NUREG/IA-0254

International Agreement Report

Suitability of Fault Modes and Effects Analysis for Regulatory Assurance of Complex Logic in Digital Instrumentation and Control Systems

Prepared by:
Luis Betancourt,[2] Sushil Birla,[2] Jean Gassino,[1] Pascal Regnier[1]

[1] Institut de Radioprotection et de Sûreté Nucléaire, France
BP 17-92262 FONTENAY aux roses cedex—France

[2] U.S. Nuclear Regulatory Commission, USA
Washington, DC 20555-0001

L. Betancourt, NRC Project Manager

Office of Nuclear Regulatory Research
U.S. Nuclear Regulatory Commission
Washington, DC 20555-0001

Manuscript completed: April 2011

Prepared as part of the agreement on technical exchange and cooperation between the U.S. Nuclear Regulatory Commission and the Institut de Radioprotection et de Sûreté Nucléaire of France in the field of nuclear safety research

Published by
U.S. Nuclear Regulatory Commission

NUREG/IA-0254

International Agreement Report

Suitability of Fault Modes and Effects Analysis for Regulatory Assurance of Complex Logic in Digital Instrumentation and Control Systems

Prepared by:
Luis Betancourt,[2] Sushil Birla,[2] Jean Gassino,[1] Pascal Regnier[1]

[1] Institut de Radioprotection et de Sûreté Nucléaire, France
BP 17-92262 FONTENAY aux roses cedex—France

[2] U.S. Nuclear Regulatory Commission, USA
Washington, DC 20555-0001

L. Betancourt, NRC Project Manager

Office of Nuclear Regulatory Research
U.S. Nuclear Regulatory Commission
Washington, DC 20555-0001

Manuscript completed: April 2011

Prepared as part of the Agreement on technical exchange and cooperation between the U.S. Nuclear Regulatory Commission and the Institut de Radioprotection et de Sûreté Nucléaire of France in the field of nuclear safety research

Published by
U.S. Nuclear Regulatory Commission

ABSTRACT

The Institut de Radioprotection et de Sûreté Nucléaire (IRSN) and the U.S. Nuclear Regulatory Commission (NRC) jointly investigated and evaluated the suitability of applying fault modes and effects analysis (FMEA), as a technique for identifying faults attributable to Complex Logic in digital instrumentation and controls for safety functions in nuclear power plants. Complex Logic refers to logic in the form of software, or in the form of programmed hardware, for which it is not practicable to ensure the correctness of all behaviors through verification alone. Whereas the term, "failure modes and effects analysis" is used in the context of the overall DI&C system , the corresponding concept for software (and other forms of complex logic) in a DI&C system is "fault modes and effects analysis." When FMEA techniques, which have been used effectively for traditional hardware, are applied to Complex Logic, such extension does not yield a similar benefit to regulatory assurance, because of the fundamental differences in the nature of faults in traditional hardware versus Complex Logic. Whereas hardwired devices (such as electromechanical relays) have only a few predetermined fault modes, the potential fault space in Complex Logic is huge; yet the actual number of faults is an extremely small fraction of the potential fault space. Finding these faults through FMEA is akin to searching for a needle in a haystack. Through analysis and examples of several real-life catastrophes, this report shows that FMEA could not have helped in the discovery of the underlying faults. The report concludes that the contribution of FMEA to regulatory assurance of Complex Logic, especially software, in a nuclear power plant safety system is marginal. Further investigations, not in the scope of the current NRC-IRSN collaborative study, are needed to understand the appropriate roles and combination of FMEA and fault tree analysis and appropriate application constraints for reliable results from such analysis techniques.

This page was intentionally left blank

FOREWORD

In March 2010, the Institut de Radioprotection et de Sûreté Nucléaire (IRSN) and the U.S. Nuclear Regulatory Commission (NRC) started technical exchange activities and cooperation in the field of digital instrumentation and control (DI&C) research. In discussions, both parties identified an interest in sharing their understanding of DI&C system fault modes attributable to Complex Logic (such as logic in the form of software or in the form of programmed hardware, for which it is not practicable to assure correctness of all behaviors through verification alone) in DI&C systems for safety functions in nuclear power plants (NPPs).

This study contributes to a part of the NRC research activity, “Analytical Assessment of DI&C Systems” described in Section 3.1.5 of the fiscal year (FY) 2010 – FY 2014 NRC Digital Systems Research Plan [1]. The NRC research activity was formulated in response to the DI&C-relevant part of the Commission’s staff requirements memorandum (SRM)-M080605B [2] dated June 26, 2008, as follows:

> The staff should report the progress made with respect to identifying and analyzing digital I&C failure modes, and discuss the feasibility of applying failure mode analysis to quantification of risk associated with digital I&C.

SRM-M080605B was triggered by Advisory Committee on Reactor Safeguards (ACRS) concern [3] that DI&C system failures were not well understood and recommendation to emphasize the importance of the identification of failure modes. Later, in its letter report on the 576th meeting [4], the ACRS recommended that:

> Software Failure Modes and Effects Analysis (FMEA) methods should be investigated and evaluated to examine their suitability for identifying critical software failures that could impair reliable and predictable DI&C performance.

NRC is continuing its research to address SRM-M080605B and the recommendation from the ACRS’ 576th meeting; however, Section 3.2 of this study has identified significant difficulties.

These findings are consistent with the NRC’s regulatory review guidance that is given in DI&C Interim Staff Guidance No. 06 [5], Section D.9.4.2.1,1 “FMEA”; for software, it refers to Sections D.4.4.1.9 “Software Safety Plan,” D.4.4.2.1 “Safety Analysis,” D.6 “Defense-in-Depth & Diversity”- this guidance does not propose failure modes and effects analysis be applied to software.

IRSN-NRC researchers jointly investigated and evaluated the suitability of fault modes and effects analysis (FMEA) as a technique for identifying faults attributable to software and other realizations of Complex Logic for safety functions in NPPs. Whereas the term, “failure modes and effects analysis” is used in the context of the overall DI&C system , the corresponding concept for software (and other forms of complex logic) in a DI&C system is “fault modes and effects analysis.” Logic does not fail in the traditional sense of degradation of a hardware component, but the system could fail, due to a pre-existing logic fault, triggered by some combination of inputs and system-internal conditions.

The IRSN and NRC conducted this study as part of their bilateral agreement on technical exchange and cooperation in the area of DI&C safety systems. In general, the scope of the IRSN-NRC cooperation is limited to exchanging information helpful in developing the technical basis, but excludes development or discussion of regulatory guidance. In this study, the scope of the investigation is limited to the role of FMEA in regulatory assurance of Complex Logic in DI&C safety systems and excludes the role of FMEA in the development process.

The study first characterizes the differences between traditional hardwired systems and current complex logic-intensive systems and the technological trends that drive these differences. Then, it discusses the issues and limitations of extending FMEA, as used in traditional hardwired devices, to current complex logic-intensive systems.

The contribution of IRSN researchers is based on a quarter century of experience with digital safety systems, spanning three generational changes in technologies. The NRC staff contributed relevant information gleaned from publicly available research publications and interviews with several of the authors. The IRSN and NRC staff collaborated in refining the analysis of their findings.

This study has also revealed areas for further investigation, but the scope of further collaborative investigation has not yet been determined. In the meantime, the NRC intends to contact experts reporting benefits from using FMEA to gain a deeper understanding of their experience relevant to regulatory assurance.

Separate from the joint investigation with IRSN, in response to the concerns mentioned above, the NRC had been investigating different ways of characterizing and classifying faults or defects in software and their potential utility in regulatory assurance of nuclear power plant safety systems. When that investigation is completed, the NRC will report its results separately. The IRSN-NRC joint study serves as an interim response to address these concerns. This report marks the successful launch of research collaboration between the NRC and IRSN. Both sides expect to continue cooperative research into the assurance of digital safety systems. to address research questions identified, but not answered in this study:

- Should the assessor accept "inadequately specified verification cases" to be "normal" and overcome these weaknesses through redundant techniques? Or,
- Should the focus be on finding and fixing underlying systemic weaknesses in the upstream review criteria?

Apart from the collaborative activities with IRSN, NRC will be addressing other related research questions:

- What is the appropriate role for techniques such as FMEA in addressing the areas of concern identified above?
- How effective are these techniques in comparison with other alternatives?

This page was intentionally left blank

CONTENTS

FIGURES

EXECUTIVE SUMMARY

This study was launched to investigate the efficacy of fault modes and effects analysis (FMEA) as a method for identifying faults attributable to Complex Logic (for which it is not practicable to ensure the correctness of all behaviors through verification alone) and leading to system failures that would impair a safety function in a nuclear power plant.

The primary method of investigation is analytical. First, the study characterizes the differences between traditional hardwired systems and current Complex Logic-intensive systems and the technological trends that drive these differences. Given the fundamental differences thus identified, the study discusses the issues and limitations of extending FMEA, as used for traditional hardwired devices, to current complex logic-intensive systems. Then, it illustrates the analytical conclusions through examples of real-life cases. The report includes an analysis of information gleaned from relevant research publications.

The investigation concludes that FMEA techniques, which have been used effectively for traditional hardware, do not yield similar benefit to regulatory assurance when applied to Complex Logic, because of fundamental differences in the nature of faults in traditional hardware and Complex Logic. Whereas hardwired devices have only a few predetermined fault modes, the potential fault space in Complex Logic is huge; yet the actual number of faults is an extremely small fraction of the potential fault space. Finding these faults through FMEA is akin to searching for a needle in a haystack.

In examples drawn from four real-life cases (Canadian Bruce-4 nuclear reactor, Palo Verde Nuclear Generating Station Unit 2, Ariane 5 launcher, and AT&T's #4ESS toll switching systems), analysis shows that FMEA could not have helped discover the underlying faults.

Many faults and fault propagation paths cannot even be identified through an examination of the design documentation because of two well-known causes of concern—(1) incomplete, inconsistent, or ambiguous requirements and (2) inadequate, unenforceable, or unverifiable architectural constraints.

This situation leads to this research question: Under what verifiable conditions could development documents be deemed dependable for the purpose of obtaining FMEA results about hypothetical software faults, when such faults are always due to development mistakes (and are in most cases undocumented behaviors)?

An initial study of research publications indicates that the techniques were useful in the system development process, rather than in the assurance process. There is no report of a successful use of FMEA for the purpose of software assurance. However, the Korean Atomic Energy Research Institute (KAERI) has reported beneficial use of fault tree analysis (FTA) in the assurance of software. Specially crafted for software, the technique is different from the FTA used for traditional hardware. Researchers applied the technique only to a small, critical module, acknowledging that the technique was redundant to two types of verification, formal verification and testing, which had been applied to the selected module earlier, but had failed to identify the defect that the FTA revealed. The KAERI case leads to several research questions, in which are beyond the scope of this NRC-IRSN collaborative study:

- Should the assessor accept "inadequately specified verification cases" to be "normal" and overcome these weaknesses through redundant techniques? Or,
- Should the focus be on finding and fixing underlying systemic weaknesses in the upstream review criteria?

