A Software Reliability Methodology Using Software Sneak Analysis, SW FMEA and the Integrated System Analysis Approach

John H. Craig, Vertex Technologies, Inc, Houston

Key Words: Software Reliability, Software Sneak Analysis, Software Failure Mode and Effects Analysis, Network Trees, Program Operation Diagram, Integrated System Analysis, Software Analysis

SUMMARY AND CONCLUSIONS

A design analysis on system software can be very beneficial towards obtaining a highly reliable system. For a system to be reliable, it is important to know how the system operates with and without failures to create compensating provisions that could increase overall reliability. Software does not "fail" like hardware where there is an object that can be examined, analyzed and improved upon. Software failures are abstract and consist of logic errors or program paths not intended by the system designers at a specific moment of time. Software based systems have been known to produce these anomalous, unexpected outputs at undesirable times not due to any hardware failures. These unexpected anomalies can be classified as Software Sneak Conditions. To provide full coverage in the analyses approach, a Software Failure Modes and Effects Analysis (SW FMEA) is also performed to examine system effects if functions of the software did not operate as intended because of a software failure.

This paper will introduce the Integrated System Analysis (ISA) method of performing a Software Sneak Analysis. The ISA process and tools that capture and model the software functionally and are then used to perform the SW FMEA. The approach discussed using the ISA tools, Software Sneak Analysis and a SW FMEA have been performed and the results of one analysis is presented.

1.0 INTRODUCTION

Our approach allows for an in-depth analysis of the system before, during and after its' manufacture. The baseline information available are the manufacturing data for the system. The data items that are generally available are schematics, parts-lists, assembly drawings etc., for the electronics hardware and "text-file" code listings for the software. In all of this list of available information for manufacturing the design, none of the it assists in understanding the "as-written" software executing within the "as-built" hardware. To uncover any anomalies possibly written into the code that may degrade reliability, knowledge of every function that it performs, whether intended or not intended, is necessary. How the code operates without failures is covered by the Software Sneak Analysis. How the software operates in situations where hardware failures produce unavailable input data or events are covered by the SW FMEA. These two analyses can be synergistically combined using the ISA methodology and its' associated graphical diagrams discussed in the paper.

2.0 SOFTWARE SNEAK ANALYSIS AND THE ISA METHODOLOGY

Software Sneak Analysis has been used in the reliability, safety and design verification areas of engineering. The intent of a Software Sneak Analysis is to uncover any unexpected conditions produced by the software. Software Sneak Analysis can detect a wide-range of software anomalies even though the method for performing the analysis is not standardized. The ISA methodology can be this standard for Software Sneak Analysis and any other analysis where the functions of the system must be examined. Another important note, ISA does allow seamless integration with system hardware so integrated analyses are possible without developing different tools and approach.

2.*1 The History of Software Sneak Analysis*

Software Sneak Analysis was developed in the early 1980s by the Boeing Company hardware Sneak Analysis group to investigate software logic problems in systems that contained simple software controls. Hardware Sneak Analysis was initially developed to discover any unintended functions that a system may produce, and/or whether the function is inhibited when expected or executed when not expected. The Boeing Sneak Analysis group initially analyzed hardware systems that contained logic functions created using relays and diodes, then later logic with discrete transistors. Sneak Conditions manifested themselves as unexpected current paths that caused the inadvertent behavior. Electronics technology advanced to logic gates and eventually to microprocessors ICs. To discover any Sneak Conditions in these and modern systems, Sneak Analysis had to be extended beyond just unintended current paths into the functions of embedded software. Thus, the basic approach of hardware Sneak Analysis using functional, topological diagrams in which clues are applied was extended to software code. Later, it was discovered that the functional representations of the system hardware and software used to perform a Sneak Analysis were useful for any analysis that required knowledge of how the system operates. Therefore, the Sneak Analysis tools, called Network Trees, Network Forests (for hardware and integrated analyses) and specifically for software, a Program Operation Diagram (POD) were combined into what is now called ISA or Integrated System Analysis.

0-7803-7717-6/03/$17.00 © 2003 IEEE

2.2 *The ISA Process and Tool Building*

The ISA tool-building process consists of two phases, partitioning the source code and Network Tree/POD construction.

