

Evaluation of Commercial Off The Shelf (COTS) Operating System (OS) Malfunction Mitigation Methods

Charles Forni, ; Alliant Techsystems; Plymouth Minnesota, USA
Bill Blake, ; Alliant Techsystems; Plymouth Minnesota, USA
Ralph Hall, ; Textron Systems; Wilmington Massachusetts, USA
Jeffrey Fornoff, ; US Army Armament, Research, Development and Engineering Center (ARDEC),
Picatinny Arsenal, NJ, USA
Larry Borshard, ; ARDEC, Picatinny Arsenal, NJ, USA
David Magidson, ; ARDEC, Picatinny Arsenal, NJ, USA

Keywords: Safety Design, COTS Software, Software Wrappers, Middleware, Operating System, OS

Abstract

The increasing demand for COTS OS use in safety critical applications has generated a lot of guidance in safety engineering process literature regarding the need for Middleware and Wrappers. Much of this discussion is focused on the need for middleware that can isolate the application from the horrors of the operating system. The assessment necessary to certify an OS, develop middleware for safety-critical applications, or to verify their integrity is a complex and costly task. For some applications, acceptable risk can be achieved by utilizing a software architecture that restricts safety-critical functionality to a subset of operations whose failure modes can be mitigated within the application, rather than in a middleware layer. This paper discusses this methodology, and provides analysis techniques to help assess the applicability of application-based mitigations when using a COTS OS in a safety-critical system. This methodology was used by the authors on a US Army remotely controlled weapon using Windows 2000 for the OS in the controller. The paper provides an approach for determining the cost benefit effectiveness of various mitigation approaches, an example of OS/application interaction partitioning, and clue lists to aid in assessing OS and application mitigations. The paper shows that Middleware for the type of systems discussed may not be needed, may not be cost benefit effective, and may actually be detrimental because it adds another layer in which malfunctions can occur. These fault conditions must also be mitigated safely and in a manner that does not reduce reliability (dependability).

Introduction

In order to determine if an OS is usable for a specific safety-critical application, an analysis must be performed to assess whether the combination of application and OS can meet the safety requirements of the system being implemented. To accomplish this task, an analysis of the interactions between the OS, the application software, and the hardware must be performed, and an assessment made of the impact of OS malfunctions. There are several characteristics of the OS that must be available when evaluating the safety impact of the OS. The OS behavior must be thoroughly specified and understood to allow an adequate qualitative analysis of the possible interactions between the OS and the rest of the system. The OS must also be capable of supporting the characteristics of the application (a real-time OS for a real-time app, for example). There are multiple methods for obtaining the information necessary to perform this assessment. Three of these methods are considered in the following paragraphs. The remainder of the paper describes a process for evaluating the use of application-based mitigations in a safety-critical system.

Using an OS with known safety-integrity: The first method is to utilize a safety-certified OS, or obtain a safety certification of the OS. In both of these cases, the OS is usually assessed using a “context-free” model that makes no assumptions about the application. Reference 2 provides information on various aspects of the assessment process using a context-free OS evaluation. The major advantage to using an OS, for which this type of assessment has been performed, is the increased level of analysis rigor, documentation of interfaces, and fault consequence information available. This allows a more detailed qualitative analysis, to be performed of the suitability of the OS for a specific safety-critical application. To assess an OS with a “context free” model requires a significant amount of work, and would likely not be cost effective unless the OS would be utilized to support many safety-critical applications.

Using an OS with middleware of known safety integrity: The second method utilizes a COTS OS with some form of middleware. Middleware is an accepted technique for implementing a buffer or filter between potentially

incompatible system functions that share data or provide a service, or to define and enforce a protocol between functions that are not necessarily designed to be compatible with each other when integrated into a system. The term middleware represents an interfacing layer (whether partial or complete) between the application and the OS. The term “Wrapper” is used to define an individual function that performs the interface processing between the software application and the OS. Thus “wrappers” are middleware. Individual “wrappers” within the middleware, can be used to protect OS components within a system from invalid or erroneous commands, without modification to those components. They can be used to functionally enhance a wrapped OS component, allowing it to meet the targeted system requirements. They can also be used to mask OS functionality that is not used. When using Middleware, it is still necessary to assess the suitability of the Middleware for use with the specific safety-critical application. As with a Context-Free” OS assessment, this approach can provide an increased level of analysis rigor, interface documentation, and fault consequence for the middleware/application analysis, if the data is provided or created during the generation of the middleware. The creation of a middleware facility, whether based on a “black box” assessment or on the supplier’s internal development process evidence, requires a significant effort. An extensive “what if “ analysis to assess all aspects of the OS function calls and possible side effects would be required to obtain a measure of the safety integrity of the middleware. In addition, subsequent evaluation of the middleware facility would require that the same analysis used for application/OS evaluation, would need to be performed to determine the suitability of the middleware to the safety-critical application. The middleware approach is not cost benefit effective unless there is adequate funding for the acquisition or development of the middleware, or specific requirements exist for its development or use.

Using an OS with all required mitigation within the Application: The third method utilizes application-based mitigation on an OS/application interface basis. It addresses possible fault conditions that could occur in the COTS OS or application. To perform this type of analysis, it is necessary to consider each individual interface between the application and the OS and determine the possible failure modes for that specific instance. In addition, possible OS failures that could have more global repercussions must be considered throughout all safety-critical portions of the application. From the perspective of the application, the OS forms a complete layer between the application software and the hardware, so a failure of the OS is similar in most cases to a hardware failure. As in a hardware analysis, it is possible to perform a “what if” analysis on black box functions and define required mitigations based on analysis results. In many cases, the level of mitigation will be more extensive due to the increased number of failure modes that must be addressed when the details of the Black Box function are not known. In all cases, however, hazardous conditions must be appropriately mitigated.

Figure 1 depicts a process flow that summarizes the decision points that must be considered when assessing the various OS / application mitigation implementation methods.

