Functional Hazard Analysis for Engineering Safe Software Requirements

Vu N. Tran
US Navy
China Lake, CA, USA
vu.n.tran1@navy.mil

Long V. Tran
University of Southern California
Los Angeles, CA, USA
longvtra@usc.edu

Viet N. Tran
University of Southern California
Los Angeles, CA, USA
vntran@usc.edu

2021 4th International Conference on Information and Computer Technologies (ICICT) | 978-1-6654-1399-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICICT52872.2021.00031

***Abstract*—Functional Hazard Analysis (FHA) is an inductive hazard analysis method used to evaluate the potential causes and hazardous consequences of a system's functional failures. Software safety uses the FHA to assess the software contribution to the system hazards and identify software improvement opportunities. The FHA integrates risk-driven and quality assurance-driven approaches into a single safety analysis method for safe software requirements engineering. Our paper reviews the use of the MIL-STD-882E FHA to support the development of software requirements for safety-critical systems. First, we summarize the distinguishing features of the FHA as a hazard analysis method. Second, we explain how to use this method to identify software safety risks and recommend software safety improvements in the software requirements engineering process.**

Keywords—Functional Hazard Analysis, Software Safety, Software Requirements, Safety Requirements Engineering

I. Introduction

Software's importance in safety-critical systems continues to increase at a rapid rate. Military aircrafts development, for instance, have reported a tenfold increase in software embedded in aircrafts within the last 2 decades. Concerns for random hardware component failures have quickly given way to concerns for complex interactions of running software within these systems. While achieving the required high hardware reliability necessary for safety assurance is often feasible, achieving the same software reliability level is difficult, if not impossible [1-2]. The introduction of software with emergent capability such as machine learning adds a new complexity level to system safety assurance and certification [3-5]. As a result, software safety has occupied a prominent position within the system safety discipline [6]. Software safety aims to bring formal safety risk management practices into all aspects of a software lifecycle, from requirements definition to design, coding, testing, integration, certification, deployment, and operation (Fig. 1). The objective of software safety is to help identify and eliminate software-induced system hazards, i.e., hazardous conditions inherent in the system due to software presence or control the software risk contribution to these hazards. The goal of software safety is to ensure minimal software risk contribution to system hazards.

System safety research finds that poorly developed requirements are the dominant causal factor of high-profile system failures [1-2][7]. For software-intensive systems, incomplete, wrong, and unsafe software requirements disproportionally contribute to system accidents [1]. It is logical to expect that requirement-level hazard analysis methods, such as the Functional Hazard Analysis (FHA), are of great interest to industries developing safety-critical systems. Industries that put a premier on software safety include automobile, aviation, space, nuclear power plant, military weaponry, and robotic surgery. Safety standards have incorporated different FHA variations over the years [6][8]. The next section of the paper discusses the challenges with engineering safe software requirements.

II. Challenges of Engineering Safe Software Requirements

Today's software engineering processes primarily rely on software testing to assure quality. Software under-development is subjected to extensive testing at the function-level, unit-level, subsystem-level, system-level, and acceptance-level to ensure meeting all required functional and non-functional requirements described in their respective Software Requirements Specifications. Code reviews, static code analysis, integration testing, structural analysis, regression testing, equivalent partitioning testing, software quality assurance, and independent verification & validation are additional quality assurance techniques often used in software engineering [9]. The quality assurance-driven approach is taught extensively in software engineering curriculums. However, the quality assurance-driven approach has its limits. Research in software reliability and system/software safety recognizes that high software reliability is fundamentally challenging to achieve because of the unique ways that software fails [1][10]. Achieving fault-free software remains an elusive goal in software engineering for nearly half a century.

Software redundancy, a common approach to improving software reliability and safety, also has its limits. Redundant software solutions such as N-version Programming and Recovery Block provide some safeguards for implementation faults. However, they are not effective against the risk of defective requirements, the dominant cause of software-related system failures [11]. Poorly implemented redundant solutions can add more complexity to the system resulting in reduced safety [12]. Methods that focus on improving the quality of the software requirements promise a better alignment between problem and solution in system safety [1][6]. The FHA is one such method (Fig. 1).