2.2.1 *Partitioning the Source Code*

In partitioning, the software code is separated by function. Initially, partitions are placed at each module since it is assumed that each module performs a specific function. However, new partitions can be added during clue application once discovering that a module contains more than one function. All the executable software (minus the comments) is partitioned and assigned a Network Tree number that includes data files, data tables, declarations, library functions and "include" files. Once all the code is partitioned, the Network Trees and POD are constructed.

2.2.2 *Network Tree and POD Construction*

The next phase is to construct the tools used by ISA and Software Sneak Analysis which are Network Trees and a POD. These diagrams are usually created by a drawing (CAD) application on the PC of choice. A software plotting routine is being written to accelerate this process, however, for the analysis discussed in this paper, the Network Trees were drawn manually.

The software code for each partition is depicted functionally where order of execution flows top to bottom, conditionals are represented as branches, and loops depicted with lines showing the loop's starting and end point. If a function or macro is called by the Network Tree using a label or function call, a cross-reference to the calling Network Tree is placed next to the instruction. Most software languages use these basic elements for logic flow and special symbols can be created for syntax that does not conform to these properties. However, there has not been a language that cannot be represented in Network Tree flow symbols and all major high-level languages such as C, C++, Ada, and Pascal have been analyzed and depicted using Network Trees. Any assembly language can also be depicted once the mnemonics are classified as either straight line, conditional branching or a loop. Figure 1 shows an example of a Software Network Tree from a C language source file with the basic graphical elements for program flow depicted.

While the Network Trees are constructed, variable/label cross-reference information is collected from each module with the specialized language interpreter application. The application extracts all variable names, labels and whether the variable and label is being defined or referenced . This information is later concatenated into a global cross-reference report once all the Network Trees are constructed. The report lists all the variables/labels represented on an individual Network Tree referenced to all other Network Trees where the variable/label also appears, followed by a "D" or "R" suffix. The "D" or Defined means the variable is assigned a new value either by calculation or a direct reassignment. For a label, the "D" is the defining macro Network Tree number or module where the actual software exists. The "R" or Referenced for a variable indicates that the variable is used in other calculations or conditionals, but not reassigned a new value within that Network Tree. Referenced labels are the routines that call the label's code during execution. The cross-reference report generated from the variable/label information are valuable in Clue Application for tracing a variable through the software and following code execution created by branches or calls to different routines.

Occasionally, the cross-reference report is resorted within the Clue Application phase when encountering look-up and jump tables missed during partitioning. Furthermore, special care concerning variable visibility is necessary with object oriented languages like Ada and C++. At times, the raw variable cross-reference files generated by the specialized interpreter during partitioning are hand-edited in a text editor to assure proper cross-referencing. Figure 2 shows an example cross-reference report.

The POD is last diagram constructed. It is similar to a call-graph and shows the Network Tree hierarchy and the order in which Network Trees are called within the hierarchy. Each box on the POD represents a Network Tree (software module at the initial partitioning time) stacked in levels where the highest level is the first module called during normal run-time execution. Lines are used to connect the boxes that are called from the adjoining levels. The boxes are also arranged left to right by the order of when they are called within their level. Below each box are more boxes, connected with lines containing the Network Tree numbers of the modules that are called by the parent module, again arranged in calling sequence left to right. The POD continues in this pyramid fashion until all software modules or Network Trees have a place in the hierarchy. Some modules like diagnostic routines, Interrupt Service Routines (ISRs), include or data files and initialization code do not get called from the run-time executive but are executed on-demand or at power-up. The POD represents these modules off to one side with a note describing how they are called and any hierarchy these routines may have with respect to each other. The POD assists in both clue application and management of the analysis, a topic that is covered in the Clue Application section of this paper. An example POD is shown in Figure 3

A Quality Assurance (QA) process is performed to verify that all the software code is represented and properly depicted on the Network Trees. The QA process is similar to a drawing standard's QA system and maintained consistency/accuracy among all the products.