Other related research questions are: What is the appropriate role for techniques such as FMEA in addressing the areas of concern identified above? How effective are these techniques in comparison with other alternatives?

These questions will be considered in the on-going research program.

ABBREVIATIONS

ACRS	Advisory Committee on Reactor Safeguards
AT&T	American Telephone & Telegraph
COTS	commercial off-the-shelf
CPC	core protection calculator
CPU	central processing unit
DI&C	digital instrumentation and control
FMEA	fault modes effects and analysis
FTA	fault tree analysis
FY	fiscal year
HAZOP	hazard and operability
I&C	instrumentation and control
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronic Engineers
IRSN	Institut de Radioprotection et de Sûreté Nucléaire
KAERI	Korean Atomic Energy Research Institute
NASA	National Aeronautics and Space Administration (U.S.)
NPP	nuclear power plant
NRC	U.S. Nuclear Regulatory Commission

1. INTRODUCTION

With the introduction of digital instrumentation and control (DI&C) in safety-related systems, some researchers and practitioners envisaged extending the traditional hardware failure analysis techniques to software. However, this extension encounters the major difficulty that software and traditional hardware fault modes are by nature quite different. Institut de Radioprotection et de Sûreté Nucléaire (IRSN) and the U.S. Nuclear Regulatory Commission (NRC) jointly investigated and evaluated the suitability of applying fault modes and effects analysis (FMEA), as a technique for identifying faults attributable to Complex Logic (such as logic in the form of software or in the form of programmed hardware, for which it is not practicable to ensure the correctness of all behaviors[1] through verification alone) in DI&C for safety functions in nuclear power plants (NPPs). The scope of the investigation is limited to the role of FMEA in regulatory assurance of Complex Logic in DI&C safety systems. The study provides a technical basis for IRSN and NRC staff to evaluate the use of FMEA for identifying faults attributable to software and other forms of Complex Logic.

The research method is primarily analytical. First, the study characterizes the differences between traditional hardwired systems and current complex logic-intensive systems and the technological trends that drive these differences. Given the fundamental differences thus identified, the study discusses the issues and limitations of extending FMEA, as used for traditional hardwired devices, to current complex logic-intensive systems. Then, it illustrates the analytical conclusions through examples of real-life cases. The investigation also includes an analysis of information gleaned from relevant research publications.

Section 2 characterizes the differences and the technological drivers behind the differences. Section 3 explains the implications of extending FMEA and fault tree analysis (FTA) from traditional hardwired devices to Complex Logic. Section 4 captures miscellaneous related observations, by first summarizing research supporting the use of FMEA for software and then summarizing other sources of uncertainty from both today's complex electronic hardware and complex software. Section 5 summarizes the findings of the study. Section 6 identifies some issues requiring further research. Appendix A, "Glossary," defines key fundamental terms as used in this document. The definition is hyperlinked at the first occurrence of a term that is not well known or may not be well accepted. Key concepts and definitions are supported with references that provide further information about these concepts. Appendix B, "Other Causes of Faults," summarizes the sources of uncertainties in complex software. Appendix C, "State of the Art in FMEA for Software," summarizes the specific viewpoints outside the U.S. NPP industry that are relevant to the suitability of FMEA-FTA for use in the safety assurance of Complex Logic. The conclusions reported in Section 5 are consistent with these findings. However, as reported in Sections 4 and 6, further investigations, not in the scope of the current NRC-IRSN collaborative study, are needed to understand the appropriate roles and combination of FMEA and FTA and appropriate application constraints for reliable results from such analysis techniques.

[1] This refers to behavior under all foreseeable operating conditions with no anomalous behavior.

2. TECHNOLOGICAL TRENDS

Traditionally, NPPs have relied on hardwired devices for their instrumentation and control (I&C) safety functions. In recent years, a shift in technology has led to the use of digital electronic systems in nuclear safety applications, because of the increased obsolescence and difficulty in maintaining analog electronic assemblies and to take advantage of functions enabled by digital logic. However, this introduction of digital electronics into nuclear safety applications, with increase in functionality, interdependencies[2], and complexity[3], increases the potential for failures due to systemic causes. This section characterizes and contrasts the technological differences.

2.1 Characteristics of past hardwired electrical technologies

When I&C systems comprised only traditional electrical hardware (where logic was hardwired), the components were functionally simple (diodes, resistors, relays, etc.), and they had to be physically placed on printed circuits of limited size. In practice, this severely constrained the number of components and therefore allowed only simple designs. This simplicity yielded two advantages: (1) practitioners could evolve designs that were not very fault prone, compared to current logic-intensive digital systems, and (2) the circuits could be verified with a higher degree of certitude. Thus, relative to system failures caused by component degradation, there was an insignificant contribution of system failures due to design faults, or, more generally, engineering mistakes and other systemic causes, compared to current logic-intensive digital systems.

Hardware component technologies used in early NPP safety systems were stable, and change in technology was slow enough that sufficient operating experience could be accumulated for understanding fault modes and their failure likelihood. In successive system design cycles, component types could be selected based on operating experience, thus avoiding component types that failed in unpredictable ways. To understand whether the system could perform its intended function when needed, the system was analyzed for its reliability (e.g., through failure analysis techniques such as FMEA and FTA). Most fault propagation paths could be understood by examining the system design (interconnections and their implied interactions); even “sneak paths” could be discovered from the available documentation. Given the reliability of each hardware component and the fault propagation paths thus determined, the system reliability could be estimated and effects on system functions could be understood. The system could be configured to monitor and detect the failure or impending failure of a critical component and bring the system to a safe state gracefully.

Hardware fault modes of traditional hardwired logic may be characterized as follows:

- Typically, faults result from physical degradation.
- The number of fault modes for basic components (e.g., relay stuck open or stuck closed) is limited, and these fault modes are well understood. (Manufacturers often give fault modes and frequencies for their basic components based on operational experience with the same and similar components.)
- Faults propagate along the electrical interconnections between components (e.g., the printed circuit tracks and wiring across panel-mounted components in electrical enclosures).

[2] For example: Interconnections; signal exchanges; resource sharing
[3] For example: Number of elementary structures grows from 10s (hardwired) to millions or billions (gates in complex logic)

- Faults occur randomly[4]; good design and maintenance practice may extend the interval between random occurrences, but in general, it is accepted that the likelihood cannot be reduced to zero.

2.2 Characteristics of current digital electronics systems

Some safety systems, including protection systems, have been implemented with digital electronics to compute complex protection functions, such as departure from nucleate boiling ratio, which allow a better and safer use of the nuclear fuel. Digital electronics systems represent signals as discrete levels, rather than as a continuous range, and perform computations using assemblies of logic gates, representing Boolean logic functions. These assemblies[5] can comprise computers or other forms of programmable or configurable hardware, as discussed in the next section.

Current digital technologies enable many benefits (e.g., more complex and accurate computations, auxiliary functions such as calibration and self-monitoring, significant reduction of cabling and signal transmitters, and ease of modification). For this reason, their use has expanded to the point that most new systems or retrofits exploit digital electronics.

2.2.1 Programmability and increasing configurability

Current digital technologies are characterized by their programmability, which may take one of the following forms or a combination of both:

- Software: The hardware is based on general-purpose, commercial off-the-shelf (COTS) components such as microprocessors, electronic memory components, and input/output circuitry. The application-specific function is implemented in a program which is stored in memory and executed by the microprocessor. Typically, application-independent software (known as the system software, which is a very simplified form of operating system) is used to interface with the computing and communication hardware, sensors, and actuators and to manage the sharing of system resources.
- Programmable hardware: The hardware is based on programmable circuits such as field-programmable gate arrays, complex programmable logic devices, or application-specific integrated circuits. These components cannot perform a safety function without first being "customized" (physically configured to a specific safety logic). This customized logic then works alone (no operating system or system software is needed).

2.2.2 Increasing complexity of hardware

The logic of DI&C system output functions also depends on complex hardware modules which are also susceptible to systemic causes of faults. In addition, current digital component technologies have very high integration densities, reaching billions of transistors in a single integrated circuit, which increases the potential for component-level faults due to engineering mistakes.

2.2.3 Notion of faults in Complex Logic

Programs, a product of human thought, are becoming more and more complex, as indicated by their increasing requirements, functions, inputs/outputs, and interdependencies. Therefore, programs are more and more fault prone.

[4] This does not imply that the causes are random.

[5] The size of the constituent elements can range from small (e.g. 10s of elementary structures as in the case of past hardwired systems) to very large (e.g. millions of logic gates).

Development of the application-specific contents of programmable hardware also involves the specification of requirements, design with dedicated languages, test, simulation, and use of complex software tools and thus has the same potential for logic faults.

For example, the National Aeronautics and Space Administration (NASA) has stated the following about complex electronics [6]:

- "Logic errors are still common in space-flight projects, with bad circuits making it into flight hardware."
- "A fundamental issue is how the complexity is managed to permit reliable design."

Similarly, the U.S. Federal Aviation Administration has stated the following [7]:

- "Not only are there the normal hardware integrity issues for safety-critical systems, but now all the issues of software correctness apply also."
- "Error-free parts can no more be guaranteed than one can promise error-free software."

Potential faults in Complex Logic may originate in any phase of the development cycle, as detailed in Section 3.1. For example, many defect types may occur just in the design phase: a missing statement, a mistake in the name of a variable, a defect in loop control (software) or similarly in register-controlled cyclic paths (programmable electronics), a wrong initial value, the use of a wrong operator, a misunderstanding of operator precedence, a bad conditional construct, or others. Defects may also occur when the requirements of the Complex Logic are missing, ambiguous, or superfluous.

Therefore, programming digital electronic systems of growing complexity, either through software or through programmable hardware, has the potential to introduce faults at any step of the development process and such potential faults, if actually introduced in the delivered Complex Logic, may trigger system failures.

In contrast with the degradation-caused fault modes of traditional hardware characterized in Section 2.1, logic does not wear and tear from repeated usage. If a system fails because of logic, it had some fault (defect or deficiency or weakness) from the time of introduction, but this fault remained latent until the occurrence of a triggering or enabling combination of inputs, state of the environment, state of the DI&C system, and state of the faulty logic.

2.2.4 Identification of potential faults in Complex Logic

To evaluate the applicability of different techniques for identifying potential faults, an estimate follows of the order of magnitude of the potential fault space.

The reactor trip output of a protection system typically depends on more than 50 analog inputs, each one digitized into more than 100 discrete values (typically 1,024). This means that the input space has more than $100^{50} = 10^{100}$ cases—more than the number of atoms in the universe,[6] even without considering the influence of past inputs on the behavior of the logic.[7] Each of these cases is either within the authorized process domain or outside it, so for each input case, there exists one right value and one wrong value for the reactor trip output. Thus, there are potentially at least 10^{100} different faulty logics, each one producing the wrong output

[6] According to current estimates, there are approximately 10^{80} atoms in the universe.

