NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

SOFTWARE FAULT TREE ANALYSIS
OF AN AUTOMATED CONTROL SYSTEM DEVICE
WRITTEN IN ADA

by

Mathias W. Winter

September 1995

Thesis Advisor: Timothy J. Shimeall

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE September 1995	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE
SOFTWARE FAULT TREE ANALYSIS OF AN AUTOMATED CONTROL SYSTEM DEVICE WRITTEN IN ADA

5. FUNDING NUMBERS

6. AUTHOR(S)
Winter, Mathias W.

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)

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 United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Software Fault Tree Analysis (SFTA) is a technique used to analyze software for faults that could lead to hazardous conditions in systems which contain software components. Previous thesis works have developed three Ada-based, semi-automated software analysis tools, the Automated Code Translation Tool (ACTT) an Ada statement template generator, the Fault Tree Editor (FTE) a graphical fault tree editor, and the Fault Isolator (FI) an automated software fault tree isolator. These previous works did not apply their tools on a real system. Therefore, the question addressed by this thesis is "Do these tools actually work on a real-world software control system?"

This thesis developed and implemented a sample Software System Analysis Methodology (SSAM) using these semi-automated software tools. The research applied this methodology to a real-world distributed control system written in Ada. The Missile Engagement Simulation Arena's (MESA) control software was developed by the Naval Air Warfare Center, Weapons Division, China Lake, CA.

The SSAM was used to show that the analysis of the Sphere-HWCI control module's 74,000 lines of code could be thoroughly analyzed in less than 100 man-hours. This practical, 740 lines-of-code per hour rate was a direct result of the incorporation of the semi-automated tools into the process.

14. SUBJECT TERMS
Software Safety, Software Fault Tree Analysis, Software Safety Methods

15. NUMBER OF PAGES
111

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

SOFTWARE FAULT TREE ANALYSIS OF AN AUTOMATED CONTROL SYSTEM DEVICE WRITTEN IN ADA

Mathias William Winter
Lieutenant Commander, United States Navy
B.S., University of Notre Dame, 1984

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

September 1995

Author: Mathias W. Winter
Mathias W. Winter

Approved By: Timothy J Shimeall
Timothy J. Shimeall, Thesis Advisor

LtCol David A. Gaitros, Second Reader

Mark H. Barber
Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

Software Fault Tree Analysis (SFTA) is a technique used to analyze software for faults that could lead to hazardous conditions in systems which contain software components. Previous thesis works have developed three Ada-based, semi-automated software analysis tools, the Automated Code Translation Tool (ACTT) an Ada statement template generator, the Fault Tree Editor (FTE) a graphical fault tree editor, and the Fault Isolator (FI) an automated software fault tree isolator. These previous works did not apply their tools on a real system. Therefore, the question addressed by this thesis is "Do these tools actually work on a real-world software control system?"

This thesis developed and implemented a sample Software System Analysis Methodology (SSAM) using these semi-automated software tools. The research applied this methodology to a real-world distributed control system written in Ada. The Missile Engagement Simulation Arena's (MESA) control software was developed by the Naval Air Warfare Center, Weapons Division, China Lake, CA.

The SSAM was used to show that the analysis of the Sphere-HWCI control module's 74,000 lines of code could be thoroughly analyzed in less than 100 man-hours. This practical, 740 lines-of-code per hour rate was a direct result of the incorporation of the semi-automated tools into the process.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First, I would like to thank Dr. Timothy Shimeall, my thesis advisor, for providing me with an excellent environment in which to conduct my research. His calm demeanor and academic acumen made him an excellent technical advisor and working mentor. His genuine interest and sincere appreciation for my work allowed this past year to be an enjoyable and worthwhile experience for me.

Second, I would like to thank LtCol David Gaitros for being gracious enough to be my second reader. Our talks on subjects ranging from software to flying provided me with much needed insight and motivation to accomplish my work.

Next, I would like to extend my deep appreciation and gratitude to the MESA project personnel at China Lake, CA. Specifically, I want to thank Mr. Bob Westbrook for his liaison help that directed me towards the MESA project, Mr. Ken Wetzel for his astute observations and invaluable insights on the analysis data and Mr. Tom Roseman for his software engineering expertise and guidance throughout the entire period of my work.

Last, but by no means least, I want to thank my wife, Joanne, and daughter, Angela for their steadfast support and unwavering dedication during this long "at-sea" shore-duty assignment. Without their love and devotion, none of this would have been possible.

I. INTRODUCTION

Computers are part of everyday life. From the complex systems found in the Space Shuttle to the Fisher Price Alphabet computer two-year-old children play with in the home. This wide spectrum of computer usage is evidence that we have become a computer-technology-dependent society. As this dependency has increased, more and more safety critical systems have become automated. This automation, relying heavily upon software control systems, increases productivity and efficiency but also greatly increases the possibility of catastrophic consequences in the event of system failures. Safety critical systems are those that manage processes that can directly impact human lives and/or expensive equipment and property. System failures directly relate into serious and usually unacceptable human and property losses. The degree of these losses depend greatly on the type of system involved. To reduce these losses, it is imperative that control systems conform to a standardized evaluation to ensure its reliability, dependability and safety. The development of useful methods to detect, isolate and eliminate the causes of these high risk failures is crucial in demonstrating the reliability and dependability of current and future safety critical software control systems.

Hardware failures and defects are well understood and documented. Through the history of the industrial age, the science of hardware failure analysis and its concepts, causes and effects have been exhaustively evaluated and quantified. The same cannot be said about the concepts, causes and effects of software induced failures. The need for standardized software safety analysis principles and techniques is undeniable. This thesis examines the requirement for and development of a practical and effective software safety analysis methodology for safety-critical software systems.

A. SOFTWARE SAFETY

Safety has been defined as "freedom from those conditions that can cause death, injury, occupational illness, or damage to or loss of equipment or property [Ref. 1]." A definition for software safety can then be given as, “freedom from software-caused death, injury, damage to or loss of equipment or property [Ref. 2].” Mishap, hazard, accident and risk are related concepts to safety that must be defined to understand the requirement for software safety. A mishap is an unplanned event or series of events that result in death, injury, occupational illness, damage to or loss of equipment or property, or environmental harm. Hazard and accident are usually used interchangeably. A hazard refers to the state or states of a system that when combined with certain environmental conditions could lead to a mishap, where accident is defined as an unwanted and unexpected release of energy. Risk is defined as a function of the probability of a hazardous state occurring, the probability of the hazard leading to a mishap, and the perceived severity of the worst potential mishap that could result from the hazard [Ref. 1]. With these stated definitions, it is no surprise that software safety has become a major concern in today’s safety critical control systems. Project managers cannot afford to take the risk of minimizing the importance of software safety in these critical systems. Though acceptable levels of software safety have been achieved, at current technology levels, no system can be guaranteed free from defects. With this inevitable human-induced limitation, software safety then involves ensuring that the system will execute within a given context without resulting in unacceptable risk. Methodologies have been developed to reduce risk to an acceptable level while increasing system safety. This is achieved by identifying potential hazards early in the development process and then establishing requirements and design features to eliminate or control these hazards [Ref. 1].

When discussing safety, the concept of reliability emerges. Though related, these two concepts are not the same. Reliability is defined as the probability that a system performs its assigned function under specified environmental conditions for a given period of time. Extended to software, the definition becomes the probability that a software system fulfils its assigned task in a given environment for a predefined number of input cases, assuming the input cases are free of errors [Ref. 3]. Therefore, it can be said that reliability requirements are concerned with making a system failure-free, where safety requirements are concerned with making it mishap-free [Ref. 1].

B. SOFTWARE DEFECTS

The American Heritage Dictionary defines defect as "an imperfection, failing or fault." As related to software, a standard definition of software defects can then be those faults, errors or failures contained in a software system [Ref. 4]. Understanding the nature and interrelationships of these defects is essential when conducting any type of software safety analysis.

1. Faults

Faults are those defects in a component or design which ultimately are responsible for a failure. Some causes include design errors, electromagnetic interference, unanticipated inputs and system misuse [Ref. 5]. Faults exhibit different classifiable properties such as duration, nature and extent. The duration of a fault can be transient, intermittent or permanent. The nature of a fault is determined by its behavior in the system. It can either be logical, producing logical values, or indeterminate, having no logical equivalent. The extent of faults determines the level of the fault, either local or global. Faults originate in the system's environment or from the interaction between the system and a user (process). Faults usually have one of several effects [Ref. 4]:

- Disappear with no perceptible effect
- Remain in place with no perceptible effect
- Lead to a sequence of additional faults that result in a failure in the system's operation (propagation to failure)
- Lead to a sequence of additional faults with no perceptible effect on the system (undetected propagation)
- Lead to a sequence of additional faults that have a perceptible effect on the system but do not result in a failure of the system's operation (detected propagation without failure)

2. Errors

The term error is sometimes interchanged with fault, however, there is a distinct difference. In the context of system states, an error is considered to be part of an erroneous state that constitutes a difference from a valid state. When a system is in an invalid state, an external state analysis of the system can determine the states that would need to be changed to make the internal state of the system valid. This internal state can be valid within itself but incompatible with its surrounding environment. This can occur when a design failure is introduced in a component or through the interaction of a component in a valid state with one in an invalid state. An undesirable effect is the propagation of errors through the system changing valid components to erroneous ones. This property tends to portray errors in a transitive nature, linking the presence of faults with the failure of the system [Ref. 6].

3. Failures

Every system is designed according to some specification. This specification defines the required operations and functions that the system is to perform. When the behavior of the system first deviates from this required specification, a failure has occurred [Ref. 6]. This definition of failure assumes the given specification is free from "errors" which could eventually lead to a failure. This assumption is necessary to be able to derive a coherent, functional definition of a failure. The specification does not, however, provide any insight to what behavior can be expected in the event of failures. This can be a problem when trying to design software that detects and copes with failures. Classifying failure behavior uses a modeling method that qualifies the disruptive nature of the failures. This is useful when not all failures are of equal consequence. These classifications of failures are as follows:

- Fail-Safe - Procedures that attempt to limit the amount of damage caused by a failure. No attempt is made to satisfy the functional specifications except where safety is concerned.
- Fail-Operational - Provides for full system functionality in the presence and migration of faults.
- Fail Soft - Provides continued system operation but at a degraded performance or

reduced functionality level until the fault is removed or the run-time conditions change [Ref. 1].

Using these classes of failures, it is possible to augment existing reliability techniques that attempt to eliminate all failures with techniques that concentrate on the high-cost failures. This effort can then ensure that these particular failures do not occur or at least their probability is minimized. Figure 1 summarizes the interrelationship between faults, errors and failures.

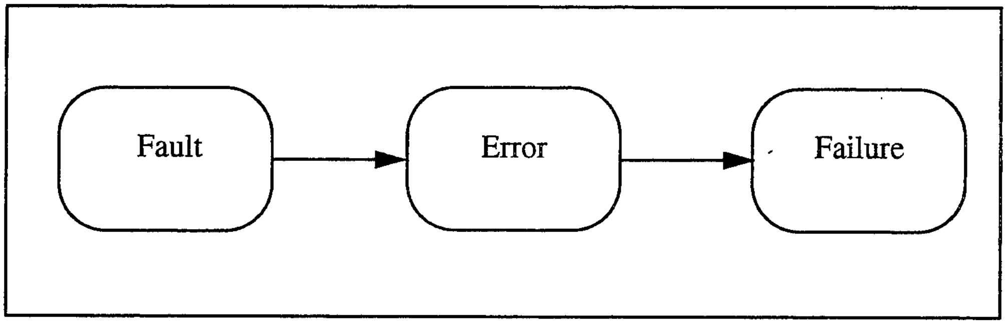


Figure 1: Software Defect Relationships

4. Software Defect Examples

It is a well-known fact that software by itself cannot directly injury, maim or cause destruction. However, it is the interaction of this software with hardware that *can* produce catastrophic events [Ref. 1]. Computers are increasingly used to monitor and control safety critical systems. Real-time software controls aircraft, shuts down nuclear power reactors in emergencies, keeps telephone networks running, and monitors hospital patients. The use of computers in such systems offers considerable benefits, but also poses serious risks to life and the environment. The following list of defective software-induced accidents and hazards is presented to expose some of these risks and to demonstrate the critical justification for complete and comprehensive software safety analysis methodologies.

- Therac-25 Radiation Therapy Machine. A man was exposed to fatal radiation level treatments due to a software modification of the control software, resulting in one human death and the manufacturer going out of business [Ref. 7].

- A French meteorological satellite computer was supposed to issue a read instruction to some high-altitude weather balloons but instead ordered an "emergency self-destruct," resulting in 72 of 141 weather balloons destroyed [Ref. 1].
- An air-to-air missile loaded on the wing of an F/A-18 jet fighter failed to separate from the launcher because a computer program signaled the missile retaining mechanism to close before the rocket had built up sufficient thrust to clear the missile from the wing. The aircraft went violently out of control resulting in loss of the aircraft [Ref. 1].

C. SOFTWARE ANALYSIS TECHNIQUES AND TOOLS

Software safety analysis and verification is required by contractors of safety critical systems. At least three Department of Defense standards include related tasks; one general safety standard [MIL-STD-882C 1993] includes tasks for software hazard analysis and verification of software safety; an Air Force standard for missile weapon systems [MIL-STD-1574A 1979] requires a complete and integrated software safety analysis; and the U.S Navy has a standard for nuclear weapons systems [MIL-STD-SNS 1986] that requires software nuclear safety analysis [Ref. 1]. Currently, a major restructuring effort of all military standards is underway. The outcome of this is unsure. However, it is almost certain that the number of military standards will decrease, consolidating numerous current safety standards.

Various software safety analysis techniques have been developed to aid in this required verification. A few of these techniques have been tested and used extensively, while others are still being developed. From these techniques have come standardized methods and procedures to achieve the required results. This procedural characteristic has led to the automation of some of these techniques.

The explosion of safety-critical software systems has put an enormous burden upon software engineers and analysts to produce "failure free" systems. This requirement alone increases the complexity and work load required to produce such systems. Manual methods, even with large software teams, can no longer provide the required effort subject to ever decreasing time and budget constraints. Today's software engineers and analysts must use automated tools to help in the process. Early automated tools were very specific in their use and were met with skepticism due to their unreliable nature and poorly proven track records. As more time, effort and resources were dedicated to the development of these tools, the concept of Computer Aided Software Engineering (CASE) tools was born. Continuing efforts at prestigious software engineering institutions like the Software Engineering Institute (SEI) at Carnegie Mellon University have made great strides in developing standard methods for software safety analysis. Automation of the entire software development life cycle from requirements development through code generation and testing is an ongoing project. It is imperative to develop correct requirement and design specifications in order to eliminate the incorporation of faults and errors in to the given software system. This is especially important in safety-critical software modules where any defect could cause loss of life and/or material. The development of correctly-implemented automated requirement and design generation methods will allow the development of virtually fault free systems. However, this is far from being a reality. Current technology in dealing with natural language requirements precludes the development of a correct automated tool. Automated tools have been proven in the code and testing generation cycles and are used extensively today. The following manual and automated analysis techniques are introduced as background for use later in developing a useful analysis methodology [Ref. 8].

1. Hazard Identification and Analysis

Hazard analysis involves identifying and assessing the criticality level of hazards and risks involved in the system design. Hazard analysis is an ongoing evolution throughout the development life cycle of the system. The different stages of hazard analysis consist of preliminary (PHA), subsystem (SSHA), system (SHA) and operating/support (OSHA). These analysis are crucial in detecting and identifying safety critical hazards. Several techniques have been developed to perform these analyses such as Failure Modes and Effects Analysis (FMEA), Fault Tree Analysis (FTA), Petri Net Modelling, Statecharts, Event Tree Analysis (ETA), design reviews/walk-throughs, checklists and other hazard/operability analysis methods. Some of these methods are described below.

a. Fault Tree Analysis

This thesis concentrates on the Software Fault Tree Analysis (SFTA) methodology extensively. Chapter II is dedicated to FTA and SFTA; their purpose, concepts and structure. In this section, numerous automated tools developed at the Naval Postgraduate School (NPGS) under the direction of Doctor Timothy J. Shimeall will be introduced. These tools will be the core of the proposed software safety analysis methodology presented later.