3.0 SOFTWARE SNEAK ANALYSIS AND THE CLUE APPLICATION PHASE

Software Sneak Analysis does not rely on simulation, an intermediate language or input-output matrices that attempt to model the software. However, it uses the ISA tools that are actual software code depicted functionally as Network Trees, a set of guidelines to properly analyze the Network Trees, and a Clue List of special items to look for that may cause Sneak

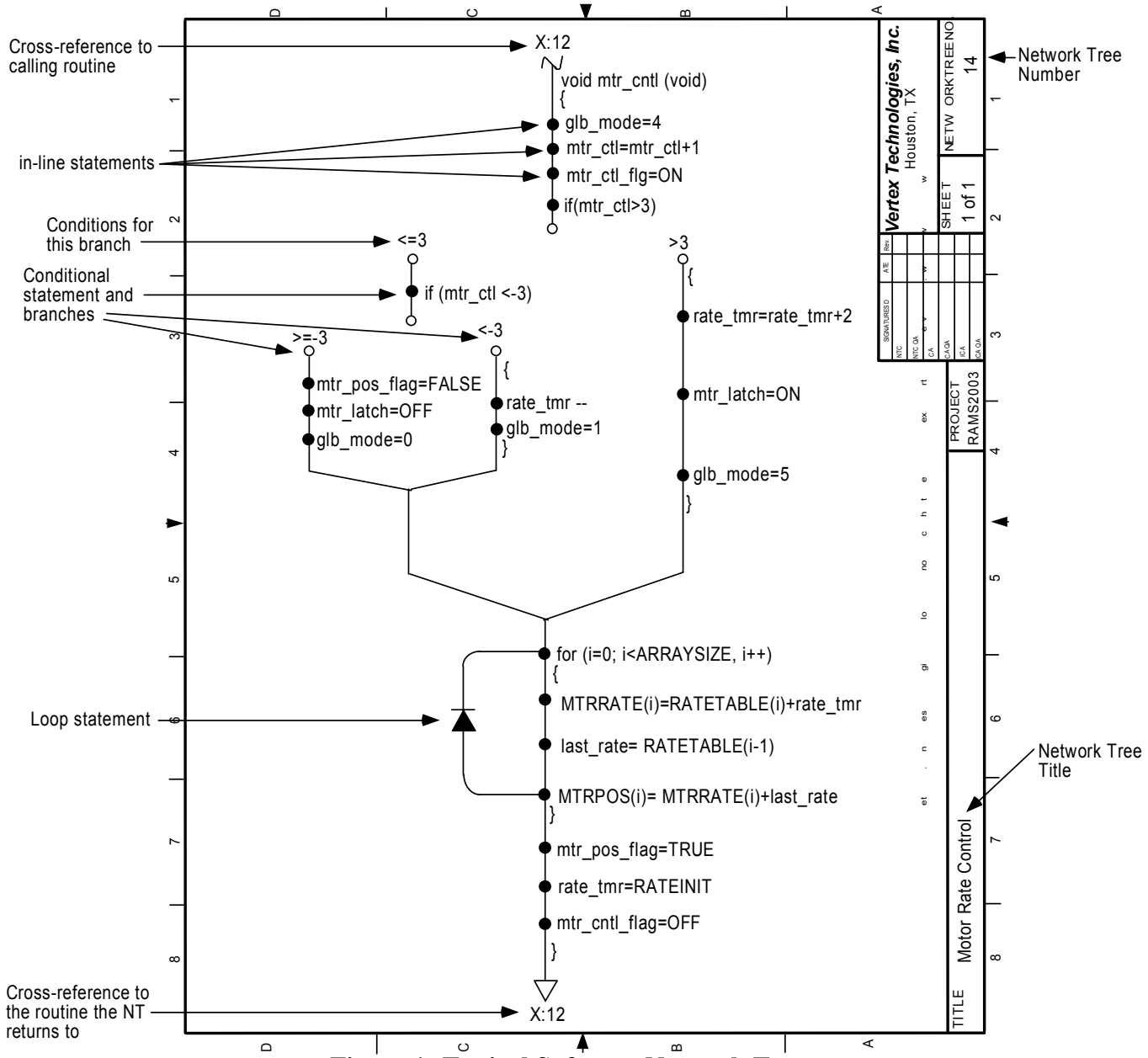

Figure 1: Typical Software Network Tree

Conditions.