[7] The behavior of a program usually depends not only on the values of its current inputs, but also on its internal state S, which may depend on arbitrarily old inputs (e.g., for ever S {if input condition X holds, then modify state S, or else keep S as it is; output = f(S, current inputs)}).

for exactly one of the 10^{100} input cases. Of course, all logics wrong for more than one input case are also faulty, leading to an even greater number of potential faulty logics.

No tool can enumerate this large number of potential faults. Current processing speeds are in the range of 10^{10} instructions per second. Even if each case could be processed in one single computing instruction cycle, the processing time would be much more than 10^{90} seconds or 3×10^{83} years. This is so much time that even a large number of computers (hundreds, thousands, or even millions) operating in parallel with 100-percent efficiency could not reduce computation time to a practicable level.

Finally, when considering the influence of past inputs,[7] the number of possible faulty logics is unbounded. Whatever the considered finite set of faulty logics, it is always possible to produce another faulty one (e.g., by delaying the occurrence of a faulty behavior by one additional clock tick).

Therefore, the number of potential faults in a Complex Logic cannot be bounded in general, and these potential faults cannot be exhaustively identified.

2.2.5 Characteristics of faults in Complex Logic

Logic faults that may trigger DI&C system failures are characterized as follows:

- The number of potential faults in Complex Logic is very high (see Section 2.2.4), and these potential faults cannot be exhaustively identified.
- If the development and assurance processes are stringent and include independent verification, only a limited number of faults is actually present in the Complex Logic of typical safety systems. This implies that the faults actually present are unknown; otherwise, they would be corrected.
- Faults that have not been introduced during the development process of the Complex Logic or that have been removed during verification and validation will never appear in use.

3. IMPLICATIONS OF EXTENDING FMEA TO COMPLEX LOGIC

Given the technological trends characterized above, IRSN-NRC jointly examined the validity of extending the traditional hardware failure analysis techniques for the assurance of Complex Logic, such as software in a DI&C system for safety functions.

Hardwired systems using past technology (diodes, relays, etc.) have a limited number of well-understood faults. As discussed in Section 2.1, these faults result from physical deterioration over time, which means that they will necessarily occur during operation unless the component is replaced or the system is removed from service. The propagation paths for these faults are known (e.g., they are derived from printed circuit tracks or cabinet wiring). Therefore, the identification of the associated fault modes and the analysis of their effects, causes, and likelihood by FMEA and FTA are feasible and useful.

In contrast, the Complex Logic in each new digital system, as characterized in Section 2.2, is not identical to the logic in any previous system, and the number of possible faults (i.e., the size of the potential fault space) can be extremely large, as shown in Section 2.2.4. A complete list of fault modes cannot be assembled [8].

Complex Logic design practice for safety systems follows principles that are intended to prevent faults in the first place; a combination of verification techniques is used to discover and remove faults or conditions that could lead to the failure of a safety function. This implies that the faults actually present in a Complex Logic are unknown; otherwise, they would be corrected during the development process.

FMEA attempts to analyze the effects for the severity of their consequences and the modes for their likelihood of occurrence, or to identify measures to avoid their occurrence, or to identify measures to mitigate the consequences. For the technique to be workable, it should be possible to identify a feasibly small number of fault modes.[8] Sections 3.1 and 3.2 analyze two approaches to arrive at a compact set of fault modes. Section 3.1 discusses fault modes abstracted from the causality perspective, and Section 3.2, from the effect perspective. In both cases, it is difficult to identify the effects. Section 3.2.1 discusses a case where faults can propagate in software unpredictably, even through functionally dependent paths. (Appendix B discusses other sources of uncertainty that are especially exacerbated when Complex Logic is implemented in software.) Section 3.3 illustrates these difficulties through some examples.

3.1 Analyzing fault modes from the causality perspective

Logic faults are rooted in human mistakes made during development, as illustrated in the process model in Figure 1, which shows the human role as a direct resource in the process, as well as having indirect roles in the creation of various resources. Mistakes may be made at the very beginning of the development during the specification of requirements or later, at any step, including transformation and introduction or omission of any information that will be included in or affect the delivered logic ultimately. Mistakes committed during the development of software tools used to build the delivered logic may also introduce faults in that logic. Compilers and synthesizers are typical examples of such software tools.

[8] Recall from the analysis in Section 2.2.4 that the potential fault space for Complex Logic can be extremely large, even without considering the uncertainties mentioned in Section 4.

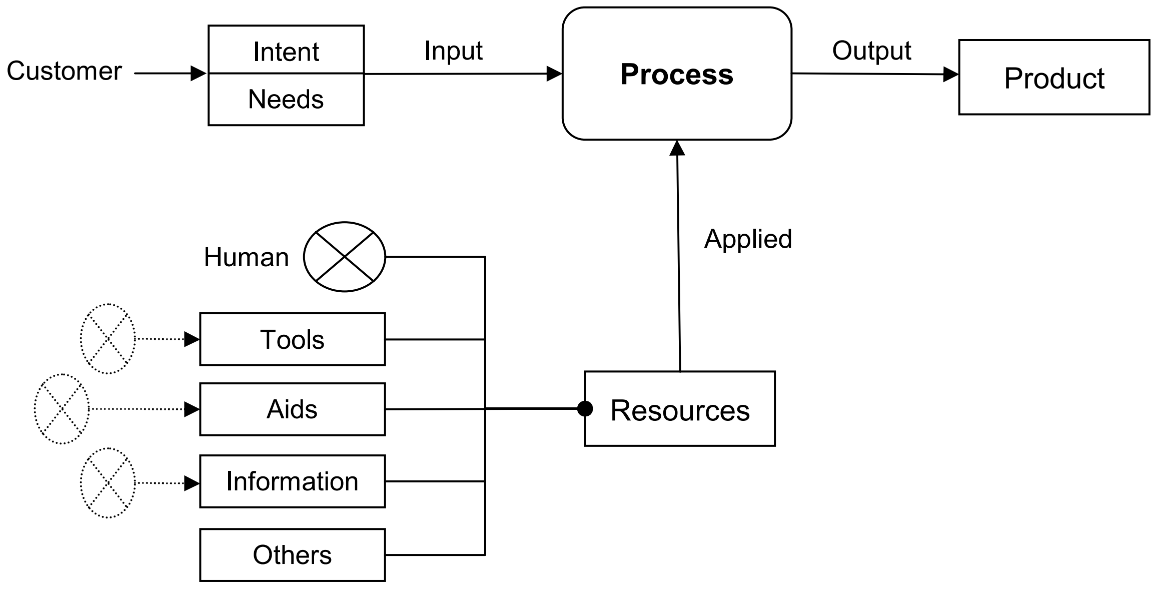

Figure 1: General process model for an engineered product

Human mistakes causing logic faults may be abstracted and categorized broadly as follows:

- missing or incorrect requirement at the beginning of the development
- incorrect translation of a requirement at some step of the development
- addition of an unspecified requirement[9]
- selection of faulty (or incorrectly or inadequately documented) COTS software or hardware

One may be tempted to use such short lists as fault modes. However, the value of this is questionable, because it is not possible to predict the fault modes' effects on a given piece of software.

Indeed, in the case of traditional hardware, a fault mode such as "stuck open" defines a specific behavior that can be explicitly propagated through the hardware to predict its effects. By contrast, a fault mode such as "unspecified requirement" does not define a specific behavior: any possible behavior should be taken into account and explicitly propagated through the software, which is not feasible. Further analysis of the "unspecified requirement" category, as an example, reveals that the number of possibilities of "unspecified requirement" cannot be enumerated exhaustively. Similarly, in the category of "incorrect translation of a requirement," the possibilities cannot be enumerated exhaustively. Then, because of the typically huge number of possibly different execution cases (as shown in Section 2.2.4), it is not possible to analyze each potentially faulty case one after the other. FMEA and FTA techniques are usable when the types of faults are a limited set, but these techniques cannot be used effectively and efficiently when the potential fault space is so large.

As a hypothetical alternative, consider analyzing only those faults that are actually present; indeed, these actual faults are very few in safety software developed, verified, and validated to

[9] Unspecified requirements arise when a developer adds functionality to a system without updating the system, software, or hardware requirements document(s). An example of adding an unspecified requirement to a system is the addition of debug statements to software code and subsequent mistake of not removing the statements from the software without updating the software requirements document to reflect the additional functionality in the code. The unspecified requirements arising from these debug statements might adversely affect system performance and resources (e.g., system timing constraints, memory needs, stack needs, or system interface requirements).

very high standards. However, if actual faults in safety systems were known, they would be corrected; then FMEA would not be needed.

When Complex Logic is implemented in software, additional sources of uncertainty arise, as discussed in Appendix B, which reduces even more the validity of the FMEA-FTA contribution to software assurance.

3.2 Difficulty in identifying the effects of faults

An FTA of an NPP as low as the safety I&C system would identify the event of interest (i.e., failure of a safety function allocated to the I&C system). An event tree analysis, internal to the I&C system, would trace the paths of functions on which this safety function was dependent. This analysis can be performed down to the finest-grained functions identified in the logical architecture and the logic modules to which these functions are allocated.

Then, the manner in which any module in these paths could malfunction (i.e., its fault mode) is of interest for understanding the effect of that fault mode on the safety function. For example, fault modes of a module are characterized in terms of the module's functions:

(1) failure to perform the module function in time (i.e., in time domain)
(2) failure to perform the module function with correct value (i.e., in value domain)
(3) performance of an unwanted function by the module
(4) interference or unexpected coupling with another module

The effects of potential Fault Modes 1, 2, and 3 of a software module on the safety output are difficult to analyze. Indeed, the exact semantic of the software and of the computing architecture has to be considered to predict the impact of each possible time or value error of a module (e.g., delivery of the results of a module too early or too late may have catastrophic impact on the safety output or no impact at all, depending on the real-time scheduling of the modules).

In the case of Fault Mode 1 (failure to perform the module function in time), performing an accurate analysis would imply studying every potential time error for any possible scheduling of the modules. In the case of Fault Mode 3 (performance of an unwanted function by the module), it is even more difficult to predict the impact of any possible unwanted function of a module on the safety output, because it would require identifying all possible unwanted functions and investigating their effects completely. As FMEA-FTA tools do not consider the exact semantic of the software and of the computing architecture, and do not identify all possible unwanted functions of each module, it is necessary to assume that the effect of these potential fault modes is that the DI&C system will fail to perform its intended safety function.

As seen in Section 2.2, a huge number of cases pertain to Fault Modes 1, 2, and 3, and there is no way to know which ones are actually present. Therefore, FMEA-FTA analyses can conclude only that there is a huge number of potential faults that potentially lead to failure of the safety function.

