2013-09

Application of architectural patterns and lightweight formal method for the validation and verification of safety critical systems

Karagiannakis, Vasileios

Monterey, California: Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

APPLICATION OF ARCHITECTURAL PATTERNS AND LIGHTWEIGHT FORMAL METHOD FOR THE VALIDATION AND VERIFICATION OF SAFETY CRITICAL SYSTEMS

by

Vasileios Karagiannakis

September 2013

Thesis Advisor: Man-Tak Shing
Co-Advisor: James Bret Michael

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704–0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704–0188) Washington, DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2013	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE APPLICATION OF ARCHITECTURAL PATTERNS AND LIGHTWEIGHT FORMAL METHOD FOR THE VALIDATION AND VERIFICATION OF SAFETY CRITICAL SYSTEMS		**5. FUNDING NUMBERS**	
6. AUTHOR(S) Vasileios Karagiannakis			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943–5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**	
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ____N/A____.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited		**12b. DISTRIBUTION CODE**	

13. ABSTRACT (maximum 200 words)

This thesis researches the role of software architectural patterns and lightweight formal methods in safety-critical software development. We present a framework that relates the different activities and products from system engineering, safety engineering, system and software requirements, and software architecture explicitly, and demonstrate the proposed framework with a case study involving the architectural design of the software to control the arming device of a fictitious Surface-to-Air Missile. We describe the safety engineering steps for the identification of the system hazards and the critical functions that the software has to provide to avoid premature detonation, resulting in four safety requirements for the software that controls the missile's Electronic Safe Arm Device (ESAD). We formalize the software safety requirements as statechart assertions and validate their correctness via JUnit test. We develop a software architecture for the control software using the Safety Executive pattern, and implement the design in C++ to support a simple time-step simulation to produce the required log files for the automated verification of the design.

14. SUBJECT TERMS Safety-critical and Software Intensive Systems, Software Architecture, Architectural Patterns, Software Safety Requirements, Validation & Verification, Formal Methods			**15. NUMBER OF PAGES** 191
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540–01–280–5500 Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

APPLICATION OF ARCHITECTURAL PATTERNS AND LIGHTWEIGHT FORMAL METHOD FOR THE VALIDATION AND VERIFICATION OF SAFETY CRITICAL SYSTEMS

Vasileios Karagiannakis
Lieutenant, Hellenic Navy
B.S., Hellenic Naval Academy, 2001

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2013

Author: Vasileios Karagiannakis

Approved by: Man-Tak Shing
Thesis Advisor

James Bret Michael
Co-Advisor

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis researches the role of software architectural patterns and lightweight formal methods in safety-critical software development. We present a framework that relates the different activities and products from system engineering, safety engineering, system and software requirements, and software architecture explicitly, and demonstrate the proposed framework with a case study involving the architectural design of the software to control the arming device of a fictitious Surface-to-Air Missile.

We describe the safety engineering steps for the identification of the system hazards and the critical functions that the software has to provide to avoid premature detonation, resulting in four safety requirements for the software that controls the missile's Electronic Safe Arm Device (ESAD). We formalize the software safety requirements as statechart assertions and validate their correctness via JUnit test. We develop a software architecture for the control software using the Safety Executive pattern, and implement the design in C++ to support a simple time-step simulation to produce the required log files for the automated verification of the design.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

AT	Autonomous
BIT	Built- In Test
C2	Components and Connectors
CW	Continuous Wave
CS	Control Section
ESAD	Electronic Safe Arm Device
ETA	Event Tree Analysis
FMEA	Failure Modes and Effects Analysis
FMECA	Failure Modes, Effects and Criticality Analysis
FTA	Fault Tree Analysis
FMD	First Motion Detection
FTD	Fuze Triggering Device
GS	Guidance Section
HAZOP	Hazard and Operability
HTFF	Hazards to Friendly Forces
HTLS	Hazards to Launcher and Ship
IDE	Integrated Development Environment
NSI	No Safety Impact
O&SHA	Operating and Support Hazard Analysis
O/M.H	Operating/Maintenance Hazards
POST	Power-On Self-Test
PHA	Preliminary Hazard Analysis
PHL	Preliminary Hazard List
POE	Projected Operational Environment
PS	Propulsion Section
SSD	Safe Separation Distance
SAT	Semi-Autonomous
SMK	**S**hing **M**ichael **K**aragiannakis

SHA	Software Hazard Analysis
SSHA	Software System Hazard Analysis
SSR	Software Safety Requirements
SS	Steering Section
SHA	System Hazard Analysis
TDD	Target Detection Device
TR	Tracking Radar
TMR	Triple Modular Redundancy
UML	Unified Modeling Language
V&V	Validation and Verification
WS	Warhead Section
WCS	Weapons Control System
XML	Extensible Markup Language

ACKNOWLEDGMENTS

Foremost, I would like to express my sincere gratitude to my advisors Prof. Man-Tak Shing and Prof. James Bret Michael for the continuous support of my Master study and research, for their patience, motivation, enthusiasm, and immense knowledge. Both helped me in all the time of research and writing of this thesis. I could not have imagined having better advisors and mentors for my master's study.

I would like to thank my parents, for supporting me throughout my life.

I thank my Lord Jesus Christ to whom I always find spiritual strength.

Last but not the least, my deepest thanks go to my spouse, Olga, who stands by me both as a wife and as a good friend supporting me selflessly through all the years that we are together. Thank you for everything you do for me.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Γηράσκω δ’ αεί πολλά διδασκόμενος – As long as I live I learn

– Solon (638 BC – 558 BC)

A. OVERVIEW

The focus of this thesis is on the use of formal methods and architectural patterns for the assurance of software system safety. It is a common practice for requirements to be initially specified in a natural language. Developers of a system must then translate the natural language specifications into engineering artifacts, such as architectures, designs, and detailed implementations. This presents a problem: Natural language statements of requirements tend to be ambiguous, allowing for multiple interpretations. This is particularly troublesome in the context of the development of safety-critical systems. In this thesis we describe and demonstrate the use of an approach, based on the application of formal verification and validation (V&V) coupled with safety hazard analysis, to assess safety requirements and their refinement into software artifacts.

The correctness of software safety requirements can only be validated within the system context and the environment in which the system will operate. It is prudent to validate the software safety requirements as early in the system life cycle as possible as it is known that errors caught in later stages of the life cycle are more expensive to fix. In addition, errors related to system safety can also be costly due to the mishaps that result from them: death and injury, damage to property, and harm to the environment. Based on the fact that architecting of a system begins early in the system life cycle, we decided to explore both the use of software safety architectural patterns to capture safety requirements and to apply formal V&V to ensure the safety requirements are fulfilled.

B. RESEARCH QUESTIONS

To develop a step-by-step framework for applying software safety architecture patterns and formal V&V, we began by posing the following questions:

- How can one derive and express precise statements of software safety requirements from the set of natural language statements of requirements for the target system?
- In the early stages of the system life cycle, how can one validate the software safety requirements in the system context and environment in which the system will operate?
- What role do safety architectural patterns play in aiding the design of the architecture in meeting the software safety requirements?
- How can one verify the correctness of the architectural design in meeting the safety requirements?

C. METHODOLOGY

Our goal of answering the preceding list of questions is to develop and experiment with a methodology for architecting safety-critical software. We created requirements for a fictitious surface-to-air missile system so that we could demonstrate our methodology. The purpose of the missile system is to release lethal energy against enemy forces. However, from a system safety perspective, the missile system must avoid releasing the lethal energy inadvertently or against friendly forces (a.k.a., friendly fire). In the case study the missile's functions are allocated to software rather than hardware. To further limit the scope of our research we focus solely on the missile system's control software. The stakeholders' expectations for the system are included in the case study.

The first step is to review the requirements and then apply hazard analysis to identify safety hazards associated with the missile system. (Note that in this thesis we address safety only and not the operational effectiveness of the missile system.) We chose to examine in depth the premature detonation of the missile's warhead. This hazard could lead to mishaps that are severe. From the stakeholders' expectations the control software of the device that arms and detonates the warhead has to decide when it is proper to perform the arming of the warhead. By understanding the way that the device is going to perform its functions safely, we develop system safety requirements that the device has to meet. From these safety requirements, we have to translate the part that concerns the software and specify the software safety requirements for the case study. Both the safety requirements and the software safety requirements are written in natural language, and we use statechart assertions to formalize the software safety requirements.

Next we use software safety architectural patterns to develop an executable architectural model for use in simulating the behavior of the proposed software in the system's deployment environment under different use case scenarios. The recorded log files from the simulation are used for verification of the architecture.

The final step in this investigation was to use formal methods for validating the software safety requirements and verifying the software's architecture. Prior to this step, we created two related artifacts: the software safety specifications expressed as formal statechart assertions and the software's architectural model, which encapsulates its design. These two artifacts are related because they refer to the same system. Using the StateRover tool, we can validate the statechart assertions by JUnit tests and then use the validated statechart assertions and the log files from the simulation runs of the architectural model to verify that the proposed software architecture meets these specifications.

D. ORGANIZATION

Chapter II provides the background information necessary for the context and the overall direction of this thesis. It defines the related terminology and identifies the gaps that exist when considering the design of the safety-critical software. This chapter also examines how software engineers address nonfunctional attributes such as safety in architecting a software-intensive system.

In Chapter III we analyze the principles the design team uses to demonstrate the concept of the software safety requirements. This chapter introduces the case study, which is named SMK for the first letter of our last names (ShingMichaelKaragiannakis), to demonstrate the safety engineering steps from the identification of a system's hazards to the critical functions that the software has to provide. We specify the four software safety requirements for the software which controls the SMK's Electronic Safe Arm Device (ESAD) (i.e., for the arming of the missile's warhead).

Chapter IV details the development of the software safety architectural design. We introduce a variety of software patterns for application to safety-critical systems. We document our use of the Safety Executive pattern in architecting the software. We

implemented the design in C++ to support a simple time-step simulation, which produces the required log files for the verification of the design.

Chapter V covers formal V&V of the requirements and our software architecture. For validation purposes, we specify the four software safety requirements using statecharts assertions. For verification purposes, we exercise those assertions using the StateRover tool. We execute the architecture code in C++, creating twenty-one log files. These log files are the inputs test cases for the StateRover tool and run against the four statechart assertions to see if C++ code violates any of the four statechart assertions in any of the twenty-one scenarios.

Chapter VI provides a summary of the results of this research, a list of lessons learned and recommendations for conducting future work.

In the Appendix A we include the source code for the simple time-step simulation. Appendix B contains a brief analysis for the twenty-one use cases we use for the simulation phase, with the produced log files and their related time diagrams. For the validation part of the statechart assertions, the test cases are implemented as JUnit tests and their code is presented in the Appendix C.

II. SAFETY-CRITICAL SOFTWARE

A. INTRODUCTION

Petroski argues that failures of systems are inevitable but that studying such failures advances our understanding of engineering design in [1]. Although the examples of system failures are couched in terms of structural and civil engineering, his argument and insights apply to the accumulation of settled knowledge within other engineering disciplines including software engineering.

The focus of this thesis is on the architecture and design of safety-critical systems whose behavior is controlled by software interacting with hardware and humans. We define the term safety-critical system to be a system that controls one or more forms of energy that if not properly controlled could cause some combination of loss of life, injury, property damage, or harm to the environment.

Although simplicity of design is a recognized best practice in engineering, software-intensive systems, which today one can argue includes everything in the Internet of Things, tend to have complex architectures and designs. That complexity tends to become embedded in the implementation of software-intensive systems during the refinement process. This is a particularly problematic situation for safety-critical systems, given that complexity inhibits our ability to adequately analyze using static and dynamic means the system safety and properties of these systems. In the remainder of this chapter we provide some examples of the operation of safety-critical software-intensive systems that resulted in mishaps and then discuss the emergence of the use of software patterns as a mechanism to reduce the complexity of safety-critical software-intensive systems.

B. EXAMPLES OF MISHAPS

There have been many incidents where under specific circumstances the systems failed to behave safely. The following are examples of some mishaps that were related to safety and software:

- The software error of a MIM-104 Patriot. The error caused its system clock to drift by one third of a second, resulting in failure to locate and intercept an incoming missile. The tragic result was 28 dead soldiers, as it is described in [10].
- A Chinook crash on Mull of Kintrye in June 1994. A Royal Air Force Chinook helicopter crashed into the Mull of Kintrye, killing 29 people. After extensive investigation the final report claimed that a bug in the software was responsible for the control of the engine's computer and caused an unexpected behavior of the engine and resulted in the accident, as it is described in [10].
- An F-22 Raptor crash. In April 1992 the first F-22 Raptor crashed while landing at Edwards Air Force Base, California. The cause of the crash was found to be a flight control software error that failed to prevent a pilot-induced oscillation, as it is described in [10].

The above mishaps and many more similar ones have created the necessity to re-evaluate the way in which we build systems and engineers implement safety-critical and safety-related functions in software. From the investigation of the related accidents and mishaps, it is reasonable to conclude that the designs were not created in such a way as to ensure the safety of these systems. Furthermore, the designs had many loopholes that could lead to unplanned situations with a chaotic and undesired behavior of the systems. There is a need to develop proper mechanisms and techniques to capture and address safety concerns and to consider how the software architecture can improve the safety of systems.

C. DEVELOPMENT OF SAFETY-CRITICAL SOFTWARE

1. Software Behavioral Modeling

From the lessons learned, there is a need for the designers to evaluate the work in progress at every stage of the development process instead of only assessing the quality of the end product, because the cost for fixing an error increases exponentially as the product reaches its deployment phase as explained in [6]. Thus, in each stage of the software's life cycle, the development team attempts to meet the user's expectations and answers the following fundamental questions: What do we have to build? How are we going to build it? Does the proposed product meet the user's expectations? For the above questions, the need to identify the path to follow to achieve these goals is crucial. Thus, there must be a clear distinction between the main parts of the system and the way that they related to each other. This study focuses on the safety attribute of software due to the fact that in many cases this attribute is responsible for undesired incidents. In safety-critical systems the occurrence of failures could be catastrophic when specific circumstances are met. For example, if the software of a 'smart bomb' detonated the fusion mechanism close to friendly forces then we would have safety-critical issues. But, in other circumstances, in which the explosion takes place in an area where no friendly forces are close, then the incident could be mission-critical. It does not violate safety issues. It is important to recognize and capture the customer's expectations to identify the proper environment in which to operate.

Historical data showed us that hardware failures are random, as opposed to software failures which are are systematic [2]. The main reason for the difference between hardware and software failures is that the software by itself does not fail. Software is a representation of human thinking about the design of a machine or how this machine should behave. Software is not a physical device; thus, it does not follow physical laws. However, the design of hardware affects software behavior and vice versa.

State machines are often needed to model the expected behavior of reactive systems for today's complex systems. Unfortunately, it is not easy to create state machine models to capture the system's behavior correctly. State machines can be deterministic or

non-deterministic. A deterministic state machine is a theoretical machine in which no randomness is involved in the transition between states of the system. A non-deterministic state machine is a theoretical machine in which ambiguity is present in the transition between the states of the system. In a deterministic machine, the next possible state is uniquely determined; in contrast, there is a set of possible next states for a non-deterministic state machine. Non-deterministic state machines are applied in cases in which there is ambiguity about the behavior of the system. Deterministic state machines are used in cases in which the uncertainty is removed.

The model of choice typically depends on the size and complexity of the system, the extent of use and experience with the machine and the risks associated with the software's systematic faults. From a review of the open literature, there appear to be many software projects in which the developers had an inability to predict faults, and as a result, there is an inability to quantify the associated risks for the software. In particular, the amount of the complexity in relation to the severity of the faults creates a mixture that can lead to ambiguous safety-critical applications. This does not mean that the design team has to detect all the potential errors, but it does mean that it is extremely important to be able to assess the effects of software with respect to the system safety. For these reasons the designers adopt the terms determinism and non-determinism to define the failure behavior of the system. Thus, the only tool that the development team could use is their degree of knowledge about the failure behavior of a component in order to choose between the use of determinism and non-determinism.

2. Software Architecture

To address the design decisions related to quality attribute requirements (e.g., requirements that deal with efficiency, usability, reliability, safety and security), which have crosscutting effects on the eventual system, the developer community has begun to focus its attention on software architecture as a means to achieve the many quality attributes of the final system. The software architecture's theoretical background and the rationale for the design of the projects come from experience, which in turn is abstracted and codified into architectural styles and patterns. Architectural styles refer to the way

that we imagine how we are going to build the system using our experience in previous problems. An example of an architectural style is the layered style.

> "The essence of a layered style is that an architecture is separated into ordered layers, wherein a program within one layer may obtain services from a layer below it" [4].

Architectural patterns are more specific in comparison with styles and define the components, the connectors and the relationships between them, to solve recurring problems, as defined by Taylor et al. in [4]. An example of the architectural patterns is the Sense-Compute-Actuator pattern that is used in structuring embedded control applications.

> "The basic idea is: A computer is embedded in some application; sensors from various devices are connected to the computer and may be sampled to determine their value. Also attached to the computer are hardware actuators" [4].

The systems community uses the styles in such a way as to decide how they are going to deal with a problem in a high-level abstraction, and the developers could use past experience, codified as patterns in new problems that are similar. However, there is not a significant set of methods available for the designers to address safety in software architecture. For these reasons, Wu and Kelly proposed in [6] the concept of safety tactics that are based on the description for other quality attributes like availability, modifiability, security, performance, usability and testability by the Software Engineering Institute. Architectural tactics provide a means to identify and codify the underlying primitives of patterns in order to solve the problem of the intractable number of existing patterns (refer to Chapter II, Section F for more details). Obviously, there are relatively fewer tactics to be handled, and there are many ways in which tactics can be combined into patterns.

3. Software Hazard Analysis

The set of safety tactics defines the connection between the software architecture with software safety and what we have to do to minimize the possibility of unplanned behaviors that they could lead to mishaps. However, effective use of safety tactics requires proper identification and management of potential hazards. The traditional validation and verification of the systems, which focuses on the testing of systems as a whole (i.e., the system's software architecture consisting of components and connectors), is not effective in assuring the system's safety. When we test the whole system as an entity, we have to define which parts of it are most likely to result in mishaps and try to simulate all possible cases that our system will face in its deployment phase. This procedure is time-consuming and requires resources thus increasing the budget. In most cases, time-pressure combined with cost increases prevents us from performing adequate fault forecasting for systematic software failures. This situation produces, at some point, the nature of the software's failure, which is systematic and is related primarily to design faults and the resources' extinction. When we refer to resources we mean both time and money.

Additionally, as systems become more complex and there is a need to deal with more complicated tasks, the risk assessment of systems becomes more complex, because we have to identify and assess all relevant faults rather than using a method to estimate the occurrence and the total number of failures. To address the issue of the intractability of failures, software engineers perform Software Hazard Analysis (SHA), which is a specified branch of Hazard Analysis proposed by Leveson in [3], to concurrently identify the potential failures, perform requirements analysis and conduct specification tasks. A requirement for a safety feature has to be met by the safety-related software. These requirements are the basis for demonstrating the satisfaction of the system-level safety requirements and meeting the user's expectations. Feature-based requirements, which can be safety functions or safety properties, may be identified based upon the requirements engineering that the developer team used. However, safety-feature requirements may also be identified to mitigate the risk of hazards occurring in other components of the system. These requirements can be used to demonstrate the adequacy of the software component

with respect to hazards. They are related to potential failures of the software within the system that may lead to system hazards.

The combination of safety tactics and the use of SHA yield a specific framework for producing the mechanisms in our software to implement the safety. That is how in this thesis we demonstrate the assurance of system safety from the system's software architecture. Our primary goal is to build a system that meets the requirements of its stakeholders and at the same time the software of this system will be able to detect, identify and properly manage the potential failures in order to avoid them or handle them and sustain safe behavior of the system.

4. Formal Methods

Relating the issues just discussed in the combination with the increasing products complexity and the likelihood of much greater subtle errors, which affect the safety attribute of the systems, there is a need for the designers to use a tool capable of specifying and verifying such systems. One way of achieving this goal is by using formal methods, which are mathematically based languages, techniques and tools. Their scope is to increase our ability to obtain a deep understanding of a system by revealing inconsistencies, ambiguities and incompleteness that might otherwise go undetected. The main advantage of the use of a formal language is that the developers can achieve specification and verification of the target system, as it is described in [2] and [11]. However, they cannot guarantee total correctness. The formal methods can be used in different phases of the system development process, helping the developers to deal with ambiguities that are hidden in these phases. For example, as Clarke and Wing state in [11]:

> "It is worth exploring how they can be used in requirements analysis, refinement, and testing."

Requirements analysis necessarily deals with customers who often have an imprecise idea of what they want; formal methods can help customers nail down their system requirements more precisely.

Refinement is the reverse of verification; it is the process of taking one level of specification (or implementation) and through a series of ‘correctness-preserving transformations’ synthesizing a lower-level specification (or implementation). Although much theoretical work on refinement has been done, the results have not yet transferred to practice.

Testing is one of the most costly areas in all software projects. Formal methods can play a role in the validation process, for example, using formal specifications to generate test suites and using model and proof-checking tools to determine formal relationships between specifications and test suites and between test suites and code.

In this study, we focus on the safety of the system from the viewpoint of a software developer, and formal specifications are needed in the phase of requirements analysis. This is because formal specification is the act of writing things down precisely, and through this process describe a system and its desired properties, as explained in [2] and [11]. This process is based on a mathematically defined syntax and semantics. These kinds of system properties might include functional behavior, timing behavior, performance characteristics or internal structure. So far, specification has been most successful for behavioral properties. The process of specifying what exactly the users/customers want the system to do helps the developers to uncover design flaws, inconsistencies, ambiguities and incompleteness. Through this process, a useful communication link is created between the customer and developer, between designer and builder and between builder and tester. It documents the answers for questions of what, how, when, why the system does, but at a higher level of description. The final outcome from this process is formally analyzed, because it is based on mathematics, and can be checked to ensure that the specified system is internally consistent.

Besides the specification, another use of formal methods by the system designers is the verification of the system’s architecture, going one-step forward to analyze a system for desired properties. Every system has architecture, which is a set of design decisions about the system, and these design decisions can be captured in models, as it is stated in [4]. The notation of the models can vary, but in the end the developers want to

use their model to express rigorously and formally the functional and non-functional aspects of the system.

Model checking, which is a technique that relies on building a finite model of a system and checking that a desired property holds in that model, as described in [11]. The check is performed as an exhaustive state space search that is guaranteed to terminate since the model is finite. There are two approaches to achieve the checking. The first is called temporal model checking, which is a technique in which the specifications are expressed in a temporal logic and systems are modeled as finite state transition systems. An efficient search procedure is used to check if a given finite state transition system is a model for the specification. In the other approach, the model checking is achieved by the comparison of the specification with the system. They are both modeled as automaton, and the purpose of the comparison is to determine whether the system's behavior conforms to that of the specification, as it is described in [11].

D. THE NATURE OF FAILURE

As mentioned above, the developer community has developed methods to identify when and how a hazard can occur. In addition they must declare the meaning of failure and determine precisely what constitutes a failure. In software, where the term failure is related to the improper way the software behaves, this term is rational and close to an abstract notion. For these reasons, and in order to be more understandable, we define the term failure using the same method that Pumfrey et al. in [9] used to classify it. In general, Pumfrey et al. classifies the failure using the logical sequence of how a failure is related to an event, how a failure is generated, how a failure behaves and which properties a failure contains. All these questions are answered in an abstract way, giving us the flexibility to adopt them in any until-now problem. For this study the following definitions are used.

> "Failure is the non-performance or inability of the system or component to perform its intended function for a specified time under specified environmental conditions," as it is defined by [3].

> "A fault is an incorrect step, process, or data definition in a computer program," as it is defined by [3].

It can be internal (e.g., fault in code, specification) or external (e.g., wrong input data, attack). Faults can cause failures but they do not have to. If a fault can only turn into a failure under conditions that are never met, no failure will be observable. Likewise external faults need internal faults in order to produce a failure. So they cannot be the sole cause of a failure. However, the fault is always a prerequisite for a failure.

> "An error is a deviation from the required operation of the system or the subsystem" as it is defined by [2].

When the failure has occurred, this means that an erroneous system state has been observed. This erroneous system state is the error, while the observation of the error (wrong output, wrong behavior, system downtime, etc.) is the failure.

Thus, the first question that must be answered is the relation between a failure and an event. We have the case where an event fails to occur, which is the failure omission, and we have the case that an event does not occur in a proper sequence, which is the failure commission. Also, we have the cases that a failure occurs in relation to the time of an event when an event occurs earlier or later than it is scheduled. And finally, we have the cases that consider the impact of a failure due to the value of the event when the received value is incorrect or when we cannot detect its value.

Continuing, we have to pinpoint the causality of a failure. To answer what the causal factors of a failure are, we have to analyze the factors that are related to it. The three major factors are the software as an entity, the related hardware and the environment in which the software is operating. The main reason that the software itself can create failures is that many issues are not well defined leading to an incorrect design and implementation, resulting in the software behaving in undesired ways. Also, the way that we try to implement the software in the real world through the hardware can lead to failures due to limitations or incorrect design of the hardware. Lastly, the environment can generate abnormal situations that the software is not able to handle. For example, it could be attributable to how humans react during the 24-hour day:

> "Safety and productivity are low at night. The fact that we are a diurnal species may explain why many of the many industrial accidents involving human error have occurred at night" as Leveson states in [3].

These kinds of situations can cause software to react in such a way that a failure can occur.

Another critical item about the nature of failures is the way that they behave when they occur. There are cases in which a component receives some inputs (or changes the original inputs) and reacts improperly, propagating failures throughout the whole system. For example, Wu and Kelly in [6] described a situation in which a random component received proper inputs, but due to the wrong mechanism, created wrong outputs, which are inputs for the next component. The next component received them and, naturally, created wrong outputs as inputs to a third component. This kind of situation propagates the first failure.

The second situation is the result of an unscheduled input. Again, taking from the paper by Wu and Kelly [6], we can face the situation where an event, due to any reason, arrives late to the proper component and results in an unscheduled sequence of events that could conflict with the proper values of other events. Thus, we have a correct event at an improper time that is transformed into failure.

To create the proper mechanisms that prevent or minimize the impact of the failures, we need to understand two important properties of each failure. The first property concerns the detectability of the failure. The second property is related to the severity and the magnitude of the failure. This property defines how tolerable a failure is, which means how easy it is for us to mitigate the results from the occurrence of a failure. Although we may not be able prevent the failures from occurring, we want alternative ways to build safety-critical software that reduces the impact of the failure.

E. HAZARD ANALYSIS TECNHIQUES AND MODES

To design a safe system, the designers must detect and identify the potential hazards. This mechanism is referred to as hazard analysis and includes a range of techniques; each of them investigates from a different perspective the system in order to identify the hidden hazards. From the publicly available literature, we can find many techniques that can be applied to particular industries and are limited in other domains. There are cases in which the development of a technique has been generated from a specific domain, but fortunately their logic can be implemented to other domains. As it is described in [2] and [3] the most widely used techniques are discussed in the following sections.

1. Fault Tree Analysis (FTA)

FTA is a deductive reasoning failure analysis (from system failure to its reasons). It is a graphical technique describing the relationship between hazards and causal factors leading up to the hazards. The designers expand the fault tree, adding more detail as the analysis becomes more thorough. The initial tree starts with the hazard and works backwards to find the causal factors. Continuing in time and as the analysis proceeds, the tree has more details. The goal for this technique is to gain a more in-depth understanding of the system as the tree's high-level nodes have been expanded deeper. The result from using this technique is a pictorial tree that includes logical operators, such as the AND and OR from Boolean algebra, to define the relationship between cause and effect. It is an easy way to picture the potential hazards and how they may occur and which modules have different effects on the potential occurrence of the hazard. An example of FTA could be the following: One desktop computer could not start. Thus, the potential faults could be: Power issue OR Booting issue. For these two issues, the computer's failure to start could have many causal factors and their proper combination led to it, as it is depicted on Figure 1.

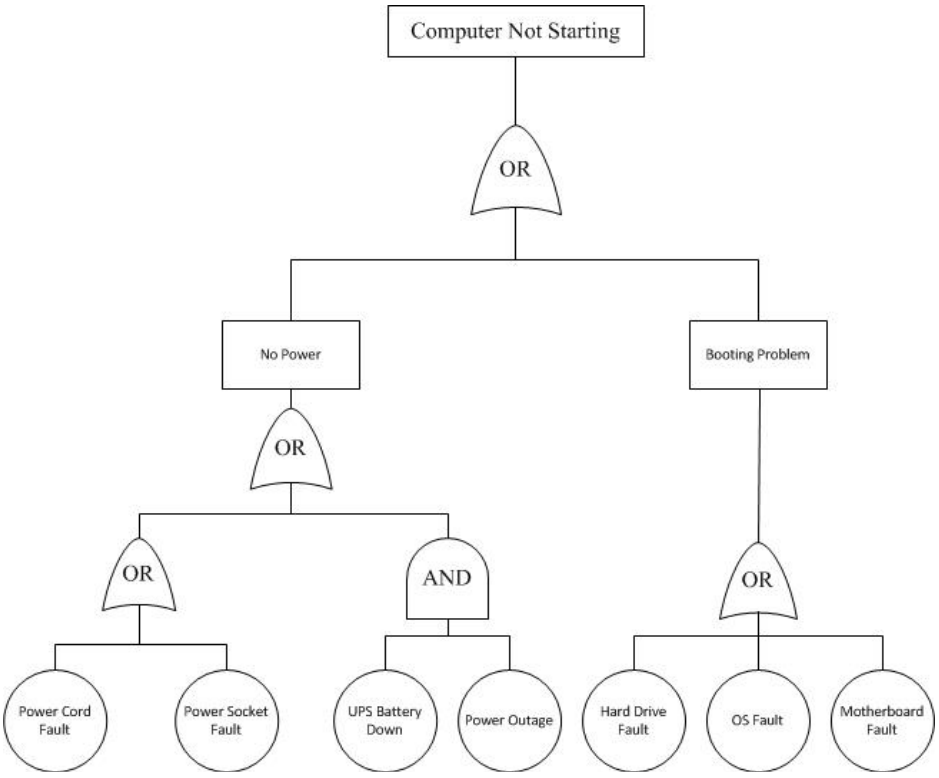


Figure 1. Fault tree analysis example. From [2] and [3].

2. Failure Modes and Effects Analysis (FMEA)

FMEA is a graphical technique for analyzing any failure of each component and relating the effects from the failures to the system. Its scope is to investigate the possible modes of failure and from them to identify and detect the consequences. Using the above technique, engineers are able to identify any structural weaknesses in the design and to rectify these before implementation begins. The benefit of using this technique is to test the product against the design, reveal the states that have failure behavior and relate them with the root, which are either the requirements or the interpretation of requirements into the design. The drawback of this technique is that it demands a lot of research, time and resources, and for these reasons it is applied at the late stage of the development phase.

3. Failure Modes, Effects and Criticality Analysis (FMECA)

FMECA continues from the outputs of the FMEA and considers the importance of the failures to the system. To measure the severity of these failures, the technique

considers the severity of each failure and its related probability of occurrence. Using the FMECA technique the engineers are able to focus on areas that the occurrence of a failure could lead to catastrophic consequences.

4. HAZard and Operability Studies (HAZOP)

HAZOP is an explanatory technique that relies on the answers to “what-if” questions to analyze the alternative behavior of each component and relate them to the system. This technique uses a group of guidewords that are related to the specific domain, helping the engineers to identify easily their scope. Also, it is very effective and helps the engineers to think deeply about their proposed systems, but it is time-consuming and demands an expert level of systems knowledge. In practice, experience is used to guide the choice of questions in each area, as it is depicted in Figure 2.

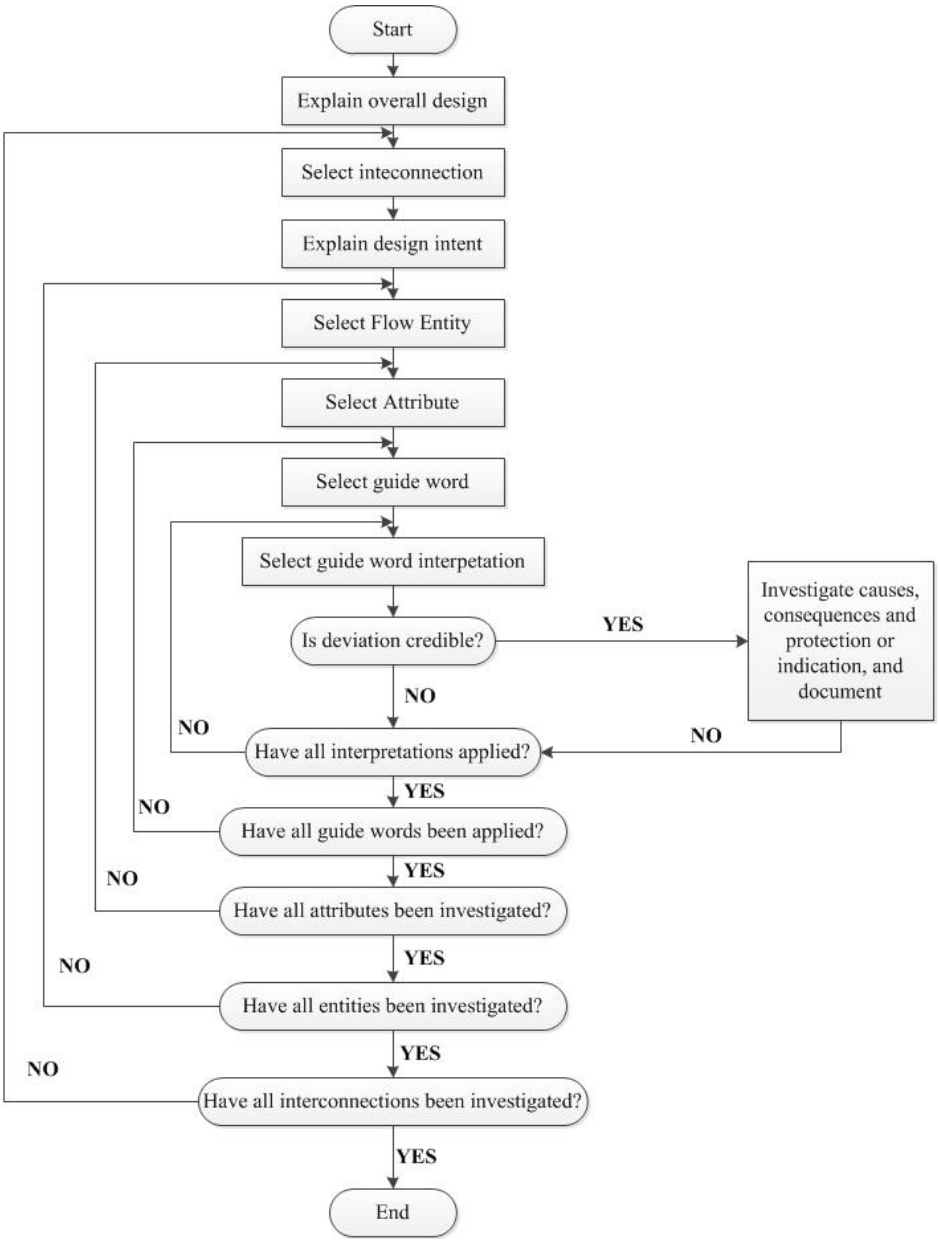


Figure 2. A flowchart of the HAZOP study process. From [2] and [3].

5. Event Tree Analysis (ETA)

ETA is an inductive reasoning failure analysis (from basic failure to its consequences) that manifests itself as a graphical technique (a dynamical expanded tree) starting with an event, which affects the system, and then continues to analyze the potential consequences. This technique tries to catch the potential propagation of an event (failure) and how this triggers a sequence of events ending in the potential results. The

benefit of using this technique is the unveiling of consequences, which are not obvious under specific situations in complex systems. An example of this technique is depicted in Figure 3, which includesthe event tree analysis for a failure of coolant pressure, as it is described in [2].

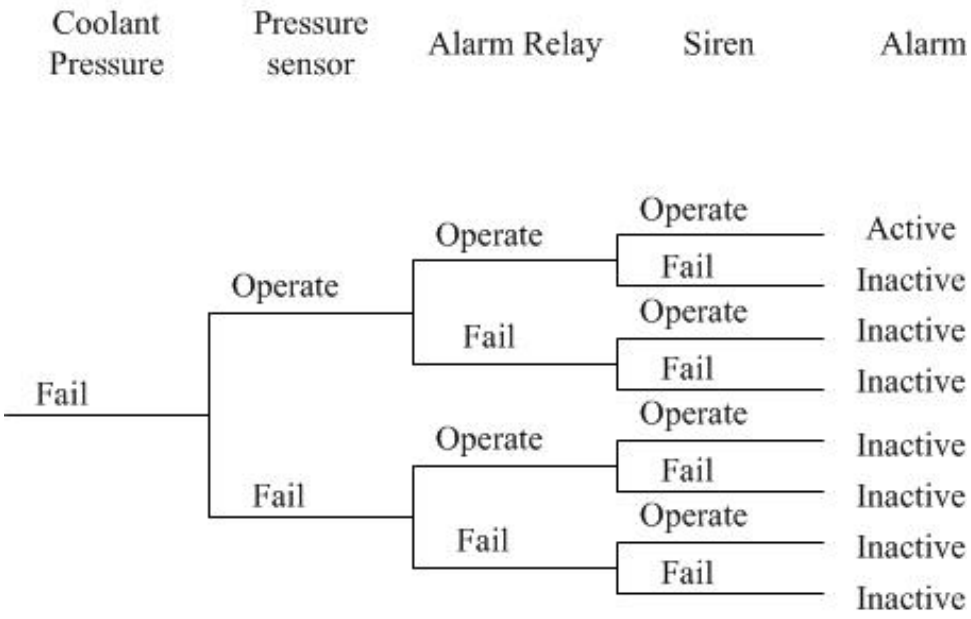


Figure 3. Event tree analysis for the coolant pressure. From [2].