All the Network Trees are required before clue application can begin and used as worksheets for the analysis. Each analyst will place notes on each Network Tree conforming to a set of annotation and analysis guidelines which assures that any possible issue concerning the Network Tree's data flow, function and purpose within the program's structure have been examined for Sneak Conditions. The POD in its' management role acts as a road-map for the analysis, allowing the lead of the analysis group to collect the modules or Network Trees by system function and assign an entire system function to an analyst. The analyst starts at the lowest level of the POD and analyzes each Network Tree, moving up a level until reaching the highest calling routine then to the right and up until all the Network Trees are complete. Other interactions between variables and labels on other Network Trees are also considered using the POD for execution sequence and the sorted variable/label cross-reference reports for recognition of where variables are being redefined or referenced. Any Sneak Conditions uncovered are reported with other issues logged on the Network Tree and a global issues list, shared by all the analysts. Many of these issues initially are questions on basic operations of the software and are resolved by the analyst as they continue analysis of their function. However, larger issues require interaction with different analysts and their respective function and are discussed/resolved between the analysts. If the issue

Mnemonic name	Module Name	NT#	Mnemonic Type L=Label V=Variable	Xrefs D=defined R=referenced
ARRAYSIZE	MTR_CNTL	14	V	10R, 10D,19D
FALSE	MTR_CNTL	14	V	4R,6R,7R,10R,18D
i	MTR_CNTL	14	V	6D,6R,11D,11R,12D,12R
last_rate	MTR_CNTL	14	V	10R, 12D,12R
mtr_ctl	MTR_CNTL	14	V	1R, 5R,6R,17R
mtr_cntl	MTR_CNTL	14	L	10R
mtr_latch	MTR_CNTL	14	V	8D,8R,10D,10R,11D,11R
mtr_pos_flag	MTR_CNTL	14	V	9D,9R,13D,13R,15D,15R,18D
MTRPOS	MTR_CNTL	14	V	5R,9D,9R,13D,13R,15D,15R,18D
MTRRATE	MTR_CNTL	14	V	3R,3D,6D,9D,9R,10D,10R,10D,18D
rate_tmr	MTR_CNTL	14	V	2D,2R,16D,16R
TRUE	MTR_CNTL	14	V	4R,6R,7R,10R,18D

Figure 2: Typical Variable Cross-Reference Report, Network Tree Sort Shown

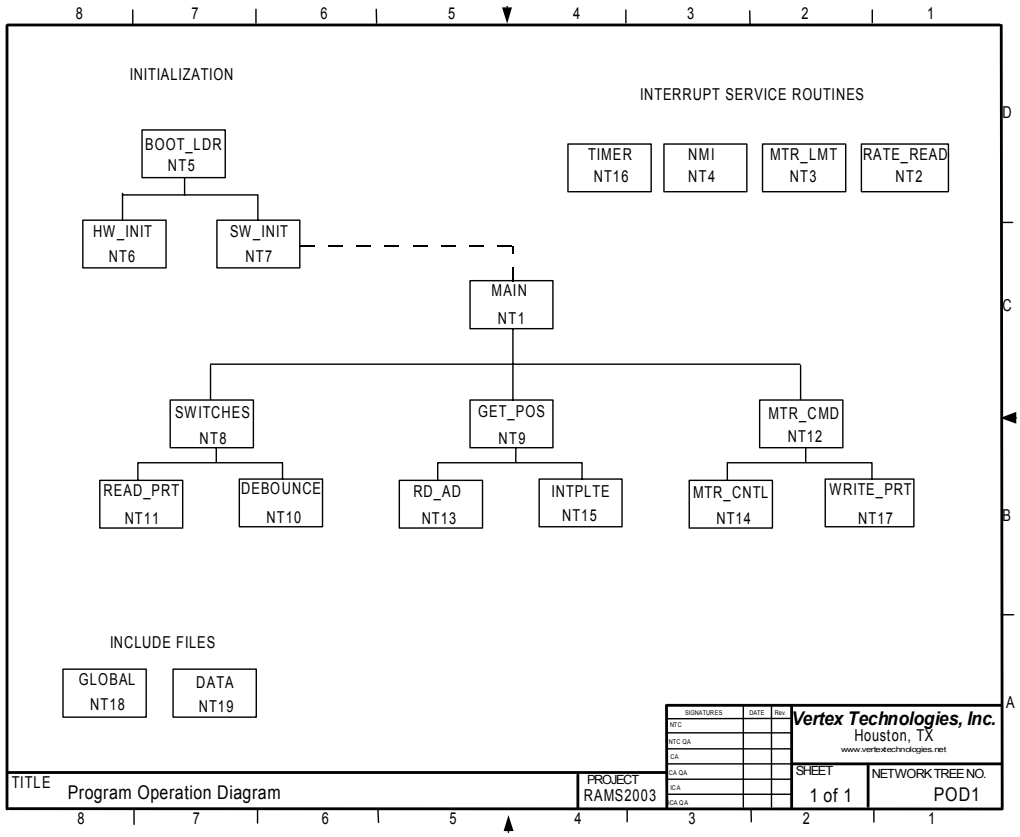

Figure 3: Program Operation Diagram

remains unresolved, it is added to the global issues list and resolved later in the Integration phase of Clue Application. The detailed Network Tree analysis is performed using the cross-reference report and annotation of the Network Trees using the guideline presented in section 3.2.