Adopting hazard analysis methods in safety-critical software requirements engineering in general and the FHA expressly has been limited [13-14]. In a recent literature review study [13], Martins and Gorschek report only 3 out of 165 experience

978-1-6654-1399-2/21/$31.00 ©2021 IEEE
DOI 10.1109/ICICT52872.2021.00031

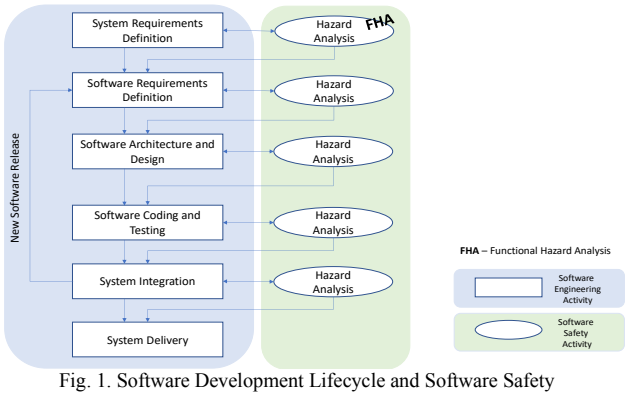

Fig. 1. Software Development Lifecycle and Software Safety

articles on hazard analysis methods published between 1983 and 2014 mentioned the FHA. The researchers found that practitioners preferred traditional methods such as Fault-Tree Analysis (FTA), Failure Mode and Effects Analysis (FMEA), Hazard and Operability Studies (HAZOPs), Failure Mode, Effects, and Criticality Analysis (FMECA), and Preliminary Hazard Analysis (PHA). Outside of these methods, little information available on the practice of the rest [13]. In their follow-up interviews with system safety experts, the authors found practitioners' preference for a particular practice method is based on familiarity or standard compliance requirements rather than fit-for-use. Their findings confirm that the FHA is still a relatively unknown hazard analysis method.

In a 2020 follow-up study on the same topic [14], the authors report that less than one-fifth of the 11 organizations involved with safety-critical systems development they interviewed has successfully incorporated hazard analysis methods into their software requirements engineering process. The authors note that one contributing factor is the lack of experience and education in software engineering among the safety experts interviewed. From their study, the authors call for tighter integration between software safety and engineering and more diversification in the publication of software safety methods in practice. They also call for greater participation of practitioners in safety requirements research to validate less-known methods such as the FHA to drive greater adoption [13-14].

Reflecting on the state of software safety, Leveson [2] points out that many challenges remain open today. These challenges include confusion about reliability and safety, lack of unrealistic risk assessments, inadequate incorporation of software safety in the software processes, faulty safety assumption in software reuse, poor mission and safety design tradeoff decisions, and weakly enforced safety standards.

Besker, Martini, and Bosch [15] report a correlation between compliance to safety regulations and software technical debt, i.e., measuring the future costs of maintaining the software. Their study finds that safety regulations disincentivize pro-active software refactoring, leading to delayed improvements and significantly higher total software rework costs. That is increasing safety regulations can lead to increasing technical debt in software. The study also finds that delayed incorporation of safety requirements mandated by regulations will add significant costs to the overall project.

The challenges identified in the studies above reflect much of what we observed as practicing software safety engineers and researchers. We found the FHA to be a suitable hazard analysis method for safe software requirements engineering. However, we have discovered many software organizations continue to struggle with understanding and adopting this method. Difficulties reported include determining the hazardous functions, understanding the system effects, and the interdependency between subsystem functions [16]. We recognize the need for a better understanding of the method within the software engineering community. This paper describes a version of the FHA, the MIL-STD-882E FHA [6], that we have used extensively in software safety projects over the years. The MIL-STD-882E FHA is a system-level hazard analysis method used in aviation, aerospace, and defense sectors in the US [6][17]. We will refer to the MIL-STD-882E FHA as the FHA going forward.

In the subsequent sections, we will describe the FHA's distinguishing features and explain how to use this method in the software requirements engineering process. The FHA can be used to analyze a whole system or individual subsystems [6]. This paper focuses on how to use the method to improve the requirements of software subsystems embedded within a safety-critical system. This application of the FHA is the most relevant to safe software engineering.

III. Distinguishing Characteristics of the FHA

As a functional safety method, the FHA concerns with potential software behavioral failures and their contributions to system hazards [18]. Functional failures that do not result in a system mishap are not relevant to the FHA. The FHA takes a list of system functions allocated to software, identifies those hazardous functions whose failure can lead to one or more system mishaps, evaluates the potential consequences of the failure, and recommends appropriate safety measures to eliminate or control the hazard risks. The FHA helps develop the safety requirements for the Software Requirements Specification (Fig. 1).