Potential Fault Mode 4 (interference with another module) is more common in software. A fault within a given module may adversely impact another module, even if those modules do not interact from the functional point of view (i.e., there is neither value exchanged, even indirectly through other modules, nor calling relation between the modules). One typical example is that a faulty module could write "out of bounds" of its allocated memory and corrupt a memory location used by another module. The failure can manifest itself, for instance, by corrupting a data value that would not be used until 6 months later in the operation of another module, and then cause a catastrophic failure.

Propagation paths of faults include not only all functional dependency links but also paths not visible from the functional point of view (i.e., not visible in the functional requirements). For example, two functionally independent modules may in fact need access to a shared resource, such as a bus to access memory and input/output. In this case, a fault in one module may put the shared resource in a faulty state and then adversely impact the other module. Since the dependency is not evident, it is difficult to verify non-interference or to identify an interference fault mode.

There are many other subtle manifestations of such potential fault modes. Therefore, FMEA-FTA techniques, as used commercially, cannot be relied on to identify the fault modes within a module, nor can these techniques accurately model their effects on other modules or on the safety output.

3.2.1 Unpredictable fault propagation in software even with known dependencies

Figure 2 illustrates an example of unpredictable propagation in software. In a program, the outputs of a given Unit A typically provide inputs to a Unit B and so on (e.g., C, D) until the signals reach one of the overall outputs (e.g., O1-O3, depending on the inputs (e.g., I_{A1}, I_{A2}, I_{A3}) and the state transitions (e.g., B-S1, B-S2, B-S3). Hence, a software fault in Unit A may lead to an erroneous input in Unit B that will possibly lead to erroneous outputs of Unit B and so on. However, this propagation is not always predictable and may depend on the precise behavior of Unit B and other units and the entire state history. For instance, if the behavior of Unit B is to not use the particular output of Unit A when it is detected as faulty, if the specific fault under analysis is detected by B, and if B may tolerate the omission of a limited number of A output samples, then the propagation may be stopped. A system-level Fault Mode (1), as identified in Section 3.2, may occur as a result of propagation through software in the following manner. The output of A occurs at a time different from the time expected in the design of the policy to schedule the execution of B. When B is executed, the value used from A is incorrect for that execution.

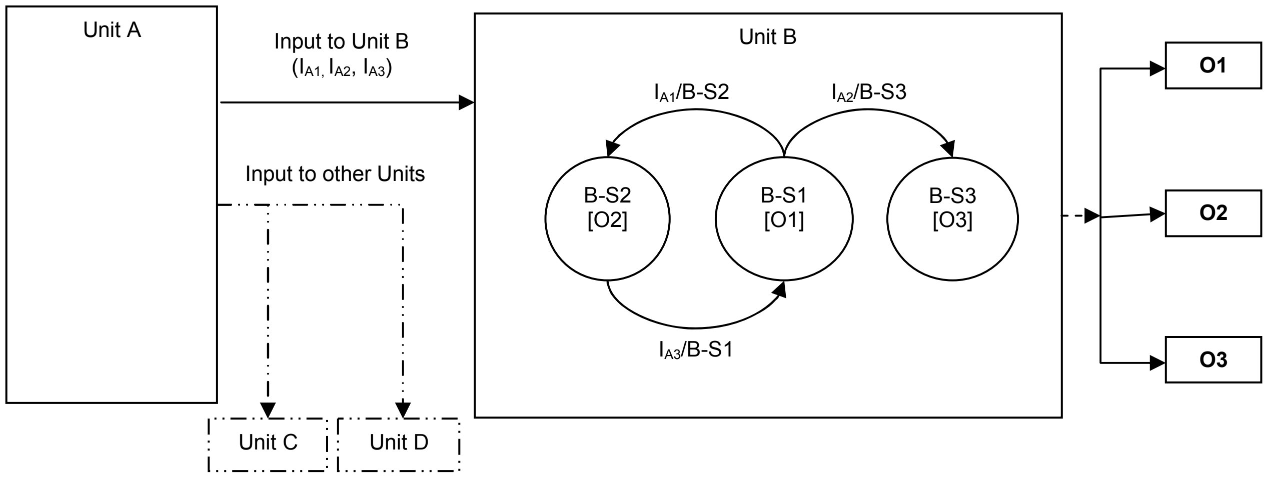

Figure 2: Unpredictable fault propagation in software

In addition, the dependency space is huge, as illustrated in the following example. The amount of data exchanged between real software units would quickly add to hundreds of elementary fields. The number of units could be hundreds. The size-complexity of each unit is typically hundreds of lines of code. Carrying out such a dependency analysis already leads to a huge number of dependency paths and is quite challenging even with the use of tools.

For example, consider a state transition model of the behavior internal to the DI&C system, down to the interactions across components, including the behavior in case of a hardware failure.[10] The traditional use of FTA, when applied to software, would be based on the designed control flow paths. However, some system failures attributable to software lead to a behavior that may not be analyzed, as they propagate erroneous values along the designed control flow paths of the software, but break that control flow and create a different set of behaviors. This is the case for software faults raising exceptions (e.g., a division by zero) or interrupts, or failures leading to another path in the binary code (e.g., consequent to a stack overflow). Here again, the possibilities are huge and are even harder to analyze systematically because the designed control flow does not capture all the propagation paths.

3.2.2 Unpredictable fault propagation in software with hidden dependencies

Dependencies may not be easy to find. For example, in a producer-consumer paradigm, especially with multiple consumers, the function chain does not reveal the propagation paths possible through the software in the system.

Hidden dependencies and couplings are a major cause of system failure, limiting the ability to identify event propagation paths. In software systems, each part could appear to be correct and fully functional, but when components are combined as a system, problems can manifest themselves. There is no single component that would be at fault. Rather, it is the combination that can be at fault or the combination of explicit and implicit assumptions. Many critical assumptions are never formally written down.

FMEA-FTA, when performed on the documented design, will not help in the discovery of such unpredictable fault propagation paths.[11]

3.3 Experience feedback

Following are some examples of system failures in different application domains, which illustrate the difficulty of identifying the fault modes and effects of Complex Logic.

3.3.1 Canadian Bruce-4 nuclear reactor

Description: In January 1990, an incident occurred at the Canadian Bruce-4 nuclear reactor in which a small loss-of-coolant accident resulted from a programming fault in the software used to control the reactor refueling machine.[12] Because of this fault, the control computer, when suspending execution of the main refueling machine positioning control subroutine in order to execute a fault-handling subroutine triggered by a minor fault condition detected elsewhere in the plant, marked the wrong return address in its memory. As a result, execution resumed at the wrong segment of the main subroutine. The refueling machine, which at the time was locked onto one of the reactor pressure tube fuel channels, released its brake and dropped its refueling assembly by about 0.9 m (3 feet), damaging both the refueling assembly and the fuel channel [9].

Why FMEA-FTA techniques are not practical in such cases: This is a typical case of module Fault Mode 4 (a fault in a module adversely impacts another module under a previously unknown specific condition). FMEA-FTA could not have identified this fault mode for the following reasons:

[10] In current practice, such models are not available for safety reviews.

[11] To appreciate the scope of assurance activities, see notes under the definition of Complex Logic.

[12] Even though the system was not qualified to safety-grade standards, this type of software fault could also occur in a safety-grade system.

- FMEA-FTA does not know the involved mechanisms of the computing architecture (stack, return address, calling mechanisms) and does not analyze the source code.
- FMEA-FTA provides no means to identify this fault rather than any other potential fault.

3.3.2 AT&T's #4ESS toll switching systems

Description: On January 15, 1990, one of American Telephone & Telegraph's (AT&T's) #4ESS toll switching systems in New York City experienced an intermittent failure that caused a major service outage on the AT&T U.S. National Telephone Network.[13] The outage occurred because of a software defect that had escaped detection even by AT&T's software test methods. The software defect was traced to an elementary programming mistake (i.e., a misplaced break statement in a "C" program switch statement [10]).

Why FMEA-FTA techniques are not practical in such cases: This is a typical case of module Fault Mode 2 (a module does not return the correct value under a previously unknown specific condition). FMEA-FTA could not have identified this fault mode for the following reasons:

- FMEA-FTA does not know the involved mechanism of the "C" programming language (fall-through conditions between cases of a "switch" statement) and does not analyze the source code.
- FMEA-FTA provides no means to identify this fault rather than any other potential fault.

3.3.3 Ariane 5 launcher (Ariane 501)

Description: On June 4, 1996, the maiden flight of the Ariane 5 launcher[14] (Ariane 501) resulted in self-destruction of the launcher. An independent inquiry board established by the European Space Agency reported that the failure of Ariane 501 was caused by the complete loss of guidance and attitude information 37 seconds after start of the main engine ignition sequence. This loss of information was the result of specification and design defects in the software of the inertial reference system. The inquiry board concluded that extensive reviews and tests carried out during the Ariane 5 development program did not include adequate analysis and testing of the inertial reference system or of the complete flight control system, which could have detected the failure [11].

Why FMEA-FTA techniques are not practical in such cases: This is a typical case of module Fault Mode 2 (a module does not return the correct value under a previously unknown specific condition). FMEA-FTA could not have identified this fault mode for the following reasons:

- FMEA-FTA does not know the involved mechanism of the programming language (overflow error when "casting" a given "floating point" value into an integer of a given size) and does not analyze the source code.
- FMEA-FTA provides no means to identify this fault rather than any other potential fault.

3.3.4 Palo Verde Nuclear Generating Station Unit 2

Description: On May 22, 2005, in the core protection calculator (CPC) of Palo Verde Nuclear Generating Station Unit 2, a software fault caused by a mistranslated system requirement when updating the CPC software was discovered. The CPC system requirements state that if

[13] Even though the system was not qualified to safety-grade standards, this type of software fault could also occur in a safety-grade system.

[14] Even though the system was not qualified to safety-grade standards, this type of software fault could also occur in a safety-grade system.

redundant analog input cards within the same channel have failed, the CPC should recognize those failures and automatically trip that channel. However, the software reverted to the “last stored good value” in case of double sensor failure, which could have masked an actual trip condition. This could have influenced multiple channels in the event of a common-cause failure affecting their sensors simultaneously [12]. After analyses and meetings between the vendor and Palo Verde staff, it was decided that the software defect common to all channels was a safety concern [13].

Why FMEA-FTA techniques are not practical in such cases: This is a typical case of a mistake in the requirement specification of the software.[15] FMEA-FTA could not have identified this fault mode for the following reasons:

- FMEA-FTA does not analyze the source code and does not compare it to the actual needs.
- FMEA-FTA provides no means to identify this fault rather than any other potential fault.

[15] The mistake was in the specification rather than in the software design.

4. MISCELLANEOUS OBSERVATIONS

FMEA-FTA has been used in hazard analysis of software-reliant systems as discussed in Section 4.1. However, in addition to the issues identified above (mostly common to various realizations of Complex Logic), FMEA-FTA for software-reliant safety systems encounters some other pitfalls. Section 4.2 discusses uncertainties in large, complex, electronic hardware and assumptions about such hardware, which can affect the validity of safety analysis concerning software.