3.1 *Initial Cross-Reference Report Analysis*

The variable/label cross-reference report is used initially during Clue Application of a Network Tree to discover variables/labels that are defined but not referenced or referenced and never defined. Undefined labels can lead to code branches that are not executable. Undefined variables are occasionally hardware registers that are defined by a hardware event in which there is no issue. Other undefined labels or variables are real problems like dead code left within the source code for diagnostic purposes and not removed, or functions that were disabled and never tested. Review of this report also familiarizes the analyst with the Network Tree's function and interactions with the other Network Trees in the application.

3.2 *Network Tree Annotation Guidelines*

Each Network Tree has a starting point where it is called from a higher level routine and an ending point where it returns to the calling (or another) routine. Most Network Trees have straight-line statements, conditional branches and loop statements. Every one of these symbols requires a different set of questions to be asked with possibly further investigation. The analyst answers these questions as notes on the Network Trees to log their findings/steps. Other analysts, the QA, and lead engineer will also review these notes to assure a complete analysis. The following are the guidelines for note annotation of a software Network Tree:

1. Calling module and how the Network Tree gets executed
2. Basic operations of each straight line instruction (or group of instructions), how it affects the overall system possibly elsewhere in the system
3. How each variable is being changed on this tree, why, when, and how it effects other modules that reference this variable
4. Each branch path, the conditions on how the branch can be executed, and the effects of this branch's execution in the overall system
5. Loops, how the program gets into the loop and out of it. The overall effects on the system of the loop. Is there a way to get stuck in the loop?
6. Any critical timing issues with other software or hardware
7. Where the Network Tree returns after execution and any unexpected return points
8. Any anomalies discovered by reviewing the Project Clue List
9. Any reportable discrepancies with a detailed description
10. Basic functional description of the Network Tree
11. All hardware interfaces (if any), their effects on the logic flow and function

Generally, the analyst will use numbered flag notes or some style of pointer for this annotation. The notes can be typed on a separate layer electronically in the drawing application or hand written on a paper copy of the Network Tree for a small project with few analysts.

3.3 *The Software Sneak Analysis Clue List*

Step number eight in the Network Tree annotation guidelines requires the analyst to review the Project Clue List for any further anomalies. The Project Clue List is a compilation of two clue lists, the software general clues and specific clues formulated for this particular analysis. General clues apply to all software where the project specific clues apply to the particular software language, host processor or other issues with the design of the system. Examples of general clues are:

1 How does the language/compiler handle mixed-mode arithmetic? Will the truncated results of a mixed-mode expression effect other routines that reference the variable?

2. If an array is accessed via a pointer or directly, can the array's index attempt to access data outside the array boundaries? Can the pointer be zero or null?

3. If a hardware read of a discrete event is occurring on the Network Tree, is the input debounced? If not debounced, is it read again within the same pass somewhere else in the real-time loop?

Project specific clues are generally discovered within the language manuals, compiler/linker manuals, the host processor's hardware data sheets, etc and the system requirements. Many of these clues are based on software operations that control the hardware. These operations should be examined against the hardware data sheet of the microprocessor to assure that the hardware has the correct data bits and the proper timing or if there are instances where the hardware should not be accessed. Any warnings or special bit operations cited in the compiler/linker manuals or the processor's data sheet should also be added to the project clue list. These clues, especially ones describing proper operation of the hardware, are most likely to produce Sneak Conditions because hardware/software interface issues are the least looked at by the designers (from my experience).

Interface issues are generally resolved by the project's integration/test team and not by the designers who have detailed knowledge of the interfaces. Examples of some project specific clues are:

1. When using the C or C++ language, assure that hardware variables intended to be checked multiple times for an event within the same routine are declared "volatile"

2. When exiting an Interrupt Service Routine (ISR), have all the hardware flags, etc. required to reset the hardware for future interrupts properly set when exiting the ISR? Are there any exit paths around the reset of these flags?

