an integrated FTA/FMEA approach for investigating potential software causes of system hazards. The chief difference between the FMEA approach and the FTA approach is a matter of depth. Wherein the FMEA looks at all failures and their effects, the FTA is applied only to those effects that are potentially safety related and that are of the highest criticality [4]. The broad coverage provided by an inductive FMEA is combined with a deductive FTA to focus the analysis. Experience has shown that the FTA/FMEA approach has been effective in identifying and mitigating potential hazards. Initiating software safety activities at the beginning of the product development life cycle facilitates the implementation of identified corrective actions such that the impact on program timing and cost is minimized.

Since FTA and FMEA are static analysis techniques, they have certain limitations. Although they may focus

attention on identified safety-critical modules, they assume that the software provides desired behavior in the absence of potential failures. Thus, design or code reviews should be performed on safety-critical modules. Software FTA and FMEA do not verify the correctness or stability of control algorithms. For these evaluations, appropriate modeling and simulation tools need to be used to verify stability and correctness of control algorithms.

Table 7: Example System-Level Software FMEA.

Software Element	Failure Mode	Local Effect	System Effect	Potential Severity	Recommendation	Projected Severity
ACQUIRE SENSOR INPUT	Fails to execute	No sensor signals are read	System will continue to use the last read sensor signal value and output calculated based on that value. Since the last read signals are within range, DIAGNOSE INPUT function will not detect the fault. If system is in Normal Operation mode, this could potentially be hazardous if desired output is different from the system calculated output. If system is in startup mode, then default values will be used. Potentially there could be no output in that case.	10	A software execution monitor that checks the execution of this software element needs to be employed	8
ACQUIRE SENSOR INPUT	Erroneous Execution	Some or all sensor signals incorrect	DIAGNOSE INPUT function will catch any out--of-range signal values. However if the sensor signals values are within range, system will continue to use the erroneous sensor signal value and hence output will be incorrect. Potentially incorrect output command send to the output hardware leading to unwanted behavior of the system.	10	This could be caused due to either erroneous behavior of the A/D peripheral, sensor failure or any memory byte corruption. Need to have checks that monitor the ADC peripheral and the related controller memory cells.	8

Table 8: Example Detailed Software FMEA.

Variables	Failure Modes	Variable Type	Software Modules Affected	Local Effect	System Effect	Potential Severity	Recommendation	Projected Severity
Flag1	TRUE when should be FALSE	Input Global	Determine SystemMode	May cause SystemState to be FAILED when it should be NORMAL, resulting in unwanted call to initiate shutdown.	System will shutdown thus causing loss of function	8		8
	FALSE when should be TRUE	Input Global	Determine SystemMode	May cause SystemState to be NORMAL when it should be FAILED, resulting in no call to shutdown when there should be one	System may provide incorrect output	10	Replace Boolean flags with enumerated data type such that '00' is FALSE and '11' is TRUE; Diverse programming of diagnostic routines; Comprehensive fault injection testing to verify diagnostics	8

REFERENCES

1. Leveson, N.G., Safeware: System Safety And Computers, ISBN 0-201-11972-2, 1995.
2. IEC 61508-3, Functional Safety Of Electrical/Electronic Programmable Electronic Safety Related Systems – Part 3 Software Requirements First Edition, 1998-12.
3. MISRA Guidelines For the Use Of The C Language In Vehicle Based Software, April 1998.
4. FAA System Safety Handbook, Dec. 2000.
5. Czerny, B.J., et al., *An Adaptable Software Safety Process for Automotive Safety-Critical Systems*, SAE World Congress 2004.
6. SAE Aerospace Recommended Practice ARP-5580, Recommended Failure Modes and Effects Analysis (FMEA) for Non-Automobile Applications, SAE International, July 2001.
7. SAE J1739, Potential Failure Modes and Effects Analysis Reference Manual, SAE International, June 2000.
8. Goddard, P.L., "Software FMEA techniques," *Proceedings of the Annual R&M Symposium 2000.*

CONTACT

Padma Sundaram
Delphi Corporation
Innovation Center
12501 E. Grand River
Brighton, Michigan 48116-8326
Phone: 810-494-2453
Email: padma.sundaram@delphi.com