ACTT

The *Automated Code Translation Tool* was developed by Captain Robert Ordonio, USA, and extended by LCDR William Reid, USN, using resources at the Naval Postgraduate School. ACTT translates Ada statements into template structures to be used in SFTA. The tool consists of four components. First, a lexical analyzer, which determines if the input consists of valid tokens. Next is a parser generator, which checks that the input uses valid Ada constructs. Next is a template generator, which transforms valid statements into templates representing possible events associated with the statement in a format suitable for SFTA. The last component is a file generator that creates a file that meets the specifications of a fault tree editor (FTE) file type [Ref. 9]. ACTT takes an Ada source code file as input and processes each Ada statement into its associated fault tree template, connecting them accordingly. The output is written as an FTE specified file for further analysis.

FTE

In general, fault tree editors are used to graphically display and modify fault trees. These editors allow software analysts to interactively manipulate the graphic representation of a fault tree by using an automated graphic editor. This usage significantly reduces the time required to draw and redraw the trees during analysis.

The *Fault Tree Editor (FTE)* used in this thesis was written by Charles P. Lombardo, Computer Systems Programmer for the Computer Science Department at NPGS. The code was written in the "C" programming language using XView, an OPEN LOOK tool kit, for the X11 Windowing System. This editor loads a user-defined fault tree file and graphically displays it using standard fault tree symbology. The input file must conform to the specifications of FTE. The output from ACTT meets this specification and allows FTE to graphically present its results. The FTE display can then be modified, printed and/or saved as a new file [Ref. 9].

FI

When dealing with software fault trees, tremendously large numbers of nodes can be generated from relatively small source code programs. To aid the analyst in managing these enormous tree sizes, a fault isolator tool, *Fault Isolator (FI)*, was designed and built. FI was written by Lieutenant Commander Russ Mason, USN, and incorporated an efficient graphical user interface using the Transportable Application Environment tool (TAE). FI processes existing FTE compatible files allowing the analyst to "prune" the original tree. This pruning process is accomplished by searching for the associated tree node that corresponds to a source code line of interest. FI searches the tree and returns results in three categories related to the source code line number, exact match, contains match and closest match. This allows the analyst to determine which node/sub-tree is of interest and which can be pruned away. FI lets the analyst save the new tree which can then be displayed in FTE. This pruning process decreases the analysts work load by eliminating tedious manual tree searches [Ref. 10].

b. Failure Modes and Effects Analysis

FMEA is an inductive technique that attempts to anticipate potential failures so that the source of those failures can be eliminated. FMEA consists of constructing a table based on the components of the system and the possible failure modes of each component. Though the exact implementation of the table can vary, the normal table consists of the following columns, component, failure mode, effect of failure, cause of failure, occurrence, severity, probability of detection, risk priority number and corrective action. A list of possible failure modes is generated for each component and inserted into the table. The remaining columns for each failure mode are then filled in using validated estimates and best guess judgements. This is strictly a manual analysis of the system that attempts to anticipate potential failures [Ref. 8].

c. Petri Nets

Petri Nets are a simple, elegant model for concurrent program analysis. The Petri Net model is a 5-tuple structured as (P,T,I,O,M). P is a finite set of places drawn as circles representing conditions. T is a finite set of transitions drawn as bars representing events. I and O are sets of input and output functions which map transitions to places and places to transitions, respectively. M is the set of initial markings (states) for the modeled net.

Places may contain zero or more tokens drawn as black circles. A marking (or state) of the Petri Nets is the distribution of tokens at a moment in time. Tokens in Petri Nets model dynamic behavior of systems. Markings change during execution of the Petri Nets as the tokens "travel" through the net as in modelling the flow of information. The execution of the Petri Nets is controlled by the number and distribution of the tokens. A transition is enabled if each of its input places contains at least as many tokens as there exists arcs from that place to the transition. When a transition is enabled it may fire. When a transition fires, all enabling tokens are removed from its input places, and a token is deposited in each of its output places.

Safety properties of Petri Nets can be analyzed without the need to necessarily generate the entire reachability graph. The idea is to work backwards from high-risk states to determine if these hazardous states are reachable, similar to FTA. This backward method uses the inverse Petri net (reversed input and output functions), and is practical only when a small number of unsafe states is considered. The idea is to work backwards from unsafe states to all critical states (i.e. states having at least two successors). When a critical state is reached, interlocks can be used to force the system to take those paths that do not lead to unsafe states [Ref. 11].

d. Statecharts Analysis

In Statecharts, a normal state transition diagram is enhanced with hierarchical and compositional features. States can then be clustered into super-states with the possibility of "zooming in" and "zooming out" of states. In an AND decomposition, states are split into concurrent subcomponents that communicate via broadcasting. An OR decomposition decomposes a state into sub-states such that control resides in exactly one sub-state. When coupled with a standard graphics package, statecharts enable viewing the description at different levels of detail. Statecharts can be used either as a stand-alone behavioral description or as part of a more general design methodology that deals with the system's other aspects, such as functional decomposition and data-flow specification [Ref. 12].

e. Others

Nuclear Safety Cross Check Analysis (NSCCA). This methodology was developed to satisfy the USAF requirements for nuclear systems. This process has two main components one technical and one procedural. The technical evaluates the software by multiple analyses and test procedures to ensure that it satisfies the systems nuclear safety requirements. The procedural implements security and control measures to protect against sabotage, collusion, compromise, or alteration of critical software components, tools, and NSCCA results. The goal of the NSCCA method is to attempt to show, with a high degree of confidence, that the software will not contribute to a nuclear mishap [Ref. 1].

Software Common Mode Analysis (SCMA). This technique is derived from its hardware predecessor. In hardware common mode analysis, redundant, independent hardware components are used to provide fault tolerance. Research has shown that there is a potential for a single hardware failure to affect more than one redundant component through a software path [Ref. 13]. Software common mode analysis uses structured walk-throughs to examine the potential for a single failure to propagate across hardware boundaries via a software path [Ref. 1].

Sneak Software Analysis. Another technique derived from its hardware counterpart. Here, the software is translated into circuit diagrams and analyzed to detect areas of unreachable code or unreferenced variables. This technique does not provide any great insight to software safety rather it provides more of a software reliability check and a poor one at that [Ref. 1].

D. THESIS PROJECT APPLICATION

1. MESA Overview

The Naval Air Warfare Center Weapons Division (NAWCWPNS), China Lake, California is developing a Missile Engagement Simulation Arena (MESA) in support of continued real-time weapons systems testing. MESA (Figure 2) is a military construction project that replaces its predecessor, the Encounter Simulation Laboratory (ESL) located in Corona, California.

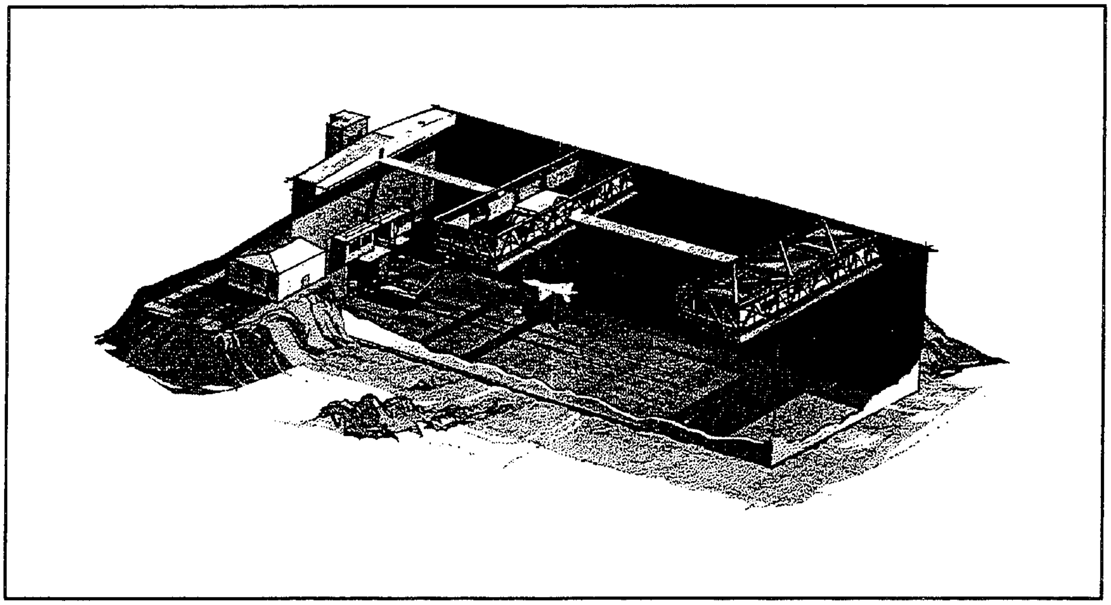


Figure 2: Missile Engagement Simulation Arena (MESA)

MESA is an indoor research, development, test and evaluation facility with the capability to simulate the engagement of various missile fuzes with airborne targets. It provides an arena for the study and analysis of the electromagnetic interactions of the missile fuze sensors with targets during simulated engagements [Ref. 14].

2. MESA System Overview

The MESA system structure is comprised of different hardware configuration items (HWCI) and computer software configuration items (CSCI) (Figure 3).

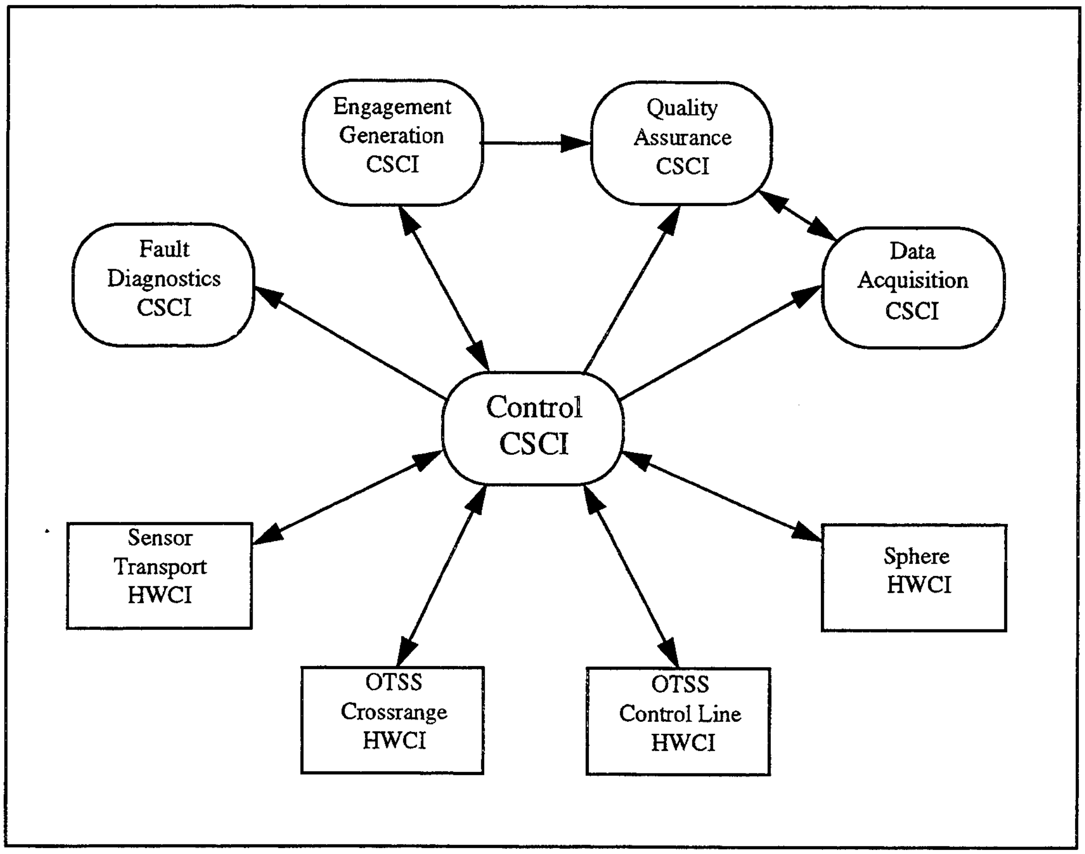


Figure 3: MESA System Structure

The main HWCI's consist of a Sensor Transport System (STS), two Overhead Target Support Systems (OTSS) and two calibration Sphere Systems. Each HWCI system is controlled by a remote computer with associated control software. The CSCI's consist of a

Control module, Fault Diagnostic module, Engagement Generation module, Data Acquisition module and a Quality Assurance module. The Control CSCI acts as the host coordinator for the MESA system. Each remote computer is capable of running in an open-loop and closed-loop mode, depending upon the current system state. The Control CSCI maintains communication with all the remote computers, sending and receiving data as required to accomplish the given test cases [Ref. 15].

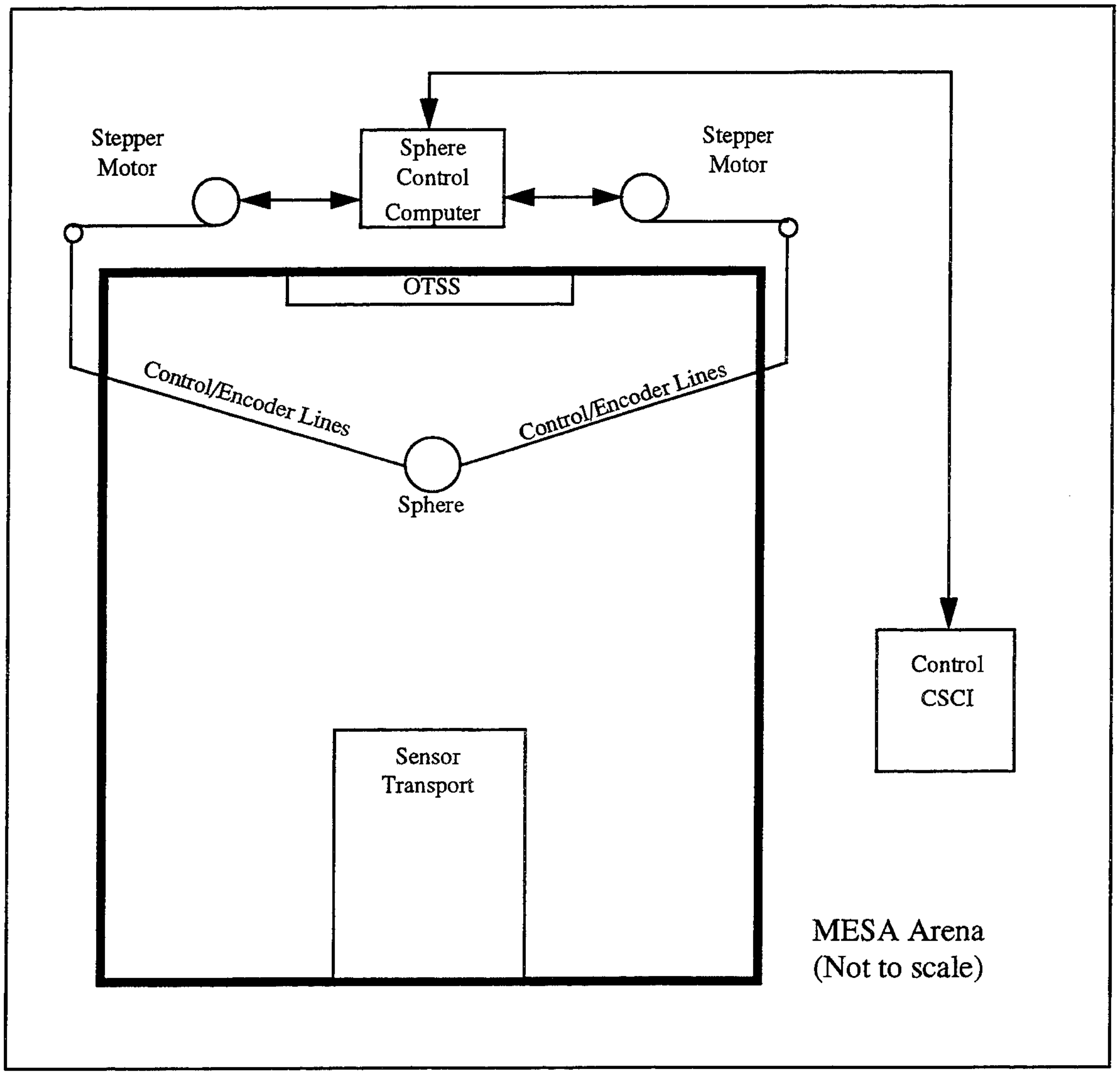


Figure 4: Calibration Sphere Hardware Diagram

E. PROBLEM STATEMENT

Due to availability of MESA software, this thesis will only analyze the sphere control system module. MESA has two calibration spheres that are suspended overhead the arena. Each sphere has two control lines and two encoder lines. The control lines position the spheres in a vertical plane perpendicular to the data-collection direction. Movement up range and down range is not under software control and requires the hoists to be moved manually. The encoder lines are used to obtain stretch-independent measurements of the control lines' length. Each sphere has a sphere computer that is part of the distributed control system. Each sphere computer controls its associated control and encoder lines. Figure 4 depicts the physical layout of one of the calibration sphere systems. The sphere is suspended by control and encoder lines connected to individual stepper motors located within the structure of the arena. The stepper motors are controlled by the remote sphere computer which operates in both an open loop mode for sphere speed control and a closed loop mode for sphere position control. The operator moves the sphere into its required position through the use of the control module interface [Ref. 15].