The FHA combines both quality assurance-driven and risk-driven approaches to software safety. For software quality assurance, the method defines additional safety assurance tasks, e.g., safety design review, code review, coding, testing, integration, and certification to incorporate into the implementation of safety-significant functions. These assurance tasks help enforce a safety implementation rigor level appropriate for the safety criticality of the hazardous functions identified [19-21].

As a risk-driven method, the FHA focuses on proactive hazard risk identification, assessment, and treatment [22]. The hazard analysis portion of the method helps identify and evaluate potential software risks inherent within the system requirements. The method then supports the generation of safety measures and their translation into formal software requirements to drive the design and implementation of improved software solutions that are risk-reduced [16]. The integration of both the quality assurance-driven and risk-driven approaches distinguishes the FHA from other hazard analysis methods.

The FHA is an inductive method [16] that works from the bottom-up, starting with analyzing all possible software failure scenarios associated with each system function allocated to the software and ending with identifying the top-level system hazards that need to be eliminated or controlled. The FHA allows for the discovering of new hazardous conditions (hazards) during the analysis process.

The FHA does not rely on probability for software risk assessment. Not using probability estimation reflects the software safety community's recognition that quantitative estimation of software risk is highly risky [2][23]. Instead, the FHA uses multiple qualitative factors to assess software contribution risk. Example factors are 1) the potential consequence of the software failure, 2) the structure of the software control that implements the system functions, 3) the risk-mitigating measures introduced, and 4) the level of safety assurance rigor of the software development activities [6].

The FHA assesses potential risk reduction for recommended safety improvements using two complementary residual risk assessments. The worst-case risk assessment assumes a failure to comply with the required safety assurance rigor, resulting in increased risk. The best-case risk assessment assumes full compliance with the recommended safety measures and the required safety-assurance rigor, resulting in reduced risk. Using these two assessments allows evaluating the potential risk reduction from the safety improvement effort.

The FHA supports iterative software development. Each development iteration can uncover new risks, recommend new requirements, introduce new safety designs, and discover new defects for fixing. As these changes make their way through the software change control process, the FHA is used to understand the risk implication, identify safety improvement opportunities, and assess their costs and benefits. Approved changes become new software requirements for a future development iteration. Approved changes are also translatable to new hardware requirements or human procedures.

Fig. 2 outlines the scope of the FHA in requirements engineering. The FHA process works between the system requirements and software requirements engineering processes. It supports identifying potential hazardous system functions allocated to the software and recommending safety measures to mitigate the safety risks to drive safety requirements development. The FHA supports the classification of software requirements into three distinct categories of safety-significant functions: hazardous system functions, also known as contributing safety-significant functions (CSSFs), tailored safety measures, also known as mitigating safety-significant functions (MSSFs), and generic safety measures, also known as generic risk-mitigating functions (GSSFs) derived from safety best-practice available from various safety handbooks [24]. MSSFs are safety measures designed to mitigate the risk of specific failure causes by the CSSFs. GSSFs are generic best-practice safety measures. The CSSFs, MSSFs, and GSSFs are translated into software requirements, i.e., CSSRs, MSSRs, and GSSRs, in the Software Requirements Specifications.

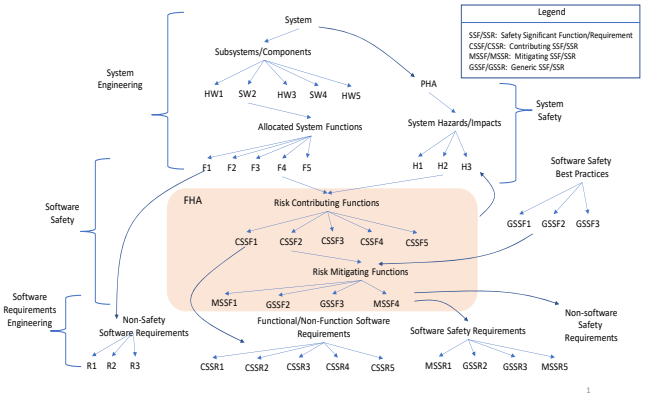

Fig. 2. Safe Software Requirements Engineering with the FHA

IV. The FHA Process

1. Identify Hazardous Functions Allocated to Software
2. Determine Software Safety Criticality
3. Determine Safety Quality-Assurance Rigor Level
4. Identify Safety Improvement Opportunities
5. Evaluate Risk Reduction Potential from the Improvement Opportunities
6. Recommend Safety Improvements

The FHA can work at the system or subsystem level in system safety [6]. For software safety, the FHA works at the software subsystem, or software, level. The pre-conditions for starting the FHA process include having one or more software subsystems with allocated system functions and an initial list of system hazards and their mishap consequences.