The engineers are able to use the previously discussed techniques simultaneously to achieve the best result for the hazard analysis. It is common to use the FMEA in combination with the FTA because the two methods can be used in a complementary way. For example, the outcomes from the FMEA can be roots for the FTA as it is described in [2] and [3]. Also, the HAZOP technique studies the interconnections between the components of the system, and using the tools of this technique, the engineers are able to determine the interactions. At its final phase, the engineers are able to prioritize the hazards, and using their outcomes, create the roots for the FTA.

Having all the above in mind, a fair question to ask is: when does the design team have to apply the hazard analysis in their work? This question has a critical meaning because it is clear that the hazard influences the behavior of the system. Especially in this study, we are interested in the software intensive systems, and the results from the hazard

analysis will affect their architecture and development. It is clear that the software is a part of a system that controls the hardware and interacts with the users. Hence the design team has to think holistically about its design. The entire system must be designed to be safe. The main parts of the system are the software, the hardware, the users, and the environment. All these parts should be considered equally as well as how they interact to each other. Functional and operational safety starts at the system level. Safety cannot be assured if efforts are focused only on software. Hazards at the system level include: hardware hazards, software hazards, procedural hazards, human factors, environmental hazards and interface hazards.

The analysis of the potential hazards starts at the requirements phase of the system with a proposed design concept. This kind of analysis is characterized as Preliminary Hazard Analysis and begins with the identification of the potential hazards associated with the proposed system. The system safety analysis continues throughout the project life cycle. The software safety analysis process needs to be performed next to review the results of the systems analyses and to assure that changes and findings at the system level are incorporated into the software as necessary. In addition, the software safety analyses provide input to the system safety analyses. The software safety analyses are a special portion of the overall system safety analyses and are not conducted in isolation.

There are many modes for hazard analysis at the life cycle of the project. In this study, we are going to follow the categorization from the Mil-Std 882E [5]. This military standard describes and declares the fundamental principles of safety in general and makes a special reference to those related to that system safety. The following techniques are part of a group where other safety-standards are given, but in this text they define in an abstract way the notion that is hidden behind them. The military standard uses the following techniques that are characterized as tasks:

- Preliminary Hazard List (PHL)
- Preliminary Hazard Analysis (PHA)
- System Hazard Analysis (SHA) and its branch Software System Hazard analysis (SSHA)

- Component SHA and Component SSHA
- Operating and Support Hazard Analysis (O&SHA)

a. Preliminary Hazard List (PHL)

PHL is an initial analysis that comes with the concepts of the project, trying to identify the hazards and the way that each of them must be confronted.

b. Preliminary Hazard Analysis (PHA)

PHA is the next step, where the main purpose is to identify and evaluate all system hazards. It is the root-step for the system and software hazard analysis giving the first results to the safety team to continue with the analytic and thorough study of the components of the system.

c. System Hazard Analysis (SHA) and its Branch Software System Hazard Analysis (SSHA)

SHA and SSHA move forward trying to relate the identified hazards to the risk's assessment, which the developers have to define to proceed in depth their analysis.

d. Component SHA and Component SSHA

Component SHA and Component SSHA study each component on an individual basis and looks to identify the associated hazards with the design of the components, and how those hazards will affect the entire system.

e. Operating and Support Hazard Analysis (O&SHA)

O&SHA investigates the relation between the project and the external users, such as humans and the environment, as concerns the safety of the system. O&SHA identifies safety requirements necessary to eliminate hazards or mitigate the risk of hazards. Using them we define the potential hazards of our system. The various system hazard analyses will attempt to eliminate as many hazards as possible, reduce the probability of occurrence of those that remain and reduce the potential damage, which may result from accidents. In some cases, software components may be assigned such

responsibility. If this occurs, software hazard analysis is a form of component hazard analysis.

F. SAFETY TACTICS AND PATTERNS

Continuing our study, safety critical failures are defined as those that can lead to hazards and thus act as a causal factor in accidents. For this reason, the failures should be avoided and in case that we cannot avoid them, we build our system in such a way to mitigate the risk of the hazards. Beginning our analysis, we are going to define the relationship between the failures and how they are identified in relation to the software architecture. In [6] Wu and Kelly categorize the failures by classification, causality, behavior and property. The above notions create the framework under which we are going to formalize the safety quality attributes for our system, giving us the ability to continue and build the protective mechanisms and mitigate any hazards. The distinction and definition of these elements are based on the logic: What (the abstract notion–classification), Why (more detailed notion–causality), How (the implementation notion–behavior) and finally Which (the properties of failures).

Using the above logic, Wu and Kelly in [6] have organized the safety tactics into three sets: failure avoidance, failure detection and failure containment. Having as a basic structure the above three sets, it was expanded into a hierarchy of techniques for constructing safety-related architectural patterns. The proposed hierarchy is depicted in Figure 4.

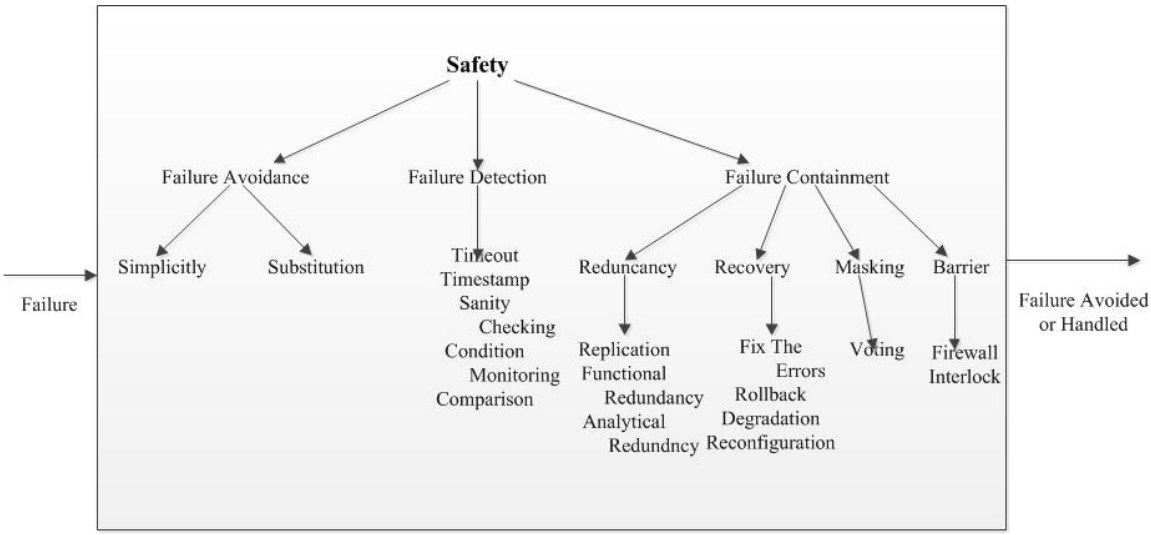


Figure 4. The hierarchy of safety tactics. From [6].

Choosing from the three above categories one or more tactics, we can apply them and provide our system with more safety. The goal of the above approach is to make our system more reliable, in order to detect the failures and avoid them or to recover from any failures and prevent the hazards from occurreing. Some of the tactics, like Rollback, do not eliminate the failures but give the opportunity to retry at another time. Moreover, we can approach the safety tactics from two different viewpoints.

1. Architectural-pattern Viewpoint

This approach uses patterns as the core safety-tactics in the context of a use-case. Having as an input the customer's expectations, we analyze the problem in terms of components and connectors, and using one of the already defined patterns, we investigate the situations where a relation leads to a hazard. One example of such a safety requirement is for our software to be able to identify the hazards and use the proper mechanisms to prevent them. From the customer's point of view this issue is concerned as granted but the detailed requirements about what exactly should be monitored is not given in all cases, as is mentioned in [7].

This is the challenging part because we have to overcome two entities that are the motivation for the architects to design their project. The first one is the group of the safety tactics, and the second is the user/customer's expectations that are expressed as

functional requirements in natural language. Both entities are too abstract for defining the source code, which is the final product and must contain the whole rationale from them. The goal is that both the developers and the stakeholders can have a meaningful view of the project. However, existing practice fails to systematize solutions to these architectural issues as it is described in [7]. Focusing on a scenario/use-case instead of the whole system may result in misleading assumptions as the complexity of the project increases. In particular, for the assumptions that are related to the safety properties of our system, we have to be absolutely sure and we have to explicitly identify and document them. Another issue concerns the selection of appropriate use-case scenarios that will determine the architecture.

2. Failure Modeling Viewpoint

From all the above, we can understand that the core of the problem is the way that we describe and define the term safety in any system. Wu and Kelly [7] proposed an alternative approach for this issue because we are not able to define which tactics must be performed a priori, and in some cases this could lead to the increase of the set of failures. They proposed the development of failure modeling, which treats the failures as ‘components’ in order to relate them to the real software components of the system. Using this approach, the architecture is combined with the safety due to the fact that it implements the results from the software hazard analysis in the design of the product. To achieve this Wu and Kelly in [7] proposed the following methodology for building failure models.

Firstly, we have to overcome any level of imprecision and ambiguity. This can be solved using formal methods to describe our requirements, but it limits our ability to be flexible in cases where the customer needs additional requirements.

Secondly, we need components that contain the safety properties of a system, and through their composition, we can succeed in obtaining the proper structure of our design.

Thirdly, we have to model our system to check its behavior in its environment (hardware, users). In this way we can achieve the Verification & Validation of our project before its deployment.

Continuing the above, we need resources (time, computing power) to capture the cases in our design assumptions that could lead to faults and re-evaluate the points that caused them to perform verification of the safety of the proposed software architecture.

The purpose of the approach proposed by Wu and Kelly in [7] is to investigate the failure behavior of the system using a bottom-up approach relating the components to the failure behavior. As already mentioned, this approach looks to design the product according to its components and not to take an existing pattern and implement safety functions on it. Using the nature of each component, it is modeled according to the way that it might propagate the failure or the way that it can generate a failure. The whole effort aims to define the events that are going to happen in such a way that the model uses them to give us as an outcome of the potential results, success or failure. Furthermore, in the case of failure we are able to distinguish the causal factors that resulted in failure. Thus, we have to design our product in such a way that we can answer the following questions from the very beginning: How can we define failure behavior of a component? How does a failure model facilitate safety analyses?

According to the literature, one of the most effective techniques about the architectural design is the combination of iteration and incremental development. However, the designers struggle to get early feedback during the iterative and incremental development to address concerns with the behavior of each component, which would mitigate as much as possible the magnitude of the software's complexity. Essentially, using this proposed methodology, they are in place to generate the model, having also the proper validation mechanisms for it. The purpose of the above task is to implement the failure behaviors of each component, and at the same time, to make the procedure more dynamic, using scenarios/use-cases that are related to the failure model in order to receive feedback. The whole procedure will return the valuable safety analysis that produces assessment results and feedback to the subsequent design process, as it is explained graphically in the Figure 5.

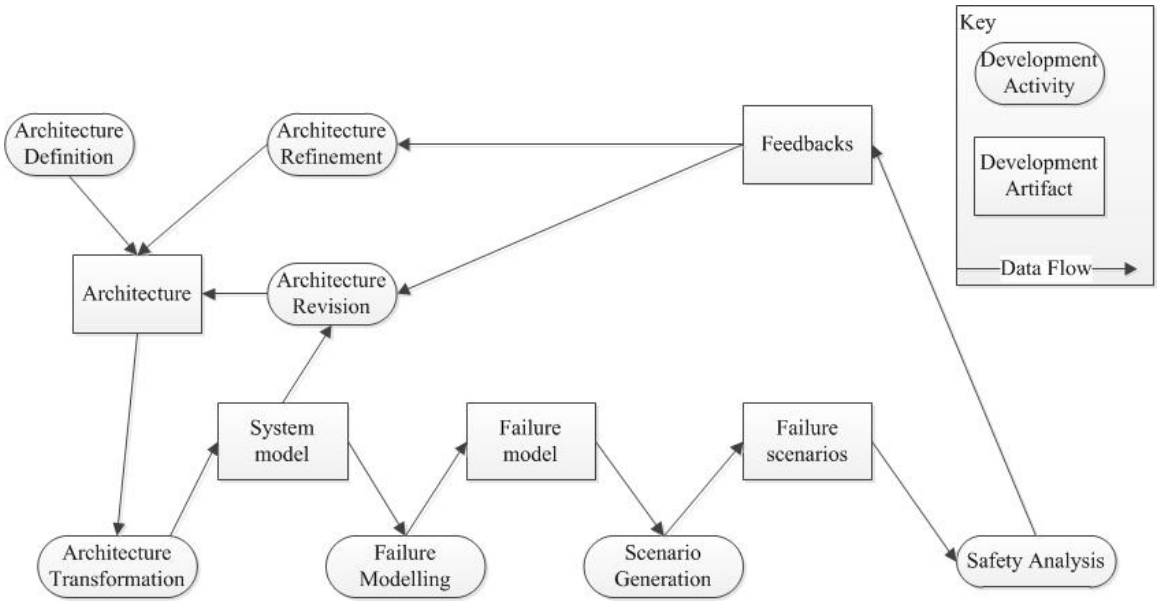


Figure 5. Failure modeling in architecture design process. From [7]

In Figure 5, we see that the output of the Architecture node is called Architecture transformation, and this is one step before the System Model. The first node contains a set of events and actions that are going to be implemented in the System Model. The challenging part of the architecture transformation is the distinction between the atomic and the composite components. The main reason is because the atomic components can be modeled separately and analyzed independently from the rest of the system. Instead, the composite must be modeled and analyzed in relation to the rest of the system due to the behaviors of enclosing components.

G. EXAMPLE OF SAFETY ARCHITECTURAL PATTERN

Wu and Kelly in [6] describe an example of how the designers should design a software product. In this example they use the complicated C2 (components and connectors) architectural style. The C2 is a combination of model-view-controller pattern in combination with the layered and event-based architectures. They used this style due to the benefits from the multiple methods of the design. In particular, the C2 style is sufficient for the failure behavior of a system. A component within this architecture has a limited relationship with others, and their hierarchy is built in a layered manner. The C2 style is characterized as an association of components linked by communication

forwarders known as connectors. This design focuses on the connector's independence in order to succeed the interchangeability and reuse of components across architectures. Components request services from components "above" them via message passing and are not in possession of knowledge of components "below" them.

Using the C2 style, we are in place to relate the architecture with the failure modeling because we want to make our modeling incremental and iterative and at the same time to distinguish between the styles, which ones are safety-related and which are not. Succeeding this as first step we are going forward with building the failure model that will give us the potential failure behavior of our system. The C2 style exhibits functional failure behavior, which is our motivation to present two aspects of failure behavior. The first concerns the individual components, and the second concerns the composition of failure behaviors between the components.

An example that explains the above rationale is the mapping of a C2 style with the triple modular redundancy (TMR) pattern as shown in Figure 6. With this approach there is a third redundant element to replace the two-way comparison with three-way "voting." In the various triple redundancy approaches, a faulty component can be identified and shut down while the remaining redundant elements can continue to operate safely.

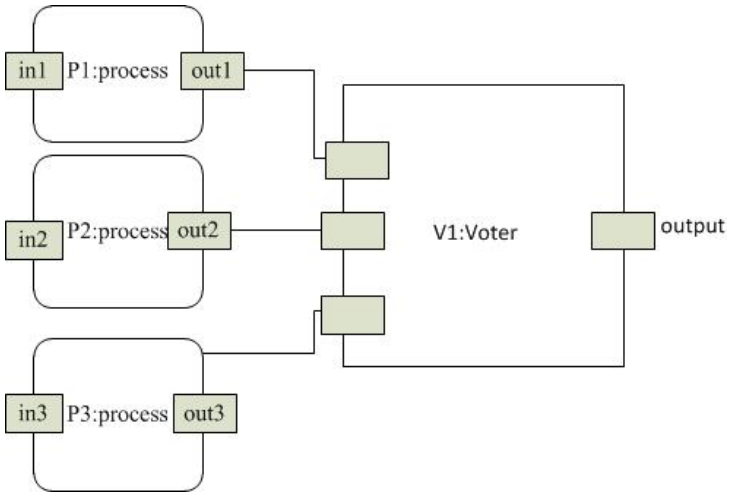


Figure 6. C2 style with the TMR pattern. From [6].

Relating the architecture above with the quality attribute safety, we distinguish between the failures events from the normal events in failure modeling. Hence, our convention rule is that all events must have one component describing normal or failure conditions. In addition, we have to define which of the already defined failure modes are in charge of the failure event component, making our analysis simpler because the complexity of the system could lead us into misunderstandings or misinterpretations. Having this in mind, Wu and Kelly investigate a possible protective mechanism to protect a processor. The most common failure behaviors of a processor can be: crash failures (i.e., permanent omission failures), transient timing failures, transient value failures and corruption failures (i.e., arbitrary timing and value failures).

They proposed a scenario in which the protective mechanism is a working watchdog timer, which is another architectural entity, and has the responsibility to detect omission and late timing failures. Creating the failure model of the architecture, we are able to simulate the potential results of any unexpected situations, depending on the implementation. In Figure 6, we see that they designed the system in such a way that the C2 style of the TMR system composes the four elementary processes P1, P2, P3 and V1. The three redundant and independent processes P1, P2 and P3 interact with the V1 process through their outputs. Each component P1, P2, P3 is responsible for computing the results based on the input data received, and propagates any incoming failure from its input to output ports. The next voter component is responsible for choosing the 'correct' result among three redundant input channels, and thus can detect one faulty input channel and stop its failure propagation. But the voting protection mechanism will be ineffective if two or more faulty channels agree with each other. The voter itself can generate omission failures (i.e., fails stop) since it cannot make a decision upon voting.

H. SUMMARY

The development of software-intensive systems is time-consuming and requires many resources. When the complexity of these systems becomes great and the systems control significant amounts of energy, then from the developers' perspective, these

systems must have safety implications, both on their design and on their use. If it is not possible to avoid or remove the hazards entirely, the risk of a mishap must be minimized.

System safety analysis is the first phase in which to identify the potential hazards for the system. If the system is software intensive, the requirements that are associated with it should be specified. The identified hazards and specified system requirements will be used to guide a safety-critical system's architectural design. Some examples of software safety requirements include limits, sequence of events, timing constraints, voting logic, hazardous hardware failure recognition, failure tolerance, caution and warning interfaces and hazardous commands.

For software-intensive safety-critical systems, software design must enforce safety constraints. Reviewers should be able to trace from requirements to lower level artifacts such as architectures, designs, code, and document and vice versa. In addition to the specific safety constraints developed for the system being designed, the design should incorporate basic safety design principles. Safety, like any quality, should be built into the system design. Operation of the system must not lead to a violation of the constraints on safe operation. The requirement for software to be safe is not that it never "fails," but that it does not cause or contribute to a violation of any of the system constraints on safe behavior. This observation leads to a group of approaches to handle software in safety-critical systems.

The first approach requires the use of current design methodologies from the existing software architectures to implicitly consider safety. Using the current architectural patterns, which are specific solutions for specific problems, the developers can reuse them on similar future issues. The literature review shows that the first approach cannot be implemented at an acceptable level in safety-critical systems. Furthermore, the existing software architectures do not succeed in implicitly considering safety. Studying the problems Wu and Kelly described in [6] discovered the primitive characteristics of the patterns defining them as tactics, which actually are the design decisions for realizing quality at the architectural design. They are abstract in that they can refer to the patterns' building blocks. The analysis of software safety using a model,

the development of safety-related tactics and eventually the method by which the tactics are implemented at the design phase is one approach to formalize design decisions.

The second approach identifies the constraints on system behavior and then designs the software to enforce the safe constraints. In this approach Wu and Kelly in [6] proposed the failure behavior of each component in terms of failure propagation and generation. The result is a failure modeling and the combination of different possible failure flows from external failures or components' internal failures to system-level failures. Having this as a baseline, they tried to create the architecture from the functionality and the operation of a software system following an architectural transformation, failure modeling, scenario generation and safety analysis for feedback. In architectural transformation they distinguish the components in elementary and composite, and focus on the components that are related to the failure modeling.

THIS PAGE INTENTIONALLY LEFT BLANK

III. ANALYSIS OF SOFTWARE SAFETY REQUIREMENTS

A. INTRODUCTION

System safety efforts for safety-critical systems often provide for the early identification of hazards and the elimination or control of those hazards through system design. Although this process has been proven reliable in providing safe and effective safety-critical systems, significant deficiencies exist when software that is utilized within the system is not adequately addressed. With the influx of software in the design, it is critical to ensure that software safety analysis is integrated into the system safety analysis process. With the proper analysis effort for all aspects of the system, and the proper integration of those efforts, a thorough identification and resolution of hazards will occur, whether those hazards are induced by a failure mode, adverse environment or software.

Thus, we have to begin with the definition of the proposed system, what this system is and for what purposes is it being built. In doing so, we will be able to continue with a general type of assessment about the potential hazards for that system. Looking further and deeper, we can describe the system's attributes, its functions and features that are not just fundamental for the design procedure, but are also essential for the safety part. When the design team has an initial view of what the system is, they will be able to look deeper and start to determine the potential causal factors that are related with the system's features and may cause or contribute to mishaps. In addition, the team should determine under what conditions the attributes, functions and features will cause a mishap.

As we discussed in Chapter II, the system safety analysis follows the system development life-cycle. The system is comprised of hardware, software and the interfaces between them. In practice, the development of a system begins with the hardware components to demonstrate the purposes of the system, and the software is created to operate the hardware [12]. In this phase, the design team with the cooperation of the stakeholders creates the systems requirements documentation, because the stakeholders have the general concepts for defining the system requirements. Having these as a

baseline, the design team will be able to define the software role and from this to specify the software requirements.

Starting from the requirements documentation, the design team understands the system, its purposes, what interactions should be operated inside the system and with the external systems, and, finally, what functions the system is able to have. An important chapter of this documentation is the one that characterizes the risk assessment of the system. This chapter identifies mishaps from the use of the system, defines the hazards and explains how a mishap could occur. This may concern the software that has direct or indirect control of the hardware [12], thus the main effort to mitigate the risk of hazard causal factors is on that. But there are cases in which, from the early stages of the development, the design team is not able to eliminate all hazards. For example, a weapon system, such as a guided missile, is designed to fly on air having a propulsion mechanism, which carries a warhead that releases energy by explosion. The release of lethal energy, under certain conditions, may be a hazard for people, products and the environment. In addition, the missile needs to travel through the air using another device that produces the proper kinematic energy to overcome gravity. This kind of energy comes as a result of the conversion of the thermochemical to the kinematic energy. The use of chemical to produce enough power to overcome gravity comes with potential hazards like explosion, high heat exposure and pressure release that could affect negatively the people, the product and the environment.

The severity associated with any mishap for each hazard is based on time, the potential for death or injury, the environmental impact and the monetary loss. A given hazard may have the potential to affect one or all of these areas. This study follows the severity and the probability categorization as they are described in [5] and are depicted in Tables 1 and 2.

SEVERITY CATEGORIES		
Description	Severity Category (S)	Mishap Result Criteria
Catastrophic	1	Could result in one or more of the following: death, permanent total disability, irreversible significant environmental impact, or monetary loss equal to or exceeding $10M.
Critical	2	Could result in one or more of the following: permanent partial disability, injuries or occupational illness that may result in hospitalization of at least three personnel, reversible significant environmental impact, or monetary loss equal to or exceeding $1M but less than $10M.
Marginal	3	Could result in one or more of the following: injury or occupational illness that may result in one or more lost work day(s), reversible moderate environmental impact, or monetary loss equal to or exceeding $100K but less than $1M.
Negligible	4	Could result in one or more of the following: injury or occupational illness not resulting in a lost work day, minimal environmental impact, or monetary loss less than $100K.

Table 1. Severity categores. From [5].

PROBABILITY LEVELS			
Description	Level (P)	Specific Individual Item	Fleet or Inventory
Frequent	A	Likely to occur often in the life of an item.	Continuously experienced.
Probable	B	Will occur several times in the life of an item.	Will occur frequently.
Occasional	C	Likely to occur sometime in the life of an item.	Will occur several times.
Remote	D	Unlikely, but possible to occur in the life of an item.	Unlikely, but can reasonably be expected to occur.
Eliminated	F	Incapable of occurrence. This level is used when potential hazards are identified and later eliminated.	Incapable of occurrence. This level is used when potential hazards are identified and later eliminated.

Table 2. Probability Levels. From [5].

Using the Tables 1 and 2, the design team is able to determine, qualitatively, the potential risks and express them as the Hazard Risk Index shown in Table 3.

RISK ASSESSMENT MATRIX				
	Severity			
Probability	Catastrophic (1)	Critical (2)	Marginal (3)	Negligible (4)
Frequent (A)	High	High	Serious	Medium
Probable (B)	High	High	Serious	Medium
Occasional (C)	High	Serious	Medium	Low
Eliminated (F)	Eliminated	Eliminated	Eliminated	Eliminated

Table 3. Risk Assessment Matrix. From [5].

The system's functions that are related to safety are defined as Safety Critical Functions and are identified during the Preliminary Hazard Analysis. These are functions within the system which are considered significant to safety, where their significance is determined by the impact of improperly performing the function. The safety critical functions are often related to the release of energy, application of power, movement of mechanical devices and movement of physical objects. To prioritize the software using probabilities is not practical as it is described in [5] and [12].

Software is generally application-specific and reliability parameters associated with it cannot be estimated in the same manner as hardware. Therefore, a different approach, which is based on the relation of the potential risk severity and the degree of control that software exercises over the hardware shall be used for the assessment of software's functions to mitigate the system's risk. The software control categories are based on [5] and are depicted in Table 4 as follows:

SOFTWARE CONTROL CATEGORIES		
Level	Name	Description
1	Autonomous (AT)	Software functionality that exercises autonomous control authority over potentially 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. (This definition includes complex system/software functionality with multiple subsystems, interacting parallel processors, multiple interfaces, and safety-critical functions that are time critical.)
2	Semi-Autonomous (SAT)	Software functionality that 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. (This definition includes the control of moderately complex system/software functionality, no parallel processing, or few interfaces, but other safety systems/mechanisms can partially mitigate. System and software fault detection and annunciation notifies the control entity of the need for required safety actions.)
3	Redundant Fault Tolerant	Software functionality that 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. (This definition assumes that there is adequate fault detection, annunciation, tolerance, and system recovery to prevent the hazard occurrence if software fails, malfunctions, or degrades. There are redundant sources of safety-significant information, and mitigating functionality can respond within any time-critical period.)
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 and 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 4. Software control categories. From [5].

Using the Tables 1 and 4, the design team is able to determine, qualitatively, the potential risks and express them as the Software Hazard Risk Index like the one shown in Table 5. It is anticipated that software with a high risk index will require thorough

analysis of system level requirements, software safety design and implementation source code to ensure adequate control of the causal factors as well as in-depth testing to ensure that the control measures are implemented correctly. Software with a medium risk index will require thorough analysis of system level requirements and software safety design as well as adequate testing to verify correct software response to errors and failure modes. Software with a moderate risk index will only require analysis of high-level requirements and verification of the satisfaction of these requirements via testing. No additional safety-related actions need to be performed for software with a low risk index [15].

RISK ASSESSMENT MATRIX				
Level of Control	Severity			
	Catastrophic (1)	Critical (2)	Marginal (3)	Negligible (4)
1	High	High	Moderate	Low
2	High	Medium	Moderate	Low
3	Medium	Moderate	Low	Low
4	Low	Low	Low	Low
5	Low	Low	Low	Low

Table 5. Software Risk Assessment Matrix. From [15].

Having the above as guidelines, we shall present a case study that involves the architectural design of a safety-critical weapon system, a fictitious Surface-to-Air Missile that is used to protect warships from attacking missiles and aircrafts. The physical description of the system is based on the description of a guided missile in [13]. The purpose of this study is not to build a new weapon but to demonstrate the process of software safety requirements engineering, safety-critical software architectural design and the formal validation and verification of the software architecture for safety.

B. SAMPLE SAFETY-CRITICAL SYSTEM – A SURFACE-TO-AIR MISSILE SMK

For the reader's convenience, we will first provide a brief description of the SMK, its purpose, its mission and its system architecture from the Systems Engineering activities. Then, we will walk the readers through the system/software safety engineering process to determine the software safety requirements for the system.

1. Context Model

a. Stakeholder Statement of Operational Need

The SMK system intends to improve ship self-defense capability against smaller, more maneuverable anti-ship missiles capable of approaching at lower altitudes. This is going to be achieved through higher maneuverability, improved sensors and a more lethal warhead. The operational needs for the development of the SMK are the following:

1) The need to increase the battle space because threats can be engaged at longer ranges due to the increased missile kinematics.

2) The need to be designed for surface launch.

3) The need to be guided by continuous wave (CW) radiation.

4) The need to be a semi-active homing missile. Homing guidance systems control the flight path by employing a device in the weapon that reacts to some distinguishing feature of the target.

b. Projected Operational Environment (POE)

The POE is the environment in which the system is expected to operate. It provides the necessary details to describe the mission areas, environment and types of locations to determine the operational capabilities under which the system will be designed. The POE provides information for establishing a context within which tasks will produce their measurable outcomes. The weather (clouds, storms, wind, rain, fog and warm/cold fronts) affects the radars, the guidance section of the missile and communications. Certain environmental conditions tend to create propagation phenomena of electromagnetic radiation such as ducting.

c. Mission Success Requirements

These requirements identify the individual activities that need to be accomplished in order to define the success of the mission. The activities identified for the success of this model will be measured in these categories:

1) Provide self-defense against anti-ship missiles.

2) Provide self-defense against air threats that are able to release air to surface missiles.

3) Detect and destroy air targets.

d. Operational Concept/Scenario

The survival of a warship from air threats requires rapid response and proper use of all sensors and use of the available weapons of the ships. It is important that the operator can quickly and unambiguously decide whether or not to fire weapons at any incoming target. For the successful treatment of an incoming target, it must be correctly identified by a sensor and then be trapped and illuminated with CW radiation by the tracking radar, allowing the shooting of the missile for the inhibition.

The following block diagram in Figure 7 depicts a typical configuration for the SMK missile onboard:

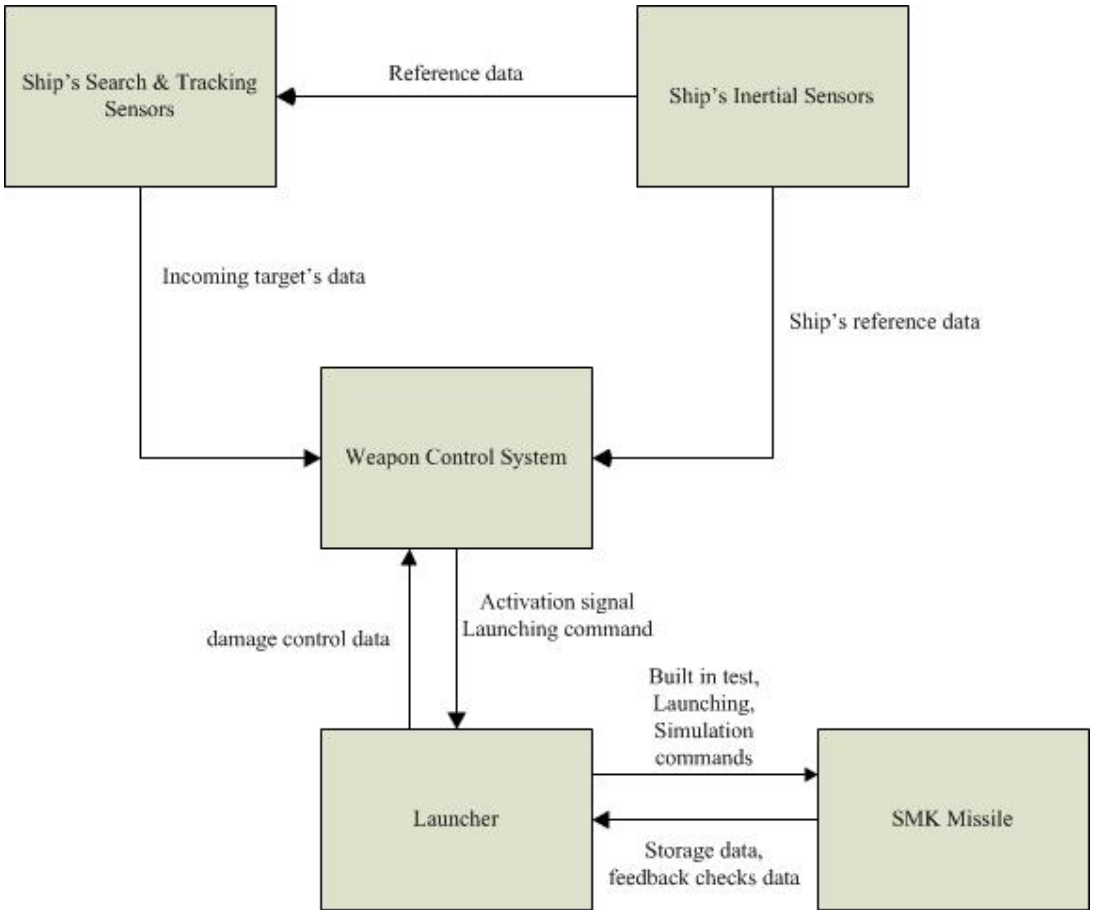


Figure 7. SMK configuration onboard.

Information on the upcoming threat can be drawn from sensors of the ship. These sensors can be used for initial detection and indication of the incoming threat and support weapon control system, which consists of tracking radar (TR) with its control console and the console control for weapons. The TR is used for the tracking of air targets that are shown on the display console for the weapons. Through the antenna of the TR, the necessary CW radiation is produced for guiding the SMK.

These sensors support the Weapons Control System (WCS) by providing real-time data for the target. The WCS is used to prepare and to provide ignition to the SMK's rocket motor for the firing procedure.

The missile should be stored in its canister, which is in a vertical launcher and via interfaces it is connected with the WCS of the ship. Physically, the canister provides storing, securing and positioning the missile before launch.

2. Physical Model

The physical context of the missile is a combination of several subsystems that are necessary for communication with the WCS, the safety launch of the missile and the successful intercept of an incoming threat. The main components/subsystems of a guided missile, like the proposed SMK, are based on the description of a weapon in [13] and are depicted in Figure 8, and their roles are described briefly.

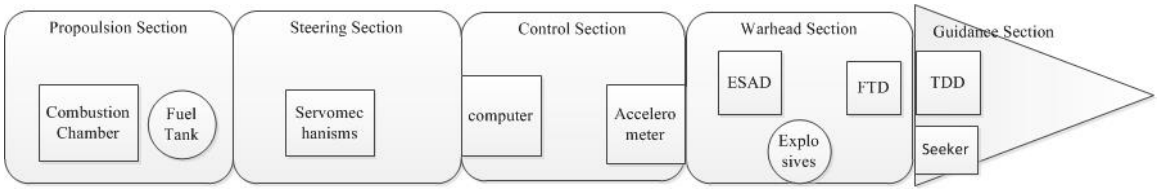


Figure 8. SMK's Block Diagram with sections and main parts. After [13].

a. Guidance Section (GS)

Due to the stakeholders' need for a semi-active missile, the Guidance Section (GS) should include a seeker device, which searches for the target and guides the missile towards the target in its terminal phase. At the semi-active homing, the target is illuminated by the tracking radar. The missile is equipped with a radar receiver (no transmitter) and by means of the reflected radar energy from the target, it formulates its own correction signals as in the active method. The major GS components consist of: the Radome, the Seeker Antenna and the Target Detection Device (TDD).

The guidance section communicates/processes prelaunch and post-launch data via the serial data bus, discrete signal paths, and analog signal paths, and performs the target acquisition and tracking functions. The TDD provides the missile with the target line of sight rate data, the homing error signals, midcourse data and other terminal missile guidance information to the missile's processor.

b. Warhead Section (WS)

The SMK missile includes a warhead. The warhead consists of a stainless steel case filled with explosive. The case contains discrete fragments bonded with its outer diameter in order to achieve the best result against missiles. The warhead case also incorporates the joints for mechanical connection to the Control and Guidance Sections.

c. Control Section (CS)

The Control Section (CS) bears this name because it has been developed to provide the transition electrical and mechanical interface connecting the other parts of the missile body through the commands provided by the onboard computer. The operating system of the computer receives the data from the other parts of the missile, makes the calculations and provides with messages the sections to act. The missile capability depends upon the components incorporated into this section. The CS carries out the major processing functions of the SMK, which include the tuning and launch sequencing, the missile's pointing commands computation necessary for the launch (superstructure avoidance-pitchover), the maneuverability control, the target acquisition and interception, and the launch simulation capabilities when operating in a Test/Training mode.

d. Propulsion Section (PS)

The rocket motor launches and accelerates the missile to the required velocity. The required power to propel a weapon to its target is obtained through the controlled release of stored energy. Every weapon requires some type of propulsion to deliver its warhead to the intended target.

A rocket motor is basically a device for converting a portion of the thermochemical energy developed in its combustion chamber into kinetic energy associated with a high-speed gaseous exhaust jet. The fuels and oxidizers are used to power the motor engine. The motor rocket consists of two basic parts: the combustion chamber, wherein the transformation of energy from chemical to thermal occurs, and the exhaust nozzle, wherein thermochemical energy is converted into the kinetic energy

necessary to produce an exhaust jet of propulsive potential. The chemical reaction between fuel and oxidizer in the combustion chamber of the jet engine produces high-pressure, high-temperature gases. These gases, when channeled through the exhaust nozzle, are converted into kinetic energy creating force acting in a direction opposite to the flow of the exhaust gases from the nozzle.

e. Steering Section (SS)

The primary function of the Steering Section (SS) is to provide pitch, roll and yaw controls during all phases of the missile flight.

3. Operational Overview

The SMK will have three modes of operation: remote, local, and test & training. In remote mode, the SMK will be fully operational and capable of launching. This is the normal mode of operation. In local mode, the SMK will be isolated from the WCS and the launcher. This mode supports maintenance and fault isolation using off-line BIT testing. In Test & Training mode, all functions between WCS and the SMK are the same as those in remote mode operations except that the CS operational program will be reconfigured to simulate a normal firing sequence.

The SMK will be designed and used to intercept and destroy the incoming threat. This requires that both fuzing and warhead detonation occur in such a way as to inflict mission critical damage to the intended targets. In order to perform this function, guidance and control systems are implemented to obtain the required terminal and intercept phase accuracy.