This thesis addresses the questions, can larger scale software control systems be efficiently and effectively analyzed using new and existing automated and semi-automated software safety analysis methodologies? Specifically, can the automated tools ACTT, FTE and FI be used in combination with the standard software fault tree analysis technique to provide accurate and meaningful software safety analysis data on a real world, currently developing project? These questions will be answered by analyzing the MESA calibration sphere subsystem using the automated tools and methodologies developed at the Naval Postgraduate School.

F. SUMMARY OF CHAPTERS

1. II. Fault Tree Analysis Process

This chapter outlines the fundamentals of fault tree analysis and its extension into the software arena with software fault tree analysis.

2. III. Software System Analysis Methodology

This chapter outlines a sample software safety analysis methodology consisting of a combination of standard manual techniques and locally developed automated techniques.

3. IV. Methodology Implementation Results

This chapter describes the implementation and presents the results of the sample software safety analysis methodology on the MESA Sphere-HWCI control software.

4. V. Conclusions

This chapter presents author derived conclusions, recommendations and desired future work areas of research.

II. FAULT TREE ANALYSIS

Fault tree analysis was developed at Bell Telephone Laboratories in 1962 by H.R. Watson. It was initially designed to be used for safety and reliability studies of the Minuteman missile system. Engineers at Boeing further developed and refined the procedures and became the method's foremost proponents as a method for performing safety analysis of complex electromechanical systems [Ref. 2]. A fault tree consists of fault events, branches and tree gates. Events are failure situations resulting from the logical interaction of primary failures or those failures of interest. Branches connect two events or a tree gate and an event. Gates are boolean logic symbols that relate the input to its output. A system is represented by a series of these components making a fault tree.

Fault tree analysis starts with defining a particular undesirable event and then provides an approach for analyzing the causes of this event. It is important to choose this event carefully. If it is too general, the fault tree becomes large and unmanageable, likewise, if the event is too specific then the analysis may not provide a sufficiently broad enough view of the system. Fault tree analysis can be extremely time consuming and expensive. Therefore some method of choosing a set of desired hazardous events must be implemented. This can be accomplished through the preliminary hazards analysis previously discussed. Each top-level hazard event is then analyzed.

Once the hazard has been chosen, it is used as the top event of a fault tree diagram. The system is them analyzed to determine all the likely ways in which that undesired hazard could occur. The fault tree is a graphical representation of the various combinations of hazards that lead to the undesired event. The faults may be caused by component failures, human failures or any other event that could lead to the undesired hazard, such as a random environment event. It should be noted that a fault tree is not a model of the system or even a model of the ways in which the system could fail. Rather it is a depiction of the logical interrelationships of basic events that may lead to a particular undesired event [Ref. 8].

The two most common gates used in fault tree analysis are the AND gate and the OR gate. If one or more events are required to produce the output event then an AND gate connector is used. The AND gate connects two or more hazards. An output occurs if all of the input hazards occur (Figure 5).

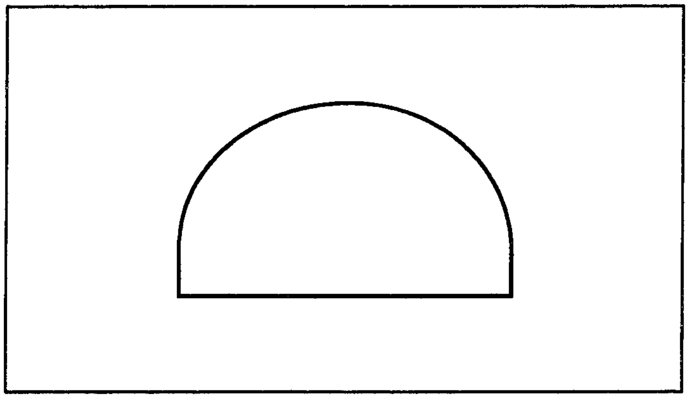


Figure 5: AND Gate

If one or more events can produce the output event then an OR gate connector is used. An output hazard occurs from an OR gate if any of the input hazards occur (Figure 6). Other gates that are occasionally used in FTA are the exclusive OR, priority and the inhibit. These will not be covered here.

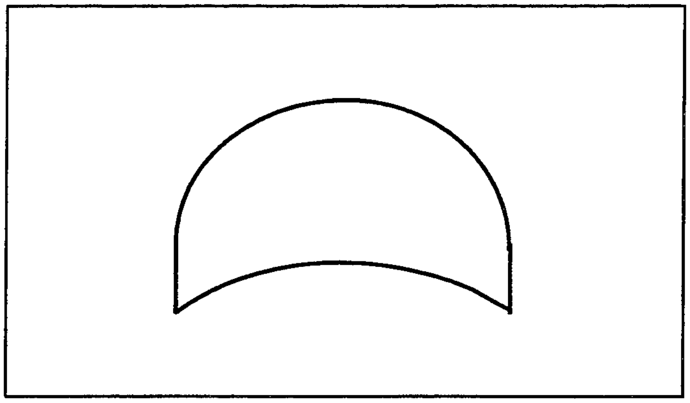


Figure 6: OR Gate

This analysis process continues until all hazards in the tree are either defined or cannot be decomposed further. The culmination of a fault tree analysis is a depiction of the required hazard sequence that must happen for the top hazardous event to occur. If no such path exists, then it is shown that the top event cannot occur [Ref. 9].

A. SOFTWARE FAULT TREE ANALYSIS

Software fault tree analysis (SFTA) was developed in 1983 through three nearly-simultaneously independent efforts by McIntee [Ref. 16], Leveson and Harvey [Ref. 17] and Taylor [Ref. 18]. Their research applied proven FTA techniques to the analysis of software. The process paralleled standard FTA principles, starting with a top event and working backwards through the tree, generating a path that showed the necessary hardware as well as software events that had to occur.

SFTA, like FTA, starts with a defined top event. This event is described through a hazard analysis and is usually a safety critical event. The process assumes that the system has failed according to the defined event and works backwards to determine the set of possible paths that allow the event to occur. This path is made up of further decomposed events connected by gates similar to those in FTA. Events are continually expanded until either they cannot be developed further due to lack of information or insufficient consequences or they no longer require analysis. Common software fault tree symbols and their associated meanings can be found in Appendix A. Once the tree has been fully expanded and analyzed, it can be shown that the program either allows or disallows the top event state to be reached. This information is then used to correct the program, if required, eliminating the undesired event's occurrence. Each event in the set of undesirable events is then analyzed in a similar fashion. It has been shown that for large systems, the use of partial SFTA can be effective in finding faults and in identifying critical modules that may need further analysis [Ref. 1].

An interesting note arises between the manner in which SFTA handles the quantification of event probabilities. Unlike hardware fault trees where each hazard/event can be assigned a given probability of failure due to centuries of historical data, software failures are in and of themselves logical, not lending themselves to a level of probability. The software either works or it does not. This distinction between probabilistic hardware fault trees and logical software fault trees is important in understanding the complexity involved in trying to conduct a complete software analysis.

In summary, SFTA can be used to determine software safety requirements, detect logic errors and identify multiple failure sequences involving different parts of the system that lead to hazardous events.

III. SOFTWARE SYSTEM ANALYSIS METHODOLOGY

Most analysis methodologies incorporate different combinations of software analysis techniques depending upon the application, available tools and experience of the analyst. Combining techniques and knowing when and which ones to use is an important part of the over all system safety analysis process. Cha discusses a safety oriented design method whose goal is to minimize the amount of safety-critical code and to produce a design whose safety can be certified [Ref. 19]. His work asserts that hazard analysis of designs allows the safety analyst to modify the software design to prevent the occurrence of hazardous states during operation. It is important that a practical, standardized methodology be implemented when performing software safety analysis. This chapter outlines a methodology consisting of a combination of standard manual techniques and automated techniques that have been developed at the Naval Postgraduate School. This methodology will be partially implemented and demonstrated in the next chapter.

A. STEP 1: CONCEPT EXPLORATION AND SYSTEM RESEARCH

The analyst must become intimately familiar with the system and its subsystems before any realistic analysis can be started. Interviews and discussions with design team personnel, project site visits, related system research and current system documentation reading are all possible sources of information. This research must be thorough and complete. Additionally, it must include any management proposed analysis constraints. These constraints should be included in the software development plan and cover analysis time lines, milestones and goals. By accomplishing this step the analyst gains a fundamental understanding of the entire system design and its relevant interfaces. Though the purpose of this step is not to make the software analyst into the system engineer, the time spent in this step will pay dividends throughout the entire analysis process.

B. STEP 2: HAZARD IDENTIFICATION

There does not seem to be any one easy way to identify hazards within a given system. Hindsight is always 20/20. After a mishap occurs, an investigation usually reveals a set of causes and the engineers learn for the next time [Ref. 8]. However, in safety-critical systems, there may be no next time. With no "systematic" process in which to look for hazards, the use of domain experts and thorough research is proposed as a "best" alternative. If the concept exploration step above is performed correctly, a decent foundation will be available to venture into this identification process. A group of "experts" should be designated and chartered to perform this process. An important pre-requisite must be that the group understands the differences between the new system and previous systems, if any, so that they can understand the new failure modes introduced by the new system. Numerous group decision methods have been proposed. The Delphi Technique and "brainstorming" are offered here as a best combination usage.

1. Delphi Technique

This method was created by the Rand corporation for the U.S. government and remained classified until the 1960's. The basic approach is to send out a questionnaire to all members of the group that enables them to express their opinions on the discussion topic. An appointed coordinator collects all the inputs, collates them and returns the summarized information to the members in an anonymous format. This process continues until a consensus is reached on the topic [Ref. 8].

Using this technique in hazards identification offers a wide range of advantages. With the dramatic increase of electronic mail, many more "experts" have become available for inclusion to software groups. The constraints of physical meetings would be eliminated and the process of collecting and distributing the results relatively painless. Group members could easily digest the required system research information and make sound judgements in a matter of days vice weeks.

2. Brainstorming

This “technique” may seem more like common sense then a defined process. But in cases where resources are limited and/or the system is big enough to prevent obtaining an exhaustive hazards list, brainstorming can actually provide a plethora of hazards that otherwise would not have been identified. No meetings are required. Experts are solicited to list all possible hazards that they envision for the system. These lists are gathered by the analyst and processed into a formal systems hazards list.

Through the combined use of the Delphi Technique and brainstorming in this step, the analyst is provided with an excellent set of potential high level system hazards to start the analysis process.

3. Program Management Input

Management plays a vital role in hazards identification. It must allow the Delphi group or the solicited experts time to produce the identified hazards, but not so much time that efforts are wasted. Some form of guidelines needs to be established. This could be in the form of a set of hazard analysis criteria or as simple as a time line. With the support of management and the use of good research and brainstorming techniques, a complete and useful set of hazards can be identified and readied for analysis.

C. STEP 3: PRELIMINARY HAZARD ANALYSIS

This step actually begins the analysis section of the process, but as its name implies, it is the precursor to the formal hazards analysis and provides a framework from which the analyst can conduct a detailed analysis.The PHA must be executed using the most current resources available. Due to its currently tedious and manual execution, any delay or re-design causes frustration and lost work man hours.

A thorough, methodical analysis of each available resource must be accomplished. By starting with the software development plan (SDP) and the software requirements specification (SRS), the analyst can isolate the areas that relate directly to the list of identified hazards. This will drastically reduce the task at hand. This initial cutting down of requirements helps reduce the scale and complexity of the analysis. Depending upon the system being analyzed, other documents may need to be inspected. This list of documents should include at a minimum, those documents used in Step 1 during the Concept Exploration and System Research phase.

Next, a table is constructed containing each of the identified requirements, its associated possible hazard, possible result if the hazard occurs and its severity level in terms of loss of life and property. An example is provided in Table 1. The idea here is to take the developed list of hazards from Step 2 and map them to their defining requirements found in the system documentation. A thorough mapping is important, however, an exhaustive one generates an unmanageable table. It is best that an ongoing dialogue between the analysts and the software developers be concurrent with the PHA to help reduce, combine and/or eliminate unnecessary mappings. By accomplishing this step the analyst narrows the scope of the analysis and begins to focus on the safety critical areas of the system. Additionally, the analyst gains a rough quantified understanding of the analysis problem and identifies the specific high severity component hazards of interest which will be used as initial starting input for the next step, Hazard Analysis.

SDP Para Ref	SDP Requirement	Possible Hazard	Possible Result	Severity
1.2.2	Control CSCI moves and positions the simulation hardware	Erroneous control signals are generated and sent to the system hardware	Hardware/Personnel damage/injury	High

Table 1: Example PHA of MESA Software Development Plan

D. STEP 4: HAZARD ANALYSIS

This step begins with the analyst determining which software modules are the most safety critical using the results of the PHA. This refined level of hazard identification at this step allows the analyst to perform a combination inductive and deductive technique. This combined process first uses Failure Modes and Effects Analysis (FMEA) as an inductive technique to determine what hazardous states are possible. Once these states are defined, a specific hazard fault tree is developed. Then the deductive technique of SFTA is applied to determine how the specific hazard can occur. Here the proposed methodology diverges from the high level software system context and starts to concentrate on specific software configuration items.

1. Step 4.a: Failure Modes Effect Analysis

As described in Chapter I, the implementation of FMEA is accomplished by manually constructing a table. Table 2 shows an example, building upon the PHA in Step 3. Even by processing only identified safety-critical items, the table can become a rather large document. A directed effort to consolidate and combine similar items through group discussion will help keep this step manageable.