Implementing the FHA requires the use of a set of mapping tables. The Appendix contains an incomplete set of tables available in the MIL-STD-882E standard [6] as examples. Key FHA mapping tables include:

1. Software Control Categories Table (Table I) for determining software safety criticality.
2. Software Safety Criticality Matrix and Level of Rigor (LOR) Table (Table II) for determining software safety criticality and the safety assurance levels for risk reduction.
3. Risk Assessment Mapping Table (Table III) for assessing the risk contribution of software.

A. Identify Hazardous Functions Allocated to Software

The FHA process starts with the identification of hazardous system functions (CSSFs) allocated to software subsystems. For each allocated system function, the process identifies one or more possible functional failure scenarios leading to a system mishap. The FHA failure scenario comprises three components: the system phase when the failure occurs, the failure mode, and the causal factors.

A failure mode is a category of failure that can trigger a particular hazardous condition leading to a system mishap. Causal factors are specific functional failure instances of a particular failure mode. The existence of causal factors for a failure mode suggests this mode may be relevant to safety analysis. Each failure mode can have zero or more causal

factors. Failure modes without a causal factor are not relevant for further hazard analysis.

The FHA uses four functional failure modes 1) Function Fails to Execute, 2) Function is Degraded, 3) Function Performed Early or Late, and 4) Function Performed Out-of-Order [6]. System functions with at least one failure mode with one or more associated causal factors leading to a hazardous condition and ultimately a mishap are CSSFs. Functions whose failure modes do not trigger a hazardous condition are not CSSFs and are also not relevant for further hazard analysis.

For each identified failure scenario, the FHA process assesses the system hazard that could be triggered, leading to a system mishap. Functions with one or more failure scenarios that can lead to one or more system mishaps are considered CSSFs. Determining CSSFs only requires the relationship between these failure scenarios and the system mishaps to be "credible" qualitatively.

Fig. 3 summarizes the process of determining hazardous functions of a software subsystem from an initial list of allocated system functions.

B. Determine Software Safety Criticality

Software safety criticality denotes the software's significance to system safety from the context of the system functions it supports. The FHA combines the worst credible mishap severity associated with each hazardous system function and software's role in implementing it into a single software safety criticality classification for the function. For example, a system function implemented in software whose failure can lead to a significant system mishap will have a high software safety criticality. This same function implemented in hardware with minimal software control will have a low software safety criticality. Software-intensive functions that are highly safety-critical are good candidates for software safety improvement.

The FHA determines software safety criticality analysis at the function level and the software subsystem level.

1) Determine Software Safety Criticality at the Function Level

The FHA measures a function's safety criticality by its failure modes' highest mishap severity level. It measures the

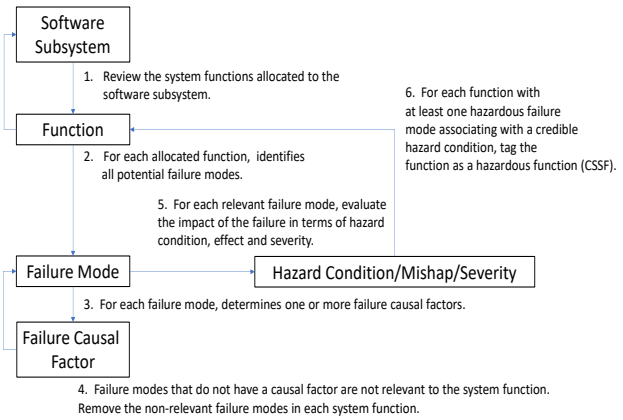

Fig. 3. Determine the CSSFs

software control criticality using five software control categories (Table I). Autonomous (AT) control means that the software implementing or controlling the hazardous function has full control over its execution. Semi-Autonomous (SAT) control means that the software has full control over the function. However, functional failure can be detected and intercepted by an entity other than that software to prevent a system mishap. Redundant Fault-Tolerant (RFT) control means that the software requires another entity's concurrence to control or execute the hazardous function. In addition to redundant decision-making, the controlling structure is fault tolerant. Influential (INF) control means that the software has limited control over the hazardous function. While the software can relay safety-related data about the function's state, neither its failure nor the data provided triggers immediate safety actions. No Safety Impact (NSI) means the software has no impact on the hazardous function [6].