4.1 Reported beneficial uses of FMEA for software

Whereas this study questions the suitability of FMEA for the assurance of Complex Logic, it is also acknowledged that others [14–25] (as discussed in Appendix C) have found these techniques useful in hazard analysis leading to the discovery or identification of safety requirements. This activity is part of the system development process, performed by the license applicant or its agent and tailored to suit the applicant's needs (e.g., system characteristics).

FMEA has been used to analyze whether the architecture will bring the system to a safe state when something in the system does not behave as intended, whether it be a hardware component[16] or a software component. The use of FMEA is a part of system-internal hazard analysis [23], which abstracts in terms of system functions (as discussed in Section 3.2) the fault modes attributable to software.

The Korean Atomic Energy Research Institute (KAERI) has implemented a digital reactor protection system, for which it performed safety analysis of the software as a part of software development [23]. KAERI devised a "software HAZOP" technique, different from conventional hazard and operability (HAZOP) analysis. In KAERI's technique, for each system hazard, it searches for adverse effects of qualitative software functional characteristics identified in the NRC's "Standard Review Plan for the Review of Safety Analysis Reports for Nuclear Power Plants: LWR Edition" (NUREG-0800), Branch Technical Position 7-14, "Guidance on Software Reviews for Digital Computer-Based Instrumentation and Control Systems," Revision 5 [26]. Limiting the search space to paths leading to critical system hazards helps reduce the analysis time compared to that of a conventional FMEA. From the result of the "software HAZOP," KAERI selected a search subspace leading to the most critical system hazard (failure to trip on demand) and applied a software FTA technique for a detailed traversal through the software logic structure. The software FTA revealed a software defect[17] that was not found in formal verification and testing. KAERI acknowledges that the use of software FTA was redundant (the "software HAZOP" could have been extended in the causal direction), but the experience on this project shows the value of redundancy in testing and formal verification techniques. This experience report raises the following questions:

- Should one accept such "misses" to be "normal" and overcome these weaknesses through redundant techniques? Or
- Should the focus be on finding and fixing systemic weaknesses[18] in the state of the art for formal verification and testing techniques[19]?

[16] An example would be the effect of a bit flip.

[17] Further investigation of KAERI's experience is needed to understand how the FTA was effective in identifying the fault.

[18] It is not a reflection on KAERI but indicates promising avenues for advancing the state of the art.

4.2 Assumptions about physical faults in digital electronics

The economically useful life of digital component technologies has become so short that it is difficult to accumulate adequate operational experience with their respective fault modes and likelihoods.

In addition, digital state switching time is being reduced continually, as well as switching voltage or energy threshold, which makes digital electronics more and more sensitive to radiation or high-energy particles. On the other hand, with the increasing use of electronics, electromagnetic emissions in the environment (e.g., from power electronics) are increasing. This makes it increasingly difficult to identify elementary physical faults in digital electronic hardware components, such as sensors, actuators, and computing devices, and to analyze how such elementary faults propagate within the circuit. Therefore, commensurate mitigating safety requirements (often for detection software) are not derivable through a well-defined, repeatable procedure.

Uncertainties about the nature and occurrence of physical faults in new digital component technologies, especially faults leading to partial or intermittent component failure, may lead to unintended effects on the functionality of the impacted integrated circuit. In particular, the propagation paths of physical faults within an integrated circuit depend on complex and undisclosed microelectronic processes at nanometric scale. However, FMEA-FTA applied to the physical faults in a high-density integrated circuit often consider only the case of total failure of all functions provided by this circuit.

These uncertainties increase system complexity and add to the challenges of validating assumptions about hardware when analyzing software.

[19] This may also apply to development constraints (see second note under the definition of Complex Logic in Appendix A).

5. CONCLUSIONS

IRSN and NRC researchers have jointly analyzed the role of FMEA in regulatory assurance of Complex Logic, especially software, in an NPP safety system and concluded that its contribution is marginal.

The root cause of this limitation lies in the fundamentally different characteristics of faults in Complex Logic, especially software, compared to physical faults for which FMEA-FTA had been developed. Whereas hardwired devices have only a few predetermined fault modes, the potential fault space in Complex Logic is huge; yet the actual number of faults is an extremely small fraction of the potential fault space. Finding these faults through FMEA is akin to searching for a needle in a haystack. Therefore, extending methods that have been successfully used for analyzing traditional hardware to Complex Logic does not yield similar benefits.

6. ADDITIONAL INVESTIGATIONS

Before further investigation into the appropriate role of FTA and FMEA in safety analysis of Complex Logic, a broader question must be addressed: Under what verifiable conditions would development documents be dependable for obtaining FMEA-FTA results[20] for hypothetical software faults, when such faults always result from development mistakes (and are in most cases undocumented behaviors)?

The dependability of documentation is in question for two well-known causes of concern:

(1) incomplete, inconsistent, or ambiguous requirements
(2) inadequate or unenforceable or unverifiable architectural constraints[21]

This leads to related research questions: What is the appropriate role for techniques such as FMEA and FTA in addressing these areas of concern? How effective are these techniques in comparison with other alternatives?

Further investigations are needed to understand and evaluate the benefits of application of FMEA-FTA for software, reported by several experts and expert groups [14–25]. An initial study of their publications indicates that the techniques were useful in support of system-internal hazard analysis and for discovering and identifying safety requirements. The reported applications were successful under certain specific conditions.

With respect to results from others' research [14–25], reported in Section 4, the NRC intends to contact these experts to gain a deeper understanding of their experience.

In the case of the KAERI application [21], in the context of the broad research questions mentioned above, the NRC has identified the following coupled research questions:

- Should the assessor accept "inadequately specified verification cases" to be "normal" and overcome these weaknesses through redundant techniques? Or,
- Should the focus be on finding and fixing underlying systemic weaknesses in the upstream review criteria?[22]

These questions will be considered in NRC's on-going research program.

[20] FMEA-FTA results can be used for software analysis.

[21] As an example of architectural concerns, if some dependency or flow path is not shown in the design, fault propagation paths or unwanted flow paths would escape analysis.

[22] See the second note under the definition of Complex Logic in Appendix A.

7. REFERENCES

1. U.S. Nuclear Regulatory Commission, “NRC Digital System Research Plan FY 2010 – FY2014,” (ML100541484), Washington, DC, February 2010.
2. U.S. Nuclear Regulatory Commission, “Commission Meeting with Advisory Committee on Reactor Safeguards,” Staff Requiement Memoranda M080605B (ADAMS Accession Number ML081780761), Washington, DC, June 26, 2008.
3. U.S. Nuclear Regulatory Commission, “Digital Instrumentation and Controls Systems Interim Staff Guidance,” Advisory Committee on Reactor Safeguards, 551th Meeting Letter Report (ML081050636), Washington, DC, April 29, 2008.
4. U.S. Nuclear Regulatory Commission, “Draft Final Digital Instrumentation & Control Interim Staff Guidane-06: Licensing Process,” Advisory Committee on Reactor Safeguards, 576th Meeting Letter Report (ML102850357), Washington, DC, October 10, 2010.
5. U.S. Nuclear Regulatory Commission, “Digital Instrumentation & Control Interim Staff Guidane-06: Licensing Process,”(ML110140103), Washington, DC, January 19, 2011.
6. Katz, R., Barto, R., and Erickson, K., “Logic Design Pathology and Space Flight Electronics,” NASA Goddard Space Flight Center, 1997.
7. U.S. Federal Aviation Administration, “Design, Test, and Certification Issues for Complex Integrated Circuits,” DOT FAA AR-95/31, Washington, DC, 1996.
8. Lutz, R.R. and Woodhouse, R.M., “Bi-Directional Analysis for Certification of Safety-Critical Software,” *Proceedings, ISACC 1999*, International Software Assurance Certification Conference, Chantilly, VA, February 28–March 2, 1999.
9. Garrett, C. and Apostolakis, G., “Context and Software Safety Assessment,” 2nd International Workshop on Human Error, Safety and System Development, Seattle, WA, 1998.
10. Toy, W.N., “Fault-Tolerant Design of AT&T Telephone Switching Systems,” in *Reliable Computer Systems: design and evaluation,* Siewiorek and Swarz, Eds., Digital Press, Burlington, MA, 1992.
11. European Space Agency, “Ariane 501—Presentation of Inquiry Board Report,” Press Release No. 33-1996, July 23, 1996.
12. Bickel, J.H., “Risk Implications of Digital RPS Operating Experience,” International Atomic Energy Agency Technical Meeting, June 19–21, 2007.
13. U.S. Nuclear Regulatory Commission, “Technical Specification Required Shutdown Due to Core Protection Calculators Inoperable,” Licensee Event Report 52992005004, Washington, DC, May 22, 2005.
14. Goddard, P.L., “Software FMEA Techniques,” *Proceedings Annual Reliability and Maintainability Symposium*, pp. 118–123, 2000.
15. Goddard, P.L., “Validating the Safety of Embedded Real-Time Control Systems Using FMEA,” *Proceedings Annual Reliability and Maintainability Symposium*, pp. 227–230, 1993.
16. Leveson, N.G., “Safeware: System Safety and Computers,” ISBN 0-201-11972, Addison-Wesley Professional, Reading, MA, April 17, 1995.

17. Lutz, R.R. and Shaw, H.Y., “Applying Integrated Safety Analysis Techniques (Software FMEA and FTA),” Jet Propulsion Laboratory (JPL) D-16168, Pasadena, CA, November 30, 1998.

18. Lutz, R.R. and Woodhouse, R.M., “Experience Report: Contributions of SFMWA to Requirements Analysis,” *Proceedings of ICRE 1996*, pp. 44–51, Colorado Springs, CO, April 1996.

19. McDermid, J.A., Nicholson, M., Pimfrey, D.J., and Fenelon, P., “Experience with the Application of HAZOP to Computer-Based Systems,” *Proceedings of COMPASS 1995*, IEEE, Gaithersburg, MD, pp. 37–48, 1995.

20. Pentti, H. and Atte, H., “Failure Mode and Effects Analysis of Software-Based Automation Systems,” STUK—Radiation and Nuclear Safety Authority, p. 37, Helsinki, August 2002.

21. Kwon, K.C. and Lee, M., “Technical Review on the Localized Digital Instrumentation and Controls Systems,” Special Issue in Celebration of the Korean Nuclear Society, *Nuclear Engineering Technology*, Vol. 40, No. 5, August 2008.

22. Park, G.Y., Koh, K.Y., Jee, E., Seong, P.H., Kwon, K.C., and Lee, D.H., “Fault Tree Analysis of KNICS RPS Software,” *Nuclear Engineering Technology*, Vol. 41, No. 4, May 2009.