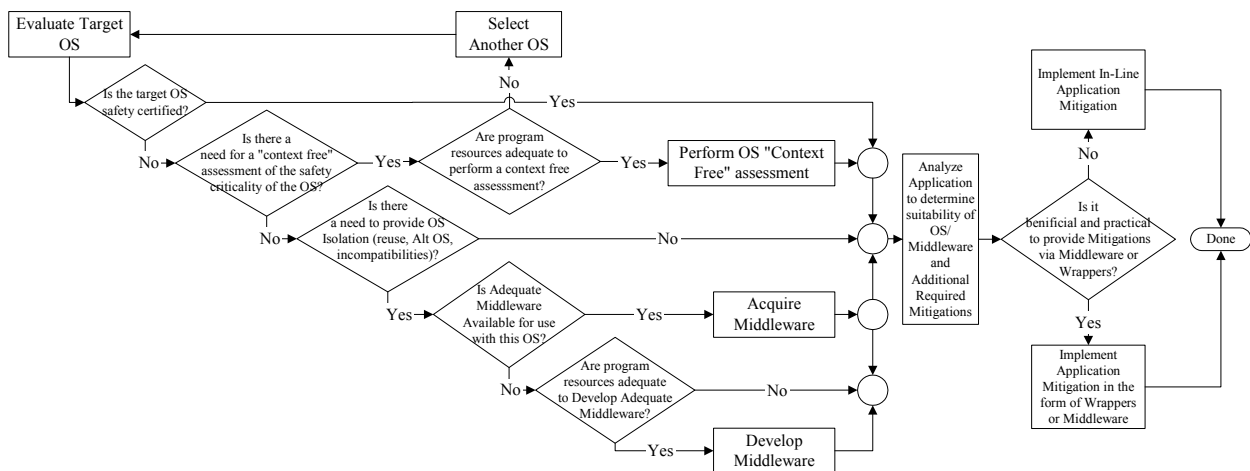


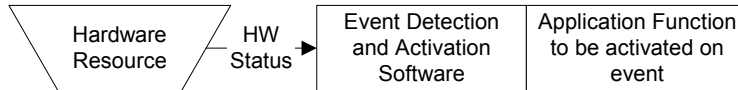
Figure 1 – Mitigation Option Selection Process flow

Evolution of OS Functionality and Hazard Causal Factors

To clarify the relationship between OS functionality and hazard causes, the following example scenario will be

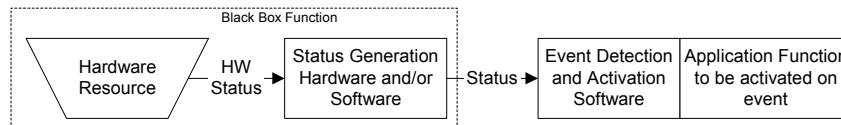
examined for 6 implementation cases. It is the intent of this analysis to assess the mitigation adequacy issue qualitatively. Addressing software reliability as a quantitative measure is currently highly speculative. The example base implementation case will be examined as it evolves through various forms of implementation, and will show how the basis of false activation changes with each form. The cases are described in the following paragraphs, with the results summarized in Table 1 at the end of this section.

Case 1 is the base implementation and consists of an application function that is to be performed when a specific event occurs in a hardware device (switch closing or key pressed). To allow the function to remain constant through the various cases, the application portion of the program has been divided into two parts. The first part is the application function (to remain constant for all cases). The second consists of the processing necessary to detect the event and to activate the application function. We will assume that a hazard could occur if the function is activated erroneously.

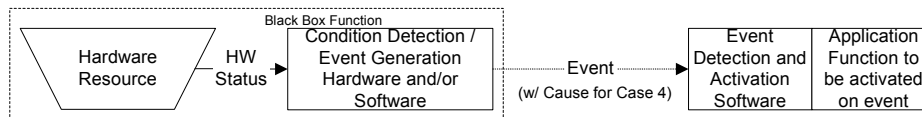


There are two erroneous activation situations that need to be examined. In the first situation, the application knows the function is not allowed. In the second, the function is allowed but the activation is erroneous. In the first situation, the application can mitigate the erroneous activation since it knows that function activation is not allowed. In the second situation, we must address the basis for the false activation to determine what mitigation will reduce the hazard to an acceptable risk level. In this case there is a contribution from both the hardware resource and the Event Detection and Activation Software. It should be noted that for the above example, the minimal basis for false activation is the probability that the hardware resource would erroneously generate the event. If the “Event Detection and Activation Software” was to pass the condition that generated the event to the application, the application could provide further mitigation and reduce the basis of false activation to that of the hardware resource alone. Whether this reduction is needed, is based of the adequacy of the mitigation available.

For Case 2, “Status Generation Hardware or Software” is included with the hardware resource. Together they make up a “black box” function that could represent an OS function called by the application. In this case the “black box” contribution to the basis for false activation consists of the contribution from the hardware resource, and the Status Generation Hardware or Software. As in Case 1, if the source of the status is the hardware resource, the contribution could be reduced to that of the hardware resource. One other point that differs from Case 1 is that the reliability of the Event Detection and Activation Software is lower than case 1 due to decreased functionality. In case 1, the processing necessary to interact with the hardware resource was in the Application. In this case that functionality resides in the “black box” function.



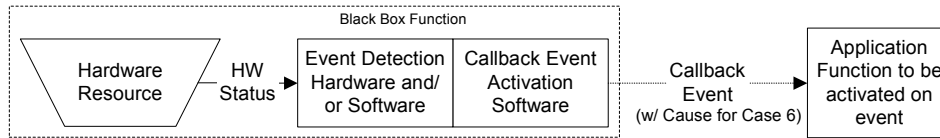
For Case 3, the detection of the initiating condition is included in the “black box” function and an event is generated to indicate the required condition has occurred. It is assumed that the Event detection and Activation portion of the application is simpler because it only has to respond to the event and invoke the application function. Since there is no status passed to the application, the probability of false activation for the “black box” function cannot be lowered.



Case 4 is similar to Case 3 with the following exception. The cause of the event is available to the application along with the event. In this case the probability of false activation for the “black box” function can be lowered to the probability of false activation at the point the event cause data is generated.

For Case 5, all the Application “Event Detection and Activation Software” functionality is moved into the “black box” function and is merged into the “Event Detection Hardware and Software” and “Callback Event Activation

Software” functions. An application configured in this fashion is referred to as a callback. A pointer to the application (referred to as a handle) is passed to the OS when the Application function is configured for execution on the specified condition. The thread monitoring the condition executes the function when the condition is met. As with case 3, the cause information is not available to reduce the probability of erroneous activation by the “black box” function.



Case 6 is similar to Case 5 with the following exception. The cause of the event is available to the application upon activation. In this case the probability of false activation for the “black box” function can be lowered to the probability of false activation at the point the event cause data is generated.