The FHA uses the Software Criticality Index (SWCI) to measure software safety criticality (Table II). Hazardous functions with high impact severity controlled entirely in software receive a high SWCI classification, e.g., SWCI 1. In contrast, functions implemented in hardware that rely on status reporting software will receive a low SWCI, e.g., SWCI 4.

2) Determine Software safety Criticality at the Software Level

System functions supported by the same software do not fail independently. Their implementations share a single run-time address space at the software process level. A software process failure for any reason can render all implemented functions unavailable. A non-hazard function can accidentally cause one or more consequential hazardous functions of the same software to fail at run-time. Similarly, a faulty function can trigger a process deadlock or starve out all available system memory at run-time, making other non-operable or unavailable functions. To account for common cause failures in the software, hazardous functions within a software subsystem will share the highest software safety criticality value at the software level. For instance, a software subsystem controlling six SWCI 4 CSSFs and one SWCI 1 CSSF is an SWCI 1, meaning that the six SWCI 4 functions may have the same risk level as the SWCI 1 CSSF due to common cause failure. Reducing the risk level of the SWCI 1 CSSF reduces the common cause failure risk for the other functions. Physically isolating high software safety-critical functions away from the rest of the system functions is a crucial software safety design practice to minimize the risk of common cause software failure [24].

C. Determine Safety Quality-Assurance Rigor Level

The Level of Rigor (LOR) represents a set of safety assurance tasks designed to assure the hazardous functions' quality implementation. These safety assurance tasks complement the software quality assurance tasks typically performed in a software project. Higher LOR translates to more safety assurance tasks, and lower LOR translates to fewer tasks. Table II (Appendix) provides a mapping between the SWCI and the LOR classifications. The Implementation Guide [25] of the MIL-STD-882E standard offers a table that maps each LOR classification to a list of best-practice safety assurance tasks tailorable for specific software projects.

This FHA step starts with determining the LOR classification for each CSSF based on its safety criticality level. The next step is to determine the software's LOR classification. The software's LOR is equal to the highest LOR applied to across the software's hazardous functions.

It is crucial to recognize that safety assurance tasks are not the same as typical software quality assurance tasks, although they could be overlapping. This difference stems from the often lack of focus on the system safety quality in a regular software development effort, which software safety as a discipline attempts to remedy. Safety assurance tasks strictly focus on the safety aspect of the software. For instance, in Table II (Appendix), the phrase "analysis of requirements" refers to the software requirements' hazard analysis. The phrase "analysis of design" refers to the software design's hazard analysis. The phrase "conduct in-depth safety testing" refers to performing code-level safety analyses and tests such as software structural coverage analysis [19], fault-injection testing [20], and mutation testing [21].

The LOR classification is translatable into the Level of Effort (LOE) needed to complete all the safety assurance tasks. Comparative, parametric, function-point, or story-point estimating methods are common LOE sizing methods in software engineering. Calculating expected LOE needs to factor in other aspects of software development planning, including organization capacity and productivity, thus usually not included in the FHA worksheet used primarily for safety requirements generation.

D. Identify the Improvement Opportunities

This FHA step identifies candidate safety measures for hazardous functions to reduce their contribution to system hazards. There are two sub-steps to perform: identify which hazardous functions need additional safety measures and determine the appropriate safety options for the selected hazardous functions that need to be improved.

1) Identify Safety Improvement Needs

The simplicity principle favors simple software functions over complex functions that deliver the same features [12]. Complex systems that are more costly to build, maintain and use are not inherently safer. Adding unneeded safety measures can lead to a more complex system that is less safe. Only hazardous functions whose run-time behaviors cannot be assured need additional safety measures. Hazardous functions that can be assured by following the LOR tasks may not need additional safety measures.

Hazardous system functions (CSSFs) that may not require additional safety measures are:

- CSSFs that are simple to understand and verify.
- Legacy CSSFs that are already in use or tested extensively in a similar environment and usage context.
- CSSFs implemented by reliable hardware.
- CSSFs have a low hazard contribution risk.