Item	Failure Mode	Effect of Failure	Cause of Failure	Occur rence	Severity	Prob Detect	Risk	Corrective Action
CSCI control signals	Host/remote computers out of synch (Closed vs. Open loops)	Inadvertent motion of hardware (ST,Sphere, OTSS)	Valid Host signal sent to Remote in an invalid mode (Closed vs. Open loop)	5	9	5	225	Incorporate a loop synchronization algorithm.

Table 2: Example FMEA of MESA Control CSCI

By accomplishing this step the analyst more succinctly defines the safety-critical scope of the analysis and determines which areas need to be further analyzed. The output of the FMEA produces a set of top-level events that are then used as input for the next step, Specific Hazard Fault Tree Generation.

2. Step 4.b: Specific Hazard Fault Tree Generation

At this point the analyst has accumulated a list of independent safety-critical events. These events are used to generate separate fault trees that identify the logical pathways from those events to their associated source code. The analyst makes each top-level event a root node in its specific hazard fault tree. The node is then decomposed into its required causes. This is continued until the final event leaf nodes succinctly define source code modules or areas. FTE is extremely useful in this step. With its graphic interface and immediate feedback, FTE provides the analyst with the generated fault tree quickly and effectively. The leaf nodes from this step become the input for the next step, SFTA.

3. Step 4.c: Software Fault Tree Analysis

In traditional methodologies, SFTA is carried out using manual methods. At most, a crude text and/or graphics editor would be used to assist the analyst in drawing and re-drawing the required fault trees with absolutely no computer-aided analysis. This paper's proposed methodology systematizes the standard SFTA technique by incorporating the use of locally-developed, automated, fault-tree tools. This automated fault-tree generation and manipulation process dramatically reduces analysis time and substantially reduces human induced errors. At this point, developed code is required for the following steps to be executed.

a. Step 4.c.1: ACTT

The analyst takes the set of top-level events generated in Step 4.a and determines which of those events have had their Ada code developed. Those not yet coded should be noted and a list kept for other processing, either when the code becomes available or using a design-analysis technique. It is important that currently-uncoded critical modules not be passed over or forgotten. Source-code-line-number labels should be generated for all coded modules. Top-level events should then be paired with their corresponding code lines. This mapping is necessary when using FI in the following step.

Each coded module should then be run through ACTT. This process is quick and generates a software fault tree in the FTE-specified format. The analyst should be prepared for a large number of output files as each module is processed. ACTT generates a separate fault tree file for the main procedure/package body, each task body and each defined exception statement. Separate working areas for each module helps in keeping the output organized. This is important as ACTT writes its output to identical file names that will over write any previously existing output from other processed modules. An example excerpt from an ACTT generated file is shown in Figure 7. For large numbers of modules, script or batch files can be written to execute this step in an even more efficient manner. After all necessary modules are processed, the output can then be manipulated and analyzed using the combination of Fault Isolator and Fault Tree Editor as described in the next step.

```
 431
Sequence of statements causes Fault
traffic.a
45 48 0 0 1 2 2

 430
Last statement causes Fault
traffic.a
45 48 0 85 1 0 1
```

Figure 7: Example ACTT Output File

It is interesting to note, that modules not fully coded can still be processed by ACTT. This "pre-processing" may prove useful in some cases where a more abstract fault tree could help to determine which detailed code structures would be less fault prone.

b. Step 4.c.2: FTE/FI

This step uses FI and FTE in combination to isolate and manipulate the fault tree generated in the Step 4.c.1. The analyst should launch both applications and position their work-space interfaces so that both are visible. The idea here is to use FTE to display the fault tree, determining which sub-tree and/or nodes need to be isolated for further processing based upon the source-code lines of interest.

FTE is used to display the ACTT generated fault tree. The same fault tree is then opened for processing using FI. FI will display the total number of nodes and the tree's depth (root node is level zero). If the tree is of small to medium size, these tree statistics can be verified by moving the FTE display around, counting the exposed nodes. The Search for Nodes option in FI is then selected. Using the source-code-line-number created in Step 4.c.1 as input, FI returns three lists of matching nodes, Exact Match, Containing Match and Closest Match. From this, the analyst can determine which node, nodes or sub-trees that need to be eliminated. Once the sub-fault tree is isolated, it can be displayed in FTE for further isolation, to be printed or saved as a new fault tree file. Figure 8 shows FI's main menu and Figure 9 shows FTE's display of an example fault tree.

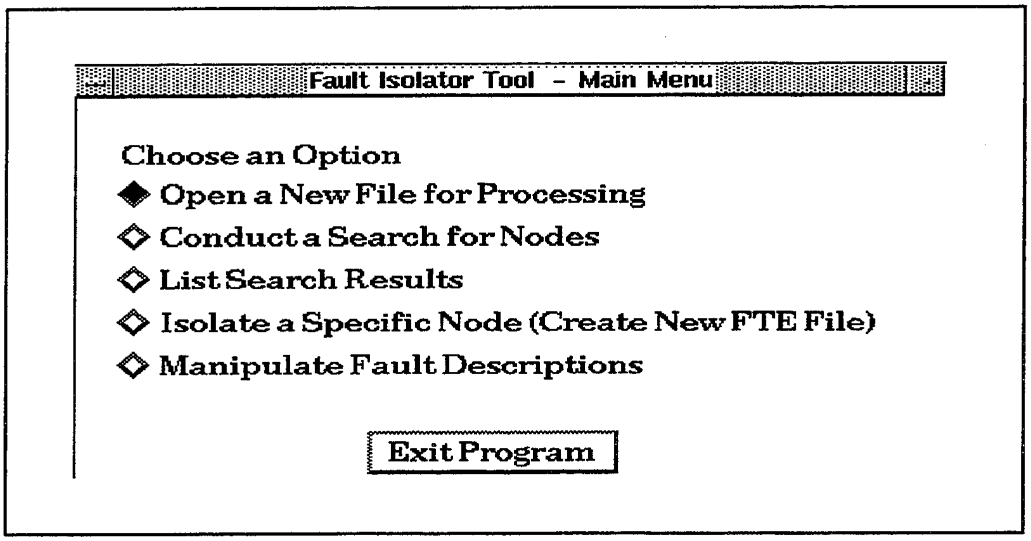
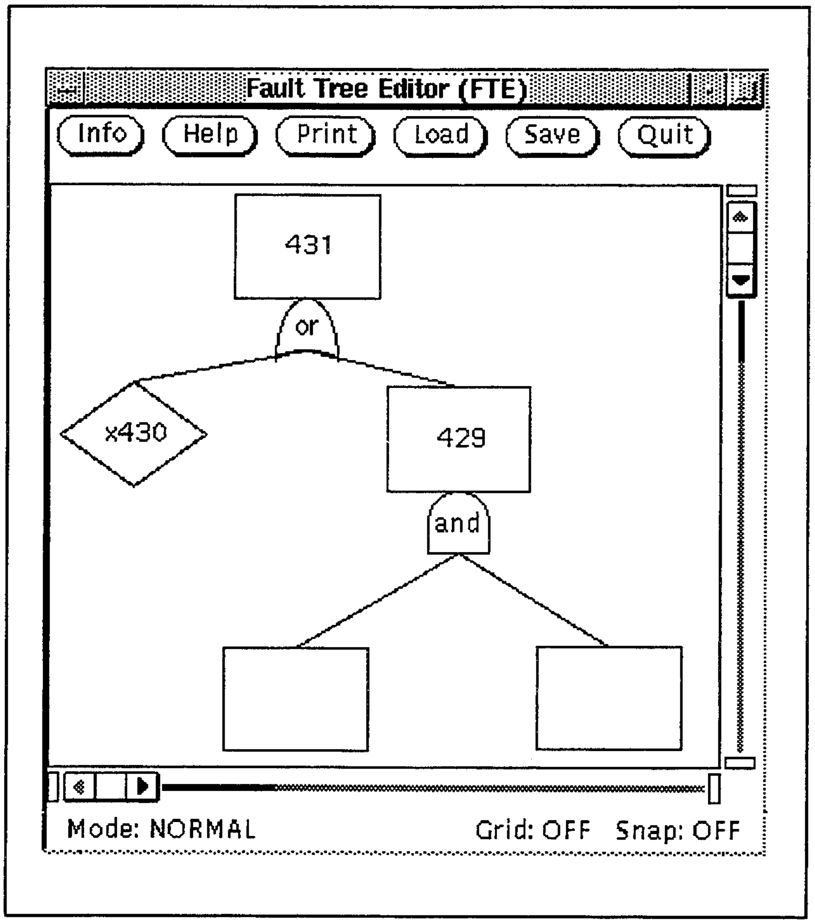

Fault Isolator Tool - Main Menu
Choose an Option
◆ Open a New File for Processing
◇ Conduct a Search for Nodes
◇ List Search Results
◇ Isolate a Specific Node (Create New FTE File)
◇ Manipulate Fault Descriptions
Exit Program

Figure 8: Fault Isolator Tool Main Menu



Mode: NORMAL Grid: OFF Snap: OFF

Figure 9: Fault Tree Editor Example Display

The combined use of these two tools gives the analyst a powerful method to perform efficient and accurate manipulation and isolation of the fault tree. These isolated sub-fault trees depict the safety-critical-software hazards and their possible paths of occurrence. With these graphical depictions the software analyst can then move onto the next step of results analysis.

4. Step 4.d: Results Analysis

The final analysis of the preceding automated and semi-automated steps is now performed. This analysis is still limited to manual methods supported by automated tools and takes on an iterative process format. As design and code decisions are made from the generated sub-fault trees, repeated manipulations and re-generations of those trees will be required using FI and FTE. Each identified hazard should be processed through these steps. A bringing together of coupled modules is anticipated and will result in the need for the merging of various generated fault trees. The current functionality of FI and FTE does not allow this yet. Future implementation is being proposed. At this point the analysis methodology has reached a logical conclusion point.

E. METHODOLOGY SUMMARY

This proposed software system analysis methodology has tried to establish a direct, efficient and effective examination process using a combination of standard manual techniques and newly constructed automated tools. The methodology steps and knowledge gained are summarized in Table 3 below. An analysis of a portion of the MESA control system will be conducted implementing this methodology in the next chapter.

Step	Process	Knowledge Gained
1	Concept Exploration and System Research	Overall system design
2	Hazards Identification • Delphi Technique and Brainstorming	Potential high level system hazards
3	Preliminary Hazard Analysis	Specific component hazards of interest
4	Hazards Analysis • 4.a. Failure Modes Effects Analysis • 4.b. Specific Hazard Fault Tree Generation • 4.c. Software Fault Tree analysis • 4.c.1. Automated Code Translation Tool • 4.c.2. Fault Tree Editor/Fault Isolator Tools • 4.d. Results Analysis	How given possible hazardous states can occur

Table 3: Software System Analysis Methodology Summary

IV. METHODOLOGY IMPLEMENTATION RESULTS

This chapter describes the implementation of the complete software safety analysis methodology. The MESA control system was selected for analysis due to its Ada programming language usage, distributed safety-critical control system structure and its apropos military application nature. Each methodology step outlines action taken and the corresponding results. This provides a concise discussion on the implementation and resulting usefulness of each step.

A. CONCEPT EXPLORATION AND SYSTEM RESEARCH

Initial system research was conducted through a combination of manual and electronic means. Extensive use of the Internet allowed system design documentation to be updated and accessed in minimal time. Additionally, constructive conversations with the MESA system engineer and software development team were made easily through the use of electronic mail. These were important factors due to the over 300 miles between the MESA facility and the Naval Postgraduate School. A project site visit was conducted to China Lake, CA to tour the facilities and meet the MESA project personnel. The discussions held were extremely useful in providing system familiarity, identifying relevant hazards and projecting system implementation availability.

Two primary documents were utilized to obtain the required system familiarity. These were the Software Development Plan (SDP) [Ref. 14] and the Software Requirements Specification (SRS) [Ref. 15]. These documents were critical to the understanding of the system, its design and the project's development plan. A thorough discussion on these documents combined with the project personnel meetings allowed sufficient details of the system interfaces to be collected and identified for use in determining possible system hazards.

B. HAZARD IDENTIFICATION

The hazard identification step was implemented through the use of brainstorming sessions. The Delphi Technique was not implemented due to the academic nature of this analysis. Numerous face-to-face and electronic brainstorming sessions were held between different combinations of the author, the author's thesis advisor and various MESA project personnel. This provided a small group of experts to analyze the system and identify potential high-level, high-severity hazards. This led the group to entertain only those hazards dealing directly with loss of life and property/material damage.

Each HWCI was analyzed and lists of possible hazards generated. The analysis was then artificially focused on the Sphere HWCI since it was the only HWCI at the time of the analysis with fully-functional Ada control code. Six high-level possible hazards were identified as listed below.

- Sphere Impacts Arena Structure
- Sphere Impacts Object Other Than Arena
- Sphere Stops At Undesired Position
- Sphere Encoder Lines Break
- Sphere Control Lines Break
- Inadvertent Sphere Line Movement

C. PRELIMINARY HAZARD ANALYSIS

The PHA step started with continued examination of the SDP and SRS. During this process, it was determined that four of the six high-level possible hazards were actually predecessors for the other two. This narrowed down the list to two specific hazards of interest. A detailed mapping of each of these specific hazards of interest to its associated defining-requirements in the SDP and SRS was made. Table 4 depicts the hazards, their defining requirements and possible result if the hazard occurs.

Though the results look somewhat simplified, this step proved to be labor intensive and time consuming. The SDP and SRS contained over 150 pages of requirements, requiring 45 hours of manual reading and analysis. The PHA narrowed the scope of the analysis, focusing on the specific hazards of interest within the Sphere HWCI. The other system HWCIs and CSCIs were also examined. As an example of a more involved PHA, Appendix B shows the PHA results on the MESA Control CSCI. To ensure the validity of this analysis step, a review of the PHA was conducted by MESA project personnel during a scheduled site visit. This review proved extremely valuable by identifying and prioritizing specific system hazards. This step demonstrated that the PHA greatly reduced the number of possible hazards for each configuration item, narrowed the scope of the analysis to concentrate on the specific hazards of interest and enabled the analysis methodology to focus on the safety-critical areas of the Sphere HWCI.

SRS Para Ref	SRS Requirement	Possible Hazard	Possible Result	Severity
3.2.2.3.2.5	Sphere Control shall control the sphere encoder and control line movements	Sphere impacts arena structure	Sphere/Personnel damage/injury	High
3.2.2.3.2.3	ST Control shall control the sensor transporter movements	Sphere impacts object other than arena	Sphere/ST/Target/ Personnel damage/ injury	High
3.2.2.3.2.4	OTS Control shall control main hoist line, control lines and encoder line movements			
3.2.2.3.2.5	Sphere Control shall control the sphere encoder and control line movements			

Table 4: Results of PHA on MESA Sphere HWCI

D. HAZARDS ANALYSIS

The hazard analysis began by looking strictly at the Sphere HWCI specific hazards of interest. A FMEA was performed followed by the “meat of the analysis” using SFTA.

1. FMEA

The Sphere HWCI FMEA produced the results depicted in Table 5. The individual FMEA items were derived from their safety-critical properties as related to the specific hazard of interest. This step provided a clear, concise listing of the specific top-level events necessary to develop the Specific Hazard Fault Trees.

Item	Failure Mode	Effect of Failure	Cause of Failure	Corrective Action
Sphere HWCI control signals	Sphere control software generates erroneous motion command	Undesired commanded motion of Sphere	• Data initialization failure • Undesired movement value generated • Invalid incremental movement calculation	Incorporate software analysis checking to ensure valid motion commands generated
Sphere hardware	Sphere encoder/control lines break	Sphere motion causes impact	• Hardware defective • Hardware limits exceeded	Ensure hardware specifications satisfied and routine hardware inspections conducted

Table 5: Results of FMEA on MESA Sphere HWCI

2. Specific Hazard Fault Tree Generation

The Specific Hazard Fault Tree Generation step was implemented using the hazards analysis results from the previous two steps. The two top-level specific-hazard faults of the Sphere HWCI determined from the PHA were designated root nodes for their respective fault trees. Through the use of FTE, the tree generation process began. Each root hazard was piece-wise decomposed into to its subsequent fault-causes. This process was accomplished through the interactive discussions between the author and the author's thesis advisor. As each hazard's fault-causes were determined, corresponding nodes were added to the fault tree. Each "Cause of Failure" result generated in the FMEA was used to help build the tree. These fault-causes ended up being interior nodes of the tree and logically linked the specific-hazard root node with the user-generated, source-code level leaf nodes.