In Table 1, the possible mitigations section includes only cases where event detection occurs when an erroneous event occurs when the event is allowed. In all cases, application-based mitigation is needed to prevent erroneous activation in situations where function activation is not allowed (by state, mode, etc), so they are not listed in the table. The case where an event does not occur when required, is not addressed in the table, since the only possible mitigation is based on knowledge (from application, operator, or other source) the event was expected.

Table 1 –Event Processing (activation of an Application Function)

Case	Implementation	Basis of false activation	Possible Mitigation methods
1	Application directly obtains status from the hardware resource to determine if activation is necessary.	Probability of erroneous activation based on hardware failure rate and the “detection and activation software” failure rate.	Mitigation for possible “event detection and activation software” failures achieved by Application verification of the HW status in cases where function activation is allowed.
2	Application obtains status from a ‘black box’ function to determine if activation is necessary.	Probability of erroneous activation based on “black box” function failure rate and the “detection and activation software” failure rate. <i>Note: Pushing “event detection and activation software” functionality into the “black box” function may result in a complexity decrease for the “event detection and activation software”.</i>	Mitigation for “event detection and activation software” failures can be achieved by Application verification of the status in cases where function activation is allowed. Mitigation for a portion (Status Generation Hardware and/or Software”) of the possible “black box” function failures can be achieved by Application verification of the HW in cases where function activation is allowed.
3	A ‘black box’ function generates an event to notify application software that the Application Function should be performed.	Probability of erroneous activation based on “black box” function failure rate and the “event activation software” failure rate.	Mitigation for “event activation software” failures can be achieved by Application Function verification of the event in cases where function activation is allowed.
4	Same as case 3 with the following exception. The event contains the source of data resulting in the event. <i>Note: Maximum mitigation would use HW Status as the Cause.</i>	Probability of erroneous activation based on “black box” function failure rate and the “event activation software” failure rate.	Mitigation for “event activation software” failures can be achieved by Application Function verification as described for Case 2. Mitigation for a portion (Event Detection Hardware and/or Software”) of the possible “black box” function failures can be achieved by Application verification of the Cause in cases where function activation is allowed.
5	Application Function is activated by a ‘black box’ function.	Probability of erroneous activation based on “black box” function failure rate.	In cases where function activation is allowed, no Mitigation for “black-box” function failures can be achieved without information about the source of the callback event.
6	Application is activated by a ‘black box’ activation function. The Activation Cause is available to the Application Function for validation. <i>Note: Maximum mitigation would use HW Status as the Cause.</i>	Probability of erroneous activation is based on the “black box” function failure rate. The cause data allows the probability of erroneous detection to be reduced to that of the “Black Box” function at the point were the cause data is generated.	In cases where function activation is allowed, mitigation for a portion (“Event Detection Hardware and/or Software” and “Callback Event Activation”) of the possible “black box” function failures can be achieved by Application Function verification of the Cause.

An added benefit realized from this examination is to provide a context in which to assess mitigation implementation approaches. An application-based mitigation coded “in-line” at the interface to the safety-critical application function could mitigate all detectable fault conditions. If the “wrapper” exists at the interface of the “black-box” OS function, there may still be possible sources of false activation that might require mitigation in the application. In addition, the “wrapper” itself may become an additional source of erroneous activation. If reusability or isolation of the “black-box” functions is desired, then “wrappers” can be used as long as adequate mitigation can be maintained for the application, but additional mitigation may be required downstream from the “wrapper”.

Mitigation Assessment For COTS OS Usage -Process Development

This section describes a generalized process for determining if application based mitigation can adequately mitigate hazards in a safety-critical system. The development of this process was performed in three steps:

The first step identified interfaces between the application and the OS, and generated a list of clues or questions to ask about the individual interface. The second step was a “what if” analysis on the various types of application / OS interface, to determine if application based mitigation could adequately handle the conditions in the associated clue lists, or if other mitigation was required. The third step generated process flow diagrams to document the results of the “what if” analysis. The interface categories used in the development of this process include both direct and indirect OS / Application interactions. OS functions that indirectly affect application operation must be addressed for all safety-critical software. Interface categories with direct OS interaction only need to be examined at the point of interaction (where the application experiences the effect of the OS malfunction).

Application / OS Interfaces: Figure 2 shows possible application / OS interfaces. The interfaces (IF1 – IF4), their possible usage, and clues used to perform mitigation analysis are described in their own subsections.

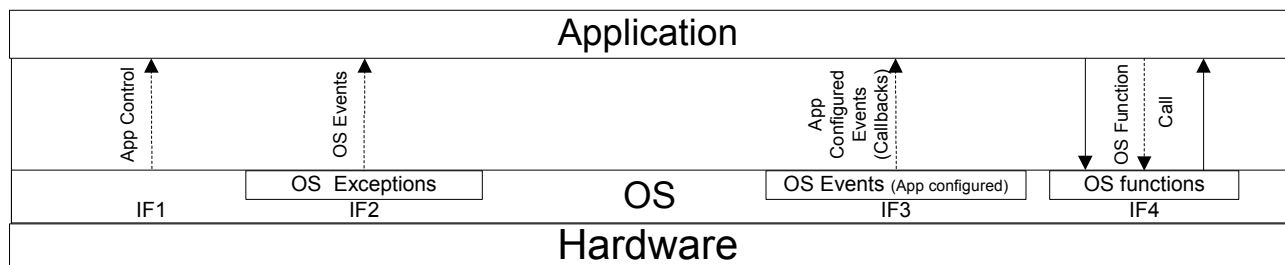


Figure 2 – OS/Application Interfaces

Application Control and Support (IF1): This interface consists of OS control of the application. It covers interactions that occur between the application and OS that are not detectable or reportable to the application. Of primary concern are failure modes that can affect the operation of the OS or application during application execution. Startup and termination must also be examined to identify potential failure modes that could lead to hazardous conditions. Activity at this interface level includes acquisition and allocation of resources to the application. Detectable failures or errors often result in an OS initiated application abort or terminate. Other faults or errors may cause the OS to fail in ways that could erroneously initiate execution or corrupt data within the OS or application software. Evaluation of these failure modes must be considered for all safety critical functions within the application. Possible “what if” clues for this interface are:

- What if the OS Aborts Application Load (insufficient resources)?
- What if the OS Aborts Application execution (insufficient resources, hardware and/or execution problems)?
- What if the OS provides insufficient Processing time (Higher priority activities, scheduler problem, etc)?
- What if the OS and/or Application memory (program or data) is corrupted?
- What if an application created thread is aborted?
- What if execution is performed at random locations in memory within the application?