Hazardous functions that may need additional safety measures are:

- CSSFs that are complex to understand and test.
- CSSFs control potentially unreliable components.
- CSSFs control third-party off-the-shelf components.
- CSSFs have high software safety criticality.
- CSSFs have a high hazard contribution risk.

2) Identify Safety Improvement Opportunities

There are different approaches to enhance software for safety, with varying degrees of complexity and effectiveness. Candidate approaches, adapted from the MIL-STD-882E standard's Order of Precedence [6], include (with one being the most preferred):

1. Eliminate high-criticality hazardous functions that are not essential to the mission since they have a high risk-to-value ratio.
2. Assign hazardous functions to software implementation with a reliable track record, e.g., legacy or COTS product.
3. Decompose hazardous functions into simpler functions can be assured in the implementation.
4. Define stringent safety pre-conditions (or interlocks) for activating the hazardous functions.
5. Limit the accessibility of hazardous functions at run-time.
6. Adopt industry best-practice software safety measures (GSSFs).
7. Add safety features that can detect the failure conditions and prevent their manifestation into a system mishap.
8. Add safety features that can detect the failure conditions and limit the severity of the system mishap.
9. Display cautions and warning signs to inform system users of the risk of using the hazardous functions.
10. Provide procedures and training on how to handle hazardous functions implemented in software.

The addition of safety measures should not result in the risk transferring or risk compounding problem. The risk transferring problem refers to the migration of the same risk contribution level from the CSSFs to the MSSFs or GSSFs. The risk compounding problem refers to the increase in total software risk contribution due to adding MSSFs and GSSFs. The MSSFs and GSSFs should not be more hazardous than the CSSFs they control, and that their addition should reduce the overall risk contribution of the software to the system hazards.

E. Evaluate Risk Reduction Potential from the Improvement Opportunities

This FHA step assesses the risk reduction potential of the recommended safety improvements. Software risk contributions assessment receives limited coverage in the MIL-STD-882E standard [6]. However, we have found this step vital in applying FHA for software safety. The decision to recommend a safety measure depends on understanding how it helps reduce the software risk contribution. Two methods for estimating the remaining, or residual, software risk are the worst-case and best-case risk assessments.

The worst-case risk assessment assumes that the safety assurance effort will have failed to meet the LOR requirements. As a result, improvement credits are not available to reduce the residual risk. In some situations, the residual risk assessed can

be higher than initially estimated due to the additional code added whose correctness could not be verified. Table III (Appendix) provides an example of how residual software risk can be determined when the software implementation process fails to comply with the LOR requirements.

The best-case risk assessment assumes that the hazardous system function and associated safety measures have been successfully implemented, assured by compliance to required LOR. Meeting the LOR requirements and implementing the MSSFs provide the needed risk reduction credit to lower the software contribution risk. Guidance for how much risk reduction credit can be provided for LOR compliance or MSSF implementation is usually defined in software safety program planning.

In the FHA, computing the risk reduction potential from implementing the safety measures is done at both the individual CSSFs level and the software level, similar to SWCI and LOR. Software risk is equal to the highest risk level for both the worst-case and best-case risk of its hazardous functions.

F. Recommend Safety Improvements

Once all the safety improvement opportunities are identified and their risk reduction potentials evaluated, a subset is recommended for incorporation into the various requirement specifications. Safety measures that can lower the safety risk of the overall system receive a higher priority. The FHA worksheet provides the following safety-related data to aid the development of safety requirements:

1. A list of hazardous functions identified from the system functions initially allocated to the software subsystem.
2. A list of hazardous functions but non-essential to eliminate to improve the system safety without losing essential functionality.
3. Each hazardous function includes anticipated failure scenarios, the system hazard conditions they triggered, the resulting system mishaps, and associated mishap severity to document the nature of the function's hazard.
4. Each hazardous function is assigned an appropriate software safety criticality and a LOR.
5. Some hazardous functions include a list of safety measures to reduce the software contribution to system hazards.
6. A list of industry best practice software safety measures.
7. Each hazardous function has a risk assessment based on the worst-case and best-case risk scenarios.