The first specific hazard fault tree was generated for the specific hazard *sphere impacts arena* (Figure 10). The top-level fault was decomposed into two independent faults, 1) *sphere control line breaks* OR 2) *controlled motion of the sphere impacts the arena*. These two faults are depicted at level one in Figure 10. The control-line-breaks fault

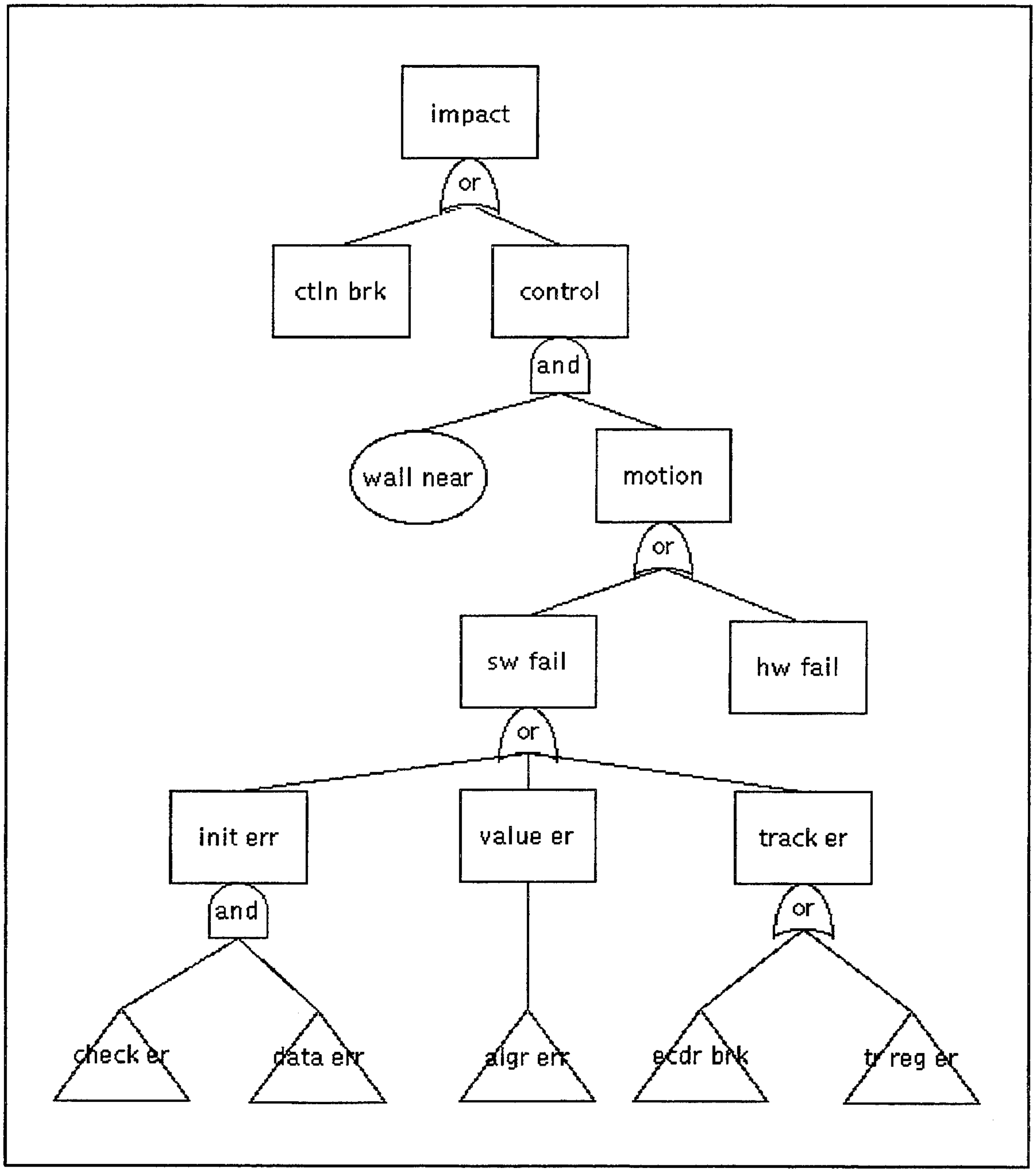


Figure 10: Sphere Impacts Arena Specific Hazard Fault Tree

was not further analyzed since it dealt more with hardware failure issues than software. The controlled-motion fault was further decomposed using the FMEA failure causes as shown in Figure 10. Eventually five refined leaf nodes were generated. These leaf nodes succinctly defined specific Sphere HWCI source-code modules.

This process was duplicated for the second specific hazard fault tree, Sphere impacts object other than arena (Figure 11). This tree generation effort paralleled that of

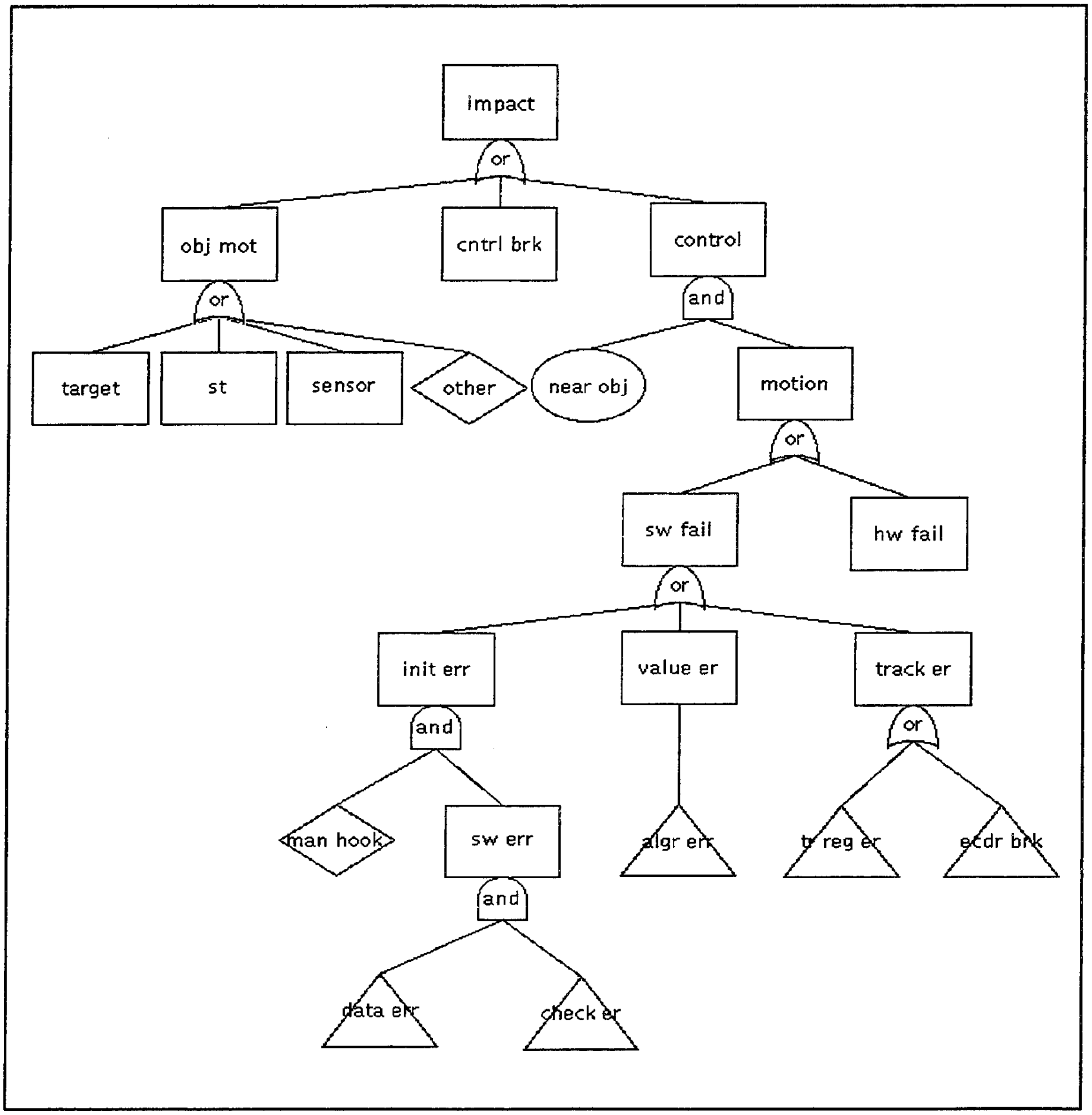


Figure 11: Sphere Impacts Object Other Than Arena Specific Hazard Fault Tree

the first but included the additional first-level independent fault of *object motion other than the sphere causes impact.* From the system design, the only logical objects that could impact the sphere were the target, sensor transporter and the sensor. Each of these objects became level-two nodes and further decomposition was performed.

The remainder of the tree in Figure 11 was identical to the first tree with the exception of the "init err" node which contained an additional fault level. This was due to the possible collision of other objects during sphere initialization. The resulting five leaf nodes were the same as the first tree.

Due to the similarities of the two trees and the fact that the sphere impacting the arena seemed more consequential, only the sphere-impacts-arena tree was further analyzed. Each of the five leaf nodes in Figure 10 were then used to start the SFTA process in the next step. The use of FTE in this step was essential. Having the ability to graphically represent these top-level hazards in a real-time manner made the process of developing these trees painlessly effective.

3. Software Fault Tree Analysis

The SFTA step began the exciting portion of the analysis. The five source-code-interface leaf nodes in Figure 10 were designated as individual, software-starting root nodes for their respective software fault trees. A mapping of these software-root nodes to their associated source code files was generated. This mapping was accomplished by using manual and semi-automated methods to search all source code files for relevance to the software-root node faults. This was extremely useful in reducing the amount of source code to be analyzed, however it was time consuming and a bit tedious. Table 6 shows the results of this mapping.

SW Root Node	Fault	Source Code File
Data Error	Wrong initialization data provided to software	Fault_monitor_s.a Remote_sphere.a

Table 6: Software Root Node Mapping to Sphere Source Code Files

SW Root Node	Fault	Source Code File
Check Error	Initialization check fails to find data incorrect	Fault_monitor_s.a Remote_sphere.a
Algorithm Error	Closed loop position algorithm generates bad values	Closed_loop_position_control.a
Tracking Register Error	Encoder line tracking not registered in software	Digital_input_s.a Digital_output_s.a Encoders_b.a Evaluate_s.a Fault_monitor_s.a Hw_reset.a Remote_sphere.a
Encoder Line Breaks	Encoder line break causes continued motion	Digital_output_s.a Encoders_b.a Evaluate_s.a Fault_monitor_s.a

Table 6: Software Root Node Mapping to Sphere Source Code Files

A second mapping was then generated to identify the specific source code procedures and functions that would fall under each software-root node. Once again, the identification of these procedures and functions was accomplished by searching each pre-mapped source code file for modules of relevance. This was an absolutely necessary step

Source Code File	Procedure of Interest	SW Root Node
Closed_loop_position_control.a	Calc_Within_Tolerance Calc_Errors Calc_Steps Steps_Or_Null Move_Steppers Spread_Of_Errors_During_Settle_And_Hold_Times	Algorithm Error
Digital_input_s.a	Get_Control_Words	Track Register Error

Table 7: Sphere Source Code Procedures of Interest Mapping to Software Root Node

Source Code File	Procedure of Interest	SW Root Node
Digital_output_s.a	Encoders_Up Encoders_Down Encoders_On Encoders_Off UnFreeze_Encoder_Readings Freeze_Encoder_Readings	Track Reg Error/ Encoder Line Break
Encoders_b.a	Get_Values	Track Reg Error
Evaluate_s.a	Evaluate	Track Reg Error/ Encoder Line Break
Fault_monitor_s.a	Initialize_Encoders_History	Data Error/Check Error
	Check_For_Malfunction	Track Register error
	Broken_Encoder_Line	Encoder Line Breaks
Hw_reset.a	HW_Reset	Track Register Error
Remote_sphere_b.a	Initialize1 Initialize2 Initial_Engagement_Conditions Go	Data Error/Check Error
	Move_Encoder_Line Set_State Stop_Motion Stop_All_Lines	Track Register Error

Table 7: Sphere Source Code Procedures of Interest Mapping to Software Root Node

in the effort to construct each individual software fault tree. Each procedure, function and task body of interest would become itself a software sub-tree connected to its parent software-root node. Table 7 shows the results of this second mapping.

a. ACTT

This step started with some initial work-area house-keeping. An electronic directory was generated for each software root node listed in Table 7. The mapped-source-code files for each node were then copied to that directory. Each source code file was run through ACTT, capturing the execution using the Unix system script utility. ACTT quickly and efficiently generated tree templates for each file. Table 7 shows each source code file and the number of ACTT generated nodes in its set of templates. These extremely bushy trees were an expected result. ACTT separated each set of templates into three different areas, a main template, exception statement templates and task body templates. Only source code with exception and task body constructs generated the latter two template types. This execution method for ACTT allowed clean, concise processing of each required file. The outputs from this step, the ACTT generated templates and the Unix script session files, fed directly into the next phase of the analysis using the FI tool.

Source Code File	Number of Generated Nodes
Closed_loop_position_control.a	1929
Digital_input_s.a	86
Digital_output_s.a	1074
Encoders_b.a	447
Evaluate_s.a	577
Fault_monitor_s.a	1573
Hw_reset.a	25
Remote_sphere_b.a	2423

Table 8: Number of ACTT Generated Fault Tree Nodes Per Source Code File

b. FI Pruning

The next step in the analysis involved doing a first-pass pruning of irrelevant sub-trees from the generated templates, making them more manageable. FI and FTE were used effectively to achieve this. Each generated template was loaded into FI and FTE. FI accurately displayed the number of nodes and tree levels and FTE provided the graphical structure. By searching the Unix script files, individual source code lines of interest were identified and entered into FI for processing. The source code lines of interest were chosen based upon the relevance to their associated initial fault statements listed in Table 6. FI accurately provided a listing of nodes in the tree relating to each source code line. New root nodes were selected and relevant sub-trees generated. Some of the trees were able to be pruned directly through FTE, though FI was still used to identify the correct nodes for pruning. This first-pruning process was extremely effective in reducing the sizes of all the fault sub-trees. This was found to be an extremely helpful step in that it allowed faster, more efficient processing of the sub-trees in the next step, Results Analysis. A comparison between the original and first-pruned sub-tree sizes is shown in Table 7.

Software Root Node	Original # of Nodes	First-Pruned # of Nodes
Check Error	647	409
Data Error	649	411
Algorithm Error	1756	1325
Encoder Break	1150	548
Tracking Register Error	1527	953

Table 9: Comparison of Original vs. Pruned Software Root Node Fault Tree Size

Appendix C contains the top-level fault trees fault descriptions and the software-root node fault trees generated in the analysis of the Sphere HWCI. A complete listing of each nodes associated fault statement is included as well.

c. *Results Analysis*

At this point, the analysis process had provided a complete fault tree of the Sphere HWCI, starting from the identified specific hazard of interest, *sphere impacts arena,* in the top-level tree, to numerous source-code-statement-construct leaf nodes generated in the software-root node sub-trees. The fault tree contained over 5700 nodes and depicted all safety-critical possible fault paths. A complete results-analysis step would take this data and systematically analyze each possible path to determine which faults could and could not occur. This step in itself would be a challenging task to say the least. Due to time limitations and the desire to demonstrate the remainder of this methodology, only one subset of fault paths was analyzed. The analyzed fault was that relating to the question, if an encoder line breaks, what happens?