OS Events (IF2): This interface consists of OS detectable faults that are related to OS provided functions and services. This category includes fault detection of shared resources (Keyboards, touch screen, display, serial IO, etc) that are controlled by the OS but used by the application. Failure of one of these services may not directly affect execution of the Application, but could indirectly result in a hazardous condition if a required piece of data or event is not generated. Ideally, failures of this type should be detectable through the called OS functions related to the service. In cases where an interface is supported by its own threads, a method must be created to notify the application or directly mitigate the problem if that failure could lead to a hazardous condition. An investigation of possible OS related exceptions should be performed to determine how these exceptions affect the application. Conditions that should be addressed include memory faults (read/write/parity/access), math coprocessor/numeric calculation faults, instruction faults, interface hardware faults, resource allocation faults (insufficient memory, stack

overruns, etc.), and OS operation faults. Possible “what if” clues for this interface are:

- What if a hardware fault related OS exception occurs (processor, interface, resource, memory access, etc)?
- What if the OS provided interface or an OS operation exception occurs?

Application Configured Events (IF3): This interface consists of OS generated events (callbacks) that are configured and controlled by the application. The thread of execution is considered to be a part of the OS. From the perspective of the callback, execution commences when the triggering event occurs. The possible conditions that are addressed when examining these events include events occurring late, early, too frequently, and not frequently enough. Possible “what if” clues for this interface are:

- What if the triggering event does not occur when it should?
- What if the triggering event occurs when it is not allowed?
- What if erroneous triggering of the event occurs when allowed?

OS Function Calls (IF4): This interface consists of calls to OS functions. Function invocation can be for the purpose of setting up data and events, initiating activity, waiting for conditions to be met, obtaining data, or any combination of the above. Analysis of functions for this category may vary. Possible “what if” clues for this interface are:

General clue

- What if the function does not return?

Clues for - function is used to setup data:

- What if the function returns without the data being properly set (exception or fault indication not provided, valid exception or fault indication provided, erroneous indication of fault status)?

Clues for - function is used to get data:

- Is the status of the acquisition function available on return?
- What if the function returns without the data being properly acquired (exception or fault indication not provided, valid exception or fault indication provided, erroneous indication of fault status)?
- What if the function returns properly with invalid data acquired (stale data, out of range, erroneous data)?

Clues for - function waits for conditions to be met:

- What if the function returns without the condition being met (exception or fault detected, erroneous condition met indication, condition met previously [stale data], timeout too early, timeout too late)?

Approaches to Mitigation Implementation: The focus of this paper is to describe a process for determining the mitigations necessary to address OS interface related hazard cases, thus, we must define the mitigation approaches that will be discussed. In the detailed analysis section, process flows will describe questions and flow paths that will result in one of three results. These paths are “No Mitigation Required”, “Implement App Mitigation”, or “Evaluate Alternate Mitigation”. The latter two results are discussed in the following subsections, so common content does not need to be repeatedly described in the analysis flowcharts.

Implement App Mitigation: The Implement App Mitigation box indicates that mitigation can be implemented within the thread associated with the OS function being evaluated or an application thread capable of mitigating the condition. The specific implementation may be either in-line, or in the form of an interface function (“wrapper”) to isolate the OS from the application. For the most part, the use of “wrappers” normally results in an increase in the probability of failure (due to extra code and function calls), and would not be recommended unless there is a specific desire for reusability of the wrapped OS function by another application, or to simplify replacement or upgrade of the OS. Figure 3 summarizes a process for evaluating and selecting an application mitigation implementation.

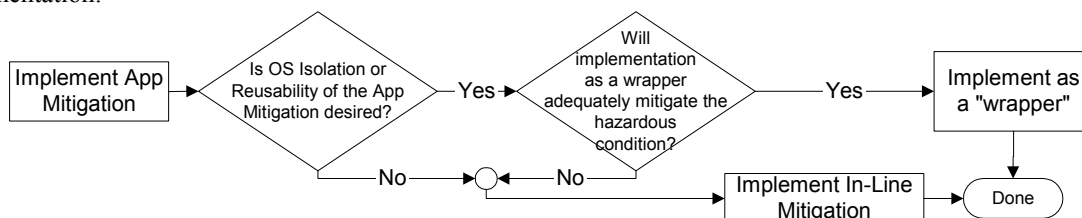


Figure 3 – Implement App Mitigation process flow

Evaluate Alternate Mitigation: The Evaluate Alternate Mitigation box indicates that mitigation cannot be implemented within the thread associated with the OS function being evaluated, or in an application thread capable of mitigating the condition. For conditions of this type, sufficient information is not usually available to properly mitigate the results of abnormal behavior by the OS function. In these situations, it may still be possible to use the OS, but the function would normally have to be replaced by a “wrapper” or corresponding middleware function. A “wrapper” in this context would consist of a replacement for the OS function in question and may or may not utilize other OS functions. A middleware function could also be used as the replacement for the OS function. In this context, the middleware would constitute a replacement for an OS service with the replacement function selected from the middleware service instead of the OS. Individual “wrapper” functions could also be included with replacement services and/or other “wrapper” function to form the middleware implementation. Should these techniques prove to be inadequate to resolve the hazard conditions, it may be necessary to use a different OS, to implement redundancy, use an independent safety kernel, or use external hardware mitigations. Figure 4 shows a process for evaluating the alternate mitigation implementations.

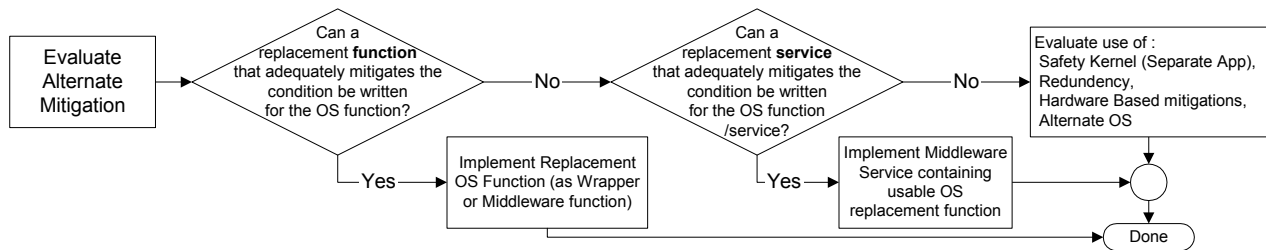


Figure 4 – Evaluate Alternate Mitigation analysis process flow

Analysis of Application to OS Interfaces:

Since the Application to OS interfaces can vary significantly with respect to analysis criteria, the OS assessment process has been partitioned into segments based on the type of interface function. The initial breakdown is based on the Application to OS Interface types described in the previous sections. An analysis of the interface type and a process tailored to the results of that analysis are provided in the following subsections. Figure 5 graphically depicts this partitioning.