V. Translating the FHA Information into Safe Software Requirements

Using an FHA Worksheet, software requirements engineers work with software safety engineers to translate the safety-related information, i.e., CSSFs, MSSFs, GSSFs, into safety-related software requirements, i.e., CSSRs, MSSRs, and GSSRs, to complete the software requirements specification. The FHA process contributes four types of safety-related enhancements to the software requirements specification [26]:

1. Tagging safety significant software requirements.
2. Adding design constraints that impose a safety limit on the design and implementation of a system function.
3. Adding safety requirements to prevent, detect or react to a safety violation.
4. Adding requirements for a safety subsystem whose primary function is to monitor, detect, and control system hazards.

We will cover the process of converting the FHA safety-related information to software safety requirements in more detail in a future article.

VI. Conclusion and Future Work

As complex systems continue to become more dependent on software, managing the software risk contribution to system hazards within the software development lifecycle has become critical for system safety [1]. Organizations need to integrate software safety methods into their existing software development processes to produce safe software and software-intensive systems. The FHA is a suitable hazard analysis method for developing safety requirements for safety-critical software. The method combines both quality assurance-driven and risk-driven software safety approaches into a single analysis process. In this paper, we discuss the FHA's features and processes to support safe software requirements engineering.

We plan to further clarify the FHA's use in safe software requirements engineering in a detailed case study in a future article. We will review the process of translating FHA recommendations into software safety requirements for inclusion in the Software Requirements Specification, and verifying the specification for completeness. Finally, we will share our observations of common mistakes organizations adopting the FHA make and recommendations for improving the method and its usage.

VII. Appendix

This section contains several MIL-STD-882E [6] tables used in the FHA process.

TABLE I. Software Control Categories (SCC) (Incomplete)

SCC	Name	Description
1	Autonomous (AT)	Software functionality exercises autonomous control authority over potential safety significant hardware systems, subsystems, or components without the possibility of predetermined safe detection and intervention by a control entity to preclude the occurrence of a mishap or hazard.
2	Semi-Autonomous (SAT)	Software functionality exercises control authority over potentially safety-significant hardware systems, subsystems, or components, allowing time for predetermined safe detection and intervention by independent safety mechanisms to mitigate or control the mishap or hazard.
3	Redundant Fault-Tolerant (RFT)	Software functionality issues commands over safety-significant hardware systems, subsystems, or components requiring a control entity to complete the command function. The system detection and functional reaction includes redundant, independent fault tolerant mechanisms for each defined hazardous condition.
4	Influential	Software generates information of a safety-related nature used to make decisions by the

		operator but does not require operator action to avoid a mishap.
5	No Safety Impact (NSI)	Software functionality that does not possess command or control authority over safety-significant hardware systems, subsystems, or components does not provide safety significant information. Software does not provide safety-significant or time sensitive data or information that requires control entity interaction. Software does not transport or resolve communication of safety-significant or time sensitive data.

TABLE II. SOFTWARE SAFETY CRITICALITY MATRIX

	Severity			
SCC	***Catastrophic (1)***	***Critical (2)***	***Marginal (3)***	***Negligible (4)***
AT	SWCI 1	SWCI 1	SWCI 3	SWCI 4
SA	SWCI 1	SWCI 2	SWCI 3	SWCI 4
RFT	SWCI 2	SWCI 3	SWCI 4	SWCI 4
INF	SWCI 3	SWCI 4	SWCI 4	SWCI 4
NSI	SWCI 5	SWCI 5	SWCI 5	SWCI 5

SWCI	**Level of Rigor Tasks**
SWCI 1	Program shall perform analysis of requirements, architecture, design, and code, and conduct in-depth safety-specific testing.
SWCI 2	Program shall perform analysis of requirements, architecture, design, and conduct in-depth safety-specific testing.
SWCI 3	Program shall perform analysis of requirements, architecture and conduct in-depth safety-specific testing.
SWCI 4	Program shall conduct in-depth safety-specific testing.
SWCI 5	Once assessed by safety engineering as Not Safety, then no safety-specific analysis or verification is required.

TABLE III. RELATIONSHIP BETWEEN SWCI, LOR TASKS, AND RISK (INCOMPLETE)

SWCI	**Risk Level**	**Software LOR Task and Risk Assessment/Acceptance**
1	HIGH	If SWCI 1 LOR tasks are unspecified or incomplete, the contributions to system risk will be documented as HIGH and provided to the PM for decision.…
2	SERIOUS	If SWCI 2 LOR tasks are unspecified or incomplete, the contributions to system risk will be documented as SERIOUS and provided to the PM for decision.…
3	MEDIUM	If SWCI 3 LOR tasks are unspecified or incomplete, the contributions to system risk will be documented as MEDIUM and provided to the PM for decision.…
4	LOW	If SWCI 4 LOR tasks are unspecified or incomplete, the contributions to system risk will be documented as LOW and provided to the PM for decision.…
5	NOT SAFETY	No safety-specific analyses or testing is required.