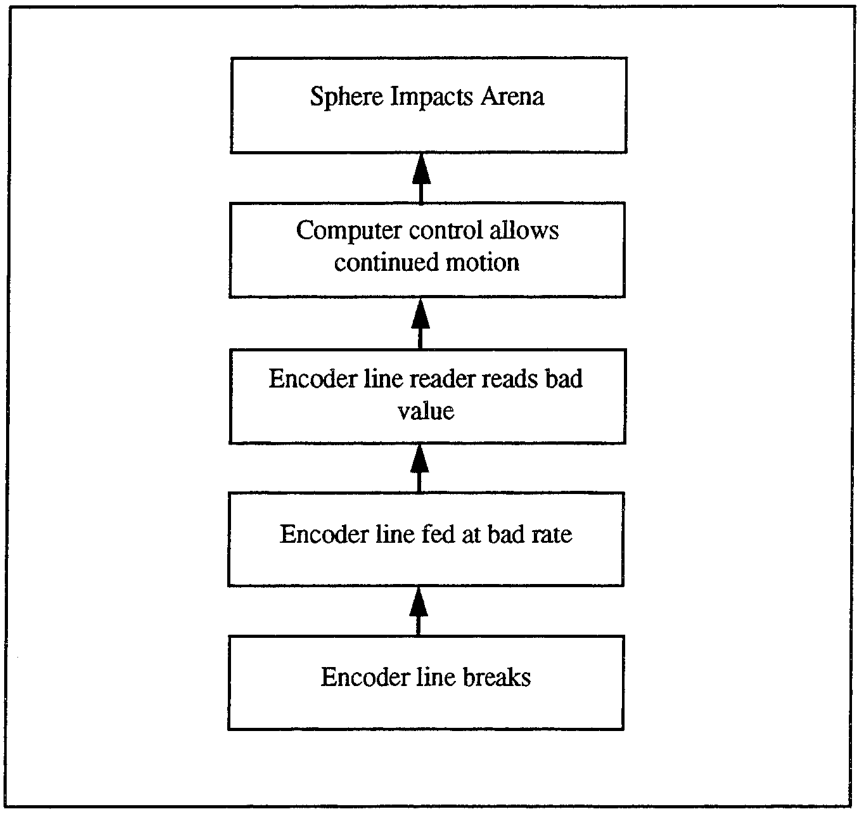

Sphere Impacts Arena
Computer control allows continued motion
Encoder line reader reads bad value
Encoder line fed at bad rate
Encoder line breaks

Figure 12: Encoder Line Breaks Causal-Link Analysis Diagram

To analyze this fault an abstract causal-link analysis diagram was constructed. Figure 12 shows the derived causal steps between the two physical events “Encoder line breaks” and “Sphere impacts arena.” When the encoder line breaks, the Sphere-HWCI control system determines that an erroneous encoder line speed is present, processes that input and sends signals to stop all motion. The analysis looked at the possible logical paths that could occur to prevent the required stop signals from being sent, thus allowing continued motion of the Sphere into the arena. Figure 13 depicts the software node

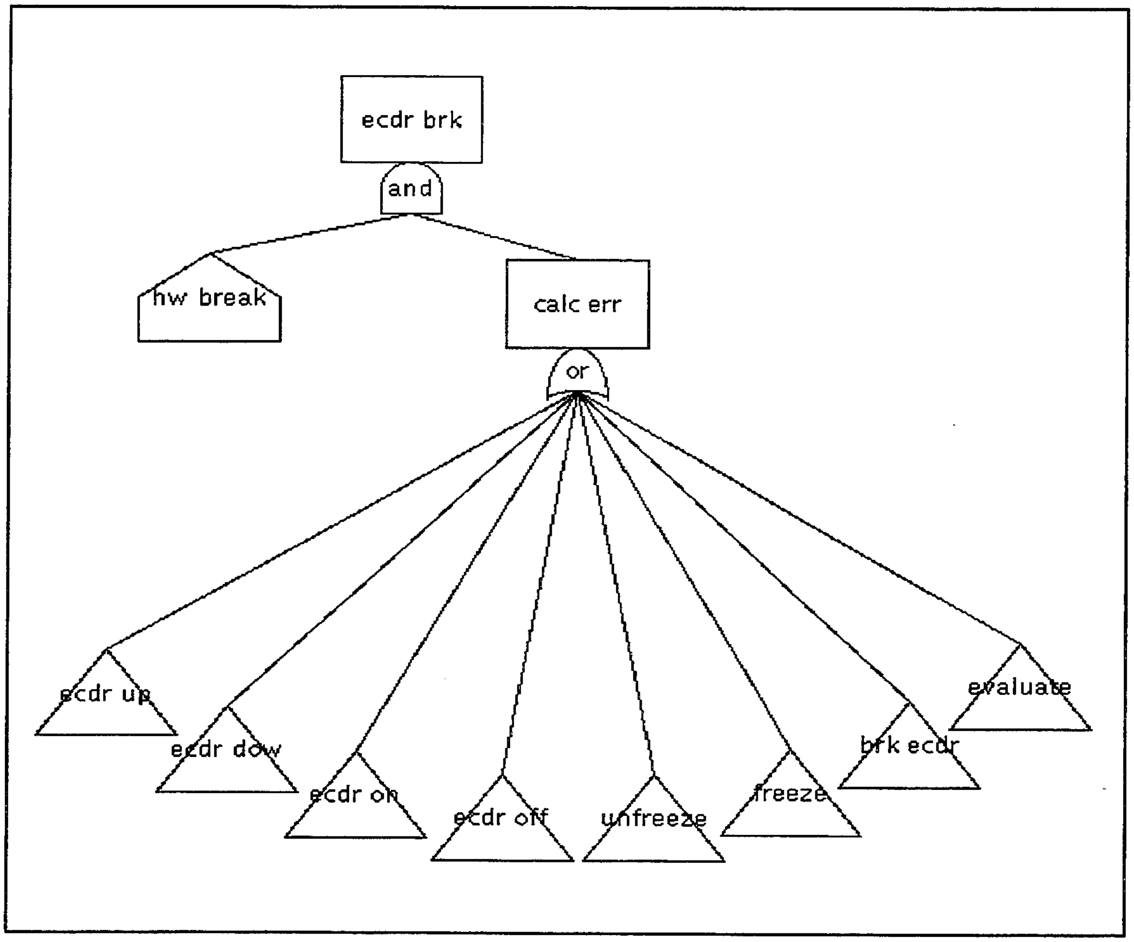


Figure 13: Encoder Line Breaks Software Node Sub-Tree

sub-tree that represents these logical paths. A calculation error in any one of the attached procedures-of-interest sub-trees could cause the "ecdr brk" node fault, *encoder line break causes continued motion*, to be true, thus leading to the Sphere impacting the arena. By using proof-by-contradiction, a systematic process was performed on each sub-tree determining if a calculation error would or would not lead to a true "ecdr brk" node fault. Irrelevant nodes and sub-trees were "diamonded out" through the use of FI and FTE. It was found that all procedure-of-interests could be diamonded out except for the "brk ecdr" and "evaluate" sub-trees. Appendix C contains these two procedure-of-interest's final "pruned" sub-trees, showing all possible, logical fault paths leading to the "ecdr brk" fault node.

Each of these fault-paths were then analyzed to determine which, if any, could occur. The fault tree depicting the calculation algorithm that determines if the stop motion procedure is executed (via setting a boolean value), is shown in Appendix C, Figure 26. Starting at each leaf node, an examination of the associated source code showed that none of the fault-paths could occur. As each node and its code were analyzed, they were diamonded out as it was determined that the robust code structures and fault-tolerant procedure calls eliminated a possible fault path. Additionally, the incorporation of hardware fairleads on the encoder lines would prevent sphere motion beyond set safety distances. Thus, both a software and hardware failure would need to occur before a broken encoder line would lead to unintended motion. This concluded the analysis of the encoder line break fault showing that the actual source code that calls the stop motion procedure was fault-free and correct.

E. EFFORT EXPENDED

This analysis methodology required many man-hours of work and a fair amount of computer resources. Though many supporting members provided invaluable insight and guidance, the day-to-day analysis crunching was basically performed single-handedly by the author. Each methodology step demanded its own unique effort and resources. The actual amount of source code analyzed was in excess of 74,000 lines and required 100 man-hours to complete.

The first phase of this methodology, Concept Exploration/System Research, Hazard Identification and Preliminary Hazard Analysis, involved a substantial amount of reading, examination and interpretation of numerous system resources with minimal computer support effort. Many man-hours were expended tracing system requirements for hazard identification and analysis. For example, the PHA on the Sphere-HWCI and

Control-CSCI alone required over 35 man-hours to complete. This phase was a manual process that required the most man-hours and will be the most challenging to automate.

The second phase of this methodology, Specific Hazard Fault Tree Generation and Software Fault Tree Analysis with ACTT, FI and FTE, involved minimal amount of manual effort and a substantial amount of computer effort. The simultaneous use of ACTT, FI and FTE was critical in this analysis. The systematic, iterative process of generating fault templates with ACTT and alternating between viewing/reviewing the associated fault trees in FTE and pruning/re-generating those trees with FI required a thorough understanding of each tool, the methodology and the supporting computer hardware and software. With this knowledge, the actual effort expended in a typical analysis of one of the given software-node fault trees was on the order of hours. This time would be further reduced for those whose life is devoted to software analysis. However, without the use of these semi-automated tools and their integration into this methodology, even the most proficient software analyst would spend more time in accomplishing the same analysis.

The third phase of this methodology, Results Analysis, involved a 50/50 split between manual and computer effort. As each individual software-node sub-tree was analyzed, an equal amount of time was spent looking through associated source code files. This phase produced the actual listing of possible faulty source code lines and was the culmination of the entire methodology process.

V. CONCLUSIONS

A. CONCLUSIONS

This thesis presented a sample methodology that demonstrated that Ada-based software control systems can be efficiently and effectively analyzed in a practical time frame through the use of existing automated and semi-automated software safety tools and methods. The semi-automated tools ACTT, FTE and FI were used in combination with standard software fault tree analysis techniques to provide accurate and meaningful software safety analysis data on a real world, currently developing software project. Through this methodology implementation, it was found that numerous "manual" holes existed in the current analysis process. Specifically, the need for an "expert systems" approach to executing the system design research step, hazards identification step and the preliminary hazard analysis step was identified. Current levels of "expert systems" technology makes this a steep task to fill. This coupled with the "industry-expected" need for a human safety analyst, shows a trend towards development of automated tools to assist, not replace, the analyst. Nevertheless, any effort towards development of a fully automated process will require substantial resources be directed towards the improvement of these analysis tools and expert systems.

B. RECOMMENDATIONS AND FUTURE WORK

Software analysis of safety-critical control-systems is an ongoing, evolutionary process. As more control systems are developed, more methods to analyze and process them will be developed. It is essential to disseminate, throughout the software analysis community, the lessons learned from those developmental processes to help promote future methodology development. The methodology presented here is dependent upon the use of the semi-automated tools, ACTT, FTE and FI. These tools and the developed methods to use them should be made available for further research and development.

These tools have demonstrated their usefulness in providing efficient software safety analysis capabilities. ACTT provides excellent software analysis data but could be improved upon as follows. First, a graphical user interface front-end will eventually become essential for wide spread use. Second, additional ACTT generated data output files would be useful such as a data session file and a generated node summary file. The FI tool gives the analyst a unique capability to manipulate and isolate fault trees. FI could be improved upon by adding additional functionality such as a Delete Sub-Tree and a Move Child/Sibling option. FTE could be improved by adding a zoom in/zoom out option, a thumbnail tree viewing option, an increased viewing window size and some sort of "pretty-tree" printing capability. A logical extension of these tools could be the combining of all three into one "Software Safety Analysis Tool Suite." This would provide the best utility of all and give the analyst one complete tool package.

Continuing research and development of semi-automated and automated software analysis tools is essential in achieving a useful, standardized software analysis methodology. Though achieving a completely automated process in the near term seems unlikely to this author, continued work in each area of safety analysis will produce the stepping stones towards that desirable goal.

APPENDIX A. SOFTWARE FAULT TREE SYMBOLOGY

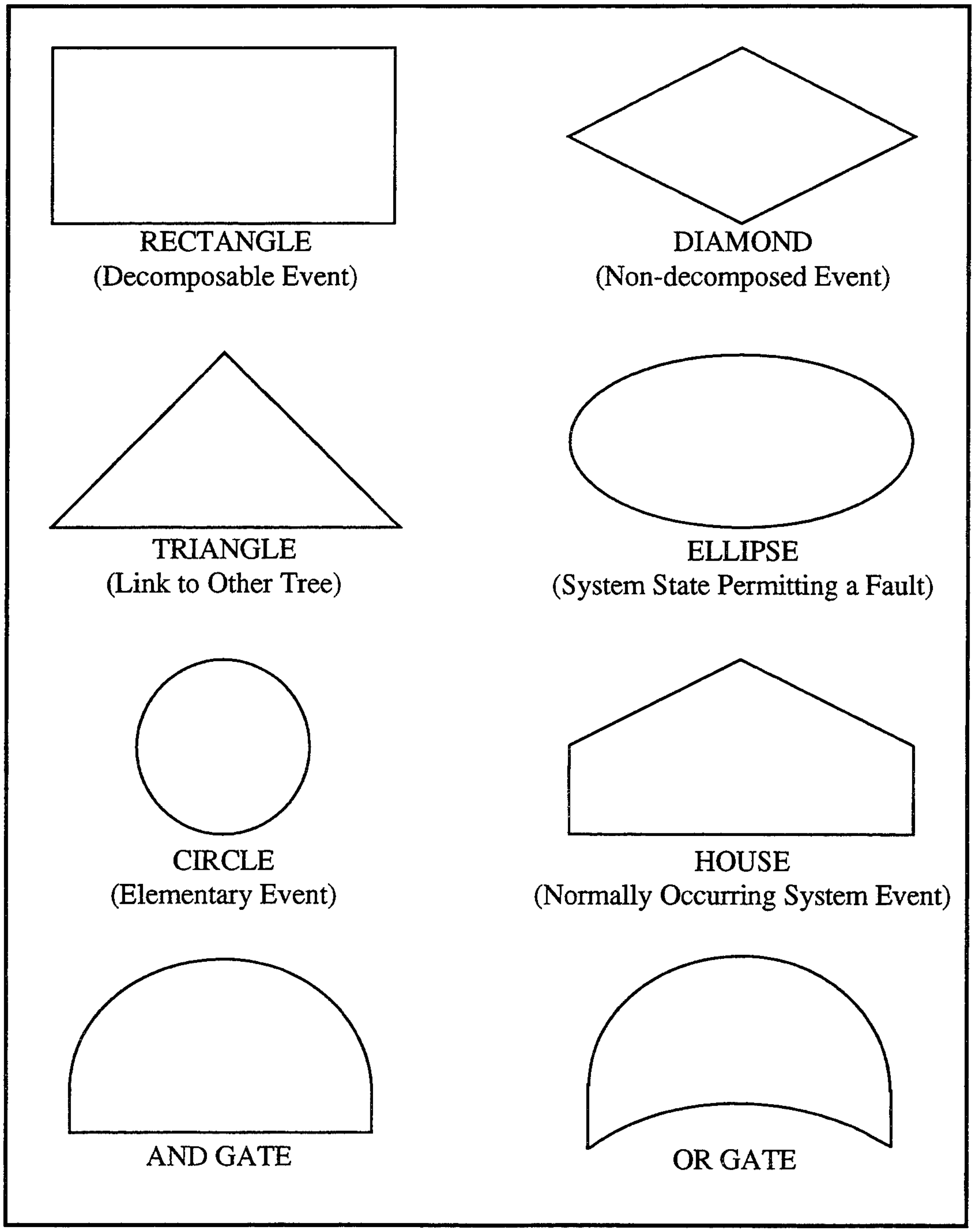


Figure 14: Software Fault Tree Symbols

APPENDIX B. MESA CONTROL CSCI PHA RESULTS

MESA Software Development Plan

SDP Para Ref	SDP Requirement	Possible Hazard	Possible Result	Severity
1.2.2	Control CSCI moves and positions the simulation hardware	Erroneous control signals are generated and sent to the system hardware	Hardware/Personnel damage/injury	High
1.2.2	Control CSCI provides operator interface to control progress of simulation	Operator directs unsafe or incorrect inputs to Control CSCI	Hardware/Personnel damage/injury	High
1.2.2	Control CSCI logs operations for future reference	Control develops erroneous log entries	Log analysis generates future hazardous procedures/actions	High
3.3	Operator has complete override capability over Control CSCI	Operator erroneously overrides Control CSCI	Hardware/Personnel damage/injury	High
3.3	Operator must determine the track in the measurement zone is free of obstructions	Operator misjudges obstructions in free zone	Hardware/Personnel damage/injury	High
3.3	GUI provides audible and visual warnings to operator on impending danger	GUI warning signals fail	Hardware/Personnel damage/injury	High
3.3	All critical data transmissions use checksums	Checksums erroneously calculated	Erroneous control signals generated - system failure, Hardware/Personnel damage/injury	Low to High