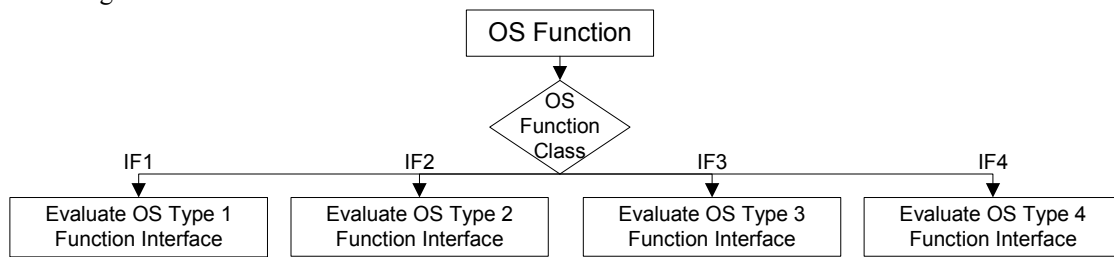


Figure 5 – OS function analysis process flow

OS Type 1 Function Interface Analysis and Evaluation Process: There are three operations that are typically performed when executing an application using an OS. The first operation is the power-up of the computer, the second operation is the start-up and initialization of the OS, and the third is preparing and initiating application execution. Similar steps are performed in reverse order when terminating operation of the application. Since the purpose of this paper is Application Mitigation, only a brief discussion will be provided for operations that occur prior to application start-up or following application shutdown. It should be noted that although these are very important considerations when evaluating the usability of an OS for use with safety-critical applications, application based mitigation is not active until application execution is initiated.

Pre-execution analysis: Hazardous conditions that could occur prior to execution of the Safety-Critical Application must be handled at the lowest level where adequate control of mitigations can be exercised. For the most part, these conditions consist of interfaces that are tied to the computer that is to run the application software. If a hardware output signal could cause a hazardous condition when not in the proper state, the output must be pulled to the safe

state and held there throughout the power-up of the computer, startup of the OS, and the loading of the application software. Conversely, the hazardous condition must remain mitigated through application termination, OS shutdown, and hardware power removal. Ideally, we would like mitigation to be performed at the lowest possible level in all cases, so a fault at any higher level could not create the hazardous condition. Since application based mitigations can't mitigate hazardous conditions that could occur prior to the start of execution of the application, they will not be addressed specifically in this paper, except to indicate they must be resolved using Alternate Mitigations.

Execution Analysis: Hazardous conditions could result from a variety of actions performed by the OS or any other program running on the target computer. The actions include erroneous execution, data or program corruption, and resource management faults (memory leaks, stack overflows, insufficient CPU time, etc), and could originate from software bugs, latent defects or hardware faults. Issues that can affect operation of the application and are not detected and handled by the OS are examined in this section. Those that are detectable and responded to by OS defined actions are examined in the section for OS Type 2 Function Interface Analysis. In order to simplify analysis of the possible fault conditions, the causes are grouped into three classes. The classes are Execution Integrity, Data integrity, and Resource Integrity.

Execution Integrity - Execution integrity includes situations where a software instruction fails, program execution occurs at an incorrect point in the program, or program execution is abruptly terminated. Some instruction failures are detectable and will result a program or OS exception and require Type 2 interface analysis. There are many instruction faults, however, that can't be detected and could result in erroneous operation of the application. The most familiar problem is aborted program execution. If a hazardous condition occurs due to this condition, no application-based mitigation will resolve the problem. A variation of this condition is the abort of an application thread. Application mitigations may be possible depending on the functionality of the aborted thread and its relationship to other threads. Another execution related problem is a jump to an incorrect address during execution. The effects of this problem are quite hard to determine due to the extremely large number of possible outcomes. The typical mitigation for this problem includes design features that require multiple actions to initiate the hazardous behavior, and software keys that can be checked prior to and during execution of safety-critical functions. Should function execution occur by any means other than the normal execution path, a check of the key value can prevent initiation of the hazardous behavior. Because erroneous execution can affect any part of the application, all safety-critical functions within the application must include mitigations for invalid execution. Figure 6 shows a generalized process flow for evaluating mitigations for hazardous conditions associated with Execution Integrity.

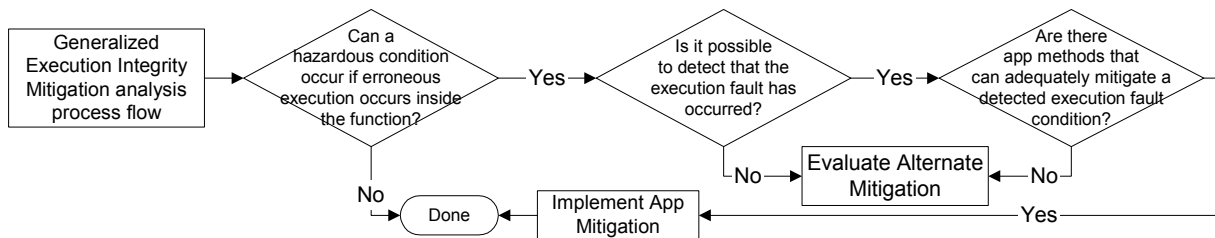


Figure 6 – Generalized Execution Integrity mitigation analysis process flow

Data integrity - Instruction related faults could also result in corruption of data anywhere within the application or OS. The fact that any particular data item may be corrupted imposes additional requirements on the analysis of the safety critical software. All safety-critical functions within the application must include mitigations for corrupted data. Figure 7 shows a general process flow to address data integrity during application execution.