REFERENCES

[1] N. G. Leveson, Engineering a Safer World. Systems Thinking Applied to a Safer World. MIT Press. 2011.

[2] N. G. Leveson, "The Therac-25: 30 years later" in *Computer*, vol. 50, no. 11, pp. 8-11, 2017.

[3] C. Barrett, D. Dill, M. Kochenderfer, and D. Sadigh, "Stanford center for AI safety," 2020. http://aisafety.stanford.edu/ whitepaper.pdf.

[4] J. Hatcliff et al., "Certifiably safe software-dependent systems: challenges and directions." Proc. Future of Software Engineering, 2014.

[5] A. C. Serban, "Designing safety-critical software systems to manage inherent uncertainty," IEEE International Conference on Software Architecture Companion, 2019.

[6] MIL-STD-882E, System Safety. Department of Defense Standard Practice, 2012.

[7] E. Wong, V. Debroy, A. Surampudi, H. Kim, and M. Siok, "Recent catastrophic accidents: investigating how software was responsible," 2010.

[8] IEC 61508-2, Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems – Part 2: Requirements for Electrical/Electronic/Programmable Electronic Safety Systems.

[9] J. J. Shen, Software Testing: Techniques, Principles, and Practices, 2019. Dahuo Books. ISBN 9781693054907.

[10] IEEE-1633, Recommended Practice on Software Reliability, IEEE Reliability Society, 2008.

[11] A. Hussain, E. Mkpojioru, and F. M. Kamal, "The role of requirements in the success or failure of software projects," International Review of Management and Marketing, 2016.

[12] L. Sha, "Using simplicity to control complexity," *IEEE Software*, 2001.

[13] L. Martins and T. Gorschek, "Requirements engineering for safety-critical systems: overview and challenges," *IEEE Software*, 2017.

[14] L. Martins and T. Gorschek, "Requirements engineering for safety-critical systems: An interview study with industry practitioners," IEEE Transactions on Software Engineering, 2020.

[15] T. Besker, A. Martini, and J. Bosch, "How regulations of safety-critical software affect technical debt," 45th Euromicro Conference on Software Engineering and Advanced Applications. 2019.

[16] J. W. Vincoli, Basic Guide to System Safety, 3rd ed. John Wiley & Sons, Inc. ISBN 978-1-118-46020-7. 2014.

[17] FAA. FAA System Safety Handbook. https://www.faa.gov/ regulations_policies/handbooks_manuals/aviation/risk_management/ss_ handbook/.

[18] P. J. Wilkinson and T. P. Kelly, "Functional hazard analysis for highly integrated aerospace systems," IEE Certification of Ground/Air Systems Seminar (Ref. No. 1998/255), 1998.

[19] K. Hayhurst, D. Veerhusen, J. Chilenski, & L. Rierson, "A practical tutorial on modified condition/decision coverage," NASA/TM-2001-210876, 2001.

[20] S. D. Carlo et al., "On enhancing fault injection's capabilities and performances for safety-critical systems," 17th Euromicro Conference on Digital System Design (DSD), Verona, Italy, 2014. pp. 583-590.

[21] R. Ramler, T. Wetzlmaier, & C. Klammer, "An empirical study on the application of mutation testing for a safety-critical industrial software system," 17th ACM Symp. on Applied Computing, 2017.

[22] IEC/FDIS31010. Risk Management – Risk Assessment Techniques. Final Draft, 2009.

[23] J. P. Blanquart et al., "Software safety assessment and probabilities," 46th International Conference on Dependable Systems and Networks Workshop. 2016, vol. 1, pp. 213-214.

[24] JSSSEH, Joint Software Systems Safety Engineering Handbook, 2010.

[25] JS-SSSA. Software System Safety – Implementation Process and Tasks Supporting MIL-STD-882E. Joint Services – Software Safety Authorities, 2018.

[26] D. Firesmith, "A Taxonomy of Safety-Related Requirements," 2004.https://resources.sei.cmu.edu/asset_files/WhitePaper/ 2004_019_001_29423.pdf