23. Park, G.Y., Lee, J.S., Cheon, S.W., Kwon, K.C., Jee, W., and Koh, K.Y., “Safety Analysis of Safety-Critical Software for Nuclear Digital Protection Systems,” *Proceedings of SAFECOMP 2007*, Lecture Notes in Computer Science 4680, pp. 148–161, 2007.

24. Kwon, K.C. and Park, G.Y., “Formal Verification and Validation of the Safety-Critical Software in a Digital Reactor Protection System,” NPIC & HMIT 2006, Albuquerque, NM, November 12–16, 2006.

25. Lee, J.S., Lindner, A., Choi, J.G., Miedl, H., and Kwon, K.C., “Software Safety Lifecycles and the Methods of a Programmable Electronic Safety System for a Nuclear Power Plant,” *Proceedings of SAFECOMP 2006*, Lecture Notes in Computer Science 4166, pp. 85–98, 2006.

26. U.S. Nuclear Regulatory Commission, “Standard Review Plan for the Review of Safety Analysis Reports for Nuclear Power Plants: LWR Edition,” Branch Technical Position 7-14, “Guidance on Software Reviews for Digital Computer-Based Instrumentation and Control Systems,” NUREG-0800, Revision 5, Washington, DC, 2007.

APPENDIX A: GLOSSARY

The scope of this glossary is limited to this document.

Complexity

(A) (software) The degree to which a system or component has a design or implementation that is difficult to understand and verify. (Definition (1)(A) in [1])

(B) (software) Pertaining to any of a set of structure-based metrics that measure the attribute in Definition (1)(A) in [1]. (Definition (1)(B) in [1])

Notes:

- There are other perspectives on the definition of complexity as illustrated below (i.e., there is no broadly accepted definition, even in the limited context of safety-critical software engineering).
- The number of linearly independent paths (one plus the number of conditions) through the source code of a computer program is an indicator of control flow complexity, known as McCabe's cyclomatic complexity. [2]
- Sometimes, the term "size-complexity" is used to refer to the effect of the number of states and number of inputs and their values and combinations.
- An ill-defined term that means many things to many people. [3]

Complex Logic

An item of logic for which it is not practicable to ensure the correctness of all behaviors[1] through verification alone.

Notes:

- This definition is derived from a combination of the definition of complexity given above and the following definition in DO-254/ED-80 in Appendix C [4], for "simple hardware item": "A hardware item is considered simple if a comprehensive combination of deterministic tests and analyses can ensure correct functional performance under all foreseeable operating conditions with no anomalous behaviour." The conditional clause "if a comprehensive combination of deterministic tests and analyses…" is summarized as "verification" (defined below in this glossary).
- Therefore, in addition to verification (see definition below), the demonstration of correctness of Complex Logic requires a combination of evidence from various phases of the development life cycle, integrated with reasoning to justify the completeness of coverage provided (summarized as development assurance). Examples include the following:
    - evaluation of the system concept (and conceptual architecture)
    - evaluation of the verification and validation plan
    - criticality analysis
    - evaluation of the architecture including requirements allocation
    - evaluation of the system-internal hazard analysis

[1] This refers to behaviour under all foreseeable operating conditions with no anomalous behaviour.

    - validation of requirements and constraints on the design and implementation
    - assessment and audit of all the processes, including supporting processes and management processes
    - certifying[2] organizations developing software
    - evaluation of the independence[3] of the assurance activities

  (See [5] for more detail.)

- Complex Logic is typically produced by techniques such as software or hardware description languages and their related tools. Thus, the assurance of correctness also requires commensurate assurance of the languages and tools.

Design Defect

Frailty or shortcoming of an item resulting from a defect in its concept, and which can be avoided only through an alteration or redesign of the item. [6]

Error

The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition (Definition (8)(A) in [1])

Failure

The termination of the ability of an item to perform a required function. [7]

Note 1: After failure, the item has a fault. [5]

Note 2: “Failure” is an event, as distinguished from “fault” which is a state. [7]

Note 3: This concept as defined does not apply to items consisting of software only. [7]

Note 4: The following definitions represent the perspectives of different disciplines to reinforce the definition given above:

- the termination of the ability of an item to perform a required function (Definition (1)(A) in [1])
- the termination of the ability of a functional unit to perform its required function (Definition (1)(N) in [1])
- an event in which a system or system component does not perform a required function within specified limits; a failure may be produced when a fault is encountered (Definition (1)(O) in [1])
- the termination of the ability of an item to perform its required function (Definition 9 in [1] from “nuclear power generating station”)
- the loss of ability of a component, equipment, or system to perform a required function (Definition 13 in [1] Safety systems equipment in “nuclear power generating stations”)

2 Certification of the development organization should be a continual process of certification and recertification much in the same manner as reactor operators are certified periodically. For example, the capability maturity model integrated certification process developed by the Software Engineering Institute focuses on assessing the capabilities of development.

3 For example, independence can be evaluated through certification of the assurance process for the Complex Logic (e.g., software).

- an event that may limit the capability of equipment or a system to perform its function(s) (Definition 14 in [1] "Supervisory control, data acquisition, and automatic control")
- the termination of the ability of an item to perform a required function (Definition 15 in [1] "nuclear power generating systems")

Failure Analysis

The logical, systematic examination of a failed item to identify and analyze the failure mechanism, the failure cause, and the consequences of failure. (191-16-12 in [7])

Fault

The state of an item characterized by inability to perform a required function, excluding the inability during preventive maintenance or other planned actions, or due to lack of external resources. (191-05-01 in [7])

Note 1: A fault is often the result of a failure of the item itself but may exist without prior failure. (191-05-01 in [7])

Note 2: Following are other definitions, relating "fault" and "defect":

- a defect or flaw in a hardware or software component (Definition 13 in [1])
- a defect in a hardware device or component; for example, a short circuit or broken wire (Definition 9 in [1])

  Synonym: physical defect

Note 3: The following definition is specific to software:

An incorrect step, process, or data definition in a computer program (Definition (7)(A) in [1])

Fault Analysis

The logical, systematic examination of an item to identify and analyze the probability, causes, and consequences of potential faults. (191-16-11 in [7])

Fault Mode

One of the possible states of a faulty item, for a given required function.

Note: The use of the term "failure mode" in this sense is now deprecated.

Fault Modes and Effects Analysis (FMEA)

A qualitative method of reliability analysis, which involves the study of the fault modes, which can exist in every subitem of the item, and the determination of the effects of each fault mode on other subitems of the item and on the required functions of the item. (191-16-03 in [7])

Note: The term "failure modes and effects analysis" is deprecated.

Fault Tree Analysis (FTA)

An analysis to determine which fault modes of the subitems or external events, or combinations thereof, may result in a stated fault mode of the item, presented in the form of a fault tree. (191-16-05 in [7])

Faulty

Pertaining to an item that has a fault.

Feasible

Capable of being done with the means at hand and circumstances as they are. [8]

Other definitions also impose such constraints as "practicably," "reasonable amount of effort, cost, or other hardship" [9], cost-effectiveness [10].

Such constraints distinguish "feasibility" from "possibility."

Hardwired

Pertaining to a circuit or device whose characteristics are permanently determined by the interconnections[4] between components[5] (Adapted from Definition 3 in [1]).

Note:

- The referred-to connections are at the printed circuit board level (or cabinet level), not internal to integrated circuits.

Item (Entity)

Any part, component, device, subsystem, functional unit, equipment, or system that can be individually considered. (191-01-01 in [7])

Notes:

- An item may consist of hardware, software, or both, and may, in particular cases, include people.
- A number of items (e.g., a population of items or a sample) may itself be considered an item.

Mistake

A human action that produces an unintended result (Definition 1 in [1] "electronic computation")

Editorial note (contrary to the note attached to Definition 1 in [1]): In the context of software engineering, this definition should be applied to mistakes concerning requirements development (including elicitation, transformation of intent into requirement or constraint specification, and explicit statement of assumptions (e.g., about the environment) and respective validation.

A human action that produces an incorrect result (Definition 3 in [1] "software")

Note: The fault tolerance discipline distinguishes between the human action (a mistake), its manifestation (a hardware or software fault), the result of the fault (a failure), and the amount by which the result is incorrect (the error). [1]

Editorial note (complementing the note in the previous definition of "mistake"): In the context of software engineering, this definition should be applied to mistakes concerning transformation of requirements specifications and constraints into successive work products and their respective verification.

[4] Examples: Wiring in cabinets; Printed paths in circuit boards

[5] Examples: Relays; AND-gates; OR-gates

Noninterference

Absence of cascading failures between two or more elements that could lead to the violation of a safety requirement [11].[6]

Example 1: Element 1 is interference-free of Element 2 if no failure of Element 2 can cause Element 1 to fail.

Example 2: Element 3 interferes with Element 4 if there exists a failure of Element 3 that causes Element 4 to fail.

Reliability (symbol : $R(t_1, t_2)$)

The probability that an item can perform a required function under given conditions for a given time interval (t_1, t_2). (191-12-01 in [7])

Note: It is generally assumed that the item is in a state to perform this required function at the beginning of the time interval.[7]

- The term “reliability” is also used to denote the reliability performance quantified by this probability (see 191-02-06 in [7]).
- This definition does not apply to items for which development mistakes can cause failures, because there is no recognized way to assign a probability to development mistakes.

Systemic

Embedded within and spread throughout and affecting a group, system, or body. Also see “systemic cause” in [12].

Systematic Failure

Failure, related in a deterministic way to a certain cause, that can be eliminated only by a modification of the design or of the manufacturing process, operational procedures, documentation, or other relevant factors. [7]

Note 1: Corrective maintenance without modification will usually not eliminate the failure cause.

Note 2: A systematic failure can be induced by simulating the failure cause.

Note 3: In International Electrotechnical Commission 61508-4 CDV 3.6.6 [13]: Examples of causes of systematic failures include human mistakes in the following areas:

- the safety requirements specification
- the design, manufacture, installation, and operation of the hardware
- the design, implementation, etc. of the software

Also, see “systemic cause” in [12].

[6] This reference uses the term “freedom from interference.”

[7] For a software component that is faulty to begin with, use of the term reliability is neither meaningful nor helpful; instead, it leads to the misapplication of analysis techniques that served well for traditional hardware.