C. IMPLEMENTATION OF HAZARD ANALYSIS FOR SMK

As mentioned previously, this study is based on the lessons-learned and the system safety engineering in Chapter II, Section E to analyze the software of the missile control software.

1. Preliminary Hazard List (PHL)

The PHL is the first step to identify and list the potential hazards and mishaps that the proposed system might face. The proposed system is a weapon, which can release destructive energy into space in order to fulfill its purposes. SMK is a guided weapon that contains chemical substances which are able to produce fire and explosion. Table 6, which is based on Appendix F of [12], lists some generic hazards in three types: Operating/Maintenance Hazards (O/MH), Hazards to Launcher and Ship (HTLS), Hazards to Friendly Forces (HTFF).

Preliminary Hazard List			
ID	Hazard	Hazard Effects	Comments
Guidance Section (GS)			
GS-1	External Shock	Staff injury or death at the handling of the missile	O/MH
GS-2	Internal Shock	Property damage	O/MH
GS-3	Static Discharge	Property damage	O/MH
GS-4	Ionizing Radiation	Staff injury or death at the handling of the missile	O/MH
GS-5	Missile mistakes friendly aircraft instead of incoming missile	Injury, death and properties damage of friendly forces	HTFF
GS-6	Missile detects false echo of the missile due to 'mirror' effect instead of incoming missile and do not provide self-defense	Injury, death and properties damage of friendly forces	HTFF
Warhead Section (WS)			
WS-1	Premature Detonation	Property damage and/ or Staff injury or death at the handling of the missile	HTLS
Control Section (CS)			
CS-1	External Shock	Staff injury or death at the handling of the missile	O/MH
CS-2	Internal Shock	Property damage	O/MH
CS-3	Static Discharge	Property damage	O/MH

Propulsion section (PS)			
PS-1	Premature Launch	Property damage and/ or Staff injury or death at the handling of the missile	HTLS and O/MH
PS-2	Chemical Change	Property and/or environmental damage	HTLS
PS-3	Fuel and Oxidizer in Presence of Pressure and Ignition Source	Property and/or environmental damage and/or Staff injury or death at the handling of the missile	HTLS and O/MH
PS-4	High Heat Source	Property and/or environmental damage and/or Staff injury or death at the handling of the missile	HTLS and O/MH
PS-5	Contamination	Property and/or environmental damage and/or Staff injury or death at the handling of the missile	HTLS and O/MH
PS-6	High pressure	Property and/or environmental damage and/or Staff injury or death at the handling of the missile death at the handling of the missile	HTLS and O/MH
PS-7	Oxidation	Property and/or environmental damage	HTLS
PS-8	Hang-fire (excessive delay between ignition and thrust)	Property and/or environmental damage	HTLS
PS-9	Hang-up (missile remains on launcher but thrusts)	Property and/or environmental damage	HTLS
Steering Section (SS)			
SS-1	Hitting the launcher due to incorrect trajectory	Property damage	HTLS
SS-2	Hitting the superstructure of the ship due to incorrect trajectory	Property damage	HTLS

Table 6. Preliminary hazard list for SMK. After [12].

2. Preliminary Hazard Analysis (PHA)

The PHA is the first source of system safety requirements that includes the hazards, their related causal factors, the level of risk and their mitigating measures. From this analysis, the design team is able to define the software system safety requirements and how the software design could control or mitigate these hazards. The purpose of the PHA is not to determine whether the hazard might occur or not, but to assume that the hazard can occur and what the consequences are. In this study we are going to examine the hazards from the list in Table 6, analyze their causal factors and link them to software functions to yield the software safety requirements.

Due to the fact that the proposed system is complex, this study will focus on one hazard with the highest priority, the premature detonation of the warhead, in order to demonstrate the software safety engineering procedure, and how to relate the safety to the software architecture and the formal validation and verification of software safety requirements and software architecture. Premature detonation is a hazard that under specific conditions could lead to mishap. When the detonation of the explosives happens in the proximity of personnel, it could lead to injury or death. In addition, when the detonation happens inside the Launcher or near the launching warship, this could lead to product damage that is serious. Due to the severity of the mishap, it is important to investigate this hazard thoroughly and mitigate the related risks. Table 7 presents an initial analysis about this hazard and the potential mishaps, based on the criteria from Tables 1, 2 and 3.

SMK Initial Analysis						
Mishap	Hazard	S	P	Hazard Risk Index	Causal Factor	Remarks
Operating and Maintenance Personnel: injury, death	Premature Warhead's Detonation	1	C	High	Personnel in proximity to launcher during maintenance use	Caused by incorrect firing command from the component that create the detonation
		1	C	High	High Heat Source	Caused by incorrect protection mechanism to prevent fire and explosion
		1	C	High	Moisture Oxidation	Caused by incorrect protection mechanism to prevent chemical change
		1	C	High	Static Electricity	Caused by incorrect protection mechanism to prevent the presence of static electricity
Warship's Launcher Destruction:	Premature Warhead's Detonation	1	C	High	Inadvertent Warhead's function	Caused by incorrect firing command from the component that create the detonation
Warship's Superstructure Destruction:	Premature Warhead's Detonation	1	C	High	Inadvertent Warhead's function	Caused by incorrect firing command from the component that create the detonation

Table 7. SMK Premature Warhead Detonation - Initial Analysis. After [15].

3. System Hazard Analysis and Software System Hazard Analysis

For the above analysis, the design team decides to add mechanisms to mitigate the risk of a premature detonation, based on the design criteria from [14]. The risk's mitigation is achieved in two ways. The first is the installation of the proper hardware (safety devices), and the second concerns the software that is installed in the control section's computer and operates the functions of the missile. Safety will be ensured by the sequence of commands leading to the actual detonation command, so that detonation does not occur during any phase of missile flight except as a result of a proper firing signal.

To eliminate the impact of the high heat source and the intrusion of moisture and other contaminants, the housing of the warhead should include forward and aft enclosures providing an environmental seal. The enclosures should provide these functions under all environmental conditions specified. In addition, because the components inherently use electrical and electronic parts, the static electrical charges are causal factors to detonate the explosives. Thus, the warhead has to contain provisions to discharge to ground any buildup of static electrical charges.

The major physical components of the warhead section are the Warhead Assembly, which contains the explosives and the fragments under environmental shield, the Electronic Safe and Arm Device (ESAD), which uses an initiation system compatible with the explosives of the warhead assembly, and the fuze triggering device (FTD) that starts the detonation of the warhead when it is armed by the ESAD. Each component performs functions to achieve the purpose of the warhead, which is the release of destructive energy.

The warhead assembly performs the storage, the environmental protection of the explosives and the production of a cloud of blast overpressure and high velocity fragments. The FTD performs the initial signal for the explosion, either when the TDD detects the missile's proximity to the target and generates a fire pulse to the ESAD (proximity) or when it generates a fire pulse upon target impact.

Finally, the ESAD is a device that has two states unarmed/safe and armed and the switch. Its initial state is unarmed, and it switches to the armed state only when it receives the commands from the CS's computer related to the arming of the warhead. In the cases that it does not receive any message or receives an abort Safe message from the computer then it remains in unarmed state (implementation of safety logic). The other major hardware parts that the ESAD incorporates to execute its mission are:

1) the accelerometer, whose function is to measure the acceleration values of the missile. The CS' computer receives these values from the accelerometer and calculates the travelling distance at the missile's long axis by the double integration of the acceleration.
2) the CS's computer processor has implemented by its software the safety logic about the arming of the missile's warhead and is responsible for the following functions:
    a) Control the execution of the Power-On Self-Test (POST). The POST ensures that all ESAD inputs and safety signals are in the correct states prior to the launch command for the SMK missile.
    b) Verify that the Launch Indicate Signal occurs.
    c) Verify that the missile's movement has been achieved Perform the integration and double integration of the incoming acceleration signals.
    d) Verify the respective acceleration profiles during the SMK post-launch target trajectory.
    e) Enable the arming circuits
    f) Process the command fire signal issued by the TDD of the Guidance Section or by the contact fuze.
3) the explosive train which contains the detonator, the lead and the booster. Its function is the detonation initiating by the secondary explosives (e.g., primer, detonator) and terminating in the main charge (high explosive, pyrotechnic compound).

Figure 9 presents the fault tree analysis for premature warhead detonation when the missile is in one of the two phases. In the left main branch, the missile is in the launcher at storage, and in the right the missile is preparing to launch. While the missile is in storage position inside the launcher, there are potential causal factors like the fire/high heat source that under specific conditions could lead to a mishap, which could result in the destruction of the launcher and death or severe injuries when personnel are in

the proximity of the detonation. Another mishap could occurr by the detonation of the warhead during the early stage of launch, which could result in the destruction of launcher, the ship's superstructure, and death and severe injuries when personnel are in the proximity of the detonation.

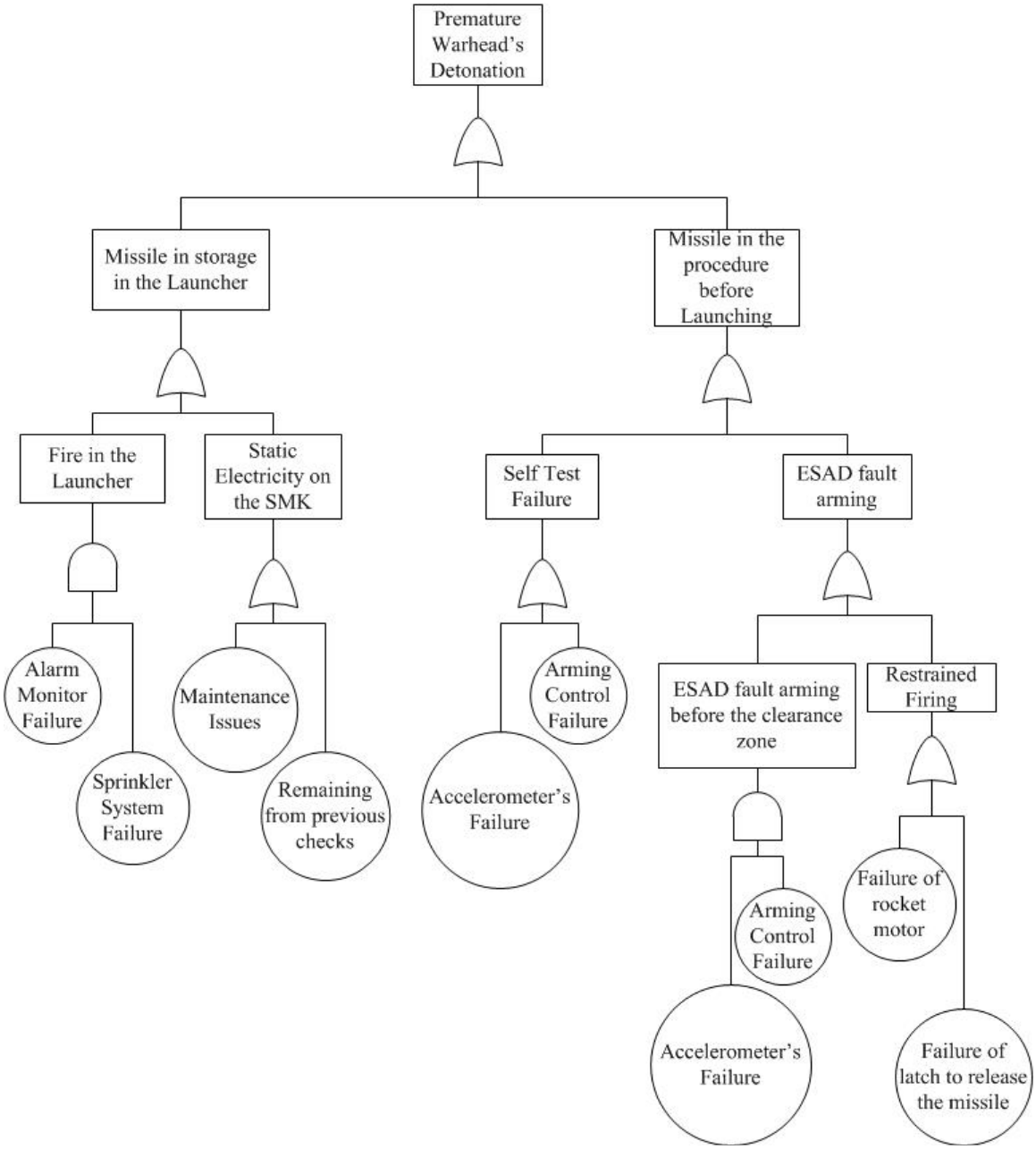


Figure 9. Fault tree analysis for premature warhead detonation.

From the fault tree analysis, the design team is able to identify the causal factors and then propose a plan to mitigate the risk of the above causal factors. Hence, the design team is able to document the safety requirements for the warhead in order to prevent the premature detonation as follows:

1) The warhead of the missile has to be in a safe-unarmed condition until the missile has intentionally been launched and has traveled a distance from the launching ship.
2) It should remain safe during launch shock and flight vibration.
3) It should only be armed after proper conditions of acceleration and time reached. These values determine the distance that the missile has to reach from its stored position until the elevation of the superstructure. This distance is characterized as clearance zone or safe separation distance from the warship.
4) Any component failure or abnormal environmental input must cause the ESAD to enter the "Fail-Safe State," dudding the missile.

Safety guidelines from [14] require that any Safe and Arm device sense two independent environments to satisfy arming requirements. One of these environments must occur only after launch; the other may be an "irreversible intent to launch."

Table 8 provides a summary of the above conditions and safety measures for preventing premature donation of the warhead.

Causal factor	Condition	Safety measure	Remarks
Power On self-Test failure	It does not pass the self-test	Disable the ESAD to arm the warhead	
	The self-test was successfully executed but it exceeded the time limit	Disable the ESAD to arm the warhead	
ESAD fault arming	The missile does not receive the launch command from the WCS in specific time	Disable the ESAD to arm the warhead	Either corrupted launch command or no launch command performed
	The missile receives the launch command but its rocket motor does not perform proper acceleration to move from its position in specific time	Disable the ESAD to arm the warhead	Proper acceleration is defined by two consecutive acceleration readings equal to 6 g's or above.
	The rocket motor does not perform proper acceleration to reach the safe distance	Disable the ESAD to arm the warhead	The double integration of the acceleration values is under the twenty meters vertical distance required to clear from the warship's superstructure

Table 8. Causal factors and conditions for the premature detonation.

4. Software Safety Requirements

The design philosophy for safety-critical systems places safety above all other considerations. Any component failure or abnormal environmental input must cause the ESAD to "Fail Safe," dudding the missile. In addition, the detonation of the warhead could only be invoked after the successful completion of the arming process and from the result of booster operation following a proper firing signal from the ESAD.

The basic function of the ESAD is the maintenance of the missile's warhead in a safe/unarmed condition until the missile has intentionally been launched and has traveled a specific distance from the launching ship. At the proper time, the ESAD arms the warhead so that upon receipt of an electric firing pulse, the firing train will initiate the warhead section.

The arming process of the ESAD begins with application of power to activate the whole missile after the CS's computer receives a command from the Weapon Control System via the connection with the launcher. The CS'S computer monitors all the missile's functions, and for the arming process, it performs checks of the environmental input lines and safety switch to verify that everything operates correctly (Power On Self-Test, POST). Failure of any of these checks will cause the ESAD to enter fail-safe, duding the missile.

The first arming environmental condition is the electrical signal Launch Indicate. Since the operator cannot intervene to abort the launch after this signal is generated, it meets the criterion for an irreversible intent to launch. The second arming environment uses accelerometer-derived data to define the movement of the missile and when the missile reaches the safe separation distance from the warship. These calculations are based on the double integration of the accelerometer in time. There is an additional check on the minimum arming distance calculation: if this distance is achieved before the independent flight timer times out, it is indicative of a fast accelerometer clock. In this event, arming is postponed until the independent flight timer times out and the delayed arm point is achieved.

Due to the fact that the SMK missile has three operational modes (remote, local and test&training), there are additional measures to increase the safety level. In the remote mode, personnel must activate the missile prior the launch in order to execute the BIT. To avoid confusion between this event and the launching procedure, prior the missile's activation personnel will select the option "BIT" from the WCS console to enable a different process for the missile. After the execution of the BIT the missile remains in an idle state waiting to be launched or to be turned-off. For this reason the WCS stops providing signals to the missile in order to avoid mishaps. When personnel receive the order to launch the missile, they activate it again selecting the option "LAUNCH" from the WCS console and push the FIRE button. The missile changes its state from idle to ready-for-launch and it follows the launching procedure. During this critical period the CS computer receives the signal's activation, launching and the data for the target (course, speed, altitude) and the data for the launching ship (course, speed,

altitude) from the WCS to create its reference system. In the case that any of the safety requirements is not met, the CS's computer commands the ESAD to abort and remain in the unarmed/safe state, thus duding the missile. In the other two operational modes, the missile does not receive the Launch Indicate signal because the switch that determines the mode on the WCS console does not allow this signal to be received by the missile. In the special case that the personnel is on training and they exercised on the launching procedure, the software of the WCS creates a simulated environment but no signal is transited to the missile.

The software in the CS's computer monitors the conditions and the signals from the WCS and the accelerometer, as indicated in Table 8, to decide on ESAD's status (SAFE or ARM). The control software must detect, identify any related erroneous states and prevent them from occurring, according to the following software safety requirements.

a. Software Safety Requirement 1 POST (Power-On Self-Test)

The CS's computer receives an activation signal from the WCS to power on the missile. It executes the POST, which must be completed within two seconds after the receipt of the activation signal and should be POST_OK in order to continue the arming procedure. If POST is overtime (>2 sec) or invalid, then it transits to the Fail-Safe state.

b. Software Safety Requirement 2 Launch Indicate

The Launch Indicate Signal represents the irreversible commitment to launch the SMK that is transmitted from the WCS. It is the significant signal to commence an actual launch of the missile. In the case of Built-In test, the command Launch does not exist in the test procedure (safety requirement). For the other cases, the order of a launch is committed. The launching signal must be received within four seconds after the powerOn.

c. Software Safety Requirement 3 First Motion Detection (FMD)

The First Motion function determines if the first two launch acceleration criteria have been satisfied. When the missile starts to move, the accelerometer sends acceleration values to the CS computer for missile displacement calculation. The detection of first motion is defined as two consecutive accelerometer readings of over 6 g's occurring within four seconds after the missile starts to move.

d. Software Safety Requirement 4 Safe Separation Distance (SSD)

The Safe Separation Distance function verifies that the acceleration of the missile is increasing during launching and the SMK reaches the minimum travelling distance (referred to as Safe Separation Distance) under any conditions (e.g., performance's fluctuations) in the specific time. If Safe Separation Distance is not satisfied, then the ESAD has to remain in safe mode. SMK should have reached the minimum distance of 20 meters within 6 sec after the missile starts to move. The minimum distance is based on the double integration of the acceleration received from the accelerometer.

IV. SOFTWARE ARCHITECTURE FOR SAFETY-CRITICAL SYSTEMS

A. INTRODUCTION

Having defined the software requirements of the system in meeting the stakeholders' expectations and software safety requirements to mitigate the risk of unsafe system behavior, the next step is to design the product. If we want to define the term 'design,' we will find many different definitions from the published literature, but all of them have the essence of a primitive version of our product. This version begins with the engineers' effort to present the elements and the structure that will comprise the proposed system.

The design of today's complex systems is time consuming, and it demands resources. The origin of systems design is the result of human experience in civil engineering over the centuries. The early step of this branch of engineering took the needs of humanity into account from the environmental conditions and, using the experience in relation to the sciences like mathematics and physics, resulted in the generation of the architecture. The term 'architecture' comes from the ancient Greek 'αρχι' (pronounced "archi") and 'τεκτων' (pronounced "tekton"), which basically mean essential and builder, respectively.

In the case software, its architecture provides the baseline for both the design and the development. The design and the architecture are closely related. We can infer that a software design is an instance of specific software architecture. As Taylor et al. pointed out in [4] that all software will have architecture, whether we plan it or not. However, it may not be well documented; it may be ad-hoc.

The dilemma that any designer can face is: "Do we need the architecture paradigm to build our system? Or can we follow our instinct or past experience to build the system without an explicitly defined architecture?" To answer these questions, we can follow a naïve path with the no-architecture option in our design. We already have the requirements from the stakeholders and we have augmented the requirements to address

the safety concerns resulting from our hazard analysis. Now, we are faced with the challenges of making many major design trade-off decisions to come up with a design that satisfies both the functional and non-functional requirements. These design decisions encompass the system structure, the functional behavior, the interaction, the nonfunctional properties (like security, safety, availability, etc.) and its implementation. We need models to reason and compare different design alternatives for achieving the desired system behaviors and properties. We also need ways to document the rationale and assumption for our choices to enable appropriate and effective changes in the design as the system evolves to incorporate additional behaviors and properties in the future. The above needs can be addressed by incorporating conscious, deliberate architectural activities in our design process, whose outcome will be a system/software architecture that captures

> "a set of principal design decision made about a system; it is a characterization of the essence and essentials of the application" [4].

In software, as in systems, the engineers try to minimize the cost of the whole program by reusing ideas and techniques from previous and similar projects. One way of effective reuse of design techniques is design patterns. As Maier and Rechtin discussed in [16]:

> "Design Patterns give abstract solutions to commonly recurring design problems, have been widely used in the software and hardware domain. As non-functional requirements are an important aspect in the design of safety-critical embedded systems, this work focuses on the integration of non-functional implications in an existing design pattern concept."

Design patterns are efficient solutions that worked in previous similar problems giving sufficient results. They all have an abstract representation, which can be customized and applied to different applications during the design phase.

B. ARCHITECTURE-BASED PATTERNS

Software architectural patterns could be generally efficient for many similar systems, but they can also be refined and specialized for each system. The designers have a pool of patterns that are applied to different areas like communications, security, etc. As

in hardware design, the decision about which patterns are going to be followed depends on the stakeholders' requirements and system constraints. Using the architectural patterns, the design team has a powerful tool to improve their productivity because they can reuse solutions that were best practices for similar problems. This reduces the development time and can improve the quality of the solutions.

The advocates of architectural patterns claim that the patterns could minimize the complexity of the product. This is particularly true for systems that are comprised of integrated objects, their processes and the frequency of interactions between them. Another merit is the improvement of the product's qualities by the incorporating practices that enable the designers to deliver the best output.

These two merits are very beneficial when the project is similar to another that has already been delivered, and the engineers have clear idea about the problems to be solved. However, the design team will face the danger of choosing the wrong patterns when they are working on problems that do not have preceding examples. The interpretation of the requirements and the transformation to specification is a non-trivial procedure. If design decisions were made without correct understanding and interpretation of the requirements, it is highly likely to yield faults and errors in the design. In particular, the intent and meaning of non-functional requirements should be considered thoroughly during the development. One of the most difficult parts of this process is the correct formulation of timing and safety requirements for safety critical systems, which is hard to do without the extensive prototyping as Shing and Drusinsky describe in [23]. It is important that the design team matches the design patterns against these stakeholder requirements, and at the same time, addresses the safety concerns from the outputs from the preliminary hazard analysis. It is also important that the design team validates and verifies the architecture design as early as possible, and as often as possible, in the development process.

C. SAFETY PATTERNS

The two mechanisms, which improve the safety, are the redundancy and the separation of safety and non-safety channels. When we use the term channel in this

particular field, we mean the medium, which is independent from its physical implementation that is used as a path for information to receive any kind of data and to produce some output under specific safety policy control [17]. What and how channels are used are principal design decisions made by the designers to realize the desired behavior and properties of the system. In many cases, an additional separation between the control and the safety-correlated entities could provide another approach for the design team to achieve the safety in the design.

There are two major approaches in achieving redundancy: to duplicate similar entities or to develop with different ways the same mechanisms. For these reasons the redundancy is either Homogeneous or Diverse. In the first case, the pattern is called a Homogeneous Redundancy Pattern, in which multiple replications of the same entity, either hardware or software, are used to run simultaneously, providing outputs that are compared at the end, as is shown in Figure 10. Using this pattern, we could spend resources to implement one channel and then replicate it. However, this kind of redundancy cannot detect and prevent errors in the design.

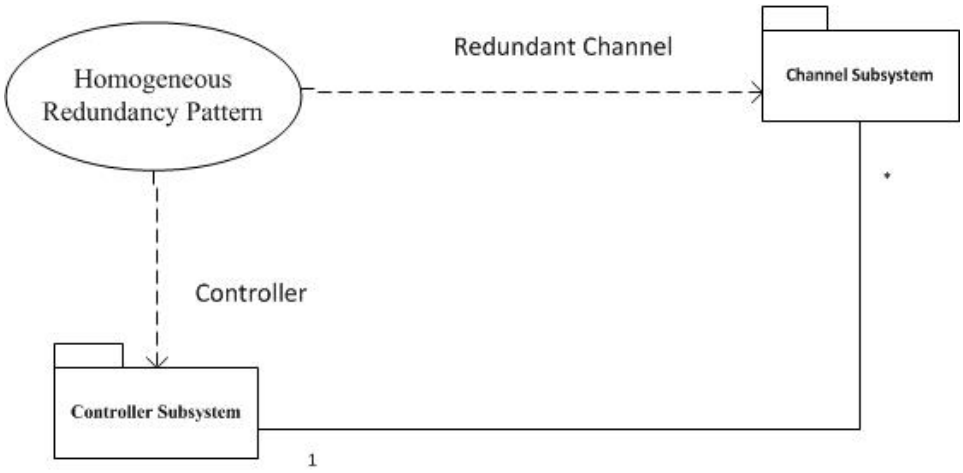


Figure 10. Homogeneous Redundancy Pattern. From [17].

The next option is the Diverse Redundancy Pattern that implements the same channels with different mechanisms using primary and secondary channels. The channels should be equal but different. There are two methods: the first one looks like the

Homogeneous pattern but each channel is implemented with different components, and they are not identical. This improves the protection against design errors because there is a different design rationale behind each channel. Correctness is determined by comparing the output produced by different algorithms using the same inputs, as is shown in Figure 11.

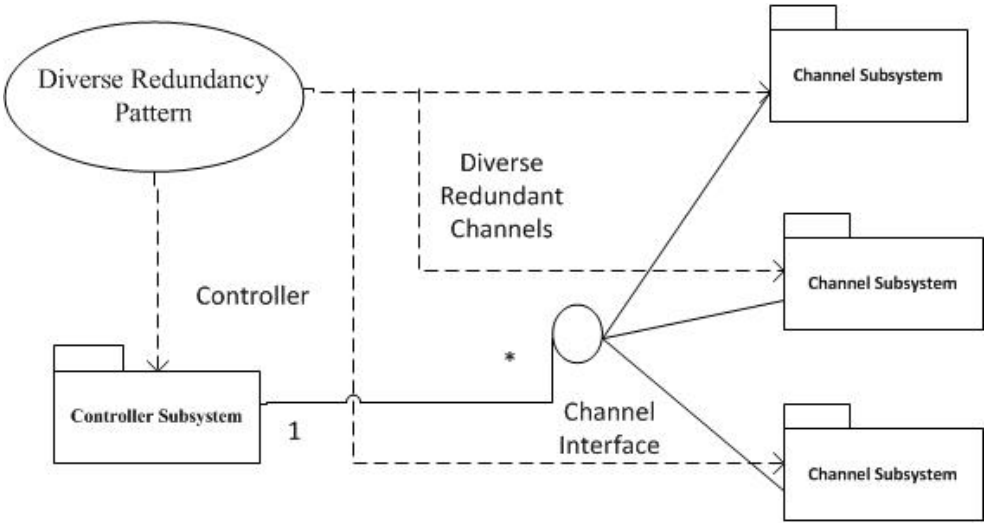


Figure 11. Diverse Redundancy Pattern. From [17].

The second method uses light-weight redundancy for the channels, having the secondary channel responsible for monitoring the actions of the primary channel and enforcing a set of policy rules for the whole system. A special case of the Light-Weight Diverse Redundancy Pattern is the Monitor-Actuator Pattern shown in Figure 12. The actuator channel receives the stimuli from the system's environment and performs the calculations to generate the actions. Simultaneously, the monitor channel ensures that the actuator's actions are proper, based on the system's specifications under the current environmental conditions. The monitor channel will detect the failures from the actuator channel and execute proper mechanisms to handle faults. As we can understand from the structure of these patterns, the Diverse Redundancy Pattern is preferred for safety-critical systems, because it is more reliable than the Homogeneous Redundancy Pattern in detecting errors in that it deals with a system's safety using multiple implementations to detect errors that could lead to failures of the system.

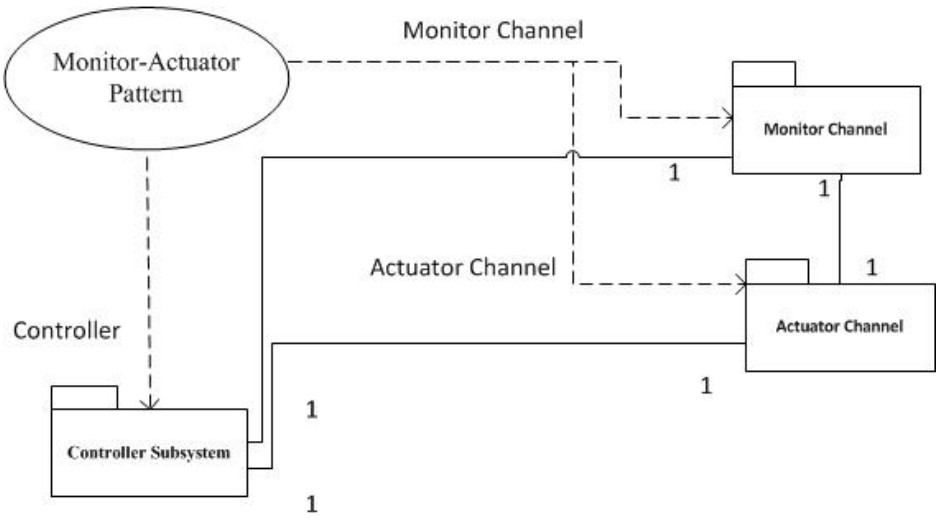


Figure 12. Monitor Actuator Pattern. From [17].

Another pattern that is commonly used on real-time embedded systems is the Watchdog Pattern, so named because it handles the timing constraints, as shown in Figure 13. The pattern uses an additional subsystem, the Watchdog, to track the timing of the events and to take corrective measures when there are illegal latencies or premature responses to the events. These corrective actions could only reset the system, shut it down, alarm the operators or initiate an error-recovery mechanism.

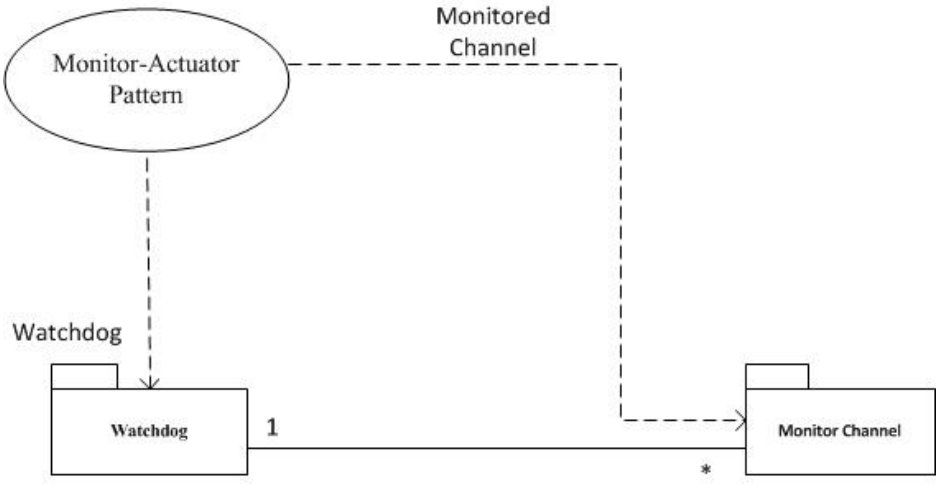


Figure 13. Watchdog Pattern. From [17].

Another pattern similar to the Watchdog is the Safety Executive or Safety Kernel Pattern shown in Figure 14. The concept of this pattern is based on the design rationale of the kernels that are used in the operating systems. The primary scope of the safety kernel is to ensure that the system cannot enter an unsafe state. Thus, it is characterized as a centralized coordinator that tracks and monitors all safety issues.

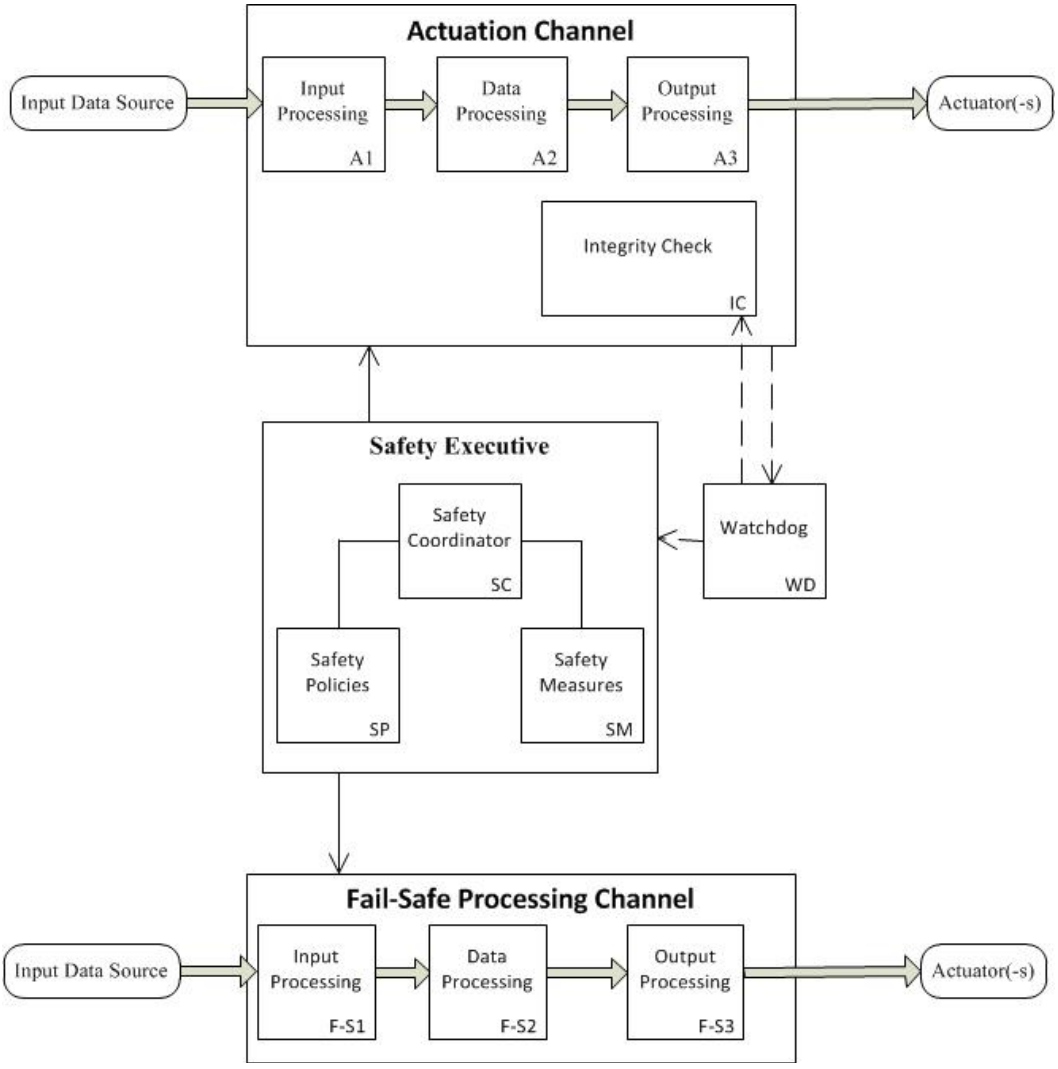


Figure 14. Safety Kernel Pattern. From [18].

This pattern uses a slightly different rationale from the Watchdog. In this case, the Actuation Channel is the path in which the information passes from the sensors or from the users to the actuators that are responsible for the execution of the commands,

providing the functionality for the system. The Fail-Safe processing channel is dedicated to executing and controlling the failures. In the case of the Watchdog pattern the role of the supervisor is played by the watchdog component, but in the Safety Executive pattern this role is dedicated to the Safety Executive component. The Safety Executive component is independent from the application programs providing the ability for the designers to focus on the safety policies and their safety measures based on the hazard analysis, as Douglass states in [17].

Input data from the Input Data Source, which can be external sensors or the user, are fed into to the processing units in the Actuation Channel and the Fail-Safe Processing Channel. Besides the three abstract processing units (Input Processing, Data Processing and Output Processing), the Actuation Channel contains a fourth computation unit, the Integrity Check. This component communicates directly with the Watchdog to check the correctness the three processing units of the actuation channel.

The most important component of this pattern is the Safety Executive. It comprises of the Safety Coordinator, the Safety Measures and the Safety Policies. The Safety Executive communicates with both the Actuation and the Fail-Safe Processing Channel. The Safety Policies consist of a set of rules, which emerge from the safety specifications. The Safety Measures contain a set of actions that are taken to prevent any identified failure from occurring. And finally, the Safety Coordinator is used to control and coordinate the safety processing policies with the measures. It also executes the control algorithms that are specified by the safety policies. The Safety Executive component does not provide the fidelity of the control or action, but it keeps track of whether the events and actions violate these policies. When this happens, it acts as a reference monitor, examining the actuator commands prior to their execution and determines which safety measures have to be executed.

In addition to the Safety Executive Component, there is the Watchdog component, which communicates with the Actuation Channel and with the Safety Executive. The watchdog receives stimuli messages from the components of the actuation channel in a predefined timeframe. If a message violates its predefined timing constraint or is invalid, as concerns its integrity, the watchdog considers this situation as a fault in

the actuation channel and it alerts the Safety Executive. Then the Safety Executive determines, through the Safety Coordinator, the corrective action by sending command signals to both the Actuation Channel and the Fail-Safe Processing Channel.