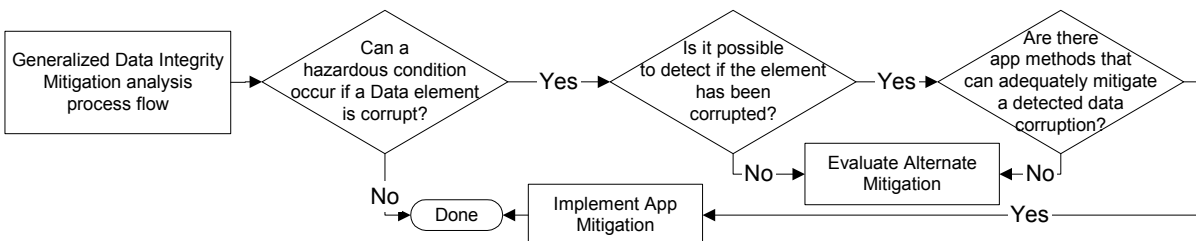


Figure 7 – Generalized Data Integrity mitigation analysis process flow

Resource Integrity - Resource management is one of the primary reasons for the use of an OS. In this type of implementation, the OS provides execution time to the various threads, memory resources, and controlled access to interfacing devices. Devices could include both timers and I/O devices. Access to controlled resources typically occurs through the other OS Interface Type functions. Assessment of mitigations for these “black box” functions is performed using the process defined for that OS Interface type. “Processing time” as a resource is not covered specifically for the “called” OS interface type functions, since these functions are assessed on a “black box” basis from the calling thread’s perspective. The use of multiple threads and various scheduling techniques may result in starvation of functions performed on certain threads. If a “processor time” starvation of any portion the Application can result in a hazardous condition, mitigations must be incorporated. The starvation condition could be caused by death of a thread, priority inversion conditions, or higher priority function “processing time” usage. Figure 8 shows a generalized process flow for evaluating mitigations for hazardous conditions associated with resource integrity.

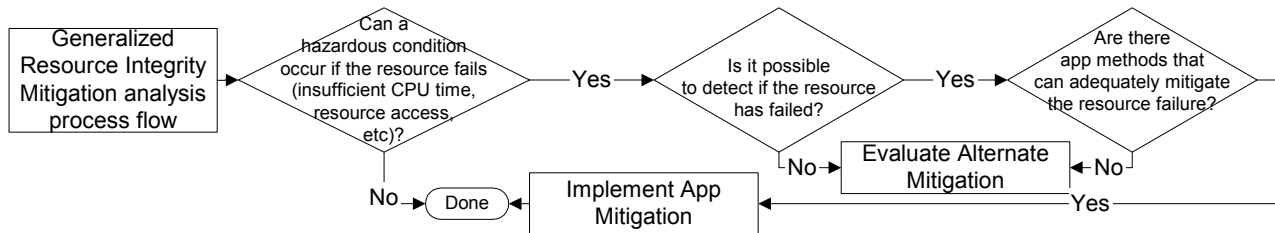


Figure 8 – Generalized Resource Integrity mitigation analysis process flow

Post-execution analysis: Hazardous conditions that could occur following execution of the Safety-Critical Application should be handled at the lowest level where adequate control of mitigations can be exercised. If implemented at any higher level, then mitigations for the hazardous condition must be available via the operation control at that level. For example: If the application terminates normally, and the application ensures the hazardous condition is mitigated before program execution is terminated, the OS must still ensure that it maintains adequate mitigations to prevent a hazardous condition from occurring while the OS is in control. If a failure of the processor, OS, or application can at any point leave the system in a hazardous condition without mitigation, the mitigation must be implemented outside the processor subsystem. Since Application based mitigations can’t mitigate hazardous conditions that occur after termination of the application, they are not be addressed specifically in this paper, except to indicate they must be resolved by alternate mitigations.

OS Type 2 Function Interface Analysis and Evaluation Process: This interface represents fault conditions detectable by the OS that could have an effect on the application. Some possibilities include non-recoverable errors in the computer hardware, the OS, or the application, or recoverable errors or operational status information that may affect application operation. Conditions that cause the application to abort cannot be addressed in terms of the application, and are evaluated under the Type 1 OS assessment process. From the applications standpoint, there are three ways to know if the condition has occurred. The first would be for the application to respond directly to an OS generated event. For this to occur the events would have to be predefined, and the application configured to handle them. These conditions are the basis of the Type 2 OS interface functions. The second way to know if the condition has occurred is by event notification via application configured events. The third way is via OS function calls. The last two cases are evaluated per the assessment processes defined for the Type 3 and 4 OS interface functions respectively. Figure 9 shows the process flow for evaluating mitigations for hazardous conditions identified for Type 2 OS interfaces.

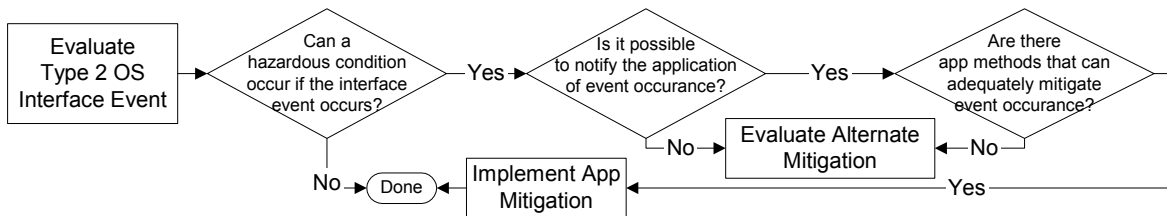


Figure 9 – Type 2 OS Interface analysis process flow

OS Type 3 Function Interface Analysis and Evaluation Process: The clue list for the Interface 3 type OS functions shows only two conditions of concern. The first condition (“What if the condition does not occur?”) cannot be mitigated by any means unless there is a secondary independent source of the required condition or an independent timeout condition capable of notifying the Callback or some other portion of the application of the fault. Callbacks are typically used for conditions where action may occur and usually do not have a terminating or secondary condition. The second condition is erroneous indication of the condition being met or is met when not allowed. If the condition causing the callback is not available, the probability of erroneous event cannot be reduced, and mitigation via knowledge available only to the application is required for mitigation. If the condition that triggered the event is available to the callback, mitigation can be implemented to verify the condition is valid. Since the data is generated closer to the source, the probability that the event is invalid can be reduced as described in the following scenario.

Assume a callback procedure is to be executed when keyboard key “A” is pressed. When the procedure is activated, it is not positively known if key “A” was pressed. It could have been that key “B” was pressed and the callback for key “A” executed or maybe no key at all was pressed and the callback just executed erroneously. If the activation reason is passed to the callback, the probability of erroneous activation is reduced since data is available which can mitigate error conditions between the point where the data is acquired and activation of the callback occurs. It should be noted that if the hardware actually generates an “A” key press condition when the “B” key is pressed, the function will be activated erroneously, and the condition cannot be mitigated without operator intervention.