Verification

The process of providing objective evidence that the software and its associated products conform to requirements (e.g., for correctness, completeness, consistency, accuracy) for all life-cycle activities during each life-cycle process (acquisition, supply, development, operation, and maintenance); satisfy standards, practices, and conventions during life-cycle processes; and successfully complete each life-cycle activity and satisfy all the criteria for initiating succeeding life-cycle activities (e.g., building the software correctly). (Definition 3.1.36 B in [5])

References for Appendix A

1. IEEE Standard 100-2000, "The Authoritative Dictionary of IEEE Standards Terms," 7th edition, 2000.
2. McCabe, T.H., "A complexity measure," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, December 1976.
3. Flake, G.W., *The Computational Beauty of Nature: Computer Explorations of Fractals, Chaos, Complex Systems, and Adaptation,* November 30, 2002, <http://mitpress.mit.edu/books/FLAOH/cbnhtml/>, October 1, 2010.
4. RTCA DO-254/Eurocae ED-80 Standard, "Design Assurance for Airborne Electronic Hardware," Radio Technical Commission for Aeronautics/EUROCAE, April 19, 2000.
5. IEEE Standard 1012-2004, "IEEE Standard for Software Verification and Validation," IEEE Computer Society, 2004.
6. BusinessDictionary.com, "Design Defect," <http://www.businessdictionary.com/definition/design-defect.html>, December 17, 2010.
7. International Electrotechnical Commission, "International Electrotechnical Vocabulary, Chapter 191: Dependability and Quality of Service," IEC 60050-191:1990-12, 1st edition, 1990.
8. WordNet, "Feasible," Princeton University, <http://wordnetweb.princeton.edu/perl/webwn?s=feasible>, December 17, 2010.
9. U.S. Department of Transportation, Federal Highway Administration, "Feasible," <http://www.fhwa.dot.gov/environment/sidewalks/appb.htm>, December 17, 2010.
10. Georgetown University, "Feasible," <http://uis.georgetown.edu/departments/eets/dw/GLOSSARY0816.html>, December 17, 2010.
11. International Organization for Standardization, "Road Vehicles—Functional Safety—Part 1: Vocabulary," ISO/DIS 26262-1, 1st edition, 2009.
12. Chris Eckert, Apollo Associated Services, LLC, "Identification and Elimination of Systemic Problems," *Proceedings of the Society of Maintenance and Reliability Professionals Annual Symposium*, St. Louis, MO, October 20–22, 2009.
13. International Electrotechnical Commission, "Functional Safety of electrical/electronic/programmable electronic safety-related systems – Part 4: Definitions and Abbreviations," IEC 61508-4:2010-04, 2nd edition, 2010.

APPENDIX B: OTHER CAUSES OF FAULTS

In analyzing how safety systems may fail, identifying the cause in terms of a distinct fault mode has been useful when different fault modes could lead to different effects requiring different means and degrees of mitigation. When the differences in effects are unclear, as in the case of the logic fault modes characterized in Section 3.2, a root cause analysis offers more information and utility. Even the identification of intermediate links in the causality chain can lead to specific corrective actions and effective reduction of the possible ways a safety system may fail. Software engineering experts have identified some such causes or types of causes, as discussed below. Some types of causes, as in Sections B.1 and B.2, apply to Complex Logic in general, while others, as in Section B.3, occur more in software.

B.1 Pervasive causes preclude localization of fault mode to item

Systemic root causes of software defects tend to have influences that pervade the product and the process. Such pervasive causes include weaknesses in the culture of an organization or a division of work and people that prevents adequate communication and weaknesses in the process of identifying safety requirements, architectural standards, process standards, management of procurement processes, and the supply chain. Effects cannot be localized as in the case of failure caused by the wear and tear of a hardware component for which fault mode and effects analysis has proven useful.

B.2 Unknown effect of change

Changes, modifications, and updates occur during the usage life of a digital instrumentation and control system; not only do the systems change, but also the resources used in their performance (e.g., people and tools). These changes can introduce defects. For example, at the Jet Propulsion Laboratory, the software used for navigation from Earth to Mars was created 30 years ago. It is robust, but revised from time to time to meet changing needs. One would expect that after 4 or 5 years, it would be defect free. However, over a 30-year period, the number of defects discovered has not decreased. It is reasonable to assume that there are still latent faults. What is the required rigor and granularity in configuration control and management, change control and management, and change impact analysis? There is no proven record of accomplishment to answer these questions.

B.3 Other causes of faults when Complex Logic is in the form of software

In most software systems, there is a significant amount of unpredictability of behavior. This may be surprising at first glance because a software program is a mathematical formula, which should provide only one well-defined result for a given input history. However, inadequately engineered software may depend not only on its functional inputs, but also on hidden, uncontrolled inputs,[1] variability of computation times (e.g., because of the state of the cache memory resulting from the activities of other programs), and other inputs. Then, the behavior of the software is nondeterministic. Some of the contributing causes are discussed below.

Assumptions about the environment of the software: In a software system that is composed of N individual components, each individual component can be designed "correctly," and yet the composition can have flaws. The flaw is generally in the specific assumptions (explicit or implicit) about the operating environment made in the design and construction of each

[1] Examples include the state of the computing hardware such as time, environment variables, or availability of resources

component. A software component can be completely robust with respect to a specific set of environmental assumptions ("environment" refers to the rest of the system, including all other software components, and the rest of the world outside the system itself that can provide events that are visible to the system). Yet, there can be subtle discrepancies in the assumptions between components that could lead to failure of the system as a whole.

Process scheduling: Another source of unpredictable behavior can come from process scheduling,[2] actual concurrency,[3] and the outside environment (providing input triggers that the software responds to, or is sensitive to). This unpredictability makes standard software testing extremely difficult. In general, this is known as the oracle problem. Any specific execution of the system as a whole is generally an unpredictable occurrence of a specific interleaving of small-grained instruction executions. If there is a vulnerability[4] of this type in the system, it can remain hidden for long periods (years), until it finally strikes when just the right interleaving of events happens. Those bugs also tend to be irreproducible even when they are known to be possible (too many things beyond the testers' control need to happen in just the right order).

Change in environmental conditions: A software component can function correctly for a long time, and just a small change in its environment (i.e., not in the component itself) can trigger a system failure. The change could be a change in the scheduler, or the speed of the CPU, or the speed at which external events occur. Concurrency issues make deterministic arguments difficult. Typically, when a race condition causes a problem, simply rebooting the system will remove all evidence of it, and the same problem may never strike again. The software may appear to work correctly and will be fully functional most of the time, even though it has vulnerabilities that can strike unpredictably at some point in its lifetime when the right conditions occur.

Unpredictable effects of triggered faults: Most importantly, the possible effects of a triggered fault (defect) (for example, one that allows a subtle race condition, which can cause data corruption somewhere) are generally unpredictable. There are many examples of major system failures that were caused by seemingly small one-line bugs. One example is the AT&T disaster in January 1990, described in Section 3.3.2, where intermittent failure caused a major service outage on the entire long-distance U.S. telephone network. Other examples of seemingly "small" problems that lead to large failures abound. The National Aeronautics and Space Agency, for instance, lost an expensive spacecraft (the Mars Global Surveyor spacecraft) recently because of 1 byte of memory having the wrong value. These observations emphasize that the consequences of even very small defects in software can be quite unpredictable. So even if we could assign a reasonably accurate probability of a defect existing in a software system (which we cannot), that in itself does not help us assess the potential effect of such a defect. The effect could be benign, or it could bring down the entire system in an unforeseeable way.

Mitigating measures are useful but limited: Intervention mechanisms outside the software (and independent of it) can (and should) be provided to mitigate the effects mentioned above. Strong mechanisms (e.g., the use of memory protection and partitioning kernels) should be provided to prevent the propagation of failures within software systems. Detection through runtime monitoring and other fault monitoring techniques can and should be exploited. Prevention techniques through defensive coding techniques, consistency checking, and static source code

[2] This refers to the way in which a process scheduler interleaves the execution of multiple tasks on a single processor.

[3] Multiple central processing units (CPUs) are running multiple tasks in parallel at unpredictable relative speeds.

[4] It is often the case.

analyzers can and should be used. Even then, mistakes can occur, but their likelihood and effects cannot be determined.

Glossary

Latent Fault

An existing fault that has not yet been recognized. (191-05-20 in [1])

Oracle

An oracle is a mechanism used by software testers and software engineers for determining whether a test has passed or failed. It is used by comparing the output(s) of the system under test, for a given test case input, to the outputs that the oracle determines that product should have. (Adapted from [2])

Examples:

- an expected result
- a requirements document
- a previous version of the product
- a human judgment

The Oracle Problem

Whether a decision procedure can be defined for interpreting the results of tests. (Adapted from 3])

Examples:

- Requirements specifications are incomplete, contain conflicting information, or are ambiguous.
- Expected results in a test case detail only a small portion of what is expected (the tiny portion of the application and functionality the test is designed to expose).

Notes:

- The oracle problem is that all oracles are fallible.
- Oracles are hard to find and require work to use effectively.

References for Appendix B

1. International Electrotechnical Commission, “International Electrotechnical Vocabulary, Chapter 191: Dependability and Quality of Service,” IEC 60050-191:1990-12, 1st edition, 1990.
2. Wikipedia, “Oracle (software testing),” <http://en.wikipedia.org/wiki/Oracle_(software_testing)>, December 17, 2010.
3. Machado, P.D.L., “Testing from Structured Algebraic Specifications: The Oracle Problem,” May 5, 2006, <http://www.lfcs.inf.ed.ac.uk/reports/00/ECS-LFCS-00-423/>, December 17, 2010.

APPENDIX C: STATE OF THE ART IN FMEA FOR SOFTWARE

Fault modes and effects analysis (FMEA) and fault tree analysis (FTA) have been used as a part of system-internal hazard analysis (HA) in nonnuclear application domains for identifying software requirements (e.g., monitoring and detection of a fault and mitigating its effect). Unlike hardware and system FMEA, a software FMEA cannot be easily used to identify system-level hazards [1]. Software analyses, reviews, and tests directed at finding faults in the software are not considered to be a direct part of software HA (i.e., verification and validation activities are not considered to be part of the HA). When performed on software, the HA considers only the following two questions [2][1]:

- If the software operates correctly (i.e., follows its specifications), what is the potential effect on system hazards?
- If the software operates incorrectly (i.e., deviates from specifications), what is the potential effect on system hazards?

Whereas the FMEA method looks at all faults and their effects, the FTA is limited to analysis of faults leading to events of interest (e.g., safety related or of the highest criticality) [3]. This appendix will briefly review specific viewpoints outside the U.S. commercial nuclear power plant industry that are relevant to a determination of the suitability of software FMEA, software FTA, or an FMEA-FTA combination, in the safety assurance of Complex Logic.

C.1 Fault Modes and Effects Analysis

FMEA is characterized as a bottom-up analysis technique that identifies the consequences of the credible fault modes for the system. The results of the FMEA are documented in a tabular format. However, this representation makes it difficult to understand the logical relationships among the causes of a failure [4] and does not group together the items causing the effects. For example, when performed on software, FMEA does not consider the correctness of algorithms or problems [5] resulting from design mistakes, but assumes that every variable may fail without regard to cause. FMEA is independent of two essential but different kinds of analysis: how the software design meets requirements and the adequacy of the requirements themselves[2][3].