As Douglass proposed in his book [17], the channels that are responsible for controlling and monitoring the data flow (Actuation channel, Fail-Processing Channel) have to be physically separated and have their own memories and processors. This separation improves the ability of the whole structure to prevent any channels' failure to affect the others. The strength of this pattern is the way that the set of the Safety Policies can be implemented. Not only are they the system's safety specifications, but they can be modified, removed or added with their related measures without changing anything from the rest of the application programs.

For the purposes of this thesis, we chose the Safety Executive Pattern to demonstrate a proposed pattern-based solution for the case study of the SMK missile.

D. A SAFETY KERNEL FOR SMK'S WARHEAD

The ESAD is the safety and arming device for the SMK, which is assembled into the SMK's warhead section. It maintains the safety of the SMK's warhead throughout the entire stored position to target interception sequence prior to intentional arming. The arming sequence is completed when the ESAD control software, which runs on the CS's computer, has received the necessary enabling signals from the WCS and all safety policies are met. If any event that violates any of the safety policies, the ESAD control software disarms the warhead (safety measure). The safety policies are shown in Table 9.

No	Description of kernel-enforced policy	Safety Measure
1	Power On Self-Test (POST) has to be occurred within 2 seconds after the receipt of the power On signal and must be valid.	If POST fails or does not occur within a 2 seconds time window after the power on of the missile, the ESAD will abort in the SAFE mode.
2	The Launch Indicate Signal represents the irreversible commitment to launch the SMK. (A Launch Indicate Signal failure prevents the ESAD from ever initiating its arming sequence)	If the Launch Indicate Signal does not occur within a 4 seconds time window after the activation of the missile (receipt of the powerOn signal), the ESAD will abort in the SAFE mode.
3	The First Motion Detection has been satisfied within the 4 second window from the moment that the missile starts to move. Due to the movement of the missile, its accelerometer starts to provide values to the CS's computer for calculations. (two consecutive acceleration readings above 6 g's within the time period)	If these criteria are not satisfied within the 4 second window, then the ESAD will abort in the SAFE mode. (A First Motion Detection failure prevents the ESAD from ever initiating its arming sequence).
4	The Safe Separation Distance processes the double integral of the missile acceleration within 6 seconds after the activation of the accelerometer to determine that the vertical distance above the warship's superstructure has been achieved. (The time period of 6 seconds is the maximum time that the missile expected to fly over the ship)	If the Safe Separation Distance is not equal or more than 20 meters within 6 seconds after the activation of the accelerometer, then the ESAD will not arm

Table 9. Safety policy and safety measures.

The components of the safety kernel for the ESAD control software are depicted in a package diagram, Figure 15, which implements the rationale of the Safety Kernel Pattern in this case study.

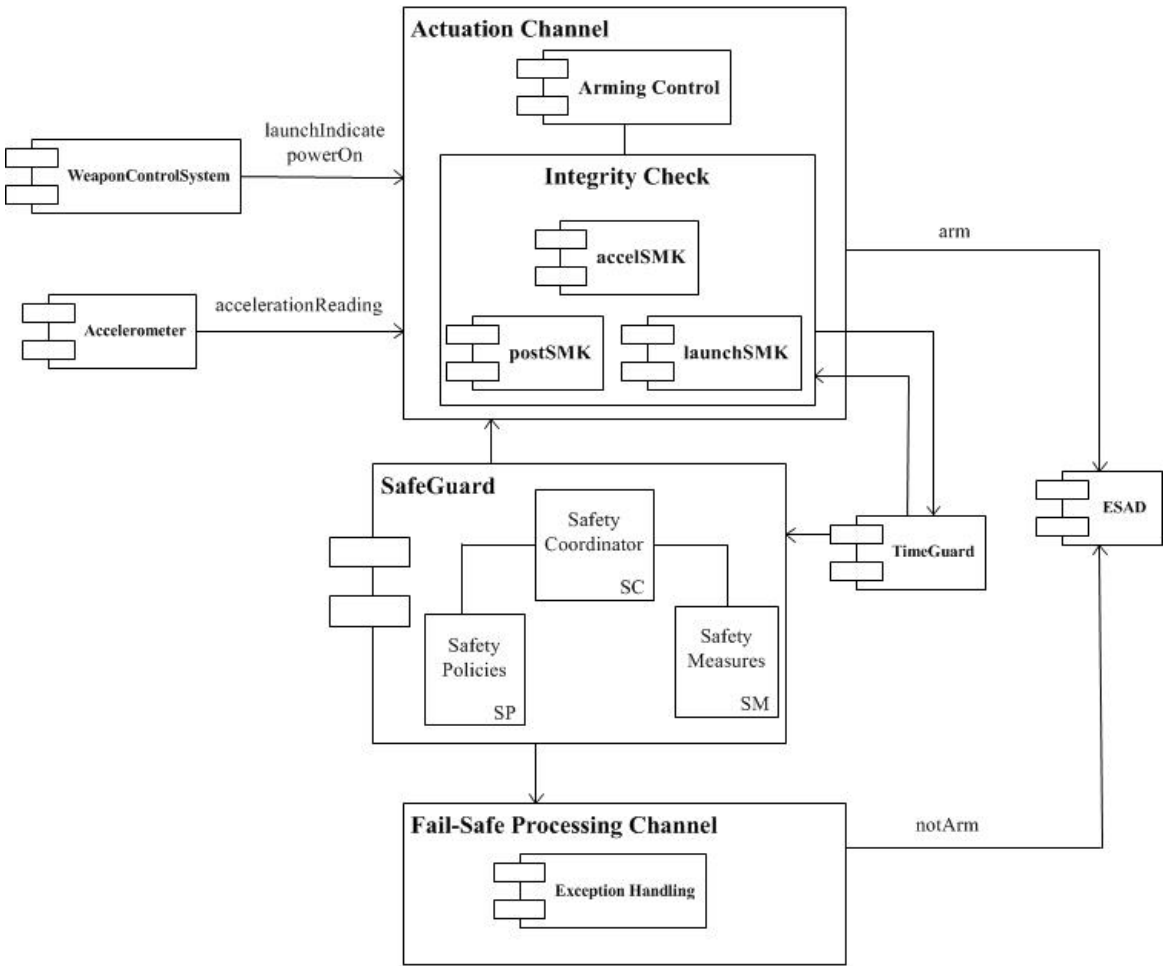


Figure 15. Safety executive pattern for the SMK's warhead. After [18].

From the above diagram the role of its component and its scope are analyzed as follows:

1) WCS: It is the Weapon Control System (hardware) that sends the powerOn and makeLaunch signals to the missile in order to initiate the launch or in a special case to make the simulation/maintenance of the missile. In the second case there must not be the makeLaunch signal in the simulation process.

2) Accelerometer: It is the second hardware that communicates through the Actuation Channel with the ESAD control software. It sends the acceleration readings due to the ignition of the rocket motor or in the simulation procedure it sends values in a specific time line to create an environment that simulates the real behavior of a launching system.

3) ESAD: It is the third hardware component that receives the command to remain unarmed or to arm from Fail-Safe Processing or Actuation channels, respectively.

4) postSMK: It is a software component that receives the activation signal from the WCS and makes the initial checks prior the launch of the SMK. It is a part of the Integrity Check component since it performs a data integrity computation and it sends its output, the endPost signal, to the Arming Control component and to the TimeGuard, respectively, as part of the required launch event sequence, when the result of the self-test is passed. In the case that the result of the self-test is failed then the postSMK sends the invalidPost signal to the TimeGuard component to activate the SafeGuard.

5) launchSMK: It is a software component that senses the Launch Indicate signal, an irreversible launch environment, from the WCS. It is part of the Integrity Check component since it performs a data integrity computation and it sends its output, the doLaunch signal, to the Arming Control component and to the TimeGuard, respectively, as part of the required launch event sequence.

6) accelSMK: It is a software component that receives the acceleration readings from the accelerometer and computes the distance travelled by the missile after the ignition of its rocket motor due to the launch command from the WCS. This component determines two conditions, the first is the proper sequence of the acceleration values and the second is the double integration of the acceleration readings. For its first condition, the missile's rocket should perform under specific conditions acceleration values that are increasing and overcome the value of 6 g's twice consecutively in order to verify that is capable to move and the holding latch is clear from its body. For the second condition, accelSMK verifies that the missile transits to the space producing acceleration readings and, through the double integration of a specific timeline, the result is equal to or more than the minimum travel distance to clear the superstructure of the warship. It is part of the Integrity Check component since it performs a data integrity computation and it sends its outputs, the endFirstMotionDection and endSafeSeparation, to the Arming Control component and to the TimeGuard, respectively, as part of the required launch event sequence.

7) ArmingControl: It is the software component responsible for changing the state of the ESAD from unarmed to arm under specific conditions. For this reason, it receives the results from the POST, the launch signal and the values of acceleration and travel distance in order to send the arm command to the ESAD.

8) SafeGuard: It is the software component that acts as the safety executive component containing the rules and measures when the safety

requirements are violated. In our example the only safety measure is to keep the ESAD unarmed when one or more safety requirements is violated.

9) TimeGuard: It is the Watchdog that counts the time sequence in order to determine that the timing of the event sequence is proper based on the safety requirements (POST, Launch, First Motion Detection, and Safe Separation from the warship).

10) Exception Handling: It is the software component in the Fail-Safe processing Channel that sends the notArm command to the ESAD to disable the warhead in the cases that a failure has occurred in the Actuation channel and at least one safety policy has been violated.

In the following sequence diagrams, Figures 16 and 17, we demonstrate two potential scenarios of events that the proposed architecture should handle. The first does not contain any failure, but in the second scenario, the first motion detection does not pass the criteria (potential restrained firing) and the ESAD receives the notArm command to abort SAFE.

1. Use Case 1: Valid Launching

The missile is activated by the WCS via the powerOn electrical signal, and the WCS sends the command for launching (makeLaunch). Due to these signals, the powerOn() and makeLaunch() events are received by the component postSMK and the component launchSMK, respectively, to make the initial checks and to create the irreversible environment for launch. Upon receiving the powerOn signal from the WCS, the postSMK makes the initial checks prior the launch. When the process completes, the postSMK sends the result to timeGuard and armingControl via the endPost() method of the components. Simultaneously the launchSMK received the Launch Indicate signal via the makeLaunch() method, making the proper verifications and through the doLaunch() method sends the result to the armingControl component and to the timeGuard component in order to have the proper event sequencing and timing checks. The accelerometer yields a sequence of acceleration values and sends them to the accelSMK component to derive the missile's state of motion from the acceleration readings for the First Motion Detection and the Safe Separation Detection checks that they must realized under specific values and timeframe. Simultaneously the timeGuard receives the

messages in specific time and checks the receiving time against the timing constraints of the safety requirements. Because no safety requirements were violated, the armingControl received all the results from the above sequence and ordered the ESAD to be armed, as shown in Figure 16.

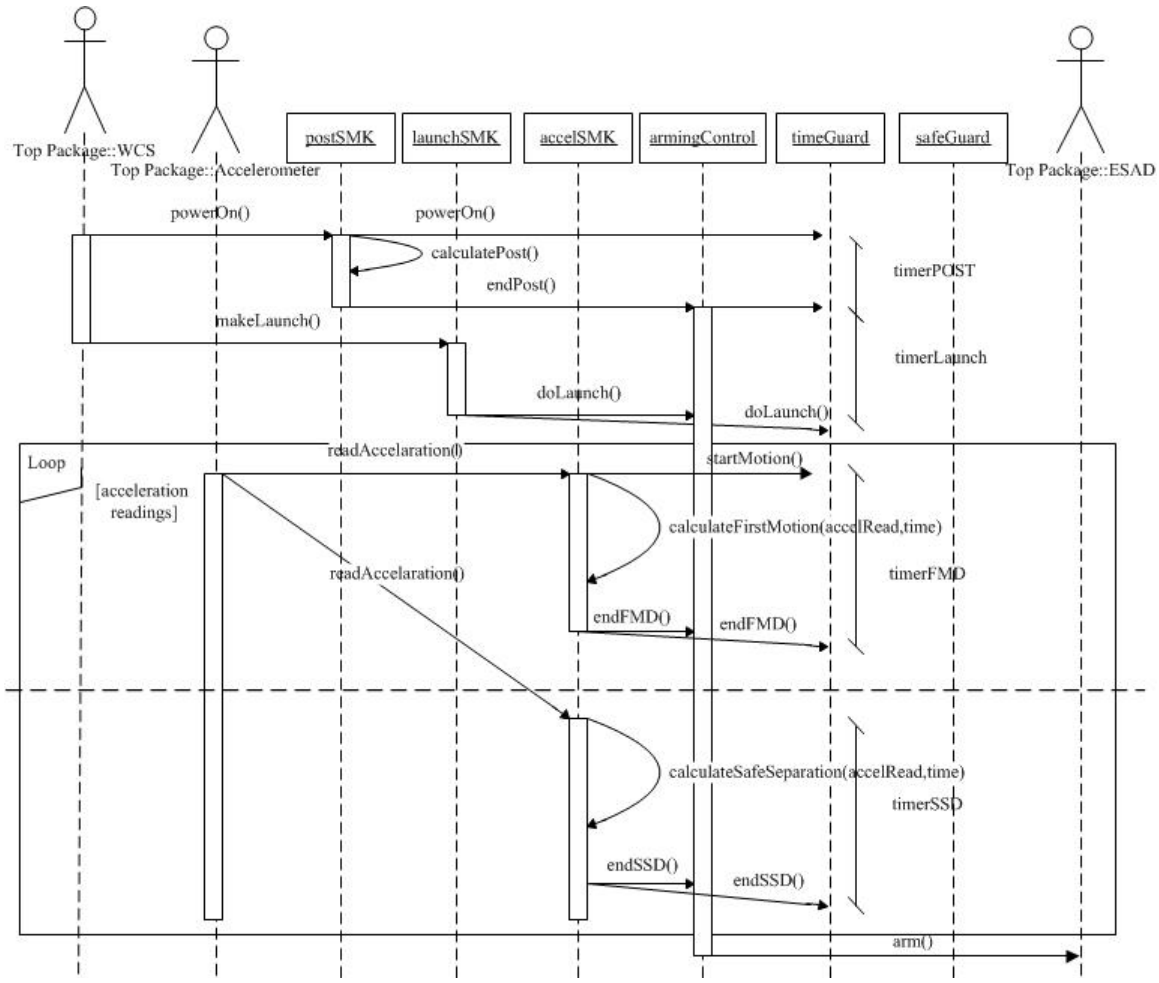


Figure 16. Sequence diagram for arming the ESAD.

2. Use Case 2: Restrained Firing

WCS activates the missile via the powerOn signal and then sends the command for launching (makeLaunch) moments later. The first steps are the same as in Use Case 1, but an accelerometer failure yields invalid values of acceleration. Thus, the accelSMK component cannot provide the valid values under the specific time constraints for the

First Motion Detection. This condition violated the third safety policy concerning the acceleration values in relation with the time. The TimeGuard detects this timing violation and sends the abort signal to the SafeGuard. This causes the SafeGuard to send the notARM signal to the ESAD and an abort signal to the Actuation Channel, as shown in Figure 17.

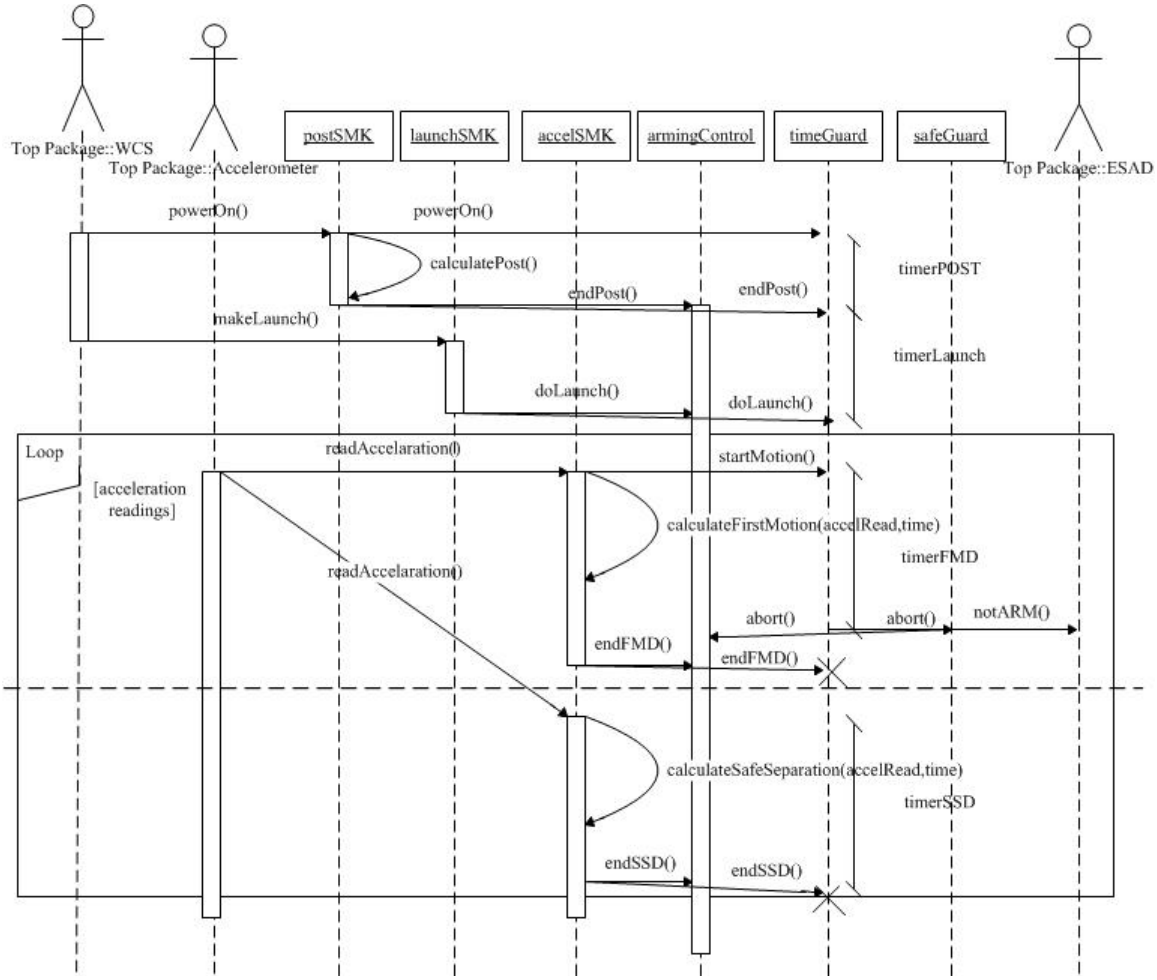


Figure 17. Sequence diagram for ESAD to remain in safe state.

E. SIMULATION

As discussed in Chapter I, one of the objectives of this thesis is to verify the correctness for the architectural design in meeting the safety requirements. To do that, we need to create an executable model for the proposed architecture. We developed a simple

time-step simulation for the missile arming control using the C++ programming language. The simulation is built in Windows 7 Enterprise edition (64-bit) using the Microsoft Visual Studio 2010, an integrated development environment (IDE) from Microsoft. The complete code is provided in Appendix A. We exercised the simulation with 21 test scenarios, which are shown in Appendix B.

The design of the simulation, as shown in Figure 19, can be divided in two parts. The first part deals with an abstract data structure, a queue that manages the exchange of messages between the objects. The second deals with the way that we implement the different objects to communicate with each other. We use the singleton design pattern [19] to create the missile's components because we want to create a single common object for each of the missile's components. In addition, we create two singleton utility classes. The first one is the IdGenerator, and the second is the RandGen. The IdGenerator creates unique identification number (ID) for all the messages in the simulation and RandGen generates random numbers that are used to vary the time increments.

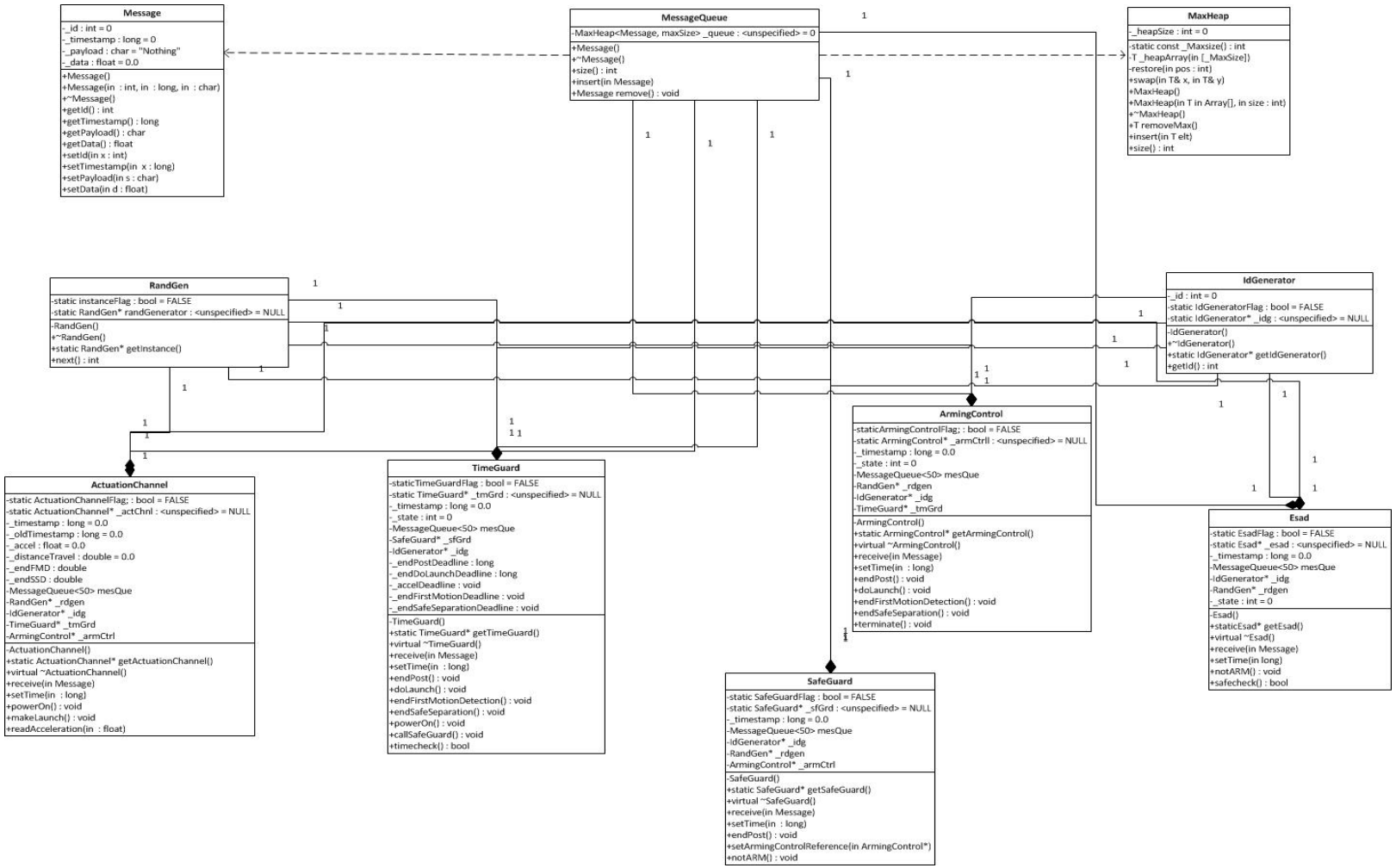


Figure 18. Class diagram.

1. Supporting Classes

For the first part, we re-used the code, with permission, created by NPS student Nahum Camacho Zamora for his class CS3021 Data Structures and Intermediate Programming, with modification to suit our needs. This part includes the header files Message.h and MessageQueue.h with their implementations (Message.cpp and MessageQueue.cpp).

a. Message

The class, Message, is responsible for creating message objects in the proper format in specific time. Each message object has four private attributes: a unique id, a timestamp, a payload and a data. The id contains an integer with value generated uniquely from the IdGenerator. The timestamp remembers the message's creation time as a long integer, whose value is equivalent to a corresponding value of the C++ time_t class. The payload contains a string taken from the set {"powerOn," "endPost," "invalidPost," "doLaunch," "startMotion," "endFirstMotionDetection," "endSafeSeparation," "abort," "arm," "notARM," "readAcceleration"}. The data field is only valid if the message's payload equals "accel," in which case the data field contains a float equal to the acceleration reading in "g." When the Accelerometer sends values to the Actuation Channel, it uses the method readAcceleration(float) that has as an argument on these values in float type.

Due to the fact that each message is unique and carries information that has to be handled easily and efficiently, we add a message's id using the IdGenerator class to create this unique id. Having this tool, we can manage them in the message queue. Each time that a message is created it has a unique id that characterizes it and can use it from the queue.

b. MessageQueue

The MessageQueue is a C++ template class that has methods to add and remove messages from the queue based on the earliest-timestamp-first policy. Messages with equal timestamp values are removed in the first-in-first-out manner. The

MessageQueue realizes its least-timestamp-first behavior using the MaxHeap template class, which implements a linked list data structure.

2. Main Function and Simulated Missile's Components

The second part of the design consists of the following classes, the: ActuationChannel, ArmingControl, TimeGuard, SafeGuard, Esad and Logger. The first five classes correspond to the components of the proposed architecture and the last component, Logger, is a utility component for log file generation.

a. Main Function/Simulation Environment

The simulation environment is our main class, in which we initialize our timer and create the instances of the singleton objects. It then increments the timer and creates messages instances (with payloads "powerOn," "makeLaunch" and "readAcceleration") according to the different test scenarios shown in Appendix B. It sends all the messages to the ActuationChannel object and also sends the "powerOn" message to the TimeGuard object.

b. Actuation Channel

We encapsulate the components in the Actuation Channel into two software classes, the ActuationChannel and the ArmingControl. The singleton ActuationChannel class simulates the functions of postSMK, the launchSMK and the accelSMK components, receiving message from the environment and sending messages to the ArmingControl class and the TimeGuard class. The singleton ActuationChannel object receives the messages "powerOn," "makeLaunch" and "readAcceleration" via the receive() method, which puts the message into its local message queue. In addition, it receives time updates via the setTime(t) method, which sets into local clock to time t and then checks to see whether the queue is empty or the queue contains messages with timestamp less than or equal to t. It will remove the messages with timestamp less or equal to t in an earliest-timestamp-first manner, and call the corresponding event handlers to handle the events.

The handler for the “powerOn” event will send a message with a random future timestamp and the payloads “endPost” or “invalidPost” to demonstrate the two possible results from the power-On self-test simulating the completion of the power-on self-test procedure. The message “endPost” represents the situation when the self-test is passed, and it is sent to the ArmingControl and to the TimeGuard, respectively. The message “invalidPost” represents the situation when the self-test is failed, and it is sent to the TimeGuard to activate the SafeGuard to keep the ESAD in the unarmed state. The handler for the message “makeLaunch” event will send a message with a random future timestamp and a payload, “doLaunch” to the ArmingControl and TimeGuard, simulating the completion of the generation of the Launch Indicate signal, which will be used by the ArmingControl object and the TimeGuard objects to mark the time and the progress of the arming sequence processing.

When ActuationChannel processes the readAcceleration message for the first time, it will remember its data value and timestamp in its private attributes, starts two timers (a 4-second timer for the endFirstMotionDetection and a 6-second timer for endSafeSeparation) and sends a “startMotion” message to the TimeGuard. For subsequent readAcceleration messages, it will check to see if there are two consecutive acceleration values above 6 g’s, compute the estimated distance travel so far and check to see if it exceeds 20 meters. If it determines that there are two consecutive acceleration values above 6 g’s before the 4-second timer expires, it will send the “endFirstMotionDetection” message to the ArmingControl and to the TimeGuard. If it determines that the missile has travelled a distance of at least 20 meters before the 6-second timer expires, it will send the message “endSafeSeparation” to the ArmingControl and to the TimeGuard.

c. Arming Control

The Arming Control class is responsible for sending the arm command to the ESAD when it receives all the required messages (“endPost,” “doLaunch,” “endFirstMotionDetection,” “endSafeSeparation”) in a timely manner.

d. TimeGuard

The TimeGuard class is responsible for keeping track of the timing constraints of the safety requirements. It will notify the SafeGuard object when it detects any timing error or when it detects the event with payload “invalidPost.” It implements the state machine shown in Figure 19, which sets different deadlines to enforce the timing constraints defined by the safety requirements when it enters each state. Like the ActuationChannel class, the singleton TimeGuard object receives the messages via the receive() method, which puts the message into its local message queue. In addition, it receives time updates via the setTime(t) method, which sets the local clock to time t, then it will process the messages with timestamp less than or equal to t in it message queue. For the messages relevant to its current state, it will first check to see if the message arrives before its deadline. If yes, it will perform the necessary state transition as defined in Figure 19. If the message arrives after its deadline, the TimeGuard object will send an “abort” message to the SafeGuard.

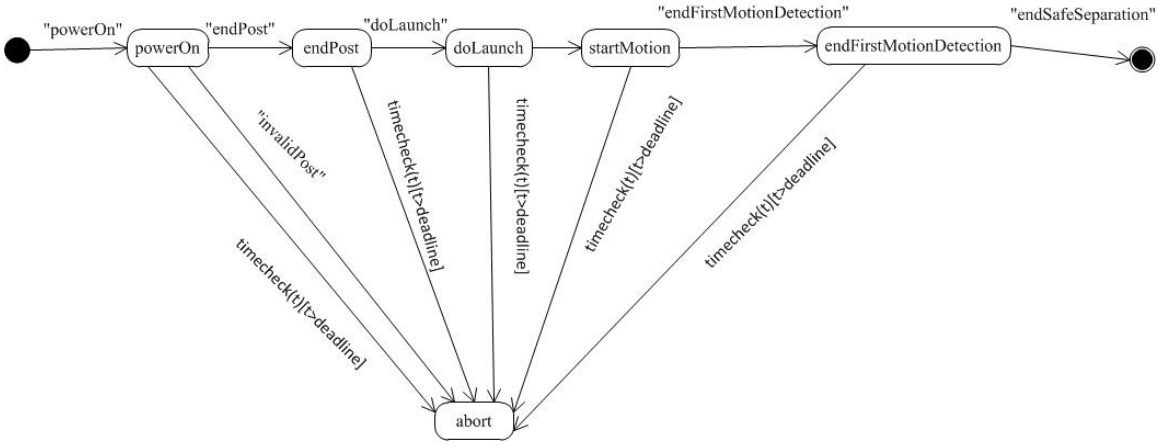


Figure 19. Statechart diagram for the TimeGuard class.

e. SafeGuard

The SafeGuard class is responsible for handling the exception when it receives the “abort” message from the TimeGuard. When this happens, the SafeGuard object will send the “notArm” message to the Esad class, and sends the “abort” message to the ArmingControl object to terminate the arming procedure.

f. Esad

The Esad class is responsible for setting the warhead state to armed and unarmed, based on the message received from ArmingControl or Safeguard.

g. Logger

The last class, Logger, keeps track of all the messages (their payloads and their times) received by the TimeGuard object and the readAcceleration messages received by the ActuationChannel object and writes them to a log file. The log file contains a trace of the event of interests and will be used for formal V&V of our architecture. The logger writes the event traces into two formats:

<string><space><@><space><receiving_time_to_TimeGuard>

<float><space><g><space><@><space><receiving_time_to_TimeGuard>.

The format is used to log all messages without the data value and the second one is used to log the readAcceleration event.

3. Test Scenarios for the Simulation

For the simulation, we need to discuss which use cases we need to simulate in the software. The set of use cases can vary depending on the designer's viewpoint about the faults that can occur during the life cycle of the missile. To manage the whole effort properly and efficiently, we started from the beginning of the arming sequence and followed the software's design to locate the events that could cause a fault to occur and eventually lead to a hazard. We wrote down all the use cases using tabular representation and binary logic in Appendix B (Tables 1 to 7), which show briefly which cases are of interest for this simulation. The tabular representation is used for our efficient management of the use cases. We use the binary logic, True/False or 1/0 respectively, to present whether the value of an event and its relative occurrence time meet the safety requirements (1 when they meet the requirements and 0 when they do not). In addition, we developed some associative use cases (Table 8), which are possible accelerometer values that calculate the First Motion Detection and the Safe Separation producing different environmental conditions that the software could face. At the end, the whole

effort resulted in 21 use cases that are independent of each other. Due to the way that we partitioned the event space, we have significantly reduced the number of potential test scenarios from approximately 1400 to 21.

Beginning from the first message that the missile should receive, we have the set of use cases about the powerOn message both to the Actuation Channel and to the TimeGuard components shown in Appendix B, Table 1. The combination of these two messages with their timing gives rise to 16 cases, but only two of them can be simulated because the other 14 do not apply to our design. The two use cases contain the sending of both messages on the Actuation Channel and TimeGuard, and the difference between them is whether the two messages are received at the same time by the components or there is a time delay for the message received by the TimeGuard. For the other 14 cases, the first eight cases do not include the message powerOn to the Actuation Channel which is not realistic for our case study. The final four cases contain a time delay to the Actuation Channel that also does not meet our design's rationale; the TimeGuard is the Watchdog component and has to start after the Actuation Channel in order to be meaningful. Hence, there are only two valid test cases in Appendix B, Table 1.

For the case of the endPost message, we have four use cases shown in Appendix B, Table 2, based on the validity of power on self-test results and whether the event arrived at the TimeGuard at the proper time. Moreover, we can combine the two cases with invalid self-test results. Thus, we have total three use cases to simulate. For the case of the launch command, which creates the irreversible condition for the missile, we have four use cases shown in Appendix B, Table 3, based on whether the makeLaunch message is generated and whether the event arrived at the TimeGuard at the proper time. We can eliminate the case that there is no makeLaunch, and there is a time delay for it to arrive at the TimeGuard, this case can never happen. Thus, we have a total of three use cases to simulate. Furthermore, there is the internal message "startMotion" that the launchSMK subcomponent sends to the TimeGuard to indicate the first detection of missile motion. Following similar logic as for the makeLaunch use cases, we keep two of the four use cases that can happen, as shown in Appendix B, Table 4.

For the acceleration values and how they are implemented in the simulation, we create three groups with different rationales. The first group is based on whether the readAcceleration message is processed properly and whether in proper time order, as shown in Appendix B, Table 5. The second group concerns the case whether the internal startMotion message is sent properly and whether there is a time delay to reach the TimeGuard, as shown in Appendix B, Table 6. And finally, the third group, as shown in Appendix B, Tables 7 and 8, deals with the acceleration values and their receiving times at the TimeGuard. For the third group, there are many different combinations with timing issues and with the acceleration values not meeting the minimum limits from the safety requirements. For the third safety requirement, there is the obligation that two consecutive values have to be equal to or more than 6 g's and for the fourth safety requirement the calculation of the travel distance has to be equal to or more than 20 meters, and this is the result from the double integration of the acceleration in time.

From the first group, which concerns with the proper receiving of the readAcceleration messages and their timing and not their contained values, we have four use cases. We only keep the case with no time delay because the other cases are unrealistic. For the second group, which concerns with the internal message "startMotion," we exclude the case in which there is a time delay but no "startMotion" signal, because it is not realistic. We include the use case in which all the acceleration values equal to zero and the two other use cases with non-zero acceleration values and with/without time delay to the receive the "startMotion" message at the TimeGuard. Thus, the total number of use cases is three for the "startMotion" message.

At the end, there are the most complicated scenarios because we have to deal with a variety of use cases. To consider all the possible cases, we write down two tables. Table 7, which concerns the use cases with/without the firstMotionDection message and their timing, as well as with/without the endSafeSeparation message and their timings and Table 8, which concerns the use cases with the combination of acceleration values and their calculated results. For this reason we simplify our rationale down in three subsets. The first subset answers the cases of the First Motion Detection criterion. For this criterion there are four cases, fulfilling it or not in relation to its receiving time from the

TimeGuard. The second subset answers the SafeSeparation cases following the same rationale. And the third subset answers the combination of the above two subsets and creates 16 use cases. Some of them, as is shown in both Tables 7 and 8 in Appendix B, can be combined minimizing the total number of the use cases to 12. We exclude the cases in which there are time delays but without messages, either for the one or for the two messages, endFirstMotionDetection and endSafeSeparation.

Consequently, we have the special case that all criteria, values and time, are fulfilled and the 20 use cases in which at least one of the criteria is not met and resulted in a fault. Thus, the total number of the simulation use cases is equal to 21, and for this reason we create 21 log files to verify and validate our proposed design in Chapter V.

THIS PAGE INTENTIONALLY LEFT BLANK

V. FORMAL V&V OF SOFTWARE SAFETY REQUIREMENTS AND ARCHITECTURE

A. INTRODUCTION

In Chapter III, we demonstrated the steps for analyzing our system according to the safety standards, and the result was the documentation of the safety requirements. Then we combined them with the hardware, and at the end we decided which hardware functions our software should check. From the results of our system hazard analysis and software system hazard analysis, we concluded that the software for controlling the ESAD arming device, which is a causal factor to premature detonation, has a high software hazard risk index, and hence will require thorough analysis of system-level requirements, software safety design and implementation source code to ensure adequate control of the causal factors as well as in-depth testing to ensure that the control measures are implemented correctly. Our requirements analysis has resulted in four software safety requirements to monitor the proper sequencing of the events from the sensors and the control software to detect and prevent any error that may cause unsafe arming of the warhead. In Chapter IV, we presented an architectural design of the control software using the safety-kernel safety pattern and created an executable model for design. We performed a detailed analysis of the different failure scenarios and came up with 21 test cases. We tested the executable architectural model in a simulation environment and produced 21 log files that capture the behavior of the software under the different situations that could lead to a premature arming of the warhead.

Because the control software is safety-critical we must thoroughly verify and validate the proposed design. In this chapter, we present a light-weight formal method for the validation and verification of the software safety requirements and the target software. The process is demonstrated in Figure 20 and is described thoroughly in [20]. We explain how this process is implemented in our case study.