In the above scenario, application mitigation is needed to prevent erroneous activation in situations where function activation is not allowed. This information is available only to the application and is not available to a middleware, or layered “wrapper” functions, unless this information is passed to them. Using application related information in the middleware would not be good design practice and would cause an increase in the number of failure modes with no mitigation benefit. An independent middleware or “wrapper” layer would make sense only if the OS is to be used with multiple applications. The level of mitigation attainable in a wrapper would in most instances be lower than that possible by performing the mitigation within the application itself. Figure 10 defines the assessment process for Type 3 OS Interface Functions.

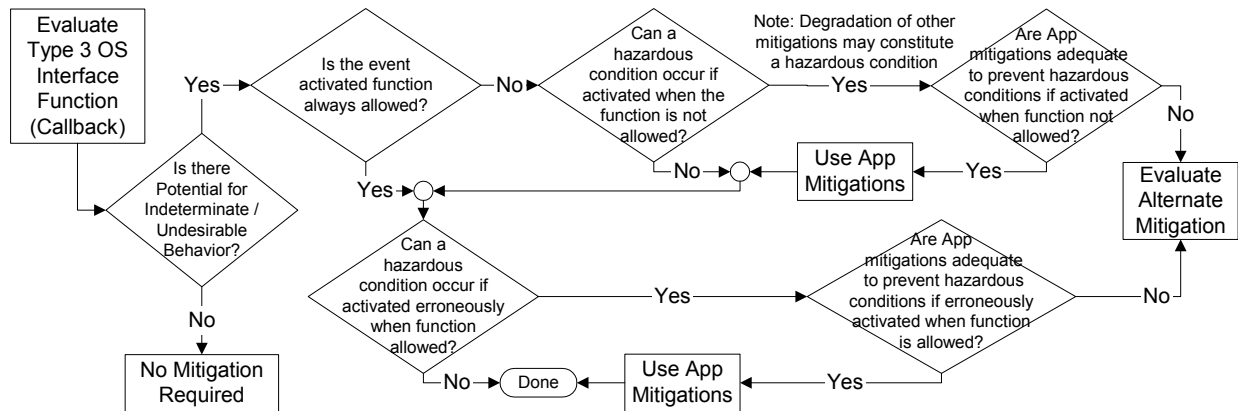


Figure 10 – Type 3 OS function analysis process flow

OS Type 4 Function Interface Analysis and Evaluation Process: This interface type is used to set up data and events, initiate activity, wait for conditions to be met, obtain data, or any combination of the above. Each function is slightly different, and the clues for each form differ. A single function may exhibit properties from one or more of the forms, so the analysis for any function in this category should be repeated with the applicable clues for each form that applies. The clue lists for these functions contain one clue that is common for all forms of called function (What if the function does not return?). This situation could occur if the calling thread is blocked indefinitely waiting for some condition to occur, running in an infinite loop, lost and executing somewhere it shouldn't be, or the thread aborted. The probability of function return failure varies significantly between the forms of the type 4 interface functions. The mitigation of resulting hazardous conditions must be external to the thread that makes the

call. No form of mitigation performed by the thread executing the non-returning function call can alleviate the lockup. This same rationale can be extended to any software function activity performed by a specific thread. If the calling thread fails for any reason, and a hazardous condition can occur, an alternate form of mitigation (external to the affected thread) must be used. There are ways to mitigate blocking function lockups due to failures of event generation processing, but the effects of these mitigations can only lower the probability of lockup due to loss of the event condition. These conditions are considered under the clue list discussion for blocking function calls. Figure 11 shows the top-level process for analyzing type 4 OS interface functions. Subsequent sections discuss the main process drivers for various forms of the type 4 function interface.

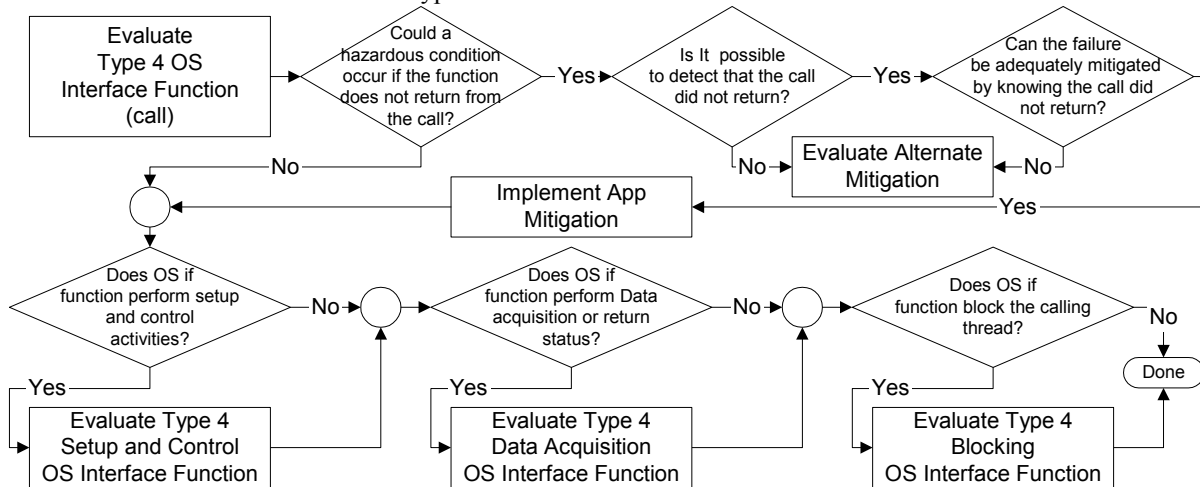


Figure 11 – Type 4 OS function analysis process flow

Figure 12 shows the process flow for type 4 functions that are used to setup data or control or to perform data acquisition. The assessment requires that each function be evaluated for hazards should the function return without the data or condition being properly set up, or invalid or incorrect data acquired. The extent of mitigation achievable is highly dependent on the availability of status information returned by the OS function.