Goddard[4] notes that the intent of software FMEA is not to verify the quality of the software, but to provide assurance that should something go wrong (whether the problem is induced by hardware or software), the software will detect it and maintain the system in a safe state.[5] According to Goddard [1], two types of software FMEA are used in embedded control systems: a system software FMEA and detailed software FMEA. A system software FMEA, performed on the architecture, can support its evaluation for effectiveness, but does not examine the implemented functional code. The detailed software FMEA, performed on the code, can be used for identifying unexpected[6] paths, which could lead to an adverse effect on safety. However,

1 This reference assumes that the computer hardware operates without failure. In addition, it assumes that a separate system HA and a separate hardware HA are performed.

2 Private communications with NRC in a teleconference with Ram Chillarege September 1, 2010

3 Private communications with NRC in a teleconference with Peter Goddard, September 10, 2010

4 Private communications with NRC in a teleconference with Peter Goddard, September 10, 2010

5 This amounts to a system hazard analysis.

6 The examination is limited to documented design. However, there could be other unexpected paths not visible in the documented design.

compared to the system software FMEA, the detailed software FMEA can be lengthy and labor intensive [1]. For example, if the system-level software FMEA consumes 6 weeks of labor, the detailed FMEA will probably take 6 or 7 man-months of labor [1]. In Goddard's experience[7], FMEA is more effective at the system level; its suitability and effectiveness at the detailed software level are questionable.

The definition of fault modes is one of the hardest parts of the FMEA of a software-based system. Unlike for hardware, a complete list of fault modes for software cannot be assembled [4, 7]. Software fault modes generally are unknown ("software modules do not fail, they only display incorrect behaviour" [7]). The analysts must apply their own knowledge about the software and postulate the relevant fault modes [8]. Banerjee [9] provided an insightful look at how teams should use FMEA in software development. However, the effectiveness depends on the domain knowledge of the review team and the accuracy of the documentation [4]. In particular, inadequate software responses to extreme conditions and boundary cases are of concern [4]. Similarly, Fenelon and McDermid [10] and Pfleeger [11] pointed out that FMEA is highly labor intensive and relies on the experience of the analysts.

The Radiation and Nuclear Safety Authority of Finland stated [7] that FMEA cannot alone provide the necessary evidence for the qualification of software-based safety critical applications in nuclear power plants, but the method should be combined with other safety and reliability engineering methods.

C.2 Fault Tree Analysis

FTA is characterized as a top-down analysis technique to identify the contributing elements that could cause the system-level undesired events (top events) [12]. Analysts have used FTA to discover design defects during the development of a system and to investigate the causes of accidents or problems that occur during system operation [13]. Software developers have used FTA to discover software defects [13, 14]. It has also been used for verifying software code, but it has been difficult and labor intensive to use it for large software [4]. A limitation of FTA is that the top event can describe only a known failure [13]. Because of lack of experience, it is difficult for analysts to select[8] an adequate set of top events, which results in the risk of leaving critical, system-level undesired events out of the analysis [12]. In addition, it cannot identify the effects of "sneak paths" not reflected in the documented design. Like FMEA, FTA is only as good as the domain and system expertise of the analyst.

Leveson states [15] that the purpose of HA is to discover or identify safety requirements (or derived requirements including design constraints and implementation constraints)—not to ensure that the logic will not lead to unsafe system failure.

According to some experts, FTA has shown weaknesses when the code has loops, but loops are common in embedded software [16]. The Korean Atomic Energy Research Institute (KAERI) has reported [17–21] the use of carefully crafted FTA (different from the FTA used in traditional hardware) on a critical software module as a technique redundant and complementary to other techniques used in the project, including HAZOP, formal verification, and testing. The software FTA revealed a defect that was not found in formal verification and testing [17]. KAERI found the redundancy worthwhile.

Fault trees are static approaches that cannot reach all the dynamic aspects of the software.

[7] Private communications with NRC in a teleconference with Peter Goddard, September 10, 2010

[8] Often, the range of scenarios leading to unwanted events is too large to consider exhaustively, and the analyst does not have enough explicit information about their likelihood to make a well-informed selection.

C.3 <u>FTA-FMEA Combination</u>

Several experts and expert groups have considered the integrated use of both bottom-up and top-down techniques to fill the gaps when both techniques are applied separately. There are two ways to perform this integrated analysis [12]:

(1) Bottom-up [16, 22–25]

In this method, the FMEA is taken as the main method and then followed with software FTA as a supplement. The method is described as follows:

1. Identify the fault modes.
2. Evaluate the impact of the fault modes on the system and the severity of the impact.
3. Select the effects with greater severity as top events for the software FTA.
4. Determine the actions needed according to the causes of the fault modes.

(2) Top-down [8, 26]

In this method, the FTA is the main method, followed with FMEA as a supplement. This method is applied at the design phase. The method is described as follows:

1. Identify the top events.
2. Evaluate the minimal cut-sets and important bottom events.
3. Perform the FMEA with the most important bottom events.
4. Determine the actions needed with the new failure effects as top events.
5. Continue analysis.

Some experts prefer the bottom-up FMEA-FTA combination [16, 22–25], while others prefer the top-down FTA-FMEA combination [8, 26]. In the latter case, the preliminary FTA and resulting minimal cut-sets direct the identification of failure modes to those that are most significant for the system reliability. Then, the effects analysis of these failures steers the refinement of the fault trees and the final detailed FMEA [7].

References for Appendix C

1. Goddard, P.L., “Software FMEA Techniques,” *Proceedings of the Annual Reliability and Maintainability Symposium*, pp. 118–123, 2000.
2. U.S. Nuclear Regulatory Commission, “Software Safety Hazard Analysis,” NUREG/CR-6430, Washington, DC, February 1996 (Agencywide Documents Access and Management System (ADAMS) Public Legacy Library Accession No. 9602290270).
3. U.S. Federal Aviation Administration, *System Safety Handbook*, Washington, DC, December 2000.
4. Lutz, R.R. and Woodhouse, R.M., “Experience Report: Contributions of SFMWA to Requirements Analysis,” *Proceedings of ICRE 1996*, Colorado Springs, CO, pp. 44–51, April 1996.
5. Czerny, B.J., D’Ambrosio, J.G., Murray, B.T., and Sundaram, P., “Effective Application of Software Safety Techniques for Automotive Embedded Controls Systems,” 2005 SAE World Congress, Detroit, MI, April 11–14, 2005.
6. Goddard, P.L., “Validating the Safety of Embedded Real-Time Control Systems Using FMEA,” *Proceedings Annual Reliability and Maintainability Symposium*, pp. 227–230, 1993.
7. Pentti, H. and Atte, H., “Failure Mode and Effects Analysis of Software-Based Automation Systems,” p.37, STUK—Radiation and Nuclear Safety Authority, Helsinki, August 2002.

8. McDermid, J.A., Nicholson, M., Pumfrey, D.J., and Fenelon, P., “Experience with the application of HAZOP to computer-based systems,” *Proceedings of COMPASS 1995*, pp. 37-48, IEEE, Gaithersburg, MD, 1995.

9. Banerjee, N., “Utilization of FMEA Concept in Software Lifecycle Management,” *Proceedings of Conference on Software Quality Management*, pp. 219–230, 1999.

10. Fenelon, P. and McDermid, K.A., “Integrated Techniques for Software Safety Analysis,” *Proceedings of the IEE Colloquium on Hazard Analysis*, Institution of Electrical Engineers, 1992.

11. Pfleeger, S.L., “Software Engineering: Theory and Practice,” Prentice Hall, Upper Saddle River, NJ, 1998.

12. Hong, Z. and Binbin, L., “Integrated Analysis of Software FMEA and FTA,” *Proceedings of the 2009 International Conference on Information Technology and Computer Science*, 2009.

13. Lutz, R.R. and Nikora, A., “Failure Assessment,” *First International Forum on Integrated System Health Engineering and Management in Aerospace (ISHEM)*, Napa Valley, CA, November 9, 2010.

14. Towhidnejad, M., Wallace, D.R., Gallo, A.M., Jr., “Validation of Object Oriented Software Design with Fault Tree Analysis,” *Proceedings of the 28th Annual NASA Goddard Software Engineering Workshop*, pp. 209–215, December 2003.

15. Leveson, N.G., “Safeware: System Safety and Computers,” ISBN 0-201-11972, Addison-Wesley Professional, Reading, MA, April 17,1995.

16. Maier, T., “FMEA and FTA to Support Safe Design of Embedded Software in Safety-Critical Systems,” CSR 12th Annual Workshop on Safety and Reliability of Software Based Systems, Bruges, Belgium, 1995.

17. Park, G.Y., Lee, J.S., Cheon, S.W., Kwon, K.C., Jee, E., Koh, K.Y., “Safety Analysis of Safety-Critical Software for Nuclear Digital Protection System,” *The 26th International Conference on Computer Safety, Reliability and Security*, Nuremberg, Germany, pp. 148-161, September 2007.

18. Kwon, K.C. and Park, G.Y., “Formal Verification and Validation of the Safety-Critical Software in a Digital Reactor Protection System,” NPIC & HMIT 2006, Albuquerque, NM, November 12-16, 2006.

19. Park, G.Y., Koh, K.Y., Jee, E., Seong, P.H., Kwon, K.C., and Lee, D.H., “Fault Tree Analysis of KNICS RPS Software,” *Nuclear Engineering Technology*, Vol. 41, No. 4, May 2009.

20. Kwon, K.C. and Lee, M., “Technical Review on the Localized Digital Instrumentation and Controls Systems,” Special Issue in Celebration of the Korean Nuclear Society, *Nuclear Engineering Technology*, Vol. 40, No. 5, August 2008.

21. Lee, J.S., Lindner, A., Choi, J.G., Miedl, H., and Kwon, K.C., “Software Safety Lifecycles and the Methods of a Programmable Electronic Safety System for a Nuclear Power Plant,” *Proceedings of SAFECOMP 2006*, LNCS 4166, pp. 85–98, 2006.

22. Lutz, R.R. and Shaw, H.Y., “Applying Integrated Safety Analysis Techniques (Software FMEA and FTA),” Jet Propulsion Laboratory (JPL) D-16168, Pasadena, CA, November 30, 1998.

23. Lutz, R.R. and Woodhouse, R.M., “Requirements Analysis Using Forward and Backward Search,” *Annals of Software Engineering*, pp. 459–475, Pasadena, CA, 1997.

24. Hecht, H., “A System Approach to Exception Handling,” *Third Annual Conference on Systems*, pp. 190–195, April 2008.

25. Hecht, H., Xuegao, A., and Hecht, M., “Computer Aided Software FMEA for Unified Modeling Language Based Software,” *Proceedings of the 2004 Annual Symposium—RAMS*, pp. 243–248, August 24, 2004.

26. Hassan, A., Goseva-Popstojanova, K., Ammar, H., “Methodology for Architecture Level Hazard Analysis, A Survey,” *ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 2003)*, pp. 68–70, Tunis, Tunisia, July 14–18, 2003.