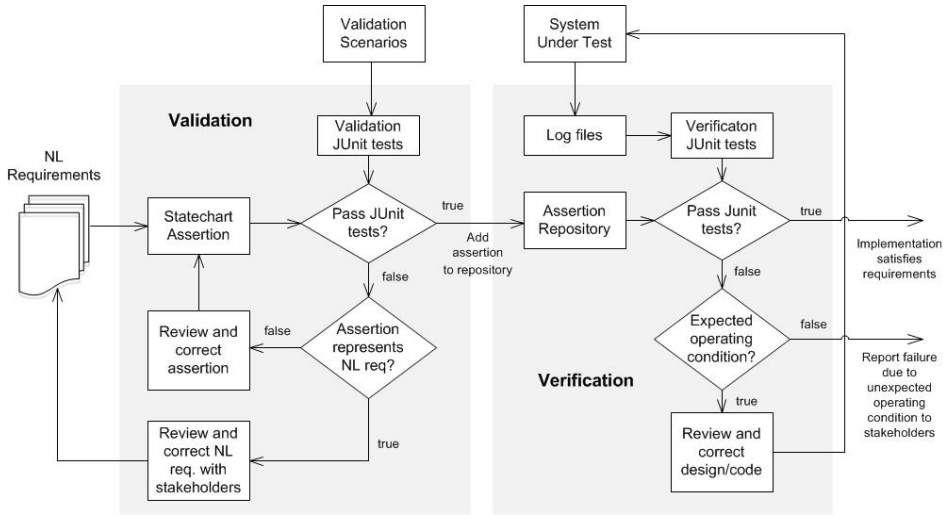


Figure 20. V&V procedure for SMK ESAD. From [20].

First, we have to translate the natural language software safety requirements into a precise specification without losing the original meaning of the requirements and expectations of the stakeholders. We accomplished this through the application of a formal method to create a mathematical model of the requirements that can be processed by a machine. Since we use state machines in our design, we choose to describe the software safety requirements as statechart assertions. The statechart assertions are extension of statecharts, which are Unified Modeling Language (UML) based diagrams to specify the behavior of reactive systems [22]. We create the statechart assertions with the StateRover tool from TimeRover, Inc. [21], which is an Eclipse integrated development environment plugin. In addition, we use the Eclipse Juno version 4.1 for the creation of the Java code and JUnit test cases.

B. SOFTWARE SAFETY REQUIREMENTS SPECIFICATION AND VALIDATION

As we described in Chapter III, the process of hazard identification makes clear the potential erroneous situations in which an error in the control software can result in a mishap. To ensure that the resultant software safety requirements adequately address the hazards posed in operating the target system, we need to validate the requirements against the potential erroneous situations identified through hazard analysis. In other words we want to make sure that the software safety requirements correctly specify what the

software must do and what it must not do. Using the StateRover tool, we are able to encode the natural language requirements into a set of executable statechart assertions, shown in Figures 21 to 24, whose behaviors can be demonstrated to the stakeholders via JUnit testing. The statechart assertions are written from an external observer's point of view. The external observer needs to observe the events: *powerOn* from the WCS, *readAcceleration* from the Accelerometer, *endPost, invalidPost, doLaunch, endFirstMotionDetection* and *endSafeSeparation* from the ActuationChannel, and *arm* from the ArmingControl. We created four statechart assertions, one for each of the software safety requirements listed in Chapter III, Section D. The initial state in each statechart assertion is the OFF state. When the proper event (powerOn or startMotion) is observed, the statechart assertions will transit to the next states as indicated in the diagrams. Because our requirements have to assure the safety of the arming sequence, we are only interested in making sure that the control software enters the SAFE_MODE state when an error is detected. Once it enters the SAFE_MODE state, the control software must never issue an arm event. Hence, if an arm event (from the armingControl) is observed while the control software is in the SAFE_MODE state, the statechart assertion will transit to an ERROR state, declaring that it has observed an error in the control software that violates the corresponding software safety requirements and may eventually lead the system to hazard. For each statechart assertion, we implement a local timer in seconds that is responsible to keep the time constraints of each Software Safety Requirement (SSR).

1. SSR 1

The CS computer receives an activation signal from the WCS to power on the missile. It executes the POST, which must be completed within two seconds after the receipt of the activation signal and should be POST_OK in order to continue the arming procedure. If POST is overtime (>2 sec) or invalid, then it transits to the Fail-Safe state. As the statechart assertion code always updates the timer and checks for timeout before processing any incoming events, we define a local timer called the twoSecTimer (shown in in the yellow box in the Figure 21). Because the StateRover tool prioritizes the timer ahead of any event, this leads to the situation that the timer expires prior to the processing

of the receiving event. For this reason we expand the time limit by one second. Doing this we capture the correct use cases when any event is received at precisely the time that the timer expires. We set the value of the twoSecTimer to 3 seconds so that it will handle the situation when the self-test result is valid and the endPost message is observed 2.0 seconds after the powerOn event is observed. The twoSecTimer processes this event so that the statechart assertion will transit from the POWER_ON state to the POST_OK state instead of the SAFE_MODE state, which would have occurred had we set the value of the timer to 2 seconds instead

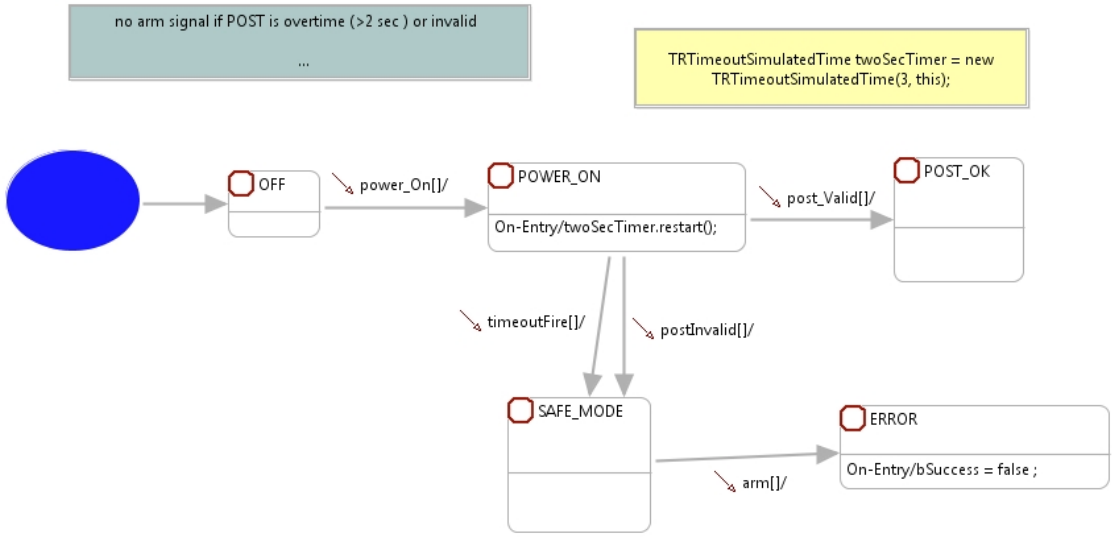


Figure 21. Statechart assertion for software safety requirement 1.

2. SSR 2

The Launch Indicate Signal represents the irreversible commitment to launch the SMK. The launching signal must be received within 4 seconds after powerOn.

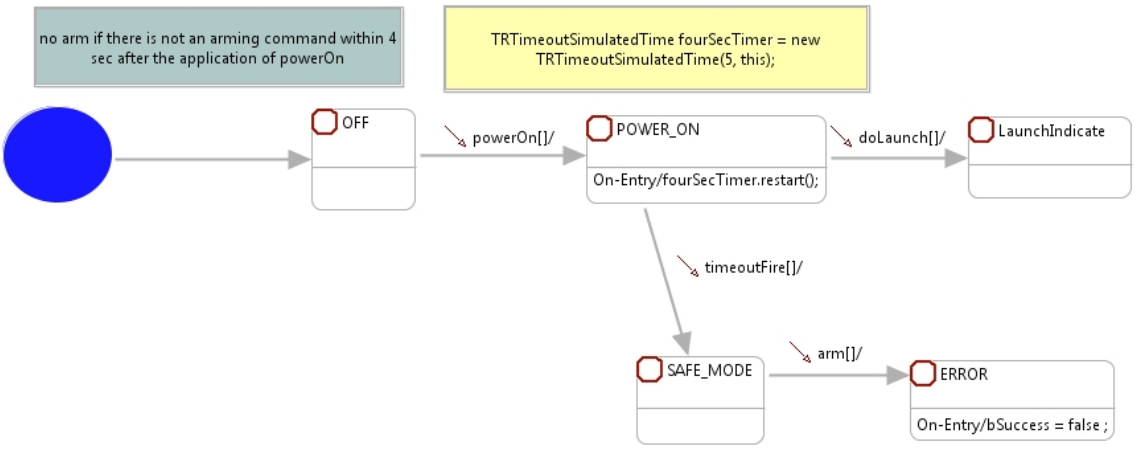


Figure 22. Statechart assertion for software safety requirement 2.

3. SSR 3

The ESAD can be armed only when the missile has received the launching signal and starts to move away from the ship. In order to determine that the missile is on the move, two consecutive accelerometer readings over 6 g must be detected within the 4-second window from the time that the accelerometer sends accelerometer values.

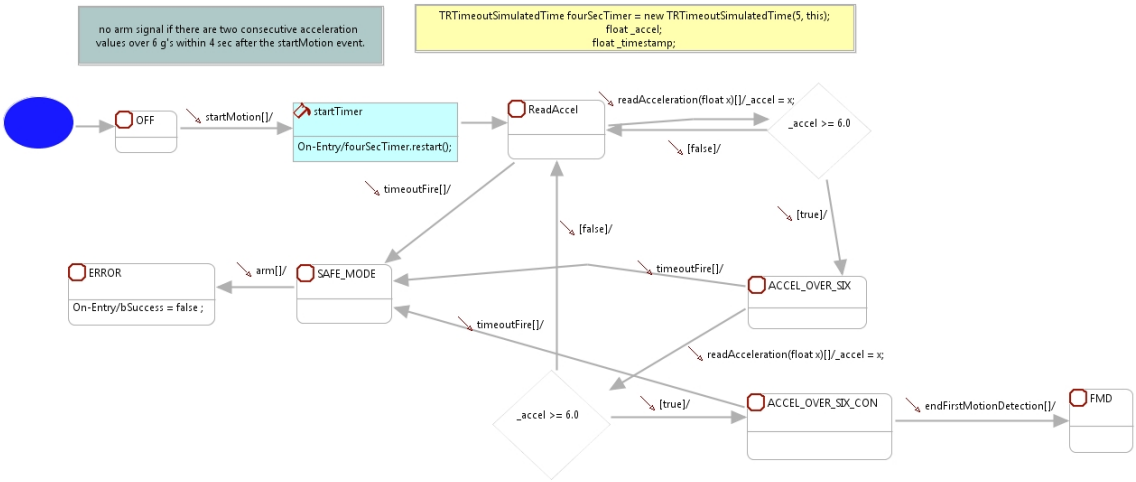


Figure 23. Statechart assertion for software safety requirement 3.

4. SSR 4

The ESAD can be armed only when the missile has reached the minimum vertical distance above the warship. If the travel distance is less than 20 meters within the 6second window after the activation of the accelerometer, then the ESAD has to remain in safe mode.

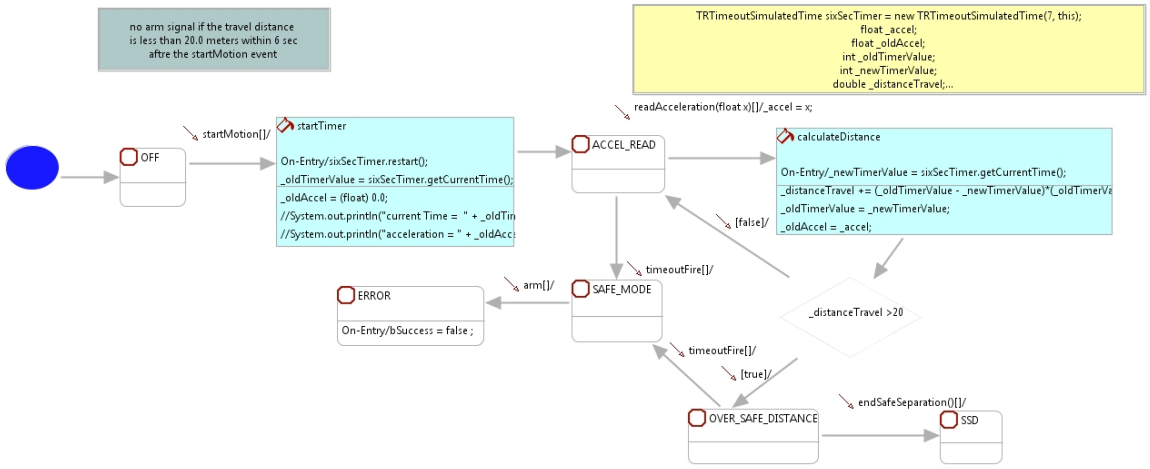


Figure 24. Statechart assertion for software safety requirement 4.

Continuing our validation process we create test cases that check whether or not our statechart assertions are able to correctly detect the various erroneous situations. For this reason we develop different test scenarios to challenge the requirements to determine whether they detect the errors as intended. This step is critical for the safety requirements analyst because it not only acts as a checker for the correct encoding of the natural language requirements but also clarifies whether the analyst correctly understands the original intent underlying the requirements.

For our case study this step was accomplished using the StateRover tool. The tool generates one Java class for each statechart assertion and allows the analyst to test the generated code using the JUnit tool. Figure 25 shows the timing diagrams and the Java code snippet for the three JUnit test cases for the statechart assertion shown in Figure 21. The first test case, which is the first one in Figure 25, represents the happy scenario in which everything is within the time limits and the result from the self-test is passed. The

second test case represents the scenario that the self-test fails, thus any arm() event should result in the ERROR state (indicated by the Java statement assureFalse(assertion.isSuccess());). The third test case represents the scenario that the self-test is passed but it exceeds the 2-second time constraint. Hence, the timer expires and any arm() event should result in the ERROR state. (Readers can refer to Appendix C for the other JUnit test cases.).

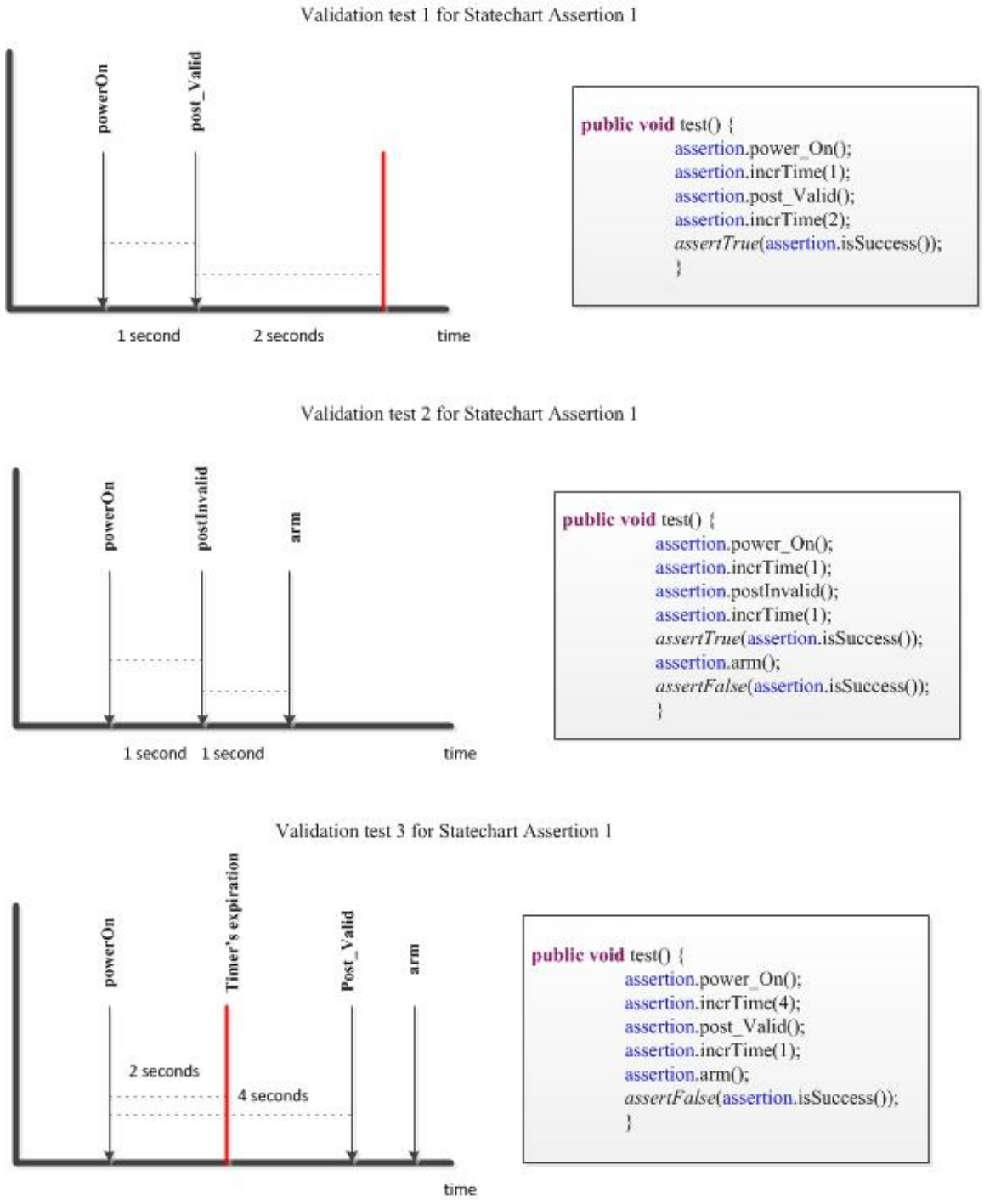


Figure 25. JUnit validation test cases for software safety requirement 2.

Following this procedure, the design team formalizes the system's requirements and uses the JUnit tool to exercise these requirements with different scenarios. This leads to modifications to the written requirements, taking the requirements to a more sufficient

level. In our case study the same people act as the designers and testers but in larger projects this could be done by two independent teams, one responsible for the design and the formalization of the requirements and the second to validate them like an IV&V team. This procedure ascertains the formality for our safety requirements' hidden rationale.

The statechart assertions, once validated, can help in the automated testing of the target software via the offline, logfile-based runtime verification process shown in the right half of Figure 20.

C. ARCHITECTURE VERIFICATION

Having implemented our architectural design in C++, our next step is to verify its correctness using logfile-based runtime verification. By doing this we are able to satisfy the last of the thesis's main objectives, which is to verify that the architectural design correctly corresponds to the safety requirements.

We use the 21 test scenarios described in Chapter IV to generate 21 log files. We followed the arming sequence from its beginning to its end, and we injected faults that could lead to failures and finally to hazards according to the hazard analysis so that we could test our model as closely as possible to the environment in which the system will be used. The 21 log files captured the events the control software received from the environment and the responses the control software generated under the scenarios. We implemented only one fault that happens each time and not a combination of them.

The next step is the formatting the log files so that the StateRover can use them to generate JUnit test code to exercise the statecharts assertions. The conversion of the log files is based on a Python program that converts the .txt file to a XML file, a code that Bonine in [21] developed for a similar purpose. Using the StateRover XML log file import tool, we are able to generate a Java JUnit test case for each XML log file from our simulation run. Since the names of the events in the log files and those in the statechart assertions may not always be identical, StateRover provides the StateRover Namespace Mapper tool for users to create a Java namespace mapper object that links the events from the log files and the associative transitions from the statechart assertions, and the

whole procedure is described explicitly in [20]. Having completed these steps, we are ready to conduct the runtime verification to verify our architecture.

Let us illustrate the process just described with the two scenarios shown in Figures 26 and 29. In the first scenario, the missile is activated normally and there is no error in the whole procedure. Running this scenario, the simulation outputs a log file, which is called logfile_no_errors, shown on the right side of Figure 26. For better understanding by the reader, we illustrate the events in the log file with the timing diagram shown on the left side of Figure 26.

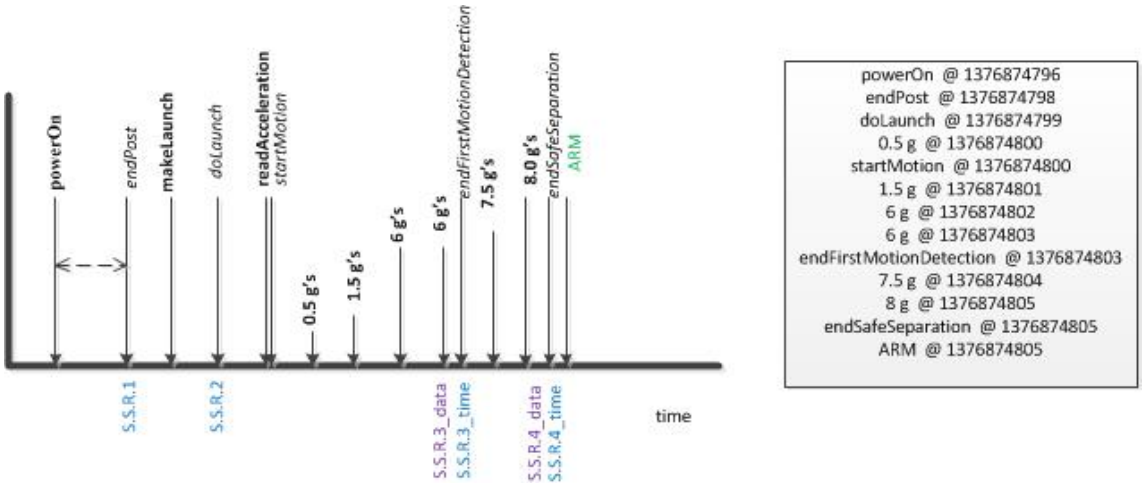


Figure 26. Log file and timing diagram for no_errors test case in simulation.

Each arrow from the timing diagram represents either an external event that stimulates our software from its environment or an internal event generated by the software in response to the external event. The external events in the timing diagram are in bold font and the internal events are in italic font. We use this convention in all the timing diagrams in Appendix B to help the reader understand the different test scenarios.

We input the log file to the StateRover tool, and we link the events from the log file and the transition names of the statechart assertions using the StateRover Namespace Mapper tool shown in Figure 27. For each log file, we should carefully map the events with the transitions to capture the rationale of each test case. Then, the Namespace Mapper initiates each event with the related transition and during the execution it decides

whether or not the log file from the simulation violates a requirement and which. In Figures 27 through 28, we demonstrate the verification procedure of the two sequence diagrams from Chapter IV, Section D, using the Namespace Mapper and the statechart Animation option from the StateRover.

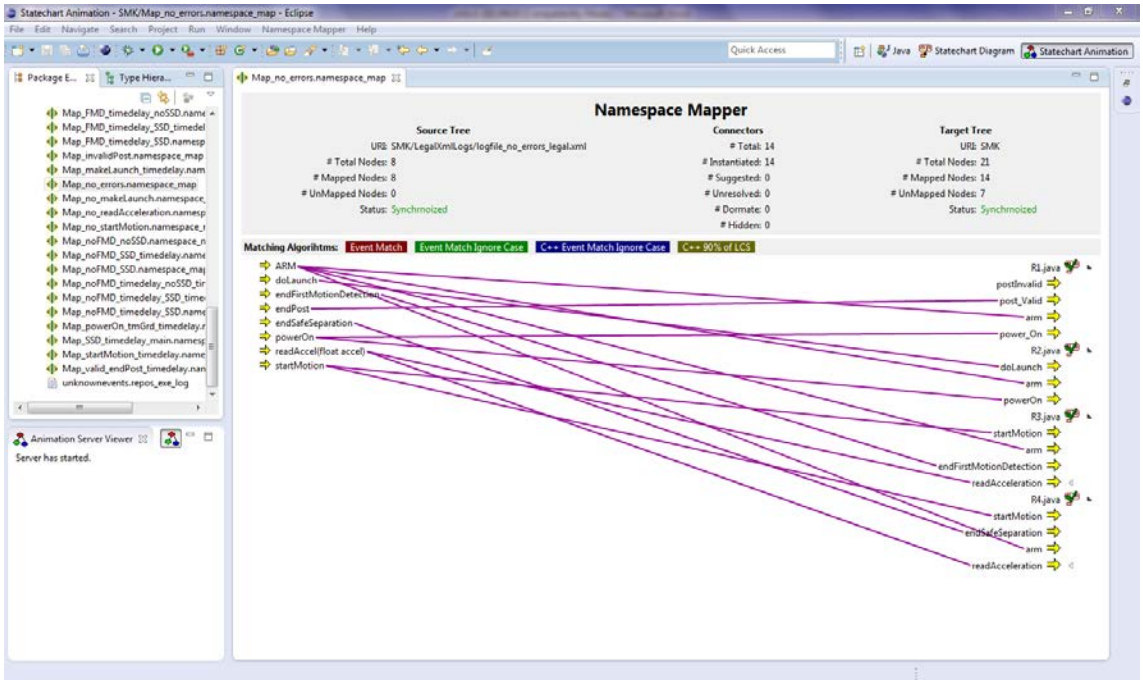


Figure 27. Namespace mapping between the simulation events and statechart assertion transitions.

After the namespace mapping, we execute the scenario and the StateRover displays the output screen (shown in Figure 28) showing that our design model works correctly as expected, meeting all four safety requirements since there is no message under the field statechart assertion failure. If the design model had violated one or more of the safety requirements, then the StateRover would indicate which requirements were violated.

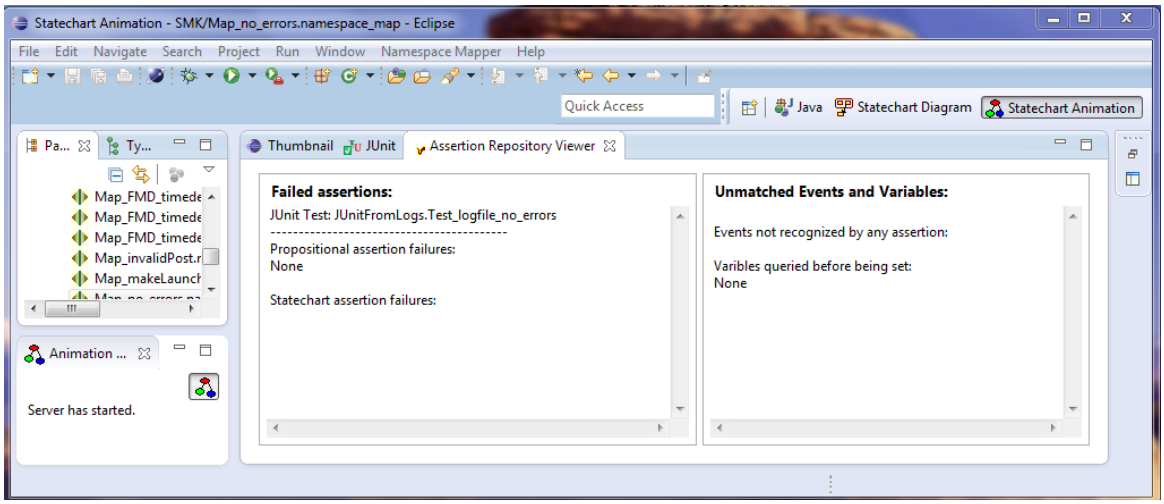


Figure 28. Verification test using the StateRover tool.

Following the same process, we ran the second scenario shown in Chapter IV and generated the log file called logfile_no_FMD_noSSD. This scenario describes a possible situation in which the missile's accelerometer does not produce proper values and SSR 3 and 4 are not fulfilled. The use case captures many different failures that have the same output. One of them is the failure of the accelerometer as hardware, in which it provides erroneous values. Another could be a restrained firing. This severe situation happens when the holding latch that holds the missile's body from its container does not release the missile during its launching procedure. The above causal factors could lead to a premature arming of the warhead and a possible detonation. Trying to prevent this from happening, our software that controls the ESAD should prevent the warhead's arming. In order to detect these situations the software has to have the proper design to deal with these events. For this reason we create an environment in which there are improper values from the accelerometer. The created log file is depicted in Figure 29 with its corresponding timing diagram. From the results of this simulation we observe that the model detects the fault and prevents a failure from occurring.

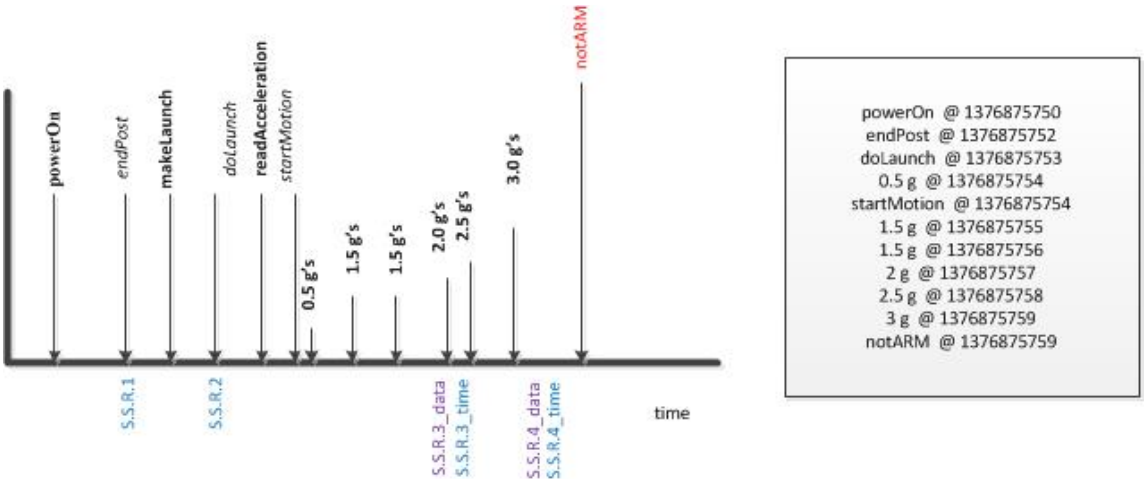


Figure 29. Log file and its relative time diagram for the no_FMD simulation case.

We repeat the procedure as we did for the logfile_no_errors and namespace mapping, with the test results shown in Figures 30 and 31.

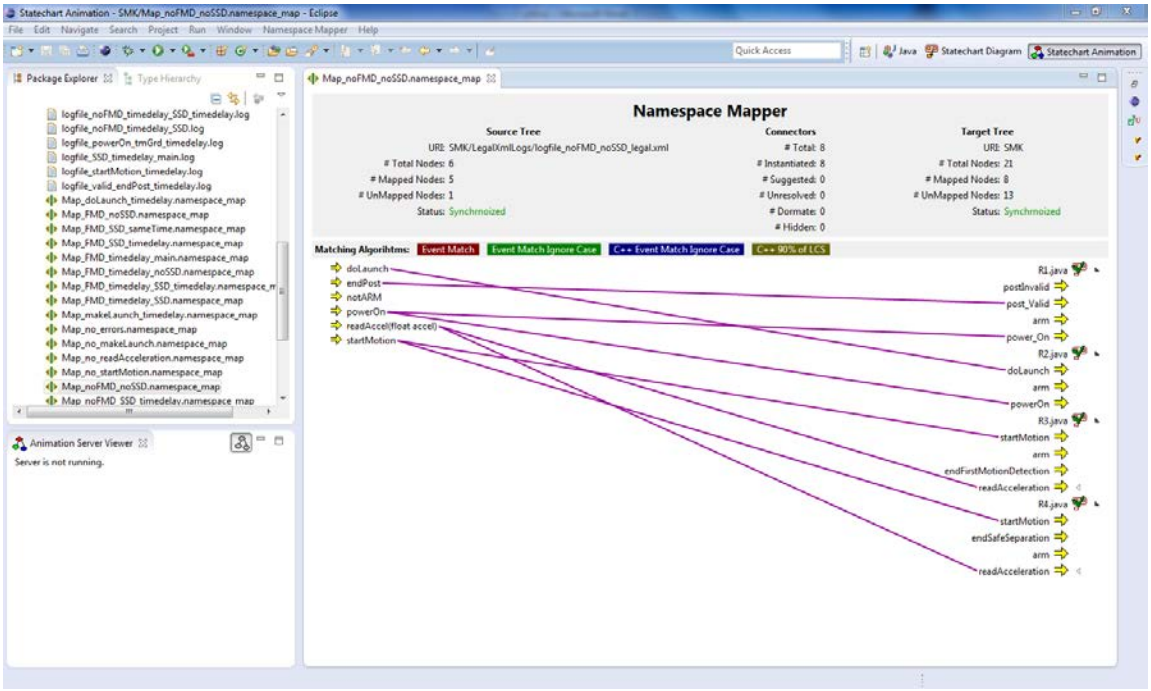


Figure 30. Namespace mapping for the second scenario.

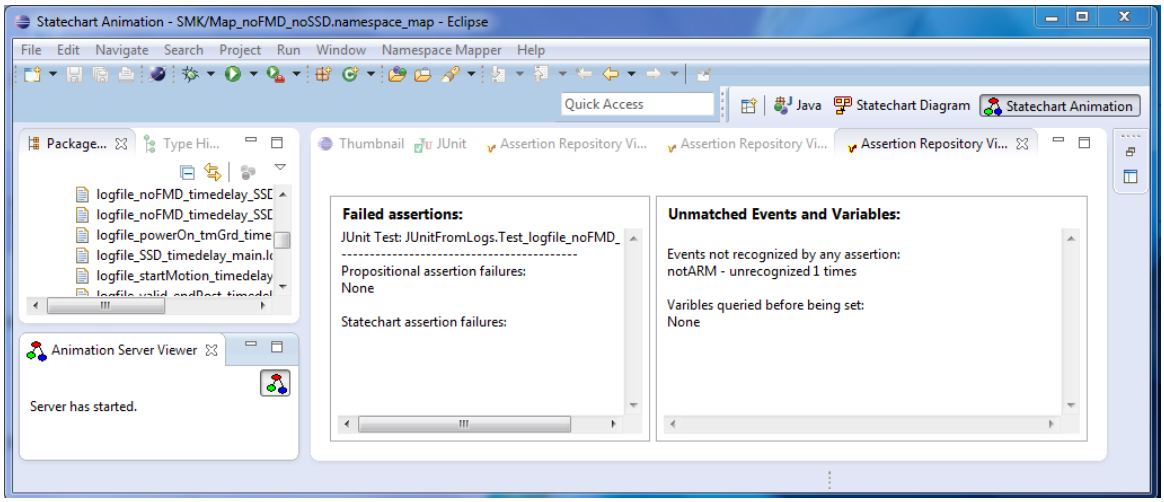


Figure 31. Test results for the second scenario.

The result from this verification test not only verifies that the warhead was not armed as we can see from the log file but that the software successfully detects the fault and prevents a failure from occurring. Hence, using the above procedure we can create as many test cases as we want to evaluate our design. The result from this process assists the design team to understand in depth what is built and provide them with feedback. Subsequently, they are able to describe clearly their viewpoint on the project and explain to the stakeholders the reasoning behind the design.

VI. CONCLUSION AND FUTURE WORK

A. SUMMARY

This study matches the different aspects of different domains such as systems engineering, hazard analysis, software requirements, and software architecture tasks in the development of the safety-critical systems. It implements the rules of safety engineering in the user's requirements, designs the product using all the above and finally verifies and validates the case study's software design. From the beginning of this study, we adopt the method that Kelly and Wu proposed in [7] to implement the system's nonfunctional attributes in the design process. We are interested in the safety of software intensive systems.

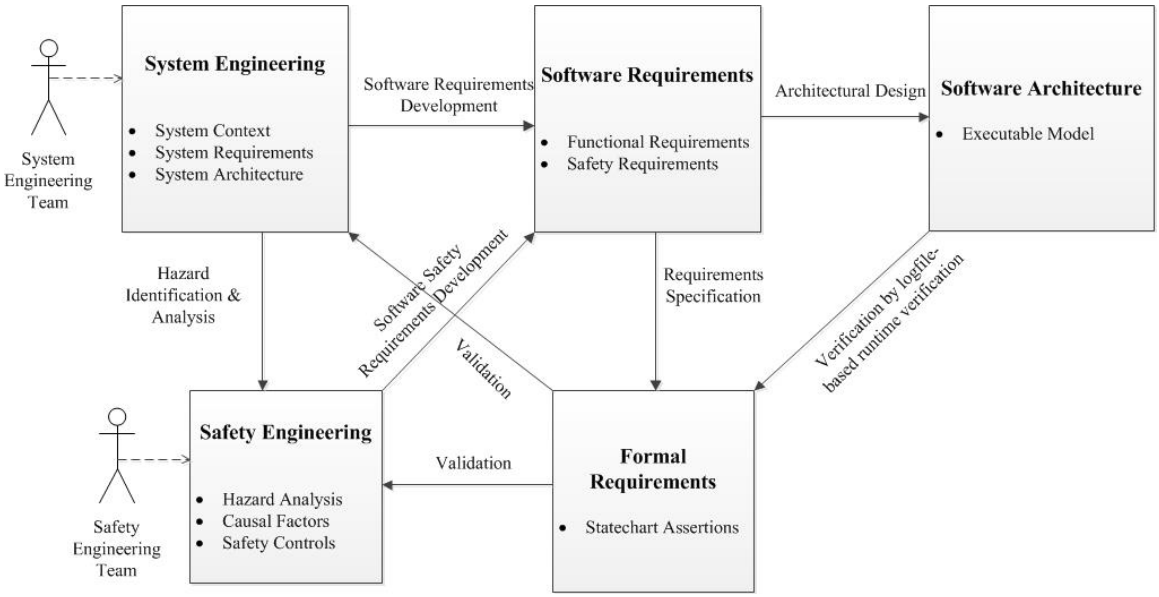


Figure 32. Formal V&V process for a safety-critical system.