3. Are all variables visible to each other in the arrays? Some arrays may not be visible to other software modules due

to the memory model that the software is compiled and linked under.

If the conditions described in these clues exist, it is an indication that an anomaly or issue may exist. These issues can eventually cause an unexpected event, or Sneak Condition to occur after investigating the effects of the anomaly on the entire application.

3.4 *The Clue Application Integration Phase*

The final step in clue application is the Integration phase that is performed after detailed analysis and annotation of all the Network Trees is complete. In this important step, the lead engineer and QA consider system anomalies uncovered in the detailed analysis to assure that they are not or lead to a Sneak Condition. All the issues within the global issues list are resolved with the analysts. The Integration phase usually uncovers the more elaborate Sneak Conditions that interact at a system level and require system knowledge to verify.

Again, a Quality Assurance (QA) process is performed to verify that all the software code was analyzed properly and that the notes were useful on the Network Trees. The QA also acts as an analyst to uncover issues across many Network Trees and provided a consistent examination of the software.

3.5 *Software Sneak Analysis General Findings*

Software Sneak Analysis has been shown to discover endless loops in software, undefined variables, array boundary violations via corrupted indexes, unused code, unreachable branches in the code, variable limit violations, software that does not function with the system hardware, interrupt collisions, timing issues with the hardware, mixed mode arithmetic errors, contradictory software functions, incorrect display and diagnostics messages, errors in measurement and computation, and requirements violations. Software Sneak Analysis can uncover anomalies not detectable by Static Dynamic or any other software code analyses. Though the static and dynamic analyses are automated and the anomalies are initially detected by a computer, these analyses only search for specific code and data patterns while some anomalies may lie outside these predetermined patterns. They also require an Intermediate Language or IL to be used to process the software. The IL translation process can hide or generate anomalous conditions that create nuisance reports. The ISA methodology and Software Sneak Analysis has the ability to go beyond these data and code pattern recognition schemes and detect functional anomalies. The ISA tools can also integrate system hardware for an analysis of the interfaces and external hardware components.

Some may object to the manual efforts required to perform ISA and a software Sneak Analysis, however, the automated tools require manual intervention also. For every anomaly report generated by the Static or Dynamic analysis application, an experienced analyst must disposition the detected issue possibly re-introducing human error. In my experience, many static code analysts that I have interviewed mentioned that 2-3% of the issues detected are real, where all the others ended up being anomalies created by code organization or programming style. ISA has added QA processes at major points to maintain consistency, accuracy and to limit the human error possibility in a manual analysis.

A major advantage of using ISA for a Software Sneak Analysis is that the annotated Network Trees are usable for any other functional analyses of the system. Any future software upgrades are easier to evaluate and changes to the hardware can be assessed with the software. This advantage leads us to our next topic, the SW FMEA.

4.0 SOFTWARE FMEA

A Software Failure Modes and Effects Analysis (SW FMEA) can discover many problems such as poor redundancy, catastrophic system effects due to malfunctioning code or erroneous inputs, insufficient fault detection and isolation, etc. The SW FMWA can also assist in devising compensating provisions for control computer failures.

The technique for performing a software FMEA is similar to the hardware FMEA. Each component is failed, as in the case of software either a specific variable or module, while an assessment of the local and global impacts to the system as it functions normally are considered. Each component (module or variable) is assessed one failure at a time and the impacts rated on a numerical severity scale. All the failures and associated effects are described and collected on SW FMEA worksheets, generally one worksheet per module or variable. The numerical severity classifications are tallied and catastrophic failures are considered for redesign or some other compensating provision to avert the issue.

To assess the impacts on the software, the annotated Network Trees from the Software Sneak Analysis are used to assist in understanding the software functions to the instruction level. Thus, the Software Sneak Analysis must first be performed. Two methods exist for performing a SW FMEA, the module failure method and variable failure method.

4.1 *Module Failure Method*

Failing the software on a module level provides the most coverage for the SW FMEA. Each module is failed, i.e., it is not processing inputs and outputs, and the impacts on the software are assessed using the Network Trees for system knowledge. Many module failures cause the software to not do nothing, such as failing the “main” routine in a C program or system timers. However, some of the specialized modules in the application that process I/O data and make decisions that control hardware are benefited by this analysis. The major shortcoming to the module failure method is that most failures produce binary results, either some function totally operates or not. Also, this method assumes the code designers created a modular design with one function per module.