Table 10: PHA on Software Development Plan

MESA Software Requirements Specification

SRS Para Ref	SRS Requirement	Possible Hazard	Possible Result	Severity
3.1.1	Control CSCI shall provide for the input of data files from Engagement Generation	Engagement generation computer maintains incorrect data files	Erroneous data collection, undesired engagement parameters resulting in hardware damage	Med to High
3.1.2	Control CSCI shall pass series and run parameters to DAC	DAC maintains incorrect parameters	Damage to sensor, erroneous data collection	High
3.2	Control CSCI shall send Start Motion, Stop Motion, Emergency Stop Motion commands within given time limits	Hardware devices continue to operate/ remain still against operator's request	Damage to hardware, loss of data	High
3.2.2.a.2 -4	Control shall update information, the State Table and the system status/positions at the System Update Rate	Inconsistent and/or erroneous data maintained in System.	Erroneous data collection, hardware devices incorrectly maneuvered resulting in damage.	High
3.2.2.3.a	Control CSCI shall download executable software to the ST, OTS, and Sphere Computers when they power up	HWCI computers fail to execute required program	Damage to hardware, loss of data	High
3.2.2.3.b	Control CSCI shall increment the run number after each engagement	Erroneous run parameters loaded	Unanticipated hardware motion and damage, erroneous data collected	High

Table 11: PHA on Software Requirements Specification

SRS Para Ref	SRS Requirement	Possible Hazard	Possible Result	Severity
3.2.2.3.c	Given a GO signal from operator, Control shall start motion of all related devices not already in motion and reset the retry counter and the motion-fault timer for those devices	Failure to start motion of devices could lead to a collision situation.	Damage to hardware could result from collision, erroneous data collected	High
3.2.2.3.d	Given a STOP signal from operator, Control shall stop motion of all related devices	Failure to stop motion of devices could lead to a collision situation.	Damage to hardware could result from collision, erroneous data collected.	High
3.2.2.3.1 .1.h	Control shall reject sensor beam run mode and Collision Risk combinations that are checked in the given table.	IRS interface incompatibility	IRS interface incompatibility	High
3.2.2.3.1 .3.a	Control shall not permit a move that exceeds the control line tension limits for the specified Target and Sphere control line in automatic mode.	OTS control line tension limit exceeded	OTS control line failure, erratic OTS motion, hardware collision and damage	High
3.2.2.3.1 .3.d	Control shall calculate Target danger zone	Erroneous danger zone calculated	Target and ST collision/damage	High
3.2.2.3.1 .3.1.a.1	Control shall calculate the ST danger zone	Erroneous danger zone calculated	Target and ST collision/damage	High

Table 11: PHA on Software Requirements Specification

SRS Para Ref	SRS Requirement	Possible Hazard	Possible Result	Severity
3.2.2.3.1 .3.1.a.2	Control shall prevent Target danger zone from intersecting with ST danger zone in Automatic mode	Target and ST danger zones overlap	Target and ST collision/damage	High
3.2.2.3.1 .4.1.e	Control shall calculate the "Check Collision" point	Erroneous Collision_Risk calculated	Collision or near miss of ST and Target, hardware damage	High
3.2.2.3.1 .4.4.b.1	When operator clicks forward run (non-collision), Control shall execute operations for a **forward run**: 1)Move ST to start point 2)Calculate ST motion profiles 3)Send run parameters to DAC 4)Program SPG 5)Start ST motion 6)Send idle to DAC when ST stops	Erroneous data calculated/passed to HWCI's Parameters out of limits condition position ST in target danger zone	Collision of ST and Target, hardware damage Collision of ST and/or Target with MESA facility structure	High

Table 11: PHA on Software Requirements Specification

SRS Para Ref	SRS Requirement	Possible Hazard	Possible Result	Severity
3.2.2.3.1 .4.4.b.2	When operator clicks reverse run (non-collision), Control shall execute operations for a **reverse run**: 1)Move ST to start point 2)Calculate ST motion profiles 3)Send run parameters to DAC 4)Program SPG 5)Start ST motion 6)Send idle to DAC when ST stops	Erroneous data calculated/passed to HWCI's Parameters out of limits condition position ST in target danger zone	Collision of ST and Target, hardware damage Collision of ST and/or Target with MESA facility structure	High
3.2.2.3.1 .4.4.b.3	When operator clicks reverse run (collision), Control shall execute operations for a **reverse run (collision)**: 1)Determine AtoD_Start 2)Calculate ST motion profiles 3)Send run parameters to DAC 4)Program SPG 5)Wait 100ms 6)Start ST motion 7)Send idle to DAC when ST stops	Erroneous data calculated/passed to HWCI's Parameters out of limits condition position ST in target danger zone	Collision of ST and Target, hardware damage Collision of ST and/or Target with MESA facility structure	High

Table 11: PHA on Software Requirements Specification

SRS Para Ref	SRS Requirement	Possible Hazard	Possible Result	Severity
3.2.2.3.1 .4.4.c.3	Control shall perform the following operations at the end of a data run: 1)Provide operator a list to choose Observed Conditions 2)For out-of-tolerance (OOT) conditions, accept run sets Set Engagement 3)For no OOT, set Run time Log, trajectory and engagement info.	Erroneous Run-time Log entry Erroneous next step set: Next engagement vice Next trajectory and vice versa.	Erroneous run-time reports, unneeded procedural changes made Collision of ST and Target, hardware damage on next run	High
3.2.2.3.1 .4.5	Control shall retrieve Next Engagement	Erroneous Engagement retrieved	Induces operator error causing collision of ST and/or Target with MESA facility structure	High
3.2.2.3.1 .4.7	When Next Trajectory selected, Control shall get next trajectory, display engagement conditions, set engagement option.	Erroneous next trajectory conditions retrieved	Collision of ST and Target with MESA facility structure, hardware damage on next run	High
3.2.2.3.1 .4.8.a.1	Control shall generate warning message for invalid next trajectory.	Failure to generate warning message, erroneous trajectory conditions used	Collision of ST and Target with MESA facility structure, hardware damage on next run	High

Table 11: PHA on Software Requirements Specification

SRS Para Ref	SRS Requirement	Possible Hazard	Possible Result	Severity
3.2.2.3.1 .4.8.a.2	Control shall pass trajectory number and engagement parameters to GUI.	Erroneous read of engagement parameters from engagement file	Collision of ST and Target with MESA facility structure, hardware damage on next run	High
3.2.2.3.2 .1.a	When EMER STOP pressed, Control shall 1)Stop all motion 2)Generate fault log 3)Request and enter operator comment	Stop motion commands erroneously or not at all generated	Collision of ST and Target with MESA facility structure, hardware damage on next run Personnel injury	High
3.2.2.3.2 .1.b	When ABORT pressed, Control shall 1)Stop ST 2)Request and enter operator comment	Stop motion commands to ST erroneously or not at all generated	Collision of ST and Target with MESA facility structure, hardware damage on next run	High
3.2.2.3.2 .1.c	When Hardware EMER STOP pressed, Control shall 1)Stop all devices 2)Generate fault log 3)Request and enter operator comment	Stop motion commands to ST erroneously or not at all generated	Collision of ST and Target with MESA facility structure, hardware damage on next run Personnel injury	High
3.2.2.3.2 .2.b	Control shall stop device motion when software limit exceeded.	Stop motion commands to ST erroneously or not at all generated	Collision of ST and Target with MESA facility structure, hardware damage on next run Personnel injury	High

Table 11: PHA on Software Requirements Specification

SRS Para Ref	SRS Requirement	Possible Hazard	Possible Result	Severity
3.2.2.3.2 .2.c	Control shall not command ST to speeds in excess of ST speed zones.	Erroneous speed commands generated	Collision of ST and Target with MESA facility structure, hardware damage on next run	High
3.2.2.3.2 .2.d	When main hoist software limit detected, Control shall 1)Stop all devices in motion 2)Generate fault log 3)Request and enter operator comment	Stop motion commands to devices erroneously or not at all generated	Collision of ST and Target with MESA facility structure, hardware damage on next run	High

Table 11: PHA on Software Requirements Specification

APPENDIX C. GENERATED FAULT TREES AND FAULT DESCRIPTION LISTINGS

1. Top-Level Specific Hazard Faults

2. Software Starting Root Node Faults

3. Encoder Line Breaks Analysis Faults

Node Label	Fault Description
impact	Sphere impacts arena
ctln brk	Control line breaks
control	Control causes sphere impact
wall near	Sphere is close to arena structure
motion	Controlled motion causes sphere impact
sw fail	Software commands motion causing sphere impact
init err	Initial position set in software causes erroneous motion
check er	Initialization check fails to find data incorrect
data err	Wrong initialization data provided to software
value er	Undesired movement value generated
algr err	Closed loop position algorithm generates bad values
track er	Invalid incremental movement calculation
tr reg er	Encoder line tracking not registered in software
ecdr brk	Encoder line break causes continued motion
hw fail	Hardware Failure produces motion causing sphere impact

Table 12: Sphere Impacts Arena Fault Description Listing

Node Label	Fault Description
impact	Sphere impacts arena
cntrl brk	Control line breaks
control	Control causes sphere impact
near obj	Sphere is close to arena structure
motion	Controlled motion causes sphere impact
sw fail	Software commands motion causing sphere impact
init err	Initial position set in software causes erroneous motion
man hook	Manual positioning causes erroneous motion
sw err	Initial position set in software causes erroneous motion
data err	Wrong initialization data provided to software
check er	Initialization data fails to find data incorrect
value er	Undesired movement value generated
algr err	Closed loop position algorithm generates bad values
track er	Invalid incremental movement calculation
tr reg er	Encoder line tracking not registered in software
ecdr brk	Encoder line break causes continued motion
hw fail	Hardware Failure produces motion causing sphere impact
obj mot	Object motion other than sphere causes impact
target	Target motion causes impact
st	Sensor transport motion causes impact
sensor	Actual sensor motion causes impact
other	Other object motion causes impact

Table 13: Sphere Impacts Object Other Than Arena Fault Description Listing

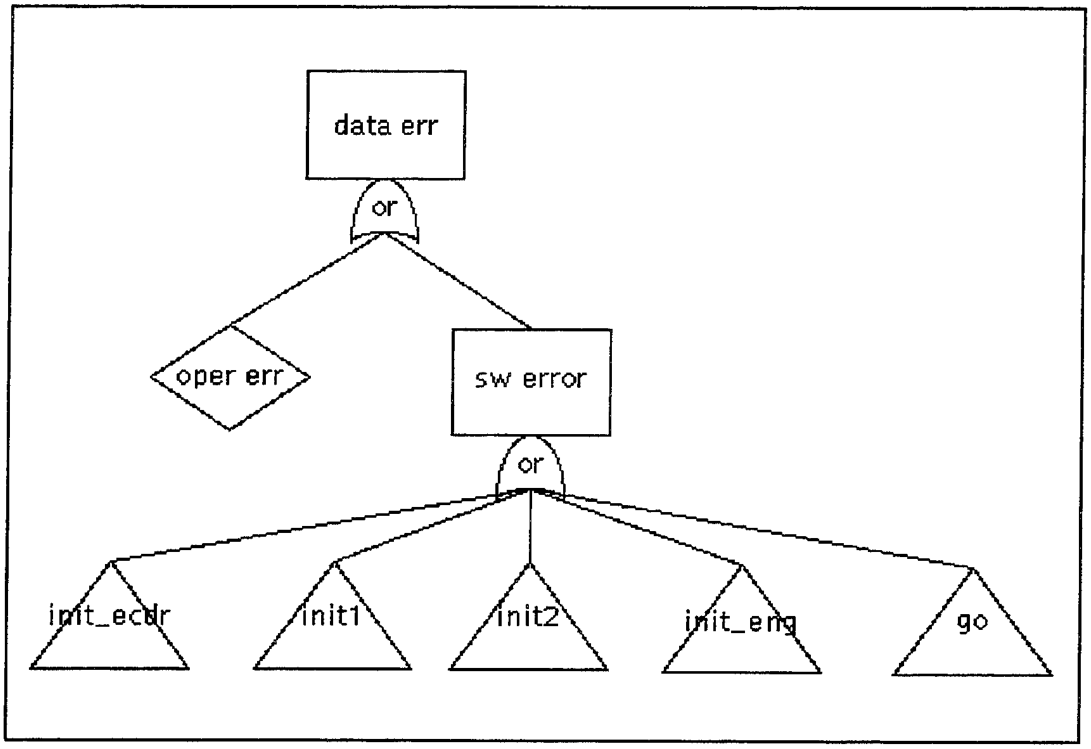


Figure 15: Data Error Fault Tree

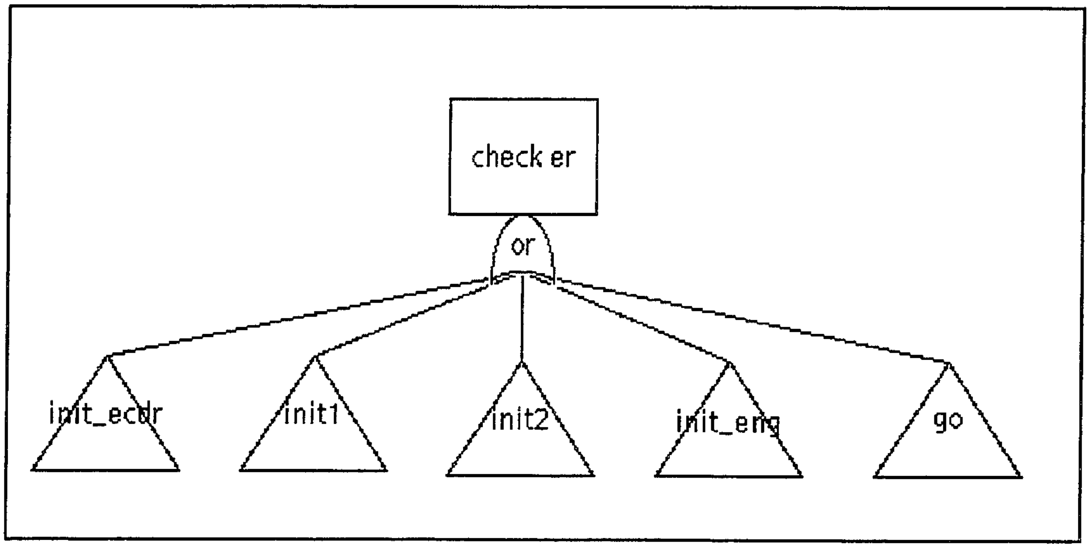


Figure 16: Check Error Fault Tree

Node Label	Fault Description
data err	Wrong initialization data provided to software
oper err	Operator manually inputs erroneous initialization data
sw error	Software generates erroneous initialization data
init_ecdr	Initialize_Encoders_History procedure in Remote_sphere.a causes fault
init1	Initialize_1 procedure in Remote_sphere.a causes fault
init2	Initialize_2 procedure in Remote_sphere.a causes fault
init_eng	Initial_Engagement_Conditions procedure in Remote_sphere.a causes fault
go	Go procedure in Remote_sphere.a causes fault