In this study, we do not introduce a new domain rather we introduce a new framework that relates the different activities and products from systems engineering, safety engineering, system and software requirements, and software architecture explicitly (Figure 32). Until now the teams from safety engineering and systems engineering have created the conditions to build a system that can handle potential

hazards. The artifact from this cooperation is the documentation of the system's safety requirements in natural language. This documentation redefines the systems engineering needs using the safety measures from safety engineering. The need for clear and understandable documentation is fulfilled by formalizing the safety requirements as statechart assertions. The innovations in our study are the inclusion of formal V&V of the software safety requirements and the software architecture to improve the existing process. We formalize the natural language software safety requirements as statechart assertions and validate the software safety specifications using the JUnit testing tool against the various potential erroneous situations that have already been identified by the domains of the hazard and safety engineering and the systems engineering. Results of the validation JUnit tests are presented to both the systems engineering team and the safety engineering team for examination and feedback. Continuing from the formalization of safety requirements, we build an executable architecture using the software safety architectural patterns to realize the software safety requirements. We then exercise the executable architecture to test its safety behavior under various scenarios and capture the interactions between the software and its environment in terms of log files, which will be converted automatically as verification JUnit tests (with the help of the StateRover tool) to verify the correctness of the software architecture using logfile-based runtime verification. Any violation detected by the verification JUnit tests will be examined by the software development team to see if it is caused by errors in the architectural design or its C++ implementation or in the encoding of the statechart assertions or due to incorrect or inadequate safety measures, as described by the natural language safety requirements. Errors in the architectural design or its C++ implementation will result in software architecture and coding revisions. Errors in statechart assertion encoding will require assertion statechart development and validation reworks, and errors in incorrect and inadequate safety measures will trigger another round of systems engineering and safety engineering activities resulting in potential changes to the system architecture and system and software safety requirements.

For the purpose of the thesis, we introduce a fictitious system which is both safety-critical and software intensive. The case study involves the architectural design of

a safety-critical weapon system, a fictitious Surface-to-Air Missile that is used to protect warships from attacking missiles and aircrafts. We focus on the need for software to control the arming device of a missile. We describe the safety engineering steps from the identification of system's hazards to the critical functions that the software has to provide to avoid premature detonation, resulting in four software safety requirements for the software which controls the missile's Electronic Safe Arm Device (ESAD) for the arming of its warhead. We formalize the software safety requirements as statechart assertions and validate their correctness via JUnit test. We develop software architecture for the control software using the Safety Executive pattern and implement the design in C++ to support a simple time-step simulation to produce the required log files for the verification of the design. While this thesis focuses on software safety, the methodology for formal V&V of software architecture proposed in this thesis is not restricted to the safety attribute alone. It can be adopted to facilitate the formal V&V of other nonfunctional attributes of systems as well.

B. LESSONS LEARNED

The focus of the research reported here is software system safety. Software system safety characterizes the system's behavior that can lead to mishaps. To understand how these mishaps can occur we first need to identify the potential hazards and the associated contributing factors. Errors can be introduced and be difficult to detect into the safety-critical and safety-related functions implemented in software, such as when the safety controls themselves introduce added complexity into the software's design. Thus, there is a need to provide safety and software engineers with means for performing assurance which can deal with varying levels of software-design complexity.

Safety and software engineers work with safety requirements initially specified in a natural language. One of the key challenges is to correctly interpret and detect problems with those requirements early in the system life cycle. Otherwise, misinterpretations of the requirements and errors in the requirements will propagate into the software architecture, design and detailed implementation. Statechart assertions can be used for this purpose, but the engineer needs to declare explicitly what the statechart assertion is

going to be from the observer's point of view, instead of capturing the complete state behavior of the software design in the assertions. Moreover, while the statechart assertions are capable of revealing weakness of the known requirements, the assertions cannot capture missing requirements. The safety and software engineers must work with the stakeholders to determine whether the results of conducting validation tests of the statechart assertions and verification tests of the software architecture indicate there has been a misinterpretation or omission of the stakeholders' expectations regarding system safety.

C. FUTURE WORK

We recommend that there be follow-on studies conducted with the aim of evaluating the effectiveness of the proposed methodology and the safety-kernel architecture in handling changes and additions in requirements and safety policies. There is also a need to determine to what extent our approach needs to be tailored to address other non-functional aspects of systems, such as security, survivability, and reliability.

The time-step simulation code reflects a minimalist approach. The design can be refactored into a better inheritance hierarchy to eliminate some of the redundant code. Another challenging problem remaining to be tackled is to develop a way to ensure that the used code does not contain unnecessary and unwanted lines of code.

A potential application of the proposed framework is to facilitate code reuse in safety critical systems. Safe code reuse requires extensive testing of the reusable code in its new environment, which includes the new system's context (the system's operating environment) and the hardware and software components the reusable code will interact with. We recommend a follow-on study on the effectiveness of the proposed framework to automate the verification of code reuse in safety-critical systems.