4.2 *Variable Failure Method*

Performing a SW FMEA on the variable level provides the most detailed results on software failures provided that the right variables are chosen. Again, the insight and knowledge gained from the Software Sneak Analysis is necessary to

choose the variables and assess the impacts when failed. The failure modes of the variable are also a consideration. If the variable is a flag with two states, the failure modes are limited to either "1" or "0". However, if the variables that are failed are byte or word length, then the combination that causes the most destructive impacts is required.

The variable failure method can be tedious and limited if too many variables are chosen. Time could be wasted examining variables that could really only fail if a portion of memory where the variable is located fails to a specific bit pattern. Therefore, the best variables to fail are based on a software function that maps directly to the hardware. Variables that represent inputs measured from sensors or discrete inputs connected to a hardware events are the most useful because these variables can simulate a software failure if the associated hardware fails. After choosing the variables to fail, the annotated Network Trees from the Software Sneak Analysis accelerate the assessment of the failure's impacts.

5.0 RESULTS FROM A REAL ANALYSIS USING THE ISA TOOLS FOR A SOFTWARE SNEAK ANALYSIS AND A SW FMEA

This approach was performed on a micro-controller system used in an automotive application. The software application was written in C and contained approximately 6500 lines of executable code. The system hardware was analyzed using the ISA tools and integrated with the Software Network Trees. Integrated Fault Trees were also constructed using several top-level events considered critical to the designers.

5.1 *Software Sneak Analysis Findings*

The Software Sneak Analysis resulted in 11 Sneak Conditions, 4 of which were entirely software and the remaining 7, hardware and software. Out of all of these Sneak Conditions, 9 resulted in actual software changes to remove the Sneak Condition. Two of the Sneak conditions produced very undesirable results. One condition caused by improper coding within an ISR caused the processor to lock-up, reboot and erase any diagnostic codes that may have been set. All hardware outputs were uncontrolled during the reboot period. Another Sneak Condition occurred when a hardware switch was read directly into the software in two places, 4 mS apart within the same pass of the executive routine. As the switch is closed, the electrical signal from the switch contacts are bouncing that could cause the first read to detect a "1" and the second read 4 mS later to detect a "0", creating a contradictory software state. This state caused a branch of code to be executed within the improper software mode resulting in erasure all the diagnostic trouble codes detected from power-up.

Other findings are uncovered from the software Sneak Analysis are classified as Design Concerns. These issues are usually violations of good design practice or Sneak Conditions that could not be proven. The analysis resulted in 15 Design Concerns with 9 being software related. Six of these concerns resulted in software code changes. One of the major Design Concerns actually explained to the developers a problem they were having with code upgrades. Some of the developers were accessing the hardware through predefined data structures while others were directly writing a bit pattern to the hardware port. When using the structure, the bits were inverted from the direct port write which confused the developers. The analysis showed how the structure is inverted in memory from the hardware due to a MSB, LSB swap in the compiler. This assisted the developers in future upgrades.

5.2 *SW FMEA Findings*

The SW FMEA was performed on the module level because of time and budget constraints. The code received was not modular enough for any real benefits from a SW FMEA thus the results were limited. The SW FMEA did reveal that many functions were placed in the "main" routine which violates modular design practices and causes software maintenance issues. Some redundancy issues concerning a software mode control function were also addressed. The mode could be switched inadvertently with a single hardware failure. However, no code changes resulted from the SW FMEA

BIOGRAPHY

John H. Craig
Vertex Technologies, Inc.
Address: P.O. Box 591522
Houston TX, 77259
www.vertextechnologies.net
Email: jhcraig@vertextechnologies.net

John Craig has been Vice President of Vertex Technologies, Inc. since 1996. He has over 20 years experience performing many design analyses such as Integrated Sneak Analysis, FMEA, Integrated Fault Trees, Requirements Traceability and other custom techniques when in search of an observed system problem. Mr. Craig has broad experience in the development of both hardware and software for embedded micro-controllers and industrial control systems with two patents in this field. He also has extensive experience repairing electronic systems providing a unique insight into the "Design for Reliability" paradigm. He holds a Bachelor of Science degree in Electrical Engineering from the University of Houston