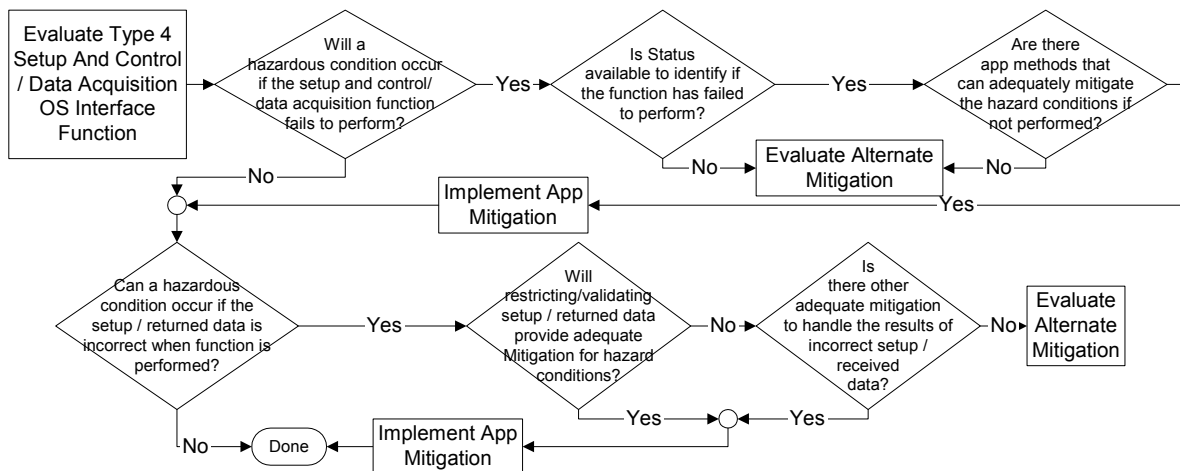


Figure 12 – Type 4 Setup and Control / Data Acquisition OS function analysis process flow

Figure 13 shows the process flow for type 4 functions that wait for conditions to be met. The primary evaluation in this process flow concerns invalid function return. Possible causes of invalid return include detected faults, erroneous condition met indication, condition previously met [stale data], return too early, or return too late.

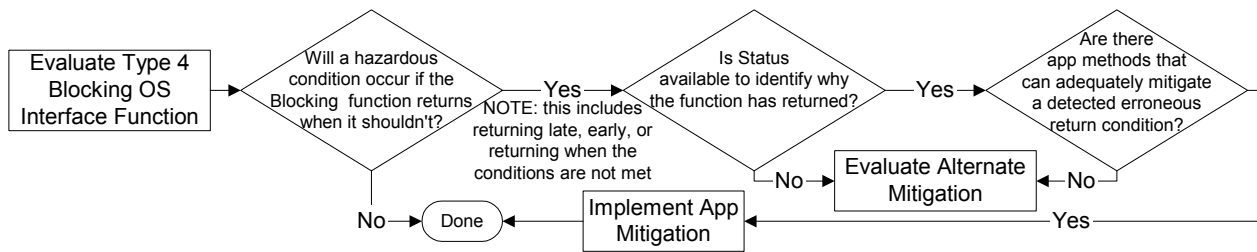


Figure 13 – Type 4 Blocking OS function analysis process flow

References

1. PROCEEDINGS of the 21st INTERNATIONAL SYSTEM SAFETY CONFERENCE – 2003, Safety Life Cycle Assessment of Integrated COTS Software, Gavin T. Watt; Raytheon Co.; Fullerton, Ca., Sally Cheung; Raytheon Co.; Fullerton, Ca.
2. PROCEEDINGS of the 21st INTERNATIONAL SYSTEM SAFETY CONFERENCE – 2003, Assessing Operating Systems for Use in Safety Related Systems, R. H. Pierce, MSc.; CSE International Ltd; Flixborough, UK, M. Nicholson, PhD.; Department of Computer Science, University of York, UK, A. G. Faulkner, MSc.; CSE International Ltd; Flixborough, UK
3. DOT/FAA/AR-02/118, Study of Commercial Off-The-Shelf (COTS) Real-Time Operating Systems (RTOS) in Aviation Applications, Office of Aviation Research, Washington, D.C. 20591

Biography

Charles Forni, Software Engineering, Alliant Techsystems, 4700 Nathan Lane N. Plymouth, MN 55442-2512, USA, telephone – (763) 744-5182, facsimile – (773) 744-5822, e-mail – charlesa.forni@atk.com.

Chuck is a Senior Principal Software Engineer with over 27 years of experience in military system software, in both software development and software safety. Chuck is currently the Joint Venture System Software Safety Engineer for the US Army Spider Program.

Bill Blake, Safety Engineering, Alliant Techsystems, 4700 Nathan Lane N. Plymouth, MN 55442-2512, USA, telephone – (763) 744-5086, facsimile – (773) 744-5822, e-mail – bill.blake@atk.com.

Bill is a Senior Principal System Safety Engineer with over 30 years of experience in military system safety engineering and safety program management. His experience is in both hardware and software systems. Bill is currently the Joint Venture System Safety Engineer for the US Army Spider Program.

Ralph Hall, Intelligent Battle Field Systems Engineering, Textron Systems, 201 Lowell Street Wilmington, MA 01887-4113, USA, telephone – (978) 657-1649, facsimile – (978) 657-1020, e-mail – rhall@systems.textron.com.

Ralph is a Software Safety Engineer with over 20 years of experience in military system quality assurance and software development. Ralph is currently a System Software Safety Engineer for the US Army Spider Program.

Larry D. Borshard, Software Quality and Safety Engineering, US Army Armament, Research, Development and Engineering Center (ARDEC), Picatinny Arsenal, NJ 07806-5000, USA, telephone – (973) 724-7847, facsimile – (973) 724-4977, e-mail – borshard@pica.army.mil.

Larry is a Software Quality and Safety Engineer with over 20 years of experience in Army weapon system software support and quality engineering. He currently provides software quality support to several tactical soldier systems, and represents the Army Fuze Safety and Review Board as the Software System Safety Advisor.

David L. Magidson, Software Quality and Safety Engineering, ARDEC, Picatinny Arsenal, NJ 07806-5000, USA, telephone – (973) 724-6958, facsimile – (973) 724-4977, e-mail – magidson@pica.army.mil.

David is a Software Quality and Safety Engineer with over 5 years experience in software quality engineering with

an extensive background in information systems management. He is currently the government software technical lead for the US Army Spider Program.

Jeffrey M. Fornoff, Army Fuze Management Office, ARDEC, Picatinny Arsenal, NJ 07806-5000, USA, telephone – (973) 724-3014, facsimile – (973) 724-4977, e-mail – fornoff@pica.army.mil.

Jeff has over 25 years experience in computer systems design, development and management. He currently performs software intensive systems analysis on military fuze applications, and assists the Army Fuze Safety Review Board as a Software System Safety Advisor.