APPENDIX A

This section includes C++ source and header files for the behavioral implementation of software that simulates the arming procedure for the ESAD.

```
//============================================================
//        Author:         Vasileios Karagiannakis
//        Naval Postgraduate School
//        Computer Science Department
//        Date:           11 Aug 2013
//        File Name:      simDriver.cpp
//============================================================
#include <iostream>
#include <string>
#include <stdlib.h>
#include <ctime>

#include “Message.h”
#include “MessageQueue.h”
#include “IdGenerator.h”
//#include “RandGen.h”

#include “ActuationChannel.h”
#include “ArmingControl.h”
#include “TimeGuard.h”
#include “SafeGuard.h”
#include “Esad.h”
#include “Logger.h”

using namespace std;

// Global Variables
IdGenerator* _idgen;
ActuationChannel* _actChnl;
ArmingControl* _armCtrl;
TimeGuard* _tmGrd;
SafeGuard* _sfGrd;
Esad* _esad;
Logger* _lgr;

/* secondary function that uses the instances of the 5 components to have the same time */
void dispatch(long t)
{
        _actChnl->setTime(t);
        _armCtrl->setTime(t);
        _tmGrd->setTime(t);
        _sfGrd->setTime(t);
        _esad->setTime(t);
}
```

```
int main() {

/* create the instance from the ActuationChannel, ArmingControl, TimeGuard, SafeGuard, Esad
and the Logger in order to create the events and write them down to the logfile*/

        _idgen = IdGenerator::getIdGenerator();
        _actChnl = ActuationChannel::getActuationChannel();
        _armCtrl = ArmingControl::getArmingControl();
        _tmGrd = TimeGuard::getTimeGuard();
        _sfGrd = SafeGuard::getSafeGuard();
        _esad = Esad::getEsad();
        _lgr = Logger::getLogger();

        _sfGrd->setArmingControlReference(_armCtrl);

        // the name of the log file describes the simulation test case
        _lgr->openLogFile("logfile.txt");

        // create the same baseline time for all the instances.
        time_t startTime;
        time(&startTime);
        long myTime = (long) startTime;
        dispatch(myTime);
        myTime++;
        cout << endl;

        // message 1 from the environment to the ActuationChannel
        Message* mes1 = new Message(_idgen->getId(),myTime,"powerOn");
        _actChnl->receive(*mes1);

        // test case: time delay between the two powerOn signals
        /*dispatch(myTime);
        myTime++;
        */
        // Message 2 from the environment to the TimeGuard
        Message* mes2 = new Message(_idgen->getId(),myTime,"powerOn");
        _tmGrd->receive(*mes2);

        // update the time three times
        dispatch(myTime);
        myTime++;
        dispatch(myTime);
        myTime++;

        // test case: time delay for the makeLaunch
        /*dispatch(myTime);
        myTime++;
        dispatch(myTime);
        myTime++;
        dispatch(myTime);
        myTime++;
```

```
*/
// message 3 from the environment to the ActuationChannel
Message* mes3 = new Message(_idgen->getId(),myTime,"makeLaunch");
_actChnl->receive(*mes3);

// update the time two times
dispatch(myTime);
myTime++;
dispatch(myTime);
myTime++;
// test case: time delay to the acceleration readings
// and also to the startMotion message
/*dispatch(myTime);
myTime++;
dispatch(myTime);
myTime++;
dispatch(myTime);
myTime++;
dispatch(myTime);
myTime++;*/
// Accelerometer sends data to the ActuationChannel

/*
// test case: no startMotion signal due to no acceleration values
Message* mes4 = new Message(_idgen->getId(),myTime,"readAcceleration,"0.0);
_actChnl ->receive(*mes4);
Message* mes5 = new Message(_idgen->getId(),myTime,"readAcceleration,"0.0);
_actChnl ->receive(*mes5);
Message* mes6 = new Message(_idgen->getId(),myTime,"readAcceleration,"0.0);
_actChnl ->receive(*mes6);
Message* mes7 = new Message(_idgen->getId(),myTime,"readAcceleration,"0.0);
_actChnl ->receive(*mes7);
Message* mes8 = new Message(_idgen->getId(),myTime,"readAcceleration,"0.0);
_actChnl ->receive(*mes8);
Message* mes9 = new Message(_idgen->getId(),myTime,"readAcceleration,"0.0);
_actChnl ->receive(*mes9);
*/

// message 4 from the environment to the ActuationChannel       accel#1
// cases 1,3,9,11 from the Appendix B description
Message* mes4 = new Message(_idgen->getId(),myTime,"readAcceleration,"0.5);
_actChnl ->receive(*mes4);

dispatch(myTime);
myTime++;

// message 5 from the environment to the ActuationChannel       accel#2
// case 1 , 9
Message* mes5 = new Message(_idgen->getId(),myTime,"readAcceleration,"1.5);
// case 3, 11 from the Appendix B description
//Message* mes5 = new Message(_idgen->getId(),myTime,"readAcceleration,"3.5);
_actChnl ->receive(*mes5);
```

```
dispatch(myTime);
myTime++;
// test case: proper values for FMD with timedelay
/*dispatch(myTime);
myTime++;
dispatch(myTime);
myTime++;
dispatch(myTime);
myTime++;*/

// message 6 from the environment to the ActuationChannel      accel#3
// case 1 from the Appendix B description
//Message* mes6 = new Message(_idgen->getId(),myTime,"readAcceleration,"1.5);
// case 3 from the Appendix B description
//Message* mes6 = new Message(_idgen->getId(),myTime,"readAcceleration,"4.5);
// case 9 , 11 from the Appendix B description
Message* mes6 = new Message(_idgen->getId(),myTime,"readAcceleration,"6.0);
_actChnl ->receive(*mes6);

dispatch(myTime);
myTime++;
// test case: proper values for FMD, SSD with timedelay during FMD
/*dispatch(myTime);
myTime++;
dispatch(myTime);
myTime++;*/

// message 7 from the environment to the ActuationChannel      accel#4
// case 1 from the Appendix B description
//Message* mes7 = new Message(_idgen->getId(),myTime,"readAcceleration,"2.0);
// case 3 from the Appendix B description
//Message* mes7 = new Message(_idgen->getId(),myTime,"readAcceleration,"5.5);
// case 9 , 11 from the Appendix B description
Message* mes7 = new Message(_idgen->getId(),myTime,"readAcceleration,"6.0);
_actChnl ->receive(*mes7);

dispatch(myTime);
myTime++;

// message 8 from the environment to the ActuationChannel      accel#5
// case 1 from the Appendix B description
//Message* mes8 = new Message(_idgen->getId(),myTime,"readAcceleration,"2.5);
// case 3 from the Appendix B description
//Message* mes8 = new Message(_idgen->getId(),myTime,"readAcceleration,"5.5);
// case 9 from the Appendix B description
//Message* mes8 = new Message(_idgen->getId(),myTime,"readAcceleration,"1.5);
// case 11 from the Appendix B description
Message* mes8 = new Message(_idgen->getId(),myTime,"readAcceleration,"7.5);
_actChnl ->receive(*mes8);

dispatch(myTime);
```

```
        myTime++;

        // test case: proper values for FMD and SSD with timedelay after FMD to SSD
        /*dispatch(myTime);
        myTime++;
        */
        // message 9 from the environment to the ActuationChannel      accel#6
        // case 1 from the Appendix B description
        //Message* mes9 = new Message(_idgen->getId(),myTime,"readAcceleration,"3.0);
        // case 3 from the Appendix B description
        //Message* mes9 = new Message(_idgen->getId(),myTime,"readAcceleration,"6.0);
        // case 9 from the Appendix B description
        //Message* mes9 = new Message(_idgen->getId(),myTime,"readAcceleration,"1.5);
        // case 11 from the Appendix B description
        Message* mes9 = new Message(_idgen->getId(),myTime,"readAcceleration,"8.0);
        _actChnl ->receive(*mes9);

        dispatch(myTime);
        myTime++;
        dispatch(myTime);
        myTime++;

        _lgr->closeLogFile();

        //system("PAUSE");
        return 0;
} // main()

//=========================================================
//      Author:         Vasileios Karagiannakis
//      Naval Postgraduate School
//      Computer Science Department
//      Date:           11 Aug 2013
//      File Name:      ActuationChannel.h
//=========================================================

#ifndef ACTUATIONCHANNEL_H_
#define ACTUATIONCHANNEL_H_

#include <string>
#include <iostream>
#include "MessageQueue.h"
#include "IdGenerator.h"
#include "RandGen.h"
#include "ArmingControl.h"
#include "TimeGuard.h"
#include "Logger.h"

using namespace std;

class ActuationChannel {
public:
```

```
        static ActuationChannel* getActuationChannel();

        virtual ~ActuationChannel();

        void receive(Message);
        void setTime(long);

        void powerOn();
        void makeLaunch();
        void readAcceleration(float);

private:
ActuationChannel();
        static bool ActuationChannelFlag;
        static ActuationChannel* _actChnl;

        MessageQueue<50> mesQue;
        long _timestamp;

        //RandGen* _rdgen;
        IdGenerator* _idg;
        TimeGuard* _tmGrd;
        ArmingControl* _armCtrl;
        Logger* _lgr;

        float _accel;
        long _oldTimestamp; /* needed to compute distance travel between 2 consecutives
acceleration readings.*/
        double _distanceTravel;
        bool _endFMD; /* boolean variable in order to send the endFirstMotionDetection only
one time to the TimeGuard*/
        bool _endSSD; /* boolean variable in order to send the endSafeSeparation only one time
to the TimeGuard*/
};

#endif /* ACTUATIONCHANNEL_H_ */

//============================================================
//      Author:         Vasileios Karagiannakis
//      Naval Postgraduate School
//      Computer Science Department
//      Date:           11 Aug 2013
//      File Name:      ActuationChannel.cpp
//============================================================

#include <iostream>
#include <time.h>
#include <stdlib.h>
#include <windows.h>
#include “ActuationChannel.h”
```

```
using namespace std;

bool ActuationChannel::ActuationChannelFlag = false;
ActuationChannel* ActuationChannel::_actChnl = NULL;

// Constructor
ActuationChannel::ActuationChannel()
{
        //_rdgen = RandGen::getInstance();
        _idg = IdGenerator::getIdGenerator();
        _tmGrd = TimeGuard::getTimeGuard();
        _armCtrl = ArmingControl::getArmingControl();
        _lgr = Logger::getLogger();
        _accel = 0.0; // values are in g's

        _distanceTravel = 0.0; // values are in meters
        _endFMD = false;
        _endSSD = false;
} // ActuationChannel()

// Destructor
ActuationChannel::~ActuationChannel()
{
        ActuationChannelFlag = false;
}// ~ActuationChannel()

void ActuationChannel::receive(Message m)
{
        if (m.getPayload() == "readAcceleration")
                _lgr->logAcceleration(m.getData(), m.getTimestamp());
        mesQue.insert(m);
}// void receive(Message m)

void ActuationChannel::setTime(long t)
{
        _oldTimestamp = _timestamp;
        _timestamp = t;
        bool done;
        done= false;
        while (!done)
        {
                if (mesQue.size() == 0)
                        done = true;
                else
                {
                        Message temp = mesQue.remove();
                if (temp.getTimestamp() <= _timestamp)
                {
                        if (temp.getPayload() == "powerOn")
                                powerOn();
                        else if (temp.getPayload() == "makeLaunch")
                                makeLaunch();
```

```
                        else if (temp.getPayload() == “readAcceleration”)
                                readAcceleration(temp.getData());
                        else
                                cout << “Unrecognized payload: “ << temp << endl;
                }else
                        {
                                mesQue.insert(temp);
                                done = true;
                        }
                }
        }
} // void ActuationChannel::setTime(long t)

void ActuationChannel::powerOn()
{
// make the power-on self test and sends the endPost message after 2 seconds to ArmingControl
        Message* temp = new Message(_idg->getId(), _timestamp + 2 , “endPost”);
// test case: violates endPostDeadline
        //Message* temp = new Message(_idg->getId(), _timestamp + 3 , “endPost”);
// test case: creates invalid endPost
        //Message* temp = new Message(_idg->getId(), _timestamp + 2 , “invalidPost”);
        //_tmGrd->receive(*temp);
        _armCtrl->receive(*temp);
} // void ActuationChannel::powerOn()

void ActuationChannel::makeLaunch()
{
/* receives the makeLaunch signal and transfers it to doLaunch signal after 1 second and sends it
to the ArmingControl*/
        Message* temp = new Message(_idg->getId(), _timestamp + 1, “doLaunch”);
// test case: violates endDoLaunchDeadline
        // Message* temp = new Message(_idg->getId(), _timestamp + 4, “doLaunch”);
        _armCtrl->receive(*temp);
}//void ActuationChannel::makeLaunch()

void ActuationChannel::readAcceleration(float d)
{
        if (_accel == 0.0 )
        //if (_accel > 0.0 )
        {
                Message* temp = new Message(_idg->getId(), _timestamp, “startMotion”);
// test case: time  delay the startMotion
        // Message* temp = new Message(_idg->getId(), _timestamp + 2, “startMotion”);
                _tmGrd->receive(*temp);
        }

        //when two consecutive values are over 6 g’s send a message to ArmingControl
        if (_accel >= 6.0 && d >= 6.0 && !_endFMD)
        {
                //create a message endFirstMotionDetection to ArmingControl
                Message*  temp  =  new  Message(_idg->getId(),  _timestamp,
“endFirstMotionDetection”);
```

```
                // test case: violates endFirstMotionDetectionDeadline
//Message* temp = new Message(_idg->getId(), _timestamp + 2, “endFirstMotionDetection”);
                _armCtrl->receive(*temp);
                _endFMD = true;
        }
        // update distance travel equals to interval times average acceleration square
        _distanceTravel += ((_timestamp - _oldTimestamp)*(_timestamp -
_oldTimestamp))*(d+_accel)/2.0;
        // 20 meters is the height of the superstructure above the highest point of the warship

        if (_distanceTravel > 20 && !_endSSD) // distance travel clears superstructure
        {
                //create a message endFirstMotionDetection to ArmingControl
                Message* temp = new Message(_idg->getId(), _timestamp,
“endSafeSeparation”);
// test case: violates endSafeSeparationDeadline
                //Message* temp = new Message(_idg->getId(), _timestamp + 5 ,
“endSafeSeparation”);
                _armCtrl->receive(*temp);
                _endSSD = true;
        }
        _accel = d; // update the acceleration readings
} // void ActuationChannel::readAcceleration(float d)

ActuationChannel* ActuationChannel::getActuationChannel()
{
        // creates the instance of a ActuationChannel
        if(!ActuationChannelFlag)
        {
                _actChnl = new ActuationChannel();
                ActuationChannelFlag = true;
                return _actChnl;
        }else
        {
                return _actChnl;
        }
}// ActuationChannel* ActuationChannel::getActuationChannel()

//=============================================================
//      Author:         Vasileios Karagiannakis
//      Naval Postgraduate School
//      Computer Science Department
//      Date:           11 Aug 2013
//      File Name:      ArmingControl.h
//=============================================================

#ifndef ARMINGCONTROL_H_
#define ARMINGCONTROL_H_

#include <string>
#include <iostream>
#include “MessageQueue.h”
```

```
#include “RandGen.h”
#include “IdGenerator.h”
#include “TimeGuard.h”
#include “Esad.h”

using namespace std;

class ArmingControl {
public:

	static ArmingControl* getArmingControl();
	virtual ~ArmingControl();

	void endPost();
	void doLaunch();
	void receive(Message);
	void setTime(long);
	void endFirstMotionDetection();
	void endSafeSeparation();
	void terminate();
	void armEsad();

private:
	ArmingControl();
	static bool ArmingControlFlag;
	static ArmingControl* _armCtrl;

	MessageQueue<50> mesQue;

	IdGenerator* _idg;
	//RandGen* _rdgen;
	TimeGuard* _tmGrd;
	Esad* _esad;

	long _timestamp;
	int _state; /* the different values the variable _state could be: 0. off; 1. endPost; 2. doLaunch; 3. endFirstMotionDetection; 4. endSafeSeparation; 5. Terminate*/
};

#endif /* ARMINGCONTROL_H_ */

//============================================================
//	Author:		Vasileios Karagiannakis
//	Naval Postgraduate School
//	Computer Science Department
//	Date:		11 Aug 2013
//	File Name:	ArmingControl.cpp
//============================================================

#include <iostream>
#include <time.h>
#include <stdlib.h>
```

```
#include <windows.h>
#include “ArmingControl.h”

using namespace std;

bool ArmingControl::ArmingControlFlag = false;
ArmingControl* ArmingControl::_armCtrl = NULL;

// Constructor
ArmingControl::ArmingControl()
{
	_idg = IdGenerator::getIdGenerator();
	_tmGrd = TimeGuard::getTimeGuard();
	_esad = Esad::getEsad();

	_state = 0;
}// ArmingControl()

// Desturctor
ArmingControl::~ArmingControl()
{
	ArmingControlFlag = false;
} // ~ArmingControl()

void ArmingControl::receive(Message m)
{
	mesQue.insert(m);
}// void ArmingControl::receive(Message m)

void ArmingControl::setTime(long t)
{
	_timestamp = t;
	bool done;
	done= false;
	while (!done)
	{
		if (mesQue.size() == 0)
			done = true;
		else
		{
			Message temp = mesQue.remove();
		if (temp.getTimestamp() <= _timestamp)
		{
			if (temp.getPayload() == “endPost”)
				endPost();
			else if (temp.getPayload() == “doLaunch”)
				doLaunch();
			else if (temp.getPayload() == “endFirstMotionDetection”)
				endFirstMotionDetection();
			else if (temp.getPayload() == “endSafeSeparation”)
				endSafeSeparation();
			else if (temp.getPayload() == “abort”)
```

```
                        terminate();
                else
                        cout << “Unrecognized payload: “ << temp << endl;
        }else
                {
                        mesQue.insert(temp);
                        done = true;
                }
        }
    }
} // void ArmingControl::setTime(long t)

void ArmingControl::endPost()
{
    if (_state == 0)
    {
        _state = 1; // endPost
        Message* temp = new Message(_idg->getId(), _timestamp, “endPost”);
        _tmGrd->receive(*temp);// send message endPost to TimeGuard
    }
} // void ArmingControl::endPost()

void ArmingControl::doLaunch()
{
    if (_state == 1)
    {
        _state = 2; // doLaunch
        Message* temp = new Message(_idg->getId(), _timestamp, “doLaunch”);
        _tmGrd->receive(*temp); // send message to doLaunch to TimeGuard
    }
}// void ArmingControl::doLaunch()

void ArmingControl::endFirstMotionDetection()
{
    if (_state == 2)
    {
        _state = 3; // endFirstMotionDetection
        Message*    temp    =    new    Message(_idg->getId(),    _timestamp,
“endFirstMotionDetection”);
        _tmGrd->receive(*temp);/*send    message    to    endFirstMotionDetection    to
TimeGuard*/
    }
}// void ArmingControl::endFirstMotionDetection()

void ArmingControl::endSafeSeparation()
{
    if (_state == 3)
    {
        _state = 4; // endSafeSeparation
        Message*    temp    =    new    Message(_idg->getId(),    _timestamp,
“endSafeSeparation”);
        _tmGrd->receive(*temp);// send message endSafeSeparation to TimeGuard
```

```
                Message* temp1 = new Message(_idg->getId(), _timestamp, “ARM”);
                _esad->receive(*temp1);// send message endSafeSeparation to TimeGuard
        }
} // void ArmingControl::endSafeSeparation()

void ArmingControl::terminate()
{
        _state = 5 ; // terminate
} // void ArmingControl::terminate()

ArmingControl* ArmingControl::getArmingControl()
{
// creates the instance of a ArmingControl
        if(!ArmingControlFlag)
        {
                _armCtrl = new ArmingControl();
                ArmingControlFlag = true;
                return _armCtrl;
        }else
        {
                return _armCtrl;
        }
}// ArmingControl* ArmingControl::getArmingControl()

//=============================================================
//      Author:         Vasileios Karagiannakis
//      Naval Postgraduate School
//      Computer Science Department
//      Date:           11 Aug 2013
//      File Name:      Esad.cpp
//=============================================================

#ifndef ESAD_H_
#define ESAD_H_

#include <string>
#include <iostream>
#include “MessageQueue.h”
#include “RandGen.h”
#include “IdGenerator.h”
#include “Logger.h”

using namespace std;

class Esad {
public:

        static Esad* getEsad();
        ~Esad();

        void receive(Message);
        void setTime(long);
```

```
        void notARM();
        void arm();

private:
        Esad();
        static bool EsadFlag;
        static Esad* _esad;

        MessageQueue<50> mesQue;

        //RandGen* _rdgen;
        IdGenerator* _idg;
        Logger* _lgr;

        long _timestamp;
        int _state; // the different values the variable _state could be: 0. umarmed; 1. arm;
};

#endif /* ESAD_H_ */

//=============================================================
//      Author:         Vasileios Karagiannakis
//      Naval Postgraduate School
//      Computer Science Department
//      Date:           11 Aug 2013
//      File Name:      Esad.cpp
//=============================================================

#include “Esad.h”
using namespace std;

bool Esad::EsadFlag = false;
Esad* Esad::_esad = NULL; // initialiazation for the pointer

// Constructor
Esad::Esad()
{
        _idg = IdGenerator::getIdGenerator();
        _lgr = Logger::getLogger();
        _state = 0;
} // Esad()

//Destructor
Esad::~Esad()
{
        EsadFlag = false;
}// ~Esad()

void Esad::receive(Message m)
{
        _lgr->logEvent(m.getPayload(),m.getTimestamp());
```

```
        mesQue.insert(m);
}// void Esad::receive(Message m)

void Esad::setTime(long t)
{
        _timestamp = t;
        bool done;
        done = false;
        while (!done)
        {
                if (mesQue.size() == 0)
                        done = true;
                else
                {
                        Message temp = mesQue.remove();
                        if (temp.getTimestamp() <= _timestamp)
                        {
                                if (temp.getPayload() == “notARM”)
                                        notARM();
                                else if (temp.getPayload() == “ARM”)
                                        arm();
                                else
                                        cout << “Unrecognized event” << endl;
                                }
                        else
                                {
                                        mesQue.insert(temp);
                                        done = true;
                                }
                }
        }
}// void Esad::setTime(long t)

void Esad::notARM()
{
        _state = 0;
} // void SafeGuard::notARM()

void Esad::arm()
{
        _state = 1;
} // void SafeGuard::arm()

Esad* Esad::getEsad()
{
        // creates the instance of a SafeGuard
        if(!EsadFlag)
        {
                _esad = new Esad();
                EsadFlag = true;
                return _esad;
        }else
```

```
	{
		return _esad;
	}
} // Esad* Esad::getEsad()

//=============================================================
//	Author:		Man-Tak Shing
//	Naval Postgraduate School
//	Computer Science Department
//	Date:		08 Aug 2013
//	File Name:	IdGenerator.h
//=============================================================

#ifndef IDGENERATOR_H_
#define IDGENERATOR_H_

#include <string>
#include <iostream>
using namespace std;

class IdGenerator {
public:

	static IdGenerator* getIdGenerator();
	~IdGenerator();

	int getId(); // Gets Id

private:
	IdGenerator();
	static bool IdGeneratorFlag;
	static IdGenerator* _idg;

	int _id; //
};
#endif /* IDGENERATOR_H_ */

//=============================================================
//	Author:		Man-Tak Shing
//	Naval Postgraduate School
//	Computer Science Department
//	Date:		08 Aug 2013
//	File Name:	IdGenerator.cpp
//=============================================================

#include “IdGenerator.h”
using namespace std;

bool IdGenerator::IdGeneratorFlag = false;
IdGenerator* IdGenerator::_idg = NULL; // initialiazation for the pointer

// Constructor
```

```
IdGenerator::IdGenerator()
{
	_id = 0;
} // IdGenerator()

// Destructor
IdGenerator::~IdGenerator()
{
	IdGeneratorFlag = false;
} // ~IdGenerator()

int IdGenerator::getId()
{
	_id++;
	return _id;
} // getId()

IdGenerator* IdGenerator::getIdGenerator()
{
	if(!IdGeneratorFlag)
	{
		_idg = new IdGenerator();
		IdGeneratorFlag = true;
		return _idg;
	}else
	{
		return _idg;
	}
}// IdGenerator* getIdGenerator()

//=========================================================
//	Author:		Vasileios Karagiannakis
//	Naval Postgraduate School
//	Computer Science Department
//	Date:		13 Aug 2013
//	File Name:	Logger.h
//=========================================================

#ifndef LOGGER_H_
#define LOGGER_H_

#include <string>
#include <iostream>
#include <fstream>
#include “IdGenerator.h”

using namespace std;

class Logger {
public:

	static Logger* getLogger();
```

```
        ~Logger();

        void logEvent(string,long);
        void logAcceleration(float,long);
        void openLogFile(char*);
        void closeLogFile();

private:
        Logger();
        static bool LoggerFlag;
        static Logger* _lgr;

        IdGenerator* _idg;
        ofstream _logfile;
};
#endif /* LOGGER_H_ */

//=============================================================
//      Author:         Vasileios Karagiannakis
//      Naval Postgraduate School
//      Computer Science Department
//      Date:           13 Aug 2013
//      File Name:      Logger.cpp
//=============================================================

#include “Logger.h”
using namespace std;

bool Logger::LoggerFlag = false;
Logger* Logger::_lgr = NULL; // initialization for the pointer

// Constructor
Logger::Logger()
{
} // Logger()

// Destructor
Logger::~Logger()
{
        LoggerFlag = false;
} // ~Logger()

void Logger::logEvent(string s,long t)
{
        _logfile << s << “ “ << “ @ “ << t << endl;
} // int Logger::logEvent(string,long)

void Logger::logAcceleration(float d,long t)
{
        _logfile << d << “ “ << “g “ << “ @ “ << t << endl;

}// void Logger::logAcceleration(string,long)
```

```
void Logger::openLogFile(char* f)
{
	_logfile.open(f);
}// void Logger::openLogFile(string)

void Logger::closeLogFile()
{
	_logfile.close();
}// void Logger::closeLogFile()

Logger* Logger::getLogger()
{
	if(!LoggerFlag)
	{
		_lgr = new Logger();
		LoggerFlag = true;
		return _lgr;
	}else
	{
		return _lgr;
	}
}// Logger* getLogger()

//============================================================
//	Author:		Man-Tak Shing modified by Nahum Camacho Zamora
//	Naval Postgraduate School
//	Computer Science Department
//	Date:		09 Feb 2013
//	File Name:	MaxHeap.h
//============================================================

#ifndef MAXHEAP_H_
#define MAXHEAP_H_

#include <string>
#include <iostream>
using namespace std;

// prototypes
template<class T, int maxSize>
class MaxHeap;

template<class T, int maxSize>
ostream& operator<<(ostream& out, MaxHeap<T, maxSize> h);

// class template
template<class T, int maxSize>
class MaxHeap {
public:
	MaxHeap();
	MaxHeap(T inArray[], int size);
```

```
        // create a MaxHeap from inArray[0..size-1]
        virtual ~MaxHeap();

        // accessor functions
        T removeMax();      // removes the largest element from MaxHeap and returns it to the
caller

        // mutator functions
        void insert(T elt); // inserts the element elt into the heap

        int size();         // return the number of elements in the heap

        friend ostream& operator<<<T, maxSize>(ostream& out, MaxHeap<T, maxSize> h);

private:
        static const int _MaxSize = maxSize;
        T _heapArray[_MaxSize];
        int _heapSize;

        void restore(int pos);
        void swap(T& x, T& y);

};

// constructors

// For an empty object
template<class T, int maxSize>
MaxHeap<T, maxSize>::MaxHeap() {
        _heapSize = 0;
} // MaxHeap()

// For a non empty object
template<class T, int maxSize>
MaxHeap<T, maxSize>::MaxHeap(T inArray[], int size) {
        if (size > _MaxSize) {
                throw “heap overflow”;
        } // if

        // set _heapSize to size
        _heapSize = size;

        // Copy the elements of the passed array in _heapArray
        for (int i = 0; i < size; i++)
                _heapArray[i] = inArray[i];

        // build the heap
        for (int i = (_heapSize - 2) / 2; i >= 0; i--) {
                restore(i);                 // Call function restore
        } // for
}  // MaxHeap(T, int)
```

```
// destructors
template<class T, int maxSize>
MaxHeap<T, maxSize>::~MaxHeap() {
        _heapSize = 0;
} // destructor

// utility functions

template<class T, int maxSize>
int MaxHeap<T, maxSize>::size() {
        return _heapSize;
} // size()

template<class T, int maxSize>
void MaxHeap<T, maxSize>::swap(T& x, T& y) {
        T temp = x; // Had temporary to swap
        x = y;
        y = temp;
} // swap(T&, T&)

template<class T, int maxSize>
void MaxHeap<T, maxSize>::restore(int pos) {

        bool done = false;      // State to validate conditions
        int current = pos;              // Initial position
        int largerChild;        // Largest child

        // Analyze at a given node the condition between father and children

        while (!done) {
                if (2 * current + 1 >= _heapSize)  // current is a leaf node
                        done = true;
                else {
                        // find larger child
                        largerChild = 2 * current + 1;
                        if (2 * current + 2 < _heapSize
                                        && _heapArray[2 * current + 2]
                                                        > _heapArray[2 * current + 1])
                                // right child is larger
                                largerChild = 2 * current + 2;

                        // compare larger child against parent
                        if (_heapArray[current] >= _heapArray[largerChild])
                                done = true;
                        else {
                                // swap elements at current and largerChild
                                // set current to largerChild
                                swap(_heapArray[current], _heapArray[largerChild]);
                                current = largerChild;
                        } // else
                } // else
        } // while
```

```
} // restore (int)

template<class T, int maxSize>
void MaxHeap<T, maxSize>::insert(T elt) {
        if (_heapSize == maxSize) {
                throw “heap overflow”;
        } // if
        int current;                    // Index
        _heapSize++;                    // Increase heap size in one space
        _heapArray[_heapSize - 1] = elt;                // Assign new element to new slot
        current = _heapSize - 1;                        // Current index is last slot
        bool done = false;                              // Initialize condition

// Compare element inserted with respective father in each node till reach root node
        while (!done) {
                if (current != 0 && _heapArray[current] > _heapArray[(current - 1) / 2]) {
                        // swap elements at current and (current-1)/2
                        // set current to (current-1)/2
                        swap(_heapArray[current], _heapArray[(current - 1) / 2]);
                        current = (current - 1) / 2;
                } // if
                else
                        done = true;
        } // while
} // insert(int)

template<class T, int maxSize>
T MaxHeap<T, maxSize>::removeMax() {
        if (_heapSize == 0) {
                throw “heap underflow”;
        } // if

        T temp;

        temp = _heapArray[0];                           // Number to return
        _heapArray[0] = _heapArray[_heapSize - 1];      // Number from last to first position
        _heapSize--;                                            // Decrease heap size
        restore(0);                                             // Call restore function
        return temp;                                            // Return maximum value
} // removeMax()


// non-member functions

template<class T, int maxSize>
ostream& operator<<(ostream& out, MaxHeap<T, maxSize> h) {
        if (h._heapSize > 0) {
                out << h._heapArray[0];
        } // if
        for (int i = 1; i < h._heapSize; i++) {
                out << ,” “ << h._heapArray[i];
        } // for
```

```
	return out;
} // operator<<(ostream& out, MaxHeap<T, maxSize> h)

#endif /* MAXHEAP_H_ */

//============================================================
//	Author:		Man-Tak Shing modified by Vasileios Karagiannakis
//	Naval Postgraduate School
//	Computer Science Department
//	Date:		04 Aug 2013
//	File Name:	Message.h
//============================================================

#ifndef MESSAGE_H_
#define MESSAGE_H_

#include <string>
#include <iostream>

using namespace std;

class Message {
public:
	Message();
	Message(int, long , string);
	Message(int id, long pr, string pd, float d);
	virtual ~Message();

	int getId();			// Gets Id
	long getTimestamp();		// Gets Timestamp
	string getPayload();		// Gets Payload
	float getData();		// Gets Data

	void setId(int x);			// Set Id
	void setTimestamp(long x);	// Set Timestamp
	void setPayload(string s);		// Set Payload
	void setData(float d);		// Set Data

	friend ostream& operator<< (ostream& out, Message m);

private:
	int _id;
	long _timestamp;
	string _payload;
	float _data;
};

// non member functions
bool operator < (Message, Message);
bool operator <= (Message, Message);
bool operator > (Message, Message);
bool operator >= (Message, Message);
```

```
bool operator == (Message, Message);

ostream& operator<< (ostream& out, Message m);

#endif /* MESSAGE_H_ */

//=============================================================
//        Author:          Man-Tak Shing modified by Vasileios Karagiannakis
//        Naval Postgraduate School
//        Computer Science Department
//        Date:            04 Aug 2013
//        File Name:       Message.cpp
//=============================================================
#include <iostream>
#include <time.h>
#include <windows.h>
#include “Message.h”
using namespace std;

// Constructor
Message::Message() {
        _id = 0;
        _timestamp = 0;
        _payload = “Nothing”;
        _data = 0.0;
} // Message()

// Constructor
Message::Message(int id, long pr, string pd) {
        _id = id;
        _timestamp = pr;
        _payload = pd;
        _data = 0.0;
} // Message(int, long, string)

Message::Message(int id, long pr, string pd, float d) {
        _id = id;
        _timestamp = pr;
        _payload = pd;
        _data = d;
} // Message(int, long, string, float)

// Destructor
Message::~Message() {

} // ~Message()


int Message::getId() {
        return _id;
} // getId()
```

```
long Message::getTimestamp() {
	return _timestamp;
} // getTimestamp()

string Message::getPayload() {
	return _payload;
} // getPayload()

float Message::getData() {
	return _data;
} // getData()

void Message::setId(int x) {
	_id = x;
} // setId(int)

void Message::setTimestamp(long x) {
	_timestamp = x;
} // setTimestamp(long)

void Message::setPayload(string s) {
	_payload = s;
} // setPayload(string)

void Message::setData(float d) {
	_data = d;
} // setData(float)

// non member functions

// friendly implementation
ostream& operator<<(ostream& out, Message m) {
	out << "[Id = " << m._id << "," ";
	out << "Timestamp = " << m._timestamp << "," ";
	out << "Payload = " << m._payload << "]";
	return out;
} // ostream& operator<<(ostream&, Message)

// non friendly implementation
bool operator <(Message a, Message b) {

	// Get the private data with the public methods
	int id1 = a.getId();
	int id2 = b.getId();
	long pr1 = a.getTimestamp();
	long pr2 = b.getTimestamp();

	return ((pr1 > pr2) || ((pr1 == pr2) && (id1 > id2))); // a timestamp is bigger than the
second timestamp
} // bool operator <(Message, Message)

bool operator <=(Message a, Message b) {
```

```
        // Get the private data with the public methods
        int id1 = a.getId();
        int id2 = b.getId();
        long pr1 = a.getTimestamp();
        long pr2 = b.getTimestamp();

        return ((pr1 >= pr2) && (id1 >= id2));
} // bool operator <=(Message, Message)

bool operator >(Message a, Message b) {

        // Get the private data with the public methods
        int id1 = a.getId();
        int id2 = b.getId();
        long pr1 = a.getTimestamp();
        long pr2 = b.getTimestamp();

        return ((pr1 < pr2) || ((pr1 == pr2) && (id1 < id2)));
} // bool operator >(Message, Message)

bool operator >=(Message a, Message b) {

        // Get the private data with the public methods
        int id1 = a.getId();
        int id2 = b.getId();
        long pr1 = a.getTimestamp();
        long pr2 = b.getTimestamp();

        return ((pr1 <= pr2) && (id1 <= id2));
} // bool operator >=(Message, Message)

bool operator ==(Message a, Message b) {

        // Get the private data with the public methods
        int id1 = a.getId();
        int id2 = b.getId();
        long pr1 = a.getTimestamp();
        long pr2 = b.getTimestamp();

        return ((pr1 == pr2) && (id1 == id2));
} // bool operator ==(Message, Message)

//=============================================================
//      Author:         Man-Tak Shing modified by Nahum Camacho Zamora
//      Naval Postgraduate School
//      Computer Science Department
//      Date:           16 Feb 2013
//      File Name:      MessageQueue.h
//=============================================================

#ifndef MESSAGEQUEUE_H_
```

```
#define MESSAGEQUEUE_H_

#include “Message.h”
#include “MaxHeap.h”
#include <iostream>
using namespace std;

// prototypes
template<int maxSize>
class MessageQueue;

template<int maxSize>
ostream& operator<< (ostream& out, MessageQueue<maxSize> q);


// class template
template <int maxSize>
class MessageQueue {
public:
        MessageQueue();
        virtual ~MessageQueue();

        void insert(Message);    // add message to queue
        Message remove();        // remove the message with highest priority
        int size();              // return number of jobs in queue

        friend ostream& operator<< <maxSize> (ostream& out, MessageQueue<maxSize> q);

private:
        MaxHeap<Message, maxSize> _queue;
};

// Constructor
template <int maxSize>
MessageQueue<maxSize>::MessageQueue(){

} // MessageQueue()

// Destructor
template <int maxSize>
MessageQueue<maxSize>::~MessageQueue(){

} // ~MessageQueue()

template<int maxSize>
void MessageQueue<maxSize>::insert(Message a){
        _queue.insert(a);
} // insert(Message)

template<int maxSize>
Message MessageQueue<maxSize>::remove(){
        return _queue.removeMax();
```

```
} // Message MessageQueue<maxSize>::remove()

template<int maxSize>
int MessageQueue<maxSize>::size(){
        return _queue.size();
} // int MessageQueue<maxSize>::size()

template<int maxSize>
ostream& operator<< (ostream& out, MessageQueue<maxSize> q) {
        return out << q._queue;
} // ostream& operator<< (ostream&, MessageQueue<maxSize>)

#endif /* MESSAGEQUEUE_H_ */

//=============================================================
//      Author:         Man-Tak Shing
//      Naval Postgraduate School
//      Computer Science Department
//      Date:           08 Aug 2013
//      File Name:      RandGen.h
//=============================================================

#pragma once
#ifndef RANDGEN_H_
#define RANDGEN_H_

class RandGen {
private:
        static bool instanceFlag;
        static RandGen *randGenerator;
        RandGen();

public:
        static RandGen* getInstance();

        int next();

        ~RandGen();
};

#endif /* RANDGEN_H_ */

//=============================================================
//      Author:         Man-Tak Shing
//      Naval Postgraduate School
//      Computer Science Department
//      Date:           08 Aug 2013
//      File Name:      RandGen.cpp
//=============================================================

#include “RandGen.h”
#include <stdlib.h>
```

```
#include <time.h>
#include <iostream>
using namespace std;

bool RandGen::instanceFlag = false;
RandGen* RandGen::randGenerator = NULL;

//Constructor
RandGen::RandGen() {
	//private constructor
	srand(time(NULL));
}// RandGen::RandGen()

// Destructor
RandGen::~RandGen() {
	instanceFlag = false;
}// RandGen::~RandGen()

int RandGen::next()
{
	return rand();
}// int RandGen::next()

RandGen* RandGen::getInstance() {
	if (!instanceFlag) {
		randGenerator = new RandGen();
		instanceFlag = true;
		return randGenerator;
	} else {
		return randGenerator;
	}
}// RandGen* RandGen::getInstance()

//=============================================================
//	Author:		Vasileios Karagiannakis
//	Naval Postgraduate School
//	Computer Science Department
//	Date:		11 Aug 2013
//	File Name:	SafeGuard.h
//=============================================================

#ifndef SAFEGUARD_H_
#define SAFEGUARD_H_

#include <string>
#include <iostream>
#include “MessageQueue.h”
#include “RandGen.h”
#include “IdGenerator.h”
#include “Esad.h”

using namespace std;
```

```
class ArmingControl; // to prevent circular dependencies, we declare explicitly the class ArmingControl
class SafeGuard {
public:

	static SafeGuard* getSafeGuard();
	~SafeGuard();

	void receive(Message);
	void setTime(long);

	void setArmingControlReference(ArmingControl*);  // to prevent circular dependencies

	void notARM();

private:
	SafeGuard();
	static bool SafeGuardFlag;
	static SafeGuard* _sfGrd;

	MessageQueue<50> mesQue;

	IdGenerator* _idg;
	ArmingControl* _armCtrl;
	Esad* _esad;

	long _timestamp;
};

#endif /* SAFEGUARD_H_ */

//============================================================
//	Author:		Vasileios Karagiannakis
//	Naval Postgraduate School
//	Computer Science Department
//	Date:		11 Aug 2013
//	File Name:	SafeGuard.cpp
//============================================================

#include “SafeGuard.h”
#include “ArmingControl.h”
using namespace std;

bool SafeGuard::SafeGuardFlag = false;
SafeGuard* SafeGuard::_sfGrd = NULL; // initialization for the pointer

// Constructor
SafeGuard::SafeGuard()
{
	_idg = IdGenerator::getIdGenerator();
	_esad = Esad::getEsad();
} // SafeGuard()
```

```
//Destructor
SafeGuard::~SafeGuard()
{
	SafeGuardFlag = false;
}// ~SafeGuard()

void SafeGuard::setArmingControlReference(ArmingControl* x)
{
	_armCtrl = x;
}//void SafeGuard::setArmingControlReference(ArmingControl* x)


void SafeGuard::receive(Message m)
{
	mesQue.insert(m);
}// void SafeGuard::receive(Message m)

void SafeGuard::setTime(long t)
{
	_timestamp = t;
	bool done;
	done = false;
	while (!done)
	{
		if (mesQue.size() == 0)
			done = true;
		else
		{
			Message temp = mesQue.remove();
			if (temp.getTimestamp() <= _timestamp)
			{
				if (temp.getPayload() == "abort")
					notARM();
				else
					cout << "Unrecognized payload: " << temp << endl;
			}else
				{
					mesQue.insert(temp);
					done = true;
				}
		}
	}
}// void SafeGuard::setTime(long)

void SafeGuard::notARM()
{
	// send message notARM to Esad
	Message* temp = new Message(_idg->getId(), _timestamp, "notARM");
	_esad->receive(*temp);
	Message* temp1 = new Message(_idg->getId(), _timestamp, "abort");
	_armCtrl->receive(*temp1);
```

```
} // void SafeGuard::notARM()

SafeGuard* SafeGuard::getSafeGuard()
{
	// creates the instance of a SafeGuard
	if(!SafeGuardFlag)
	{
		_sfGrd = new SafeGuard();
		SafeGuardFlag = true;
		return _sfGrd;
	}else
	{
		return _sfGrd;
	}
} // SafeGuard* SafeGuard::getSafeGuard()

//=============================================================
//	Author:		Vasileios Karagiannakis
//	Naval Postgraduate School
//	Computer Science Department
//	Date:		08 Aug 2013
//	File Name:	TimeGuard.h
//=============================================================

#ifndef TIMEGUARD_H_
#define TIMEGUARD_H_

#include <string>
#include <iostream>
#include “MessageQueue.h”
#include “RandGen.h”
#include “IdGenerator.h”
#include “SafeGuard.h”
#include “Logger.h”

using namespace std;

class TimeGuard {
public:

	static TimeGuard* getTimeGuard();
	~TimeGuard();

	void receive(Message);
	void setTime(long);
	void callSafeGuard();

	bool timecheck();

	void powerOn();
	void endPost();
	void doLaunch();
```

```
        void startMotion();
        void endFirstMotionDetection();
        void endSafeSeparation();

private:
        TimeGuard();
        static bool TimeGuardFlag;
        static TimeGuard* _tmGrd;

        MessageQueue<50> mesQue;
        IdGenerator* _idg;
        SafeGuard* _sfGrd;
        Logger* _lgr;

        long _timestamp;
        int _state; /* the different values the variable _state could be: 0. off; 1. powerOn; 2. endPost; 3. doLaunch; 4. startMotion; 5. endFirstMotionDetection; 6. Terminate*/

        long _endPostDeadline;
        long _endDoLaunchDeadline;
        long _accelDeadline;
        long _endFirstMotionDeadline;
        long _endSafeSeparationDeadline;
};

#endif /* TIMEGUARD_H_ */

//============================================================
//      Author:         Vasileios Karagiannakis
//      Naval Postgraduate School
//      Computer Science Department
//      Date:           08 Aug 2013
//      File Name:      TimeGuard.cpp
//============================================================
#include “TimeGuard.h”
using namespace std;

bool TimeGuard::TimeGuardFlag = false;
TimeGuard* TimeGuard::_tmGrd = NULL; // initialization for the pointer

// Constructor
TimeGuard::TimeGuard()
{
        _idg = IdGenerator::getIdGenerator();
        _sfGrd = SafeGuard::getSafeGuard();
        _lgr = Logger::getLogger();

        _state = 0;
} // TimeGuard()

//Destructor
TimeGuard::~TimeGuard()
```

```
{
	TimeGuardFlag = false;
}// ~TimeGuard()

void TimeGuard::receive(Message m)
{
	string payload = m.getPayload();
	if (payload == “powerOn” || payload == “endPost” || payload == “invalidPost” ||
payload == “doLaunch” || payload == “startMotion” || payload == “endFirstMotionDetection” ||
payload == “endSafeSeparation”)
		_lgr->logEvent(payload, m.getTimestamp());
	mesQue.insert(m);
 }// void TimeGuard::receive(Message m)

bool TimeGuard::timecheck()
{
	if (_state == 0 )
		return true;
	else if(_state == 1 && _timestamp <= _endPostDeadline)
		return true;
	else if (_state == 2 && _timestamp <= _endDoLaunchDeadline)
		return true;
	else if (_state == 3)
		return true;
	else if (_state == 4 && _timestamp <= _endFirstMotionDeadline)
		return true;
	else if (_state == 5 && _timestamp <= _endSafeSeparationDeadline)
		return true;
	else if (_state == 6)
		return true;
	else
		return false;
}// bool TimeGuard::timecheck()

void TimeGuard::setTime(long t)
{
	_timestamp = t;
	bool done;
	done = false;

	if (!timecheck())
		{
			//send message to SafeGuard
			callSafeGuard();
			_state = 6;
		}
	else
		{
			while (!done)
			{
				if (mesQue.size() == 0)
					done = true;
```

```
                                        else
                                        {
                                                Message temp = mesQue.remove();
// if the time of the message is less than the time of the TimeGuard's then it changes its states
                                                if (temp.getTimestamp() <= _timestamp)
                                                {
                                                        if (temp.getPayload() == "powerOn")
                                                                powerOn();
                                                        else    if      (temp.getPayload()      ==
"endPost")
                                                                endPost();
else if (temp.getPayload() == "invalidPost")
                                                                callSafeGuard();
                                                        else    if      (temp.getPayload()      ==
"doLaunch")
                                                                doLaunch();
                                                        else    if      (temp.getPayload()      ==
"startMotion")
                                                                startMotion();
                                                        else    if      (temp.getPayload()      ==
"endFirstMotionDetection")
                                                                endFirstMotionDetection();
                                                        else    if      (temp.getPayload()      ==
"endSafeSeparation")
                                                                endSafeSeparation();
                                                }else
                                                        {
                                                                mesQue.insert(temp);
                                                                done = true;
                                                        }
                                        }
                                }
                        }
} // void TimeGuard::setTime(long t)

void TimeGuard::powerOn()
{
        if (_state == 0)
        {
                _state = 1; // powerOn
                _endPostDeadline = _timestamp + 2 ; //receives message powerOn from
ActuationChannel
        }
}// void TimeGuard::powerOn()

void TimeGuard::endPost()
{
        if (_state == 1)
        {
                _state = 2; // endPost
                // receives message endPost from ArmingControl
                _endDoLaunchDeadline = _timestamp + 2 ;
```

```
        }
} // void TimeGuard::endPost()

void TimeGuard::doLaunch()
{
        if (_state == 2)
        {
                _state = 3; // doLaunch
        }
} // void TimeGuard::doLaunch()

void TimeGuard::startMotion()
{
        if (_state == 3)
        {
                _state = 4; // start Motion
                _endFirstMotionDeadline = _timestamp + 4 ;
                _endSafeSeparationDeadline = _timestamp + 6 ;
        }
} // void TimeGuard::startMotion()

void TimeGuard::endFirstMotionDetection()
{
        if (_state == 4)
        {
                _state = 5; // endFirstMotionDetection
        }
} // void TimeGuard::endFirstMotionDetection()

void TimeGuard::endSafeSeparation()
{
        if (_state == 5)
        {
                _state = 6; // enter Terminate State
        }
} // void TimeGuard::endSafeSeparation()

void TimeGuard::callSafeGuard()
{
        Message* temp = new Message(_idg->getId(), _timestamp , “abort”);
        _sfGrd->receive(*temp);
}// void TimeGuard::callSafeGuard()

TimeGuard* TimeGuard::getTimeGuard()
{
        // creates the instance of a TimeGuard
        if(!TimeGuardFlag)
        {
                _tmGrd = new TimeGuard();
                TimeGuardFlag = true;
                return _tmGrd;
        }else
```

```
        {
                return _tmGrd;
        }
} // TimeGuard* TimeGuard::getTimeGuard()
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B

We create different test cases that will be implemented as possible environments for the design of the software which controls the arming of the warhead.

A. TABLES FOR THE SIMULATION CASES ANALYSIS

1. Simulation Cases Analysis for the PowerOn Message

0 in signal means **no signal** 1 in signal means **signal** 0 in time means **no delay** 1 in time means delay **over time constrain**					
Cases	powerOn actChnl	time powerOn actChnl	powerOn tmGrd	time powerOn tmGrd	Comments
1	0	0	0	0	No signals / **No Simulation**
2	0	0	0	1	No signals / **No Simulation**
3	0	0	1	0	Not realistic for the TimeGuard / **No Simulation**
4	0	0	1	1	Not realistic for the TimeGuard / **No Simulation**
5	0	1	0	0	No signals / **No Simulation**
6	0	1	0	1	No signals / **No Simulation**
7	0	1	1	0	Not realistic for the Actuation Channel / **No Simulation**
8	0	1	1	1	Not realistic for the Actuation Channel / **No Simulation**
9	1	0	0	0	Not realistic for the whole test without the initialization of TimeGuard **No Simulation**
10	1	0	0	1	Not realistic for the whole test without the initialization of TimeGuard **No Simulation**
11	1	0	1	0	**Ok**, case with no errors
12	1	0	1	1	**Ok**, with timeGuard delay
13	1	1	0	0	Not applicable to powerOn for the Actuation Channel / **No Simulation**

14	1	1	0	1	Not applicable to powerOn for the Actuation Channel **No Simulation**
15	1	1	1	0	Not applicable to powerOn for the Actuation Channel / **No Simulation**
16	1	1	1	1	Not applicable to powerOn for the Actuation Channel / **No Simulation**

Table 1

2. Simulation Cases Analysis for the EndPost Message

0 in signal means **POST_Invalid** 1 in signal means **POST_Valid** 0 in time means **no delay** 1 in time means delay **over time constrain**			
Cases	endPost	time endPost	comments
1	0	0	Invalid endPost, no delay / **Ok**
2	0	1	Invalid endPost, with time delay / **No Simulation** / It can be combined with case 1
3	1	0	Ok, case with no errors
4	1	1	Valid endPost, with timeGuard delay / Ok

Table 2

3. Simulation Cases Analysis for the MakeLaunch Message

0 in signal means **no makeLaunch** 1 in signal means **makeLaunch** 0 in time means **no delay** 1 in time means delay **over time constrain**			
Cases	makeLaunch	time makeLaunch	comments
1	0	0	No makeLaunch , no delay / **Ok**
2	0	1	**Not realistic** for the Actuation Channel /**No Simulation**
3	**1**	**0**	**Ok,** case with no errors
4	1	1	makeLaunch , with time delay / **Ok**

Table 3

4. Simulation Cases Analysis for the DoLaunch Message

0 in signal means **no doLaunch** 1 in signal means **doLaunch** 0 in time means **no delay** 1 in time means delay **over time constrain**			
Cases	doLaunch	time doLaunch	comments
1	0	0	Not realistic for the TimeGuard / **No Simulation**
2	0	1	Not realistic for the TimeGuard / **No Simulation**
3	1	0	Ok, case with no errors
4	1	1	doLaunch , with time delay / **Ok**

Table 4

5. Simulation Cases Analysis for the ReadAcceleration Message

0 in signal means **no readAcceleration (float accel)** 1 in signal means **readAcceleration (float accel)** 0 in time means **no delay** 1 in time means **delay between the acceleration readings over 1 second**			
Cases	readAcceleration (float accel)	time readAcceleration (float accel)	comments
1	0	0	No readAcceleration , no delay / **Ok** (accelerometer failure to provide readings)
2	0	1	Not realistic for the Actuation Channel / **No Simulation**
3	1	0	**Ok**, case with no errors
4	1	1	readAcceleration, with time delay is the same as some cases from Table 7, in which the messages are delayed. Thus, both cases are combined./ **No Simulation**

Table 5

6. Simulation Cases Analysis for the StartMotion Message

0 in signal means **no startMotion** 1 in signal means **startMotion** 0 in time means **no delay** 1 in time means delay **over time constrain**			
Cases	**startMotion**	time **startMotion**	comments
1	0	0	**Ok, (accel = 0.0)**
2	0	1	**Not realistic** for the TimeGuard /**No Simulation**
3	**1**	**0**	**Ok,** case with no errors
4	1	1	**startMotion**, with time delay / **Ok**

Table 6

7. Simulation Cases Analysis for the EndFirstMotionDetection and EndSafeSeparationDistance Messages

0 in signal means invalid values(not 2 consecutive 6 g' && not travel distance over 20 m) 1 in signal means valid values 0 in time means no delay 1 in time means delay over time constrain from actuation Channel to Time guard					
Cases	endFMD	time endFMD	endSSD	time endSSD	comments
1	0	0	0	0	**Ok,** (no proper accelerations)-
2	0	0	0	1	Not realistic / **No Simulation**
3	0	0	1	0	**Ok,** invalid FMD values but valid for SSD
4	0	0	1	1	**Ok,** same accel values as case 3
5	0	1	0	0	(similar to case 2)/ **No Simulation**
6	0	1	0	1	**Ok,** same values as case 1
7	0	1	1	0	**Ok,** same accel values as case 3
8	0	1	1	1	**Ok,** same accel values as case 3
9	1	0	0	0	**Ok -**
10	1	0	0	1	Not realistic / **No Simulation**
11	1	0	1	0	**Ok ,** case with no errors
12	1	0	1	1	**Ok,** same values as case 11
13	1	1	0	0	**Ok,** same accel values as case 9
14	1	1	0	1	**No Simulation/** Can be

					combined as case 9
15	1	1	1	0	**Ok,** same accel values as case11
16	1	1	1	1	**Ok**

Table 7

8. Acceleration Values for the Different Scenarios

case	A1	A2	A3	A4	A5	A6
1	**0.5**	**1.5**	**1.5**	**2.0**	**2.5**	**3.0**
3	**0.5**	**3.5**	**4.5**	**5.5**	**5.5**	**6.0**
9	**0.5**	**1.5**	**6.0**	**6.0**	**1.5**	**1.5**
11	**0.5**	**3.5**	**6.0**	**6.0**	**7.5**	**8.0**

Table 8

9. Final Table about the number of the Simulation Test Cases

Case studies	**Active simulation cases (with combination more than one failures)**	**Thesis simulation cases (At least one to be fault)**
powerOn	2	1case OK + 1 case wrong
endPost	3	1case OK + 2 cases wrong
makeLaunch	3	1case OK + 2 cases wrong
doLaunch	2	1case OK +1 case wrong
startMotion	3	1case OK +2 cases wrong
readAcceleration	2(1+1) mutual exclusive with the endFMD_endSSD	1case OK +1 cases wrong
endFMD_endSSD	12	1case OK +11 cases wrong
Total cases	**1404**	**21**

Table 9

B. AGGREGATE LOG FILE TABLE WITH CALCULATION REMARKS

Use Case	Name of log file	Remarks
1.	logfile _no_errors	Everything operates as supposed to do
2.	logfile _powerOn_tmGrd_timedelay	Time delay to TimeGuard to receive the powerOn message in main()
3.	logfile _valid_endPost_timedelay	Time delay to Actuation Channel to send the endPost message to TimeGuard

4.	logfile _ invalid_endPost	Change the message that the ActuationChannel sends to the ArmingControl
5.	logfile _ no_makeLaunch	No makeLaunch in main()
6.	logfile _ makeLaunch_timedelay	Time delay in main() to send the makeLaunch in ActuationChannel
7.	logfile _ doLaunch_timedelay	Time delay to Actuation Channel to send the doLaunch message to TimeGuard
8.	logfile _no_readAcceleration	No readAcceleration in main()
9.	logfile _ no_startMotion	All readAcceleration equal to 0.0 in main()
10.	logfile _ startMotion_timedelay	Time delay to Actuation Channel to send the startMotion message to TimeGuard
11.	logfile _noFMD_noSSD	readAcceleration in main() like case 1
12.	logfile _noFMD_SSD	readAcceleration in main() like case 3
13.	logfile _noFMD_SSD_timedelay	readAcceleration in main() like case 3 and time delay to actuation channel to send the message to TimeGuard
14.	logfile _ noFMD_timedelay_noSSD_timedelay	readAcceleration in main() like case 1 and time delay to actuation channel to send the messages to TimeGuard
15.	logfile _ noFMD_timedelay_SSD	readAcceleration in main() like case 3 and time delay to actuation channel to send the message to TimeGuard
16.	logfile _ noFMD_timedelay_SSD_timedelay	readAcceleration in main() like case 3 and time delay to actuation channel to send the message to TimeGuard
17.	logfile _FMD_noSSD	readAcceleration in main() like case 9
18.	logfile _FMD_SSD_timedelay	readAcceleration in main() like case 11 and time delay to actuation channel to send the message to TimeGuard
19.	logfile _ FMD_timedelay_noSSD	readAcceleration in main() like case 9 and time delay to actuation channel to send the message to TimeGuard

20.	logfile _ FMD_timedelay_SSD	readAcceleration in main() like case 11 and time delay to actuation channel to send the message to TimeGuard
21.	logfile _ FMD_timedelay_SSD_timedelay	readAcceleration in main() like case 11 and time delay to actuation channel to send the messages to TimeGuard

1. Log files and their Timing Diagrams

- logfile _no_errors

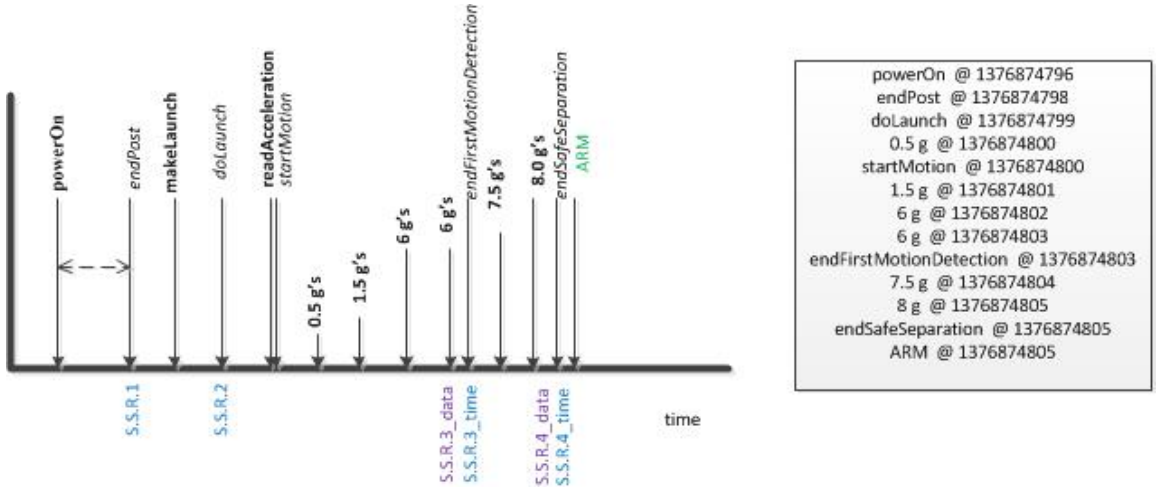


Figure 1

- logfile _powerOn_tmGrd_timedelay

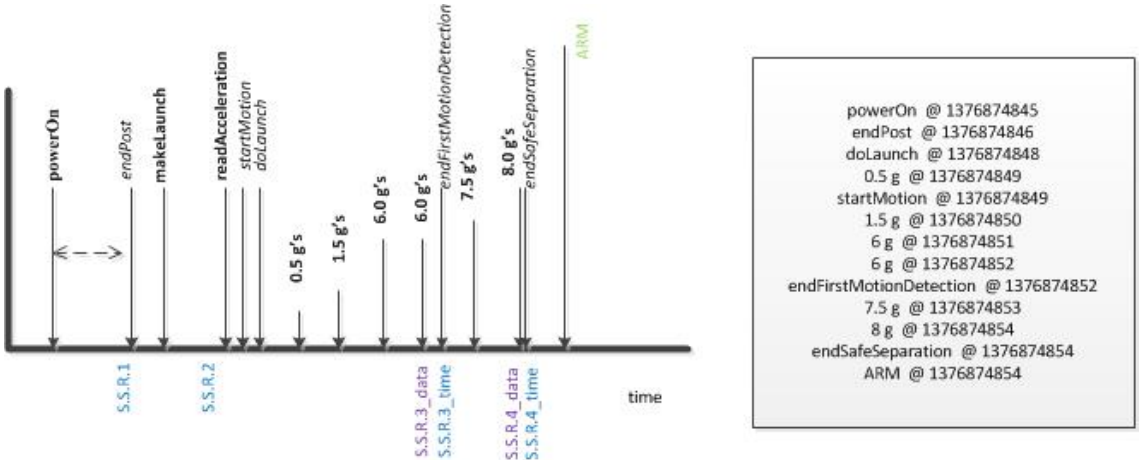
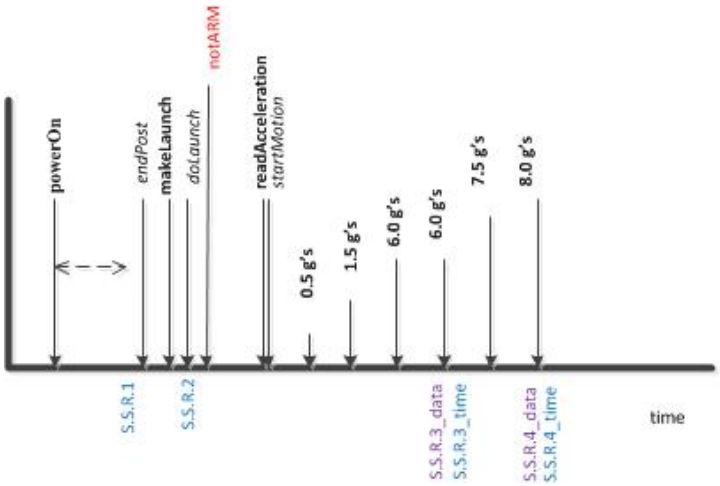


Figure 2

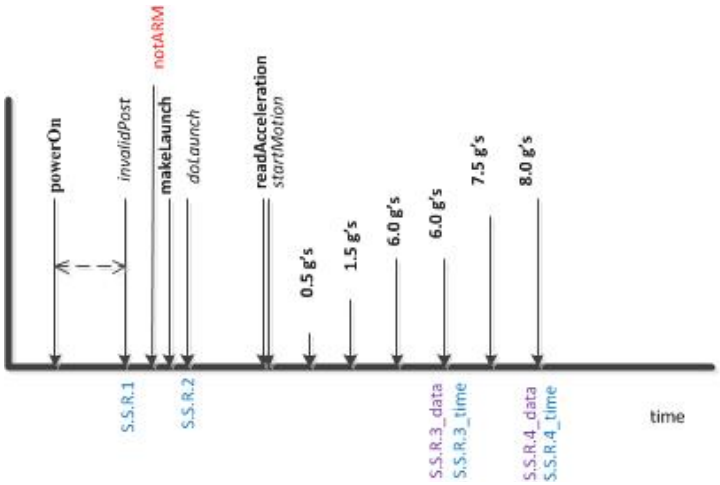
- logfile _valid_endPost_timedelay



powerOn @ 1376874908
endPost @ 1376874911
doLaunch @ 1376874911
notARM @ 1376874911
0.5 g @ 1376874912
startMotion @ 1376874912
1.5 g @ 1376874913
6 g @ 1376874914
6 g @ 1376874915
7.5 g @ 1376874916
8 g @ 1376874917

Figure 3

- logfile _ invalidPost



powerOn @ 1377849981
invalidPost @ 1377849983
notARM @ 1377849983
notARM @ 1377849984
0.5 g @ 1377849985
startMotion @ 1377849985
1.5 g @ 1377849986
6 g @ 1377849987
6 g @ 1377849988
7.5 g @ 1377849989
8 g @ 1377849990

Figure 4

- logfile _ no_makeLaunch

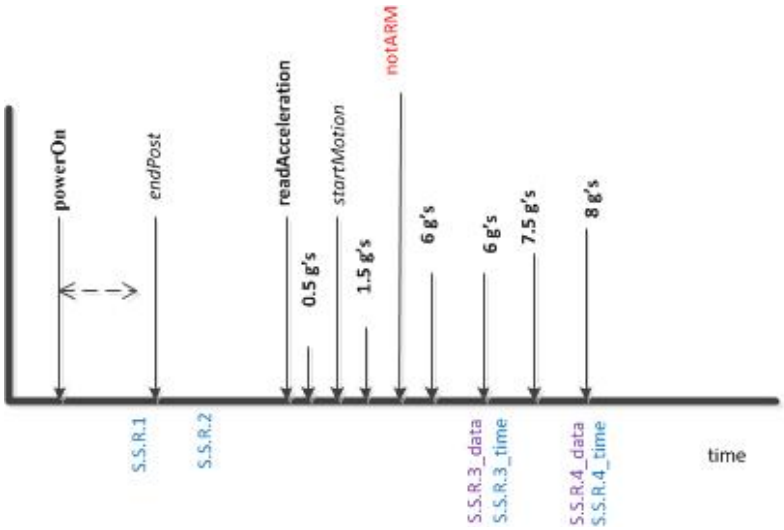


Figure 5

- logfile _ makeLaunch_timedelay

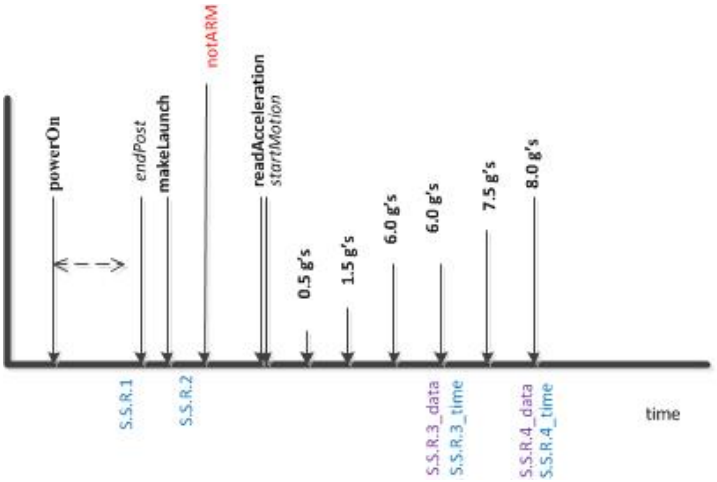
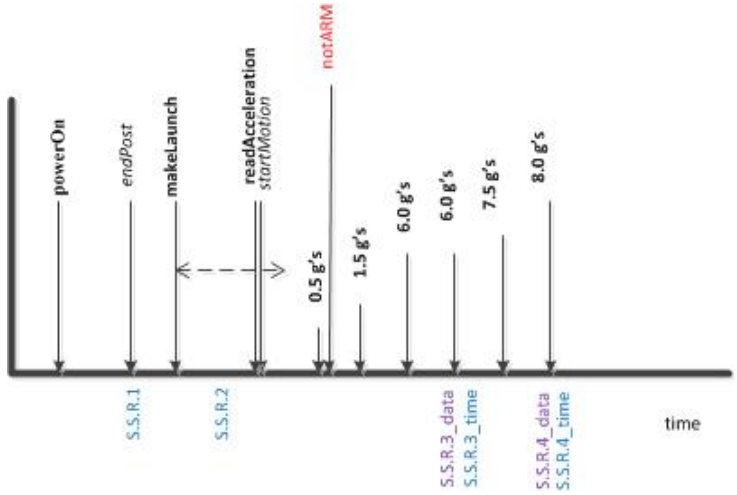


Figure 6

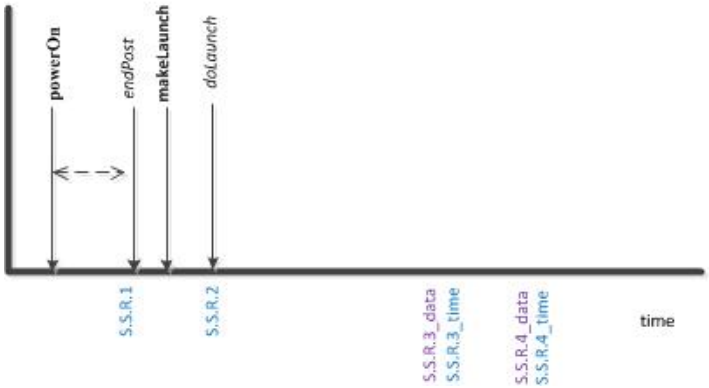
- logfile _ doLaunch_timedelay



powerOn @ 1376875098
endPost @ 1376875100
0.5 g @ 1376875102
startMotion @ 1376875102
1.5 g @ 1376875103
notARM @ 1376875103
6 g @ 1376875104
6 g @ 1376875105
7.5 g @ 1376875106
8 g @ 1376875107

Figure 7

- logfile _no_readAcceleration



powerOn @ 1376875204
endPost @ 1376875206
doLaunch @ 1376875207

** remains in unarmed state

Figure 8

- logfile _ no_startMotion

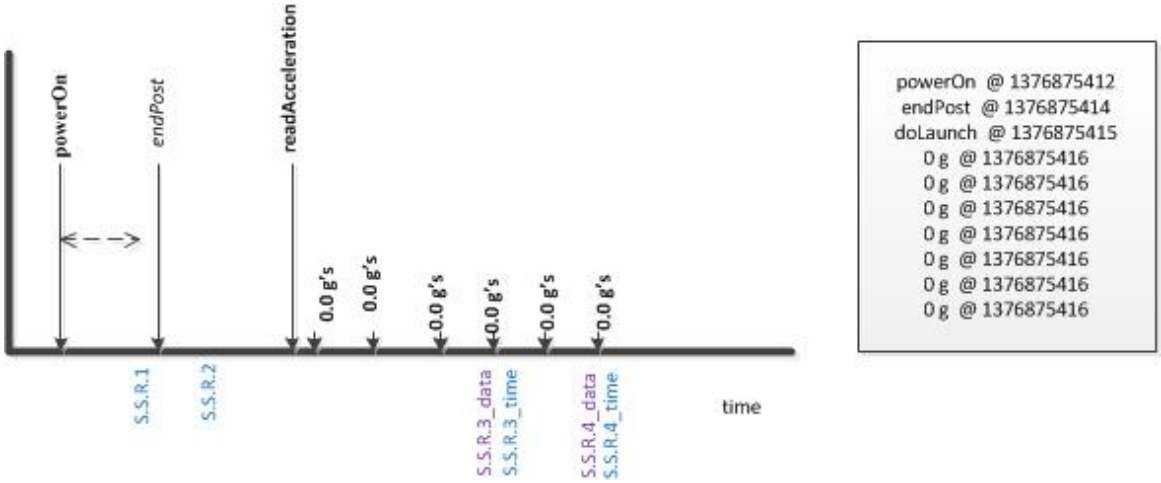


Figure 9

- logfile _ startMotion_timedelay

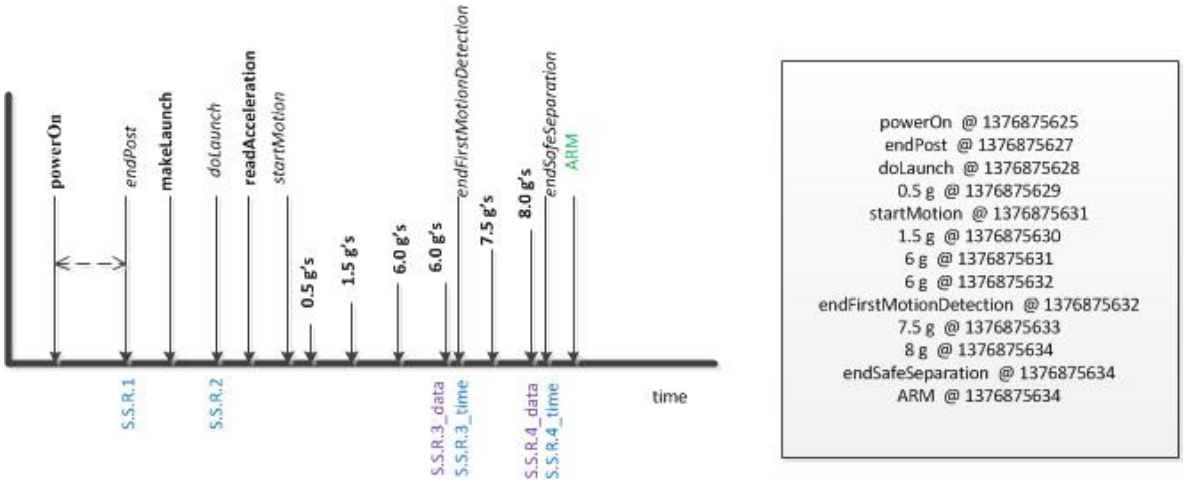
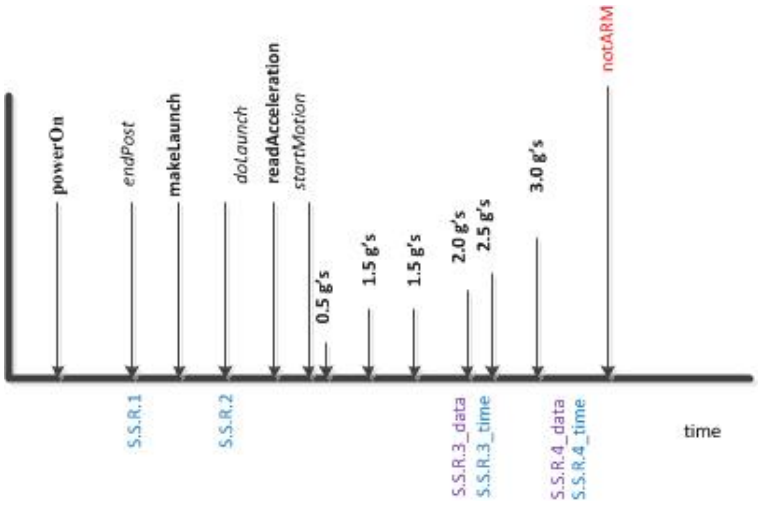


Figure 10

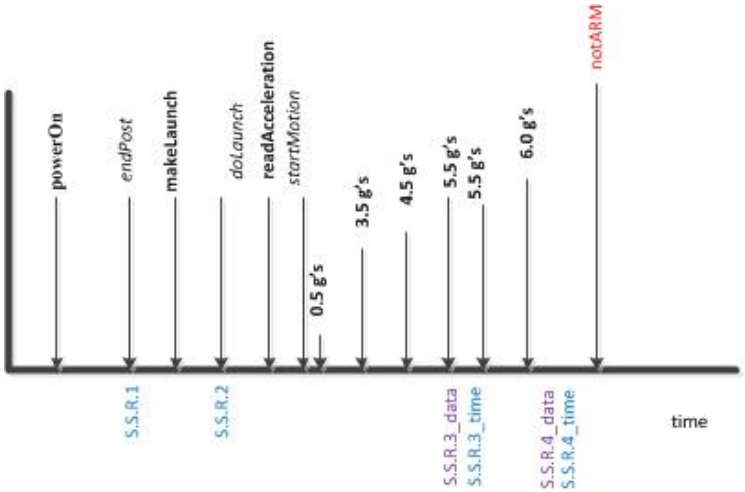
- logfile _noFMD_noSSD



powerOn @ 1376875750
endPost @ 1376875752
doLaunch @ 1376875753
0.5 g @ 1376875754
startMotion @ 1376875754
1.5 g @ 1376875755
1.5 g @ 1376875756
2 g @ 1376875757
2.5 g @ 1376875758
3 g @ 1376875759
notARM @ 1376875759

Figure 11

- logfile _noFMD_SSD



powerOn @ 1376875821
endPost @ 1376875823
doLaunch @ 1376875824
0.5 g @ 1376875825
startMotion @ 1376875825
3.5 g @ 1376875826
4.5 g @ 1376875827
5.5 g @ 1376875828
5.5 g @ 1376875829
6 g @ 1376875830
notARM @ 1376875830

Figure 12

- logfile _noFMD_SSD_timedelay

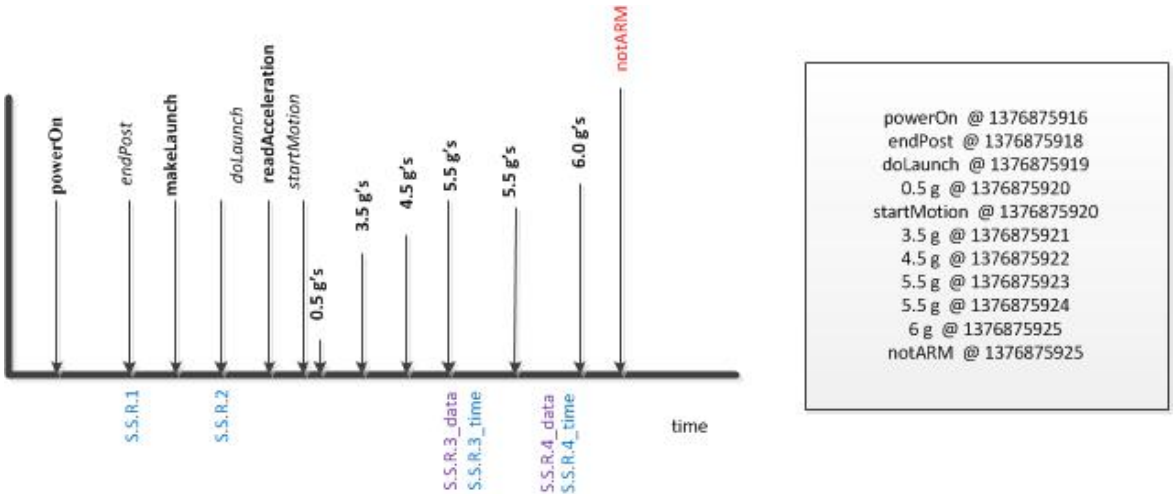


Figure 13

- logfile _ noFMD_timedelay_noSSD_timedelay

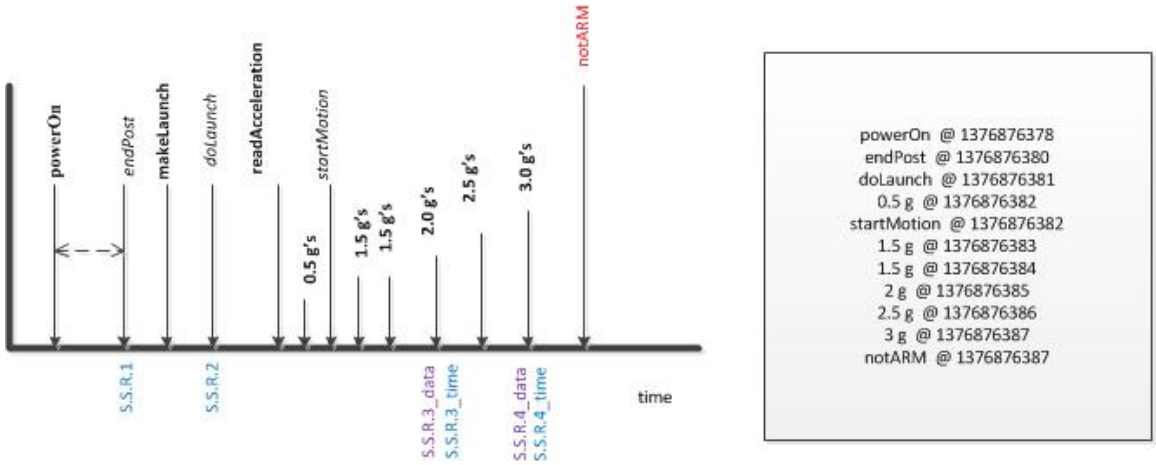


Figure 14

- logfile _ noFMD_timedelay_SSD

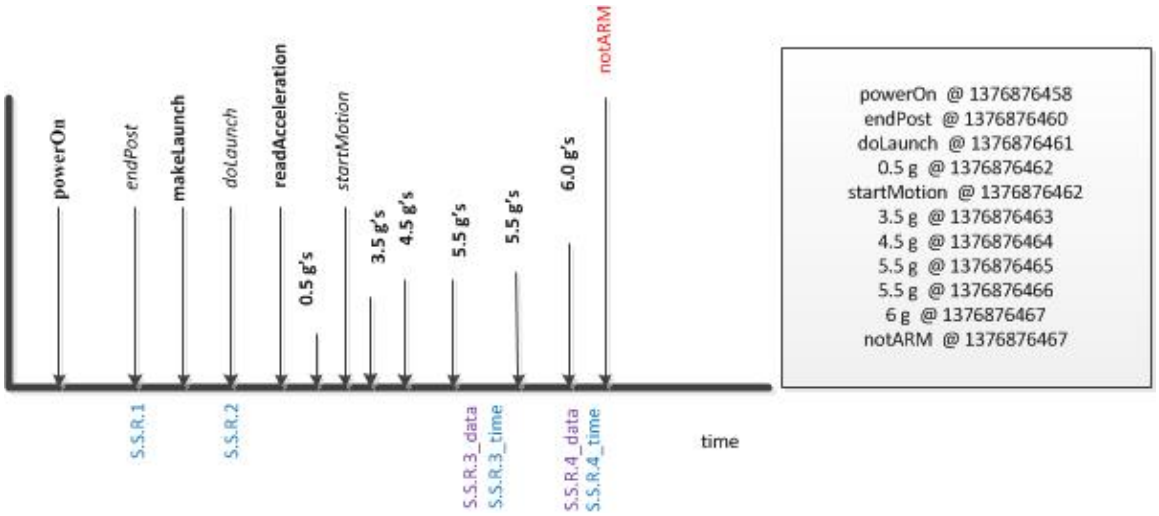


Figure 15

- logfile _ noFMD_timedelay_SSD_timedelay

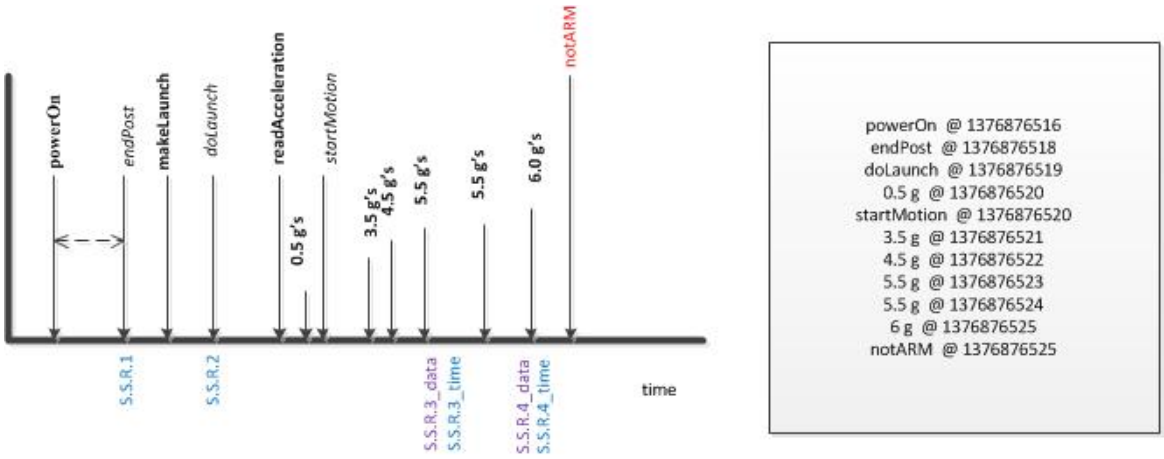


Figure 16

- logfile _FMD_noSSD

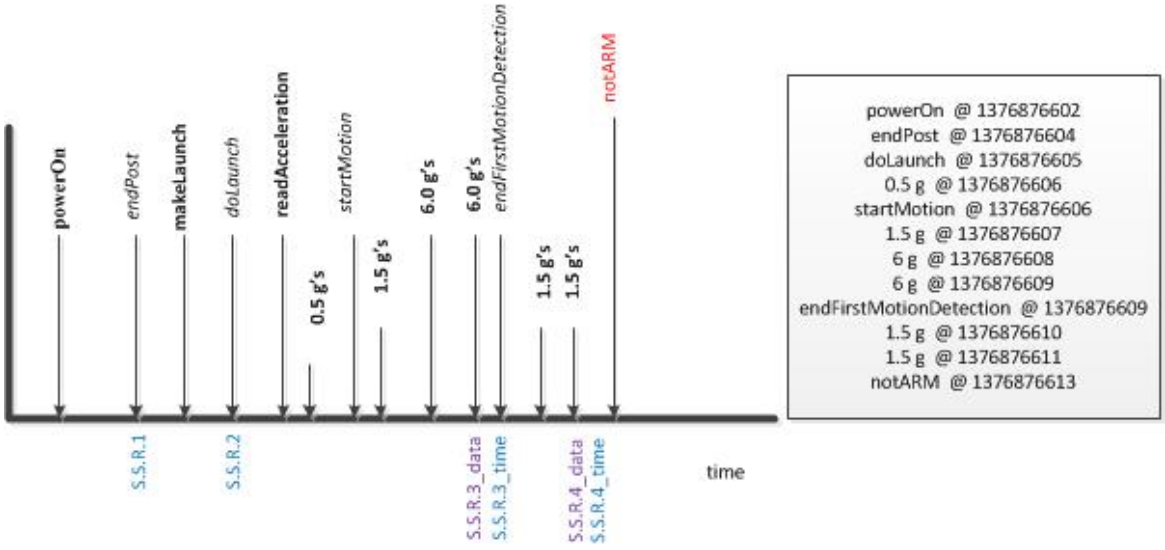


Figure 17

- logfile _FMD_SSD_timedelay

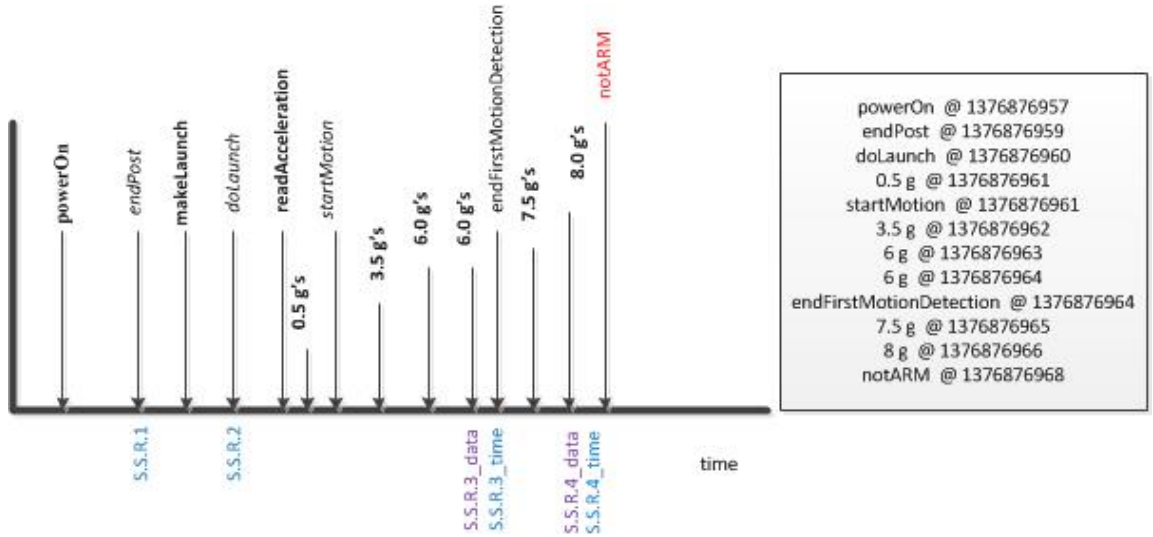


Figure 18

- logfile _ FMD_timedelay_noSSD

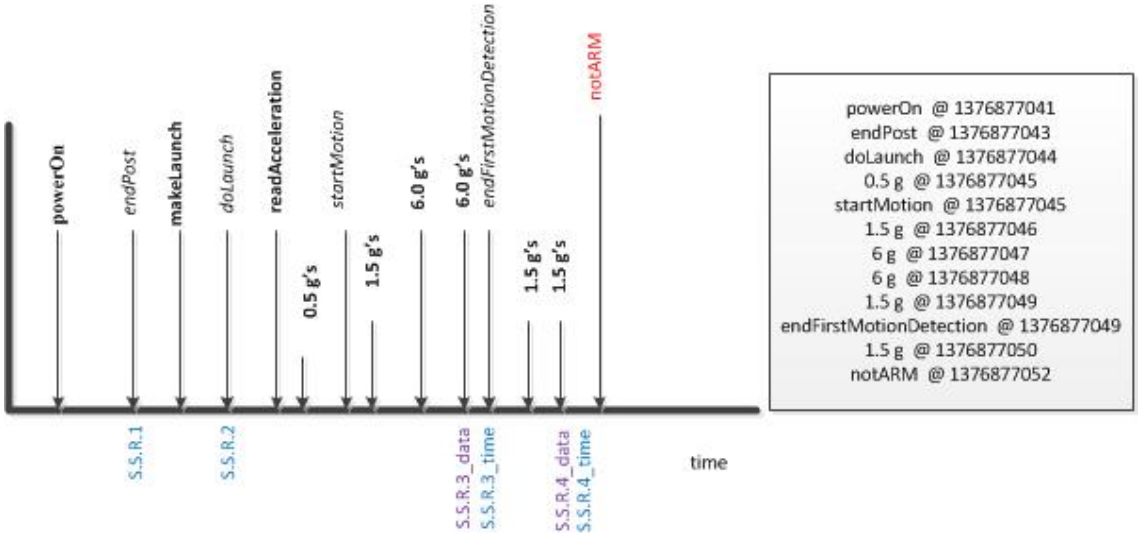


Figure 19

- logfile _ FMD_timedelay_SSD

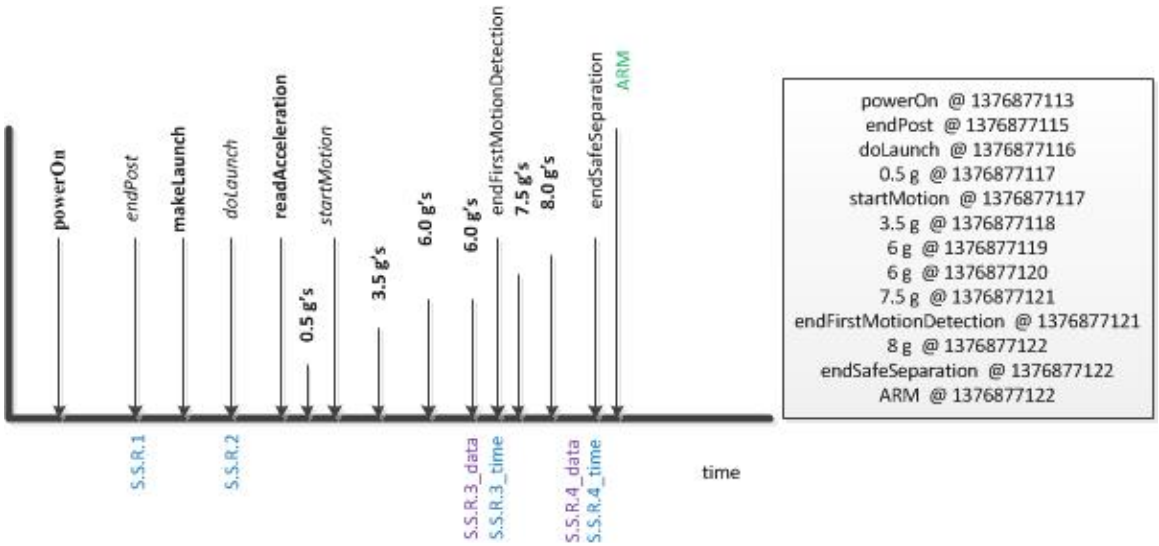


Figure 20

- logfile _ FMD_timedelay_SSD_timedelay

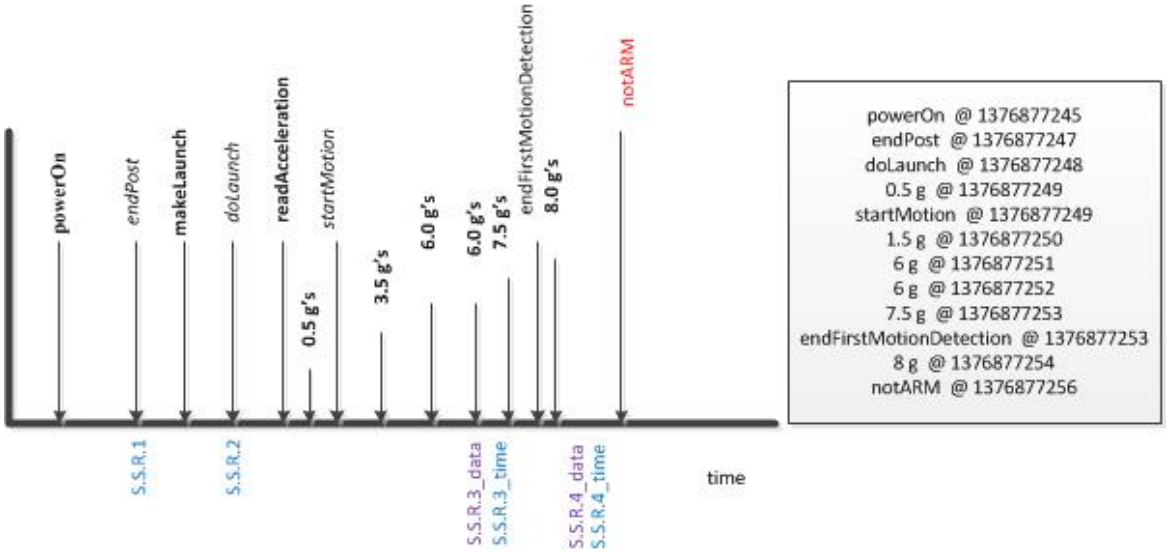


Figure 21

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C

JUnit test cases for the Statecharts assertions' Validation

A. SOFTWARE SAFETY REQUIREMENT 1: POST

1. Test Case 1: Everything is Correct

```
package r1_validation;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class r1_test1 {
	private r1.R1 assertion;

	@Before
	public void setUp() throws Exception {
		assertion = new r1.R1();

	}

	@After
	public void tearDown() throws Exception {
	assertion = null;
	}

	@Test
	public void test() {
		assertion.power_On();
		assertion.incrTime(1);
		assertion.post_Valid();
		assertion.incrTime(2);
		assertTrue(assertion.isSuccess());
	}
}
```

2. Test Case 2: The Self-Test is Failed

```
package r1_validation;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class r1_test2 {
	private r1.R1 assertion;

	@Before
```

```
	public void setUp() throws Exception {
		assertion = new r1.R1();
	}

	@After
	public void tearDown() throws Exception {
		assertion = null;}

	@Test
	public void test() {
		assertion.power_On();
		assertion.incrTime(1);
		assertion.postInvalid();
		assertion.incrTime(1);
		assertTrue(assertion.isSuccess());
		assertion.arm();
		assertFalse(assertion.isSuccess());
	}
}
```

3. Test Case 3: The Self-Test is Passed but the Timer expires

```
package r1_validation;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class r1_test3 {
	private r1.R1 assertion;

	@Before
	public void setUp() throws Exception {
		assertion = new r1.R1();
	}

	@After
	public void tearDown() throws Exception {
		assertion = null;}

	@Test
	public void test() {
		assertion.power_On();
		assertion.incrTime(4);
		assertion.post_Valid();
		assertion.incrTime(1);
		assertion.arm();
		assertFalse(assertion.isSuccess());
	}
}
```

B. SOFTWARE SAFETY REQUIREMENT 2: LAUNCH INDICATE

1. Test Case 1: Everything is Correct

```
package r2_validation;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class r2_test1 {

	private r2.R2 assertion;

	@Before
	public void setUp() throws Exception {
		assertion = new r2.R2();

	}

	@After
	public void tearDown() throws Exception {
	assertion = null;
	}

	@Test
	public void test() {
		assertion.powerOn();
		assertion.incrTime(3);
		assertion.doLaunch();
		assertTrue(assertion.isSuccess());
	}
}
```

2. Test Case 2: The Timer expires, before the DoLaunch Signal is received

```
package r2_validation;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class r2_test2 {

	private r2.R2 assertion;

	@Before
	public void setUp() throws Exception {
		assertion = new r2.R2();
```

```
	}

	@After
	public void tearDown() throws Exception {
	assertion = null;
	}

	@Test
	public void test() {
		assertion.powerOn();
		assertion.incrTime(5);
		assertion.doLaunch();
		assertion.arm();
		assertFalse(assertion.isSuccess());
	}
}
```

3. Test Case 3: There is not DoLaunch Signal

```
package r2_validation;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class r2_test3 {

	private r2.R2 assertion;

	@Before
	public void setUp() throws Exception {
		assertion = new r2.R2();

	}

	@After
	public void tearDown() throws Exception {
	assertion = null;
	}

	@Test
	public void test() {
		assertion.powerOn();
		assertion.incrTime(5);
		assertion.arm();
		assertFalse(assertion.isSuccess());
	}
}
```

C. SOFTWARE SAFETY REQUIREMENT 3: FIRST MOTION DETECTION

1. Test Case 1: Everything is Correct

```
package r3_validation;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class r3_test1 {
	private r3.R3 assertion;

	@Before
	public void setUp() throws Exception {
		assertion = new r3.R3();
	}

	@After
	public void tearDown() throws Exception {
		assertion = null;
	}

	@Test
	public void test() {
		assertion.startMotion();
		assertion.incrTime(1);
		assertion.readAcceleration((float)0.5);
		assertion.incrTime(1);
		assertion.readAcceleration((float)3.5);
		assertion.incrTime(1);
		assertion.readAcceleration((float)6.0);
		assertion.incrTime(1);
		assertion.readAcceleration((float)6.5);
		assertTrue(assertion.isSuccess());
		}
}
```

2. Test Case 2: There is no Acceleration Value over 6 g's

```
package r3_validation;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class r3_test2 {
	private r3.R3 assertion;

	@Before
	public void setUp() throws Exception {
		assertion = new r3.R3();
```

```
	}

	@After
	public void tearDown() throws Exception {
		assertion = null;
	}

	@Test
	public void test() {
		assertion.startMotion();
		assertion.incrTime(1);
		assertion.readAcceleration((float)0.5);
		assertion.incrTime(1);
		assertion.readAcceleration((float)2.0);
		assertion.incrTime(1);
		assertion.readAcceleration((float)3.5);
		assertion.incrTime(1);
		assertion.readAcceleration((float)4.5);
		assertion.incrTime(1);
		assertion.arm();
		assertFalse(assertion.isSuccess());
		}
}
```

3. Test Case 3: Only one Acceleration Value is over 6 g's before the Timer expires

```
package r3_validation;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class r3_test3 {
	private r3.R3 assertion;

	@Before
	public void setUp() throws Exception {
		assertion = new r3.R3();
	}

	@After
	public void tearDown() throws Exception {
		assertion = null;
	}

	@Test
	public void test() {
		assertion.startMotion();
		assertion.incrTime(1);
		assertion.readAcceleration((float)1.5);
		assertion.incrTime(1);
```

```
		assertion.readAcceleration((float)6.0);
		assertion.incrTime(1);
		assertion.readAcceleration((float)3.5);
		assertion.incrTime(1);
		assertion.readAcceleration((float)4.5);
		assertion.incrTime(1);
		assertion.arm();
		assertFalse(assertion.isSuccess());
	}
}
```

4. Test Case 4: The two Signals EndFirstMotionDetection and EndSafeSeparation are Received at the Same Time from the TimeGuard

The acceleration contain the proper values to fulfill the SSR but there are time delays and (violates the time requirement for the SSR3 because the endFirstMotionDetection has to be within 4 sec after the startMotion event)

```
package r3_validation;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class r3_test4 {
	private r3.R3 assertion;

	@Before
	public void setUp() throws Exception {
		assertion = new r3.R3();
	}

	@After
	public void tearDown() throws Exception {
		assertion = null;
	}

	@Test
	public void test() {
		assertion.startMotion();
		assertion.incrTime(1);
		assertion.readAcceleration((float)0.5);
		assertion.incrTime(1);
		assertion.readAcceleration((float)1.0);
		assertion.incrTime(1);
		assertion.readAcceleration((float)6.0);
		assertion.incrTime(1);
		assertion.readAcceleration((float)6.0);
		assertion.incrTime(1);
```

```
                assertion.readAcceleration((float)7.5);
                assertion.incrTime(1);
                assertion.readAcceleration((float)8.0);
                assertion.incrTime(1);
                assertion.arm();
                assertFalse(assertion.isSuccess());
        }
}
```

5. Test Case 5: The Acceleration Contain the Proper values but there are Time Delays and the Timer expires

```
package r3_validation;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class r3_test5 {
        private r3.R3 assertion;

        @Before
        public void setUp() throws Exception {
                assertion = new r3.R3();
        }

        @After
        public void tearDown() throws Exception {
                assertion = null;
        }

        @Test
        public void test() {
                assertion.startMotion();
                assertion.incrTime(1);
                assertion.readAcceleration((float)0.5);
                assertion.incrTime(2);
                assertion.readAcceleration((float)1.0);
                assertion.incrTime(1);
                assertion.readAcceleration((float)6.0);
                assertion.incrTime(2);
                assertion.readAcceleration((float)6.0);
                assertion.incrTime(1);
                assertion.arm();
                assertFalse(assertion.isSuccess());
        }
}
```

D. SOFTWARE SAFETY REQUIREMENT 4: SAFE SEPARATION

1. Test Case 1: Everything is Correct

```
package r4_validation;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class r4_test1 {
	private r4.R4 assertion;

	@Before
	public void setUp() throws Exception {
		assertion = new r4.R4();
	}

	@After
	public void tearDown() throws Exception {
		assertion = null;
	}

	@Test
	public void test() {
		assertion.startMotion();
		assertion.incrTime(1);
		assertion.readAcceleration((float)0.5);
		assertion.incrTime(1);
		assertion.readAcceleration((float)1.5);
		assertion.incrTime(1);
		assertion.readAcceleration((float)6.0);
		assertion.incrTime(1);
		assertion.readAcceleration((float)6.5);
		assertion.incrTime(1);
		assertion.readAcceleration((float)7.5);
		assertion.incrTime(1);
		assertion.readAcceleration((float)8.5);
		assertion.incrTime(1);
		assertTrue(assertion.isSuccess());
		}
}
```

2. Test Case 2: The Values are Correct but There Are Time Delays and the Timer Expires

```
package r4_validation;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
```

```
public class r4_test2 {
	private r4.R4 assertion;

	@Before
	public void setUp() throws Exception {
		assertion = new r4.R4();
	}

	@After
	public void tearDown() throws Exception {
		assertion = null;
	}

	@Test
	public void test() {
		assertion.startMotion();
		assertion.incrTime(1);
		assertion.readAcceleration((float)0.5);
		assertion.incrTime(3);
		assertion.readAcceleration((float)1.5);
		assertion.incrTime(2);
		assertion.readAcceleration((float)2.5);
		assertion.incrTime(2);
		assertion.readAcceleration((float)3.5);
		assertion.incrTime(1);
		assertion.arm();
		assertFalse(assertion.isSuccess());
		}
}
```

3. Test Case 3: The Calculated Distance Does Not Reach the Minimum Value of 20 Meters Due To Acceleration Values

```
package r4_validation;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class r4_test3 {
	private r4.R4 assertion;

	@Before
	public void setUp() throws Exception {
		assertion = new r4.R4();
	}

	@After
	public void tearDown() throws Exception {
		assertion = null;
	}
```

```
	@Test
	public void test() {
		assertion.startMotion();
		assertion.incrTime(1);
		assertion.readAcceleration((float)0.5);
		assertion.incrTime(1);
		assertion.readAcceleration((float)1.5);
		assertion.incrTime(1);
		assertion.readAcceleration((float)6.0);
		assertion.incrTime(1);
		assertion.readAcceleration((float)6.0);
		assertion.incrTime(1);
		assertion.readAcceleration((float)0.5);
		assertion.incrTime(1);
		assertion.readAcceleration((float)0.5);
		assertion.incrTime(1);
		assertion.arm();
		assertFalse(assertion.isSuccess());
		}
}
```

4. Test Case 4: The Two Signals EndFirstMotionDetection and EndSafeSeparation are Received at the Same Time from the TimeGuard

```
package r4_validation;
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class r4_test4 {
	private r4.R4 assertion;

	@Before
	public void setUp() throws Exception {
		assertion = new r4.R4();
	}

	@After
	public void tearDown() throws Exception {
		assertion = null;
	}

	@Test
	public void test() {
		assertion.startMotion();
		assertion.incrTime(1);
		assertion.readAcceleration((float)0.5);
		assertion.incrTime(1);
		assertion.readAcceleration((float)1.0);
```

```
                assertion.incrTime(1);
                assertion.readAcceleration((float)6.0);
                assertion.incrTime(1);
                assertion.readAcceleration((float)6.0);
                assertion.incrTime(1);
                assertion.readAcceleration((float)7.5);
                assertion.incrTime(1);
                assertion.readAcceleration((float)8.0);
                assertion.incrTime(1);
                assertion.arm();
                assertFalse(assertion.isSuccess());
                }
}
```

LIST OF REFERENCES

[1] H. Petroski, *To Engineer is Human: The Role of Failure in Successful Design*, New York: Vintage Books, 1992.

[2] N. Storey, *Safety Critical Computer Systems*, Boston: Addison-Wesley Longman Publishing Co., 1996.

[3] N.G. Leveson, *Safeware, System Safety and Computers*, Boston: Addison-Wesley Longman Publishing Co., 1995.

[4] R.N. Taylor, N. Medvidovic, E.M. Dashofy, *Software Architecture, Foundations, Theory and Practice*, Hoboken, NJ: John Wiley & Sons, 2009.

[5] *Military Standard 882E Standard Practice For System Safety*, MIL-STD- 882E, 2012.

[6] W. Wu and T. Kelly, "Safety tactics for software architecture design," *in Proc.28th Annu. Int. Computer Software Conf.*, 2004, pp.368–375.

[7] W. Wu and T. Kelly, "Failure modeling in software architecture design for safety," *in Proc. Workshop on Architecting Dependable Systems*, 2005, New York, NY, pp. 1–7.

[8] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, A.L. Wolf, "An architecture-based approach to self-adaptive software," *IEEE Intell. Syst., vol.14*, 1999, pp. 54–62.

[9] J. Pumfrey, P. Fenelon, J.A. McDermid, M. Nicholson, "Towards integrated safety analysis and design," *ACM Computing Reviews, vol. 2, no. 1*, 1994, pp. 21–32.

[10] "List of Software Bugs." [Online]. Available: http://en.wikipedia.org/wiki/List_of_software_bugs

[11] M. Clarke and J. M. Wing, "Formal methods: State of the art and future directions," *ACM Computing Surveys, vol. 28, no. 4*, 1996.

[12] *National Aeronautics and Space Administration, NASA Software Safety Guidebook*, NASA-GB-8719.13, 2004.

[13] Weapons and Systems Engineering Department, Fundamentals of Naval Weapons Systems, United States Naval Academy, url: http://www.fas.org/man/dod-101/navy/docs/fun/index.html.

[14] *Department of Defense, Design Criteria Standard 1316E, Safety Criteria for Fuse Design*, MIL-STD-1316E, 1998.

[15] Prof. Michael's slides from the course SW4582: Weapon Systems Software Safety, Module 2, Part 3, Slide 10, Jan 2010.

[16] M. W. Maier and E. Rechtin, *The Art of Systems Architecting 3rd ed.*, Danvers, MA: CRC Press Taylor & Francis Group, 2009.

[17] B.P. Douglass, *Doing Hard Time: Developing Real-time Systems with UML, Objects, Framework and Pattern*. New York: Addison-Wesley Longman Publishing, 1999.

[18] A. Armoush, F. Salewski and S. Kowalewski, "Design pattern representation for safety-critical embedded systems," *Journal of Software Engineering and Applications,* vol. 2, no. 1, 2009, pp. 1–12.

[19] E.J. Braude, *Software Design from Programming to Architecture,* Hoboken, NJ: John Wiley & Sons, 2004.

[20] C. Bonine, M. Shing, T.W. Otani, "Computer-aided process and tools for mobile software acquisition," NPS, Monterey, CA, Tech. Rep. NPS-SE-13-C10P07R05–075, 2013.

[21] C. Bonine, "Specification, validation and verification of mobile application behavior," M.S. thesis, Dept. Comp. Science, NPS, Monterey, CA, 2013.

[22] D. Harel, "Statecharts: A visual approach to complex systems," *Science of Computer Programming,* vol. 8, no. 3, 1987, pp. 231–274.

[23] M. Shing and D. Drusinsky, "Architectural design, behavior modeling and run-yime verification of network embedded systems," in *Proc. of the 12th Conference on Reliable Systems on Unreliable Networked Platforms,* Monterey, 2005, pp. 281–303.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California