Table 14: Data Error Fault Description Listing

Node Label	Fault Description
check er	Initialization check fails to find data incorrect
init_ecdr	Initialize_Encoders_History procedure in Fault_monitor_s.a causes fault
init2	Initialize_2 procedure in Remote_sphere.a causes fault
init1	Initialize_1 procedure in Remote_sphere.a causes fault
init_eng	Initial_Engagement_Conditions procedure in Remote_sphere.a causes fault
go	Go procedure in Remote_sphere.a causes fault

Table 15: Check Error Fault Description Listing

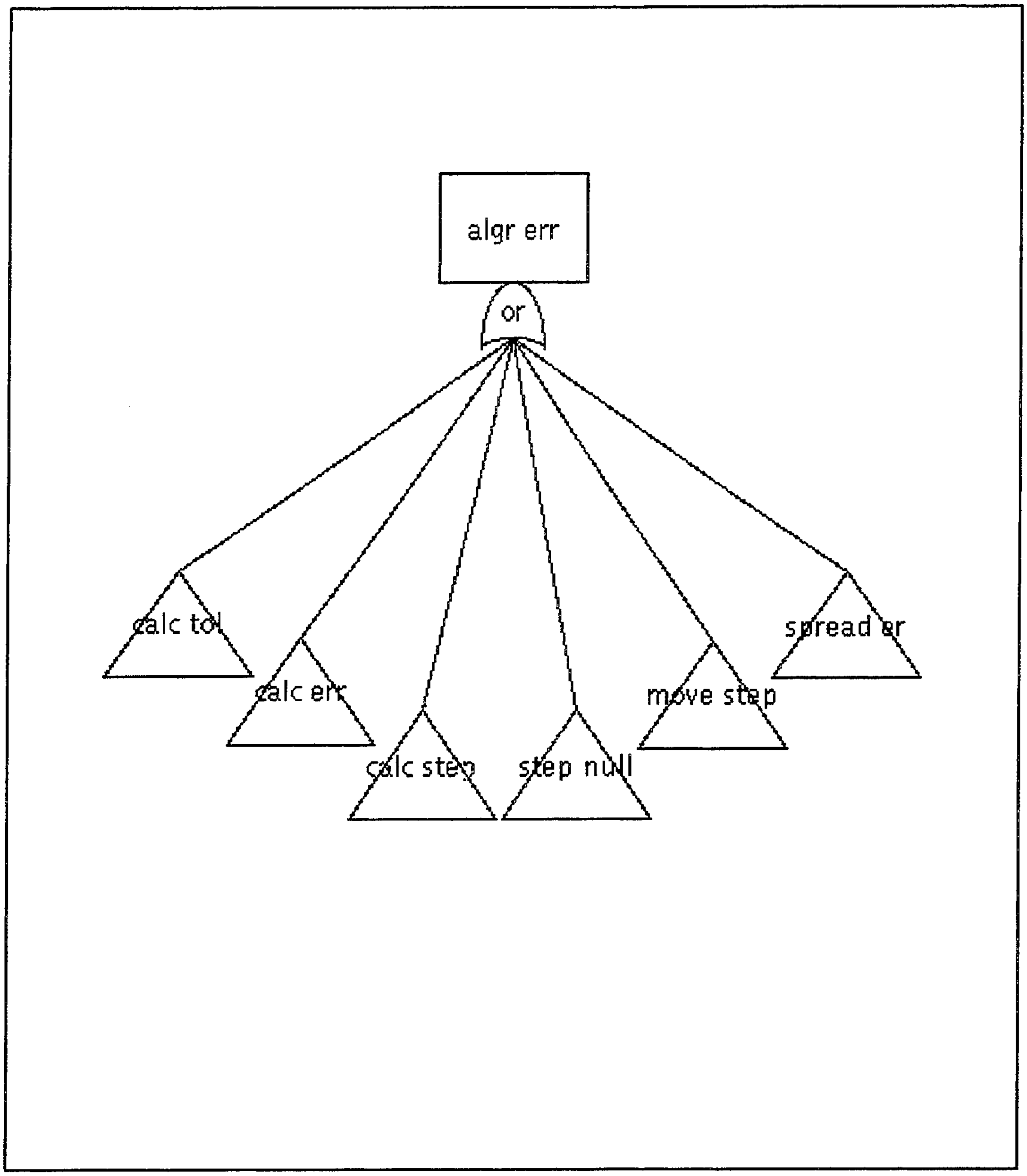


Figure 17: Algorithm Error Fault Tree

Node Label	Fault Description
algr err	Closed loop position algorithm generates bad values
calc tol	Calc_Within_Tolerance procedure causes fault
calc err	Calc_Errors procedure causes fault
calc step	Calc_Steps procedure causes fault
step null	Steps_Or_Null procedure causes fault
move step	Move_Steppers procedure causes fault
spread err	Spread_Of_Errors_During_Settle_And_Hold_Times procedure causes fault

Table 16: Algorithm Error Fault Description Listing

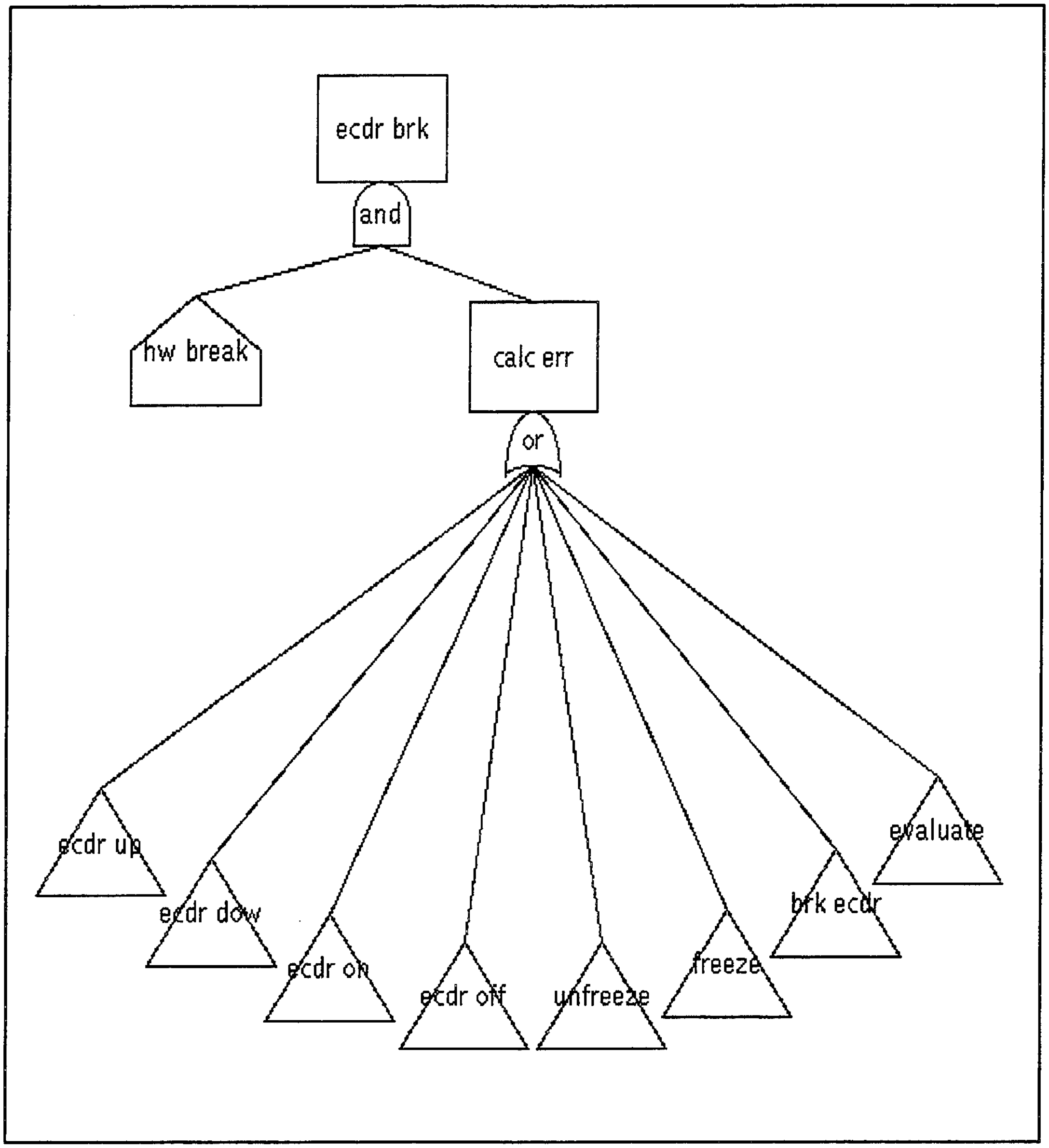


Figure 18: Encoder Line Break Error Fault Tree

Node Label	Fault Description
ecdr brk	Encoder line break causes continued motion
hw break	Hardware causes encoder line to break
calc err	Software generates erroneous data causing encoder line to break
ecdr off	Encoders_Off procedure in Digital_output_s.a causes fault
unfreeze	Unfreeze_Encoder_Readings procedure in Digital_output_s.a causes fault
freeze	Freeze_Encoder_Readings procedure in Digital_output_s.a causes fault
brk ecdr	Broken_Encoder_Line procedure in Fault_Monitor_s.a causes fault
evaluate	Evaluate procedure in Evaluate_s.a causes fault
ecdr on	Encoders_On procedure in Digital_output_s.a causes fault
ecdr down	Encoders_Down procedure in Digital_output_s.a causes fault
ecdr up	Encoders_Up procedure in Digital_output_s.a causes fault

Table 17: Encoder Line Break Error Fault Description Listing

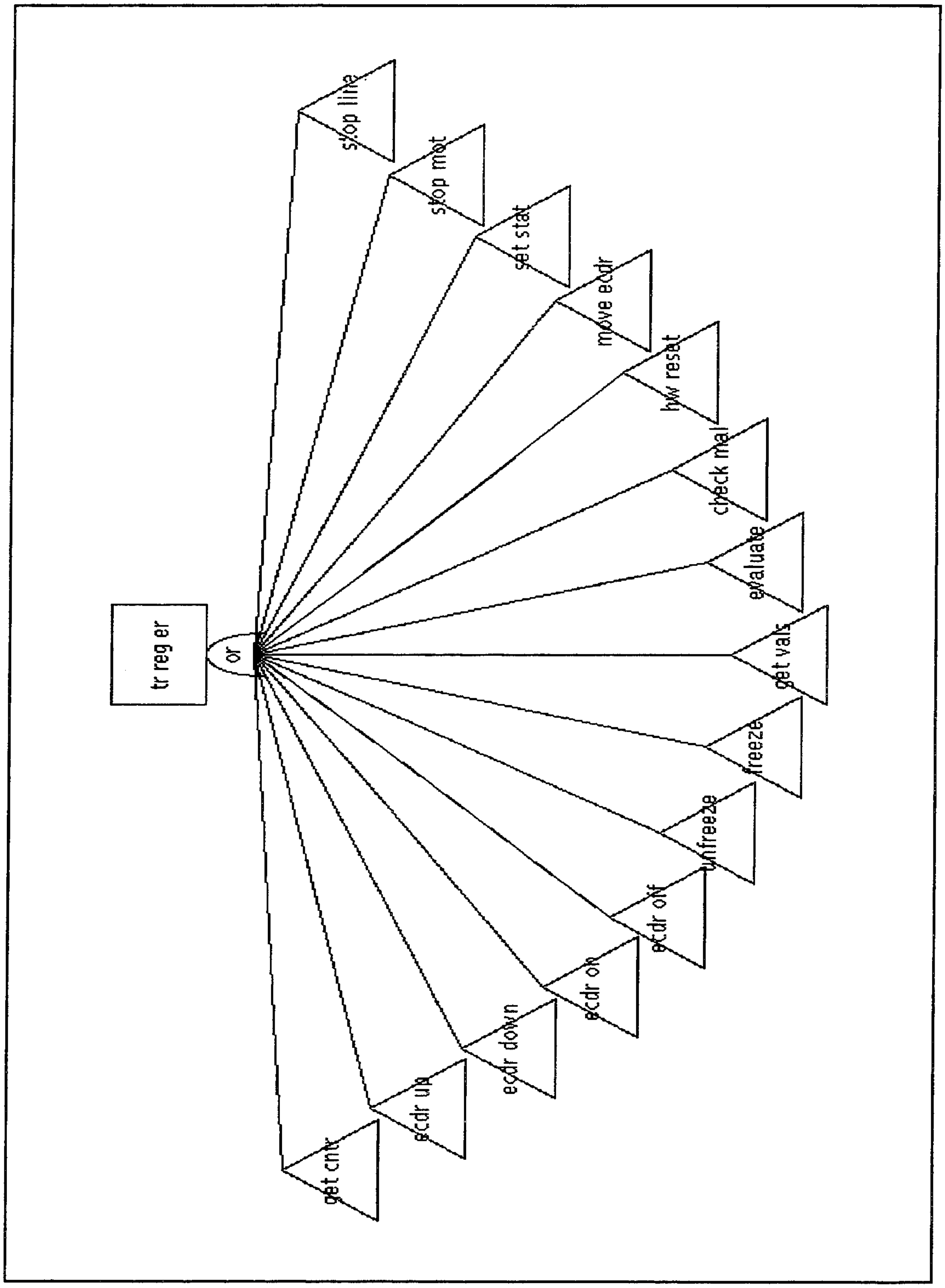


Figure 19: Encoder Line Tracking Error Fault Tree

Node Label	Fault Description
tr reg er	Encoder line tracking not registered in software
get cntr	Get_Control_Words procedure in Digital_input_s.a causes fault
ecdr up	Encoders_Up procedure in Digital_output_s.a causes fault
ecdr down	Encoders_Down procedure in Digital_output_s.a causes fault
ecdr on	Encoders_On procedure in Digital_output_s.a causes fault
ecdr off	Encoders_Off procedure in Digital_output_s.a causes fault
unfreeze	Unfreeze_Encoder_Readings procedure in Digital_output_s.a causes fault
freeze	Freeze_Encoder_Readings procedure in Digital_output_s.a causes fault
get vals	Get_Values procedure in Encoders_b.a causes fault
evaluate	Evaluate procedure in Evaluate_s.a causes fault
check mal	Check_For_Malfunction procedure in Fault_monitor_s.a causes fault
hw reset	HW_Reset procedure in Hw_reset.a causes fault
move ecdr	Move_Encoder_Line procedure in Remote_sphere.a causes fault
set stat	Set_State procedure in Remote_sphere.a causes fault
stop mot	Stop_Motion procedure in Remote_sphere.a causes fault
stop line	Stop_All_Lines procedure in Remote_sphere.a causes fault

Table 18: Encoder Line Tracking Error Fault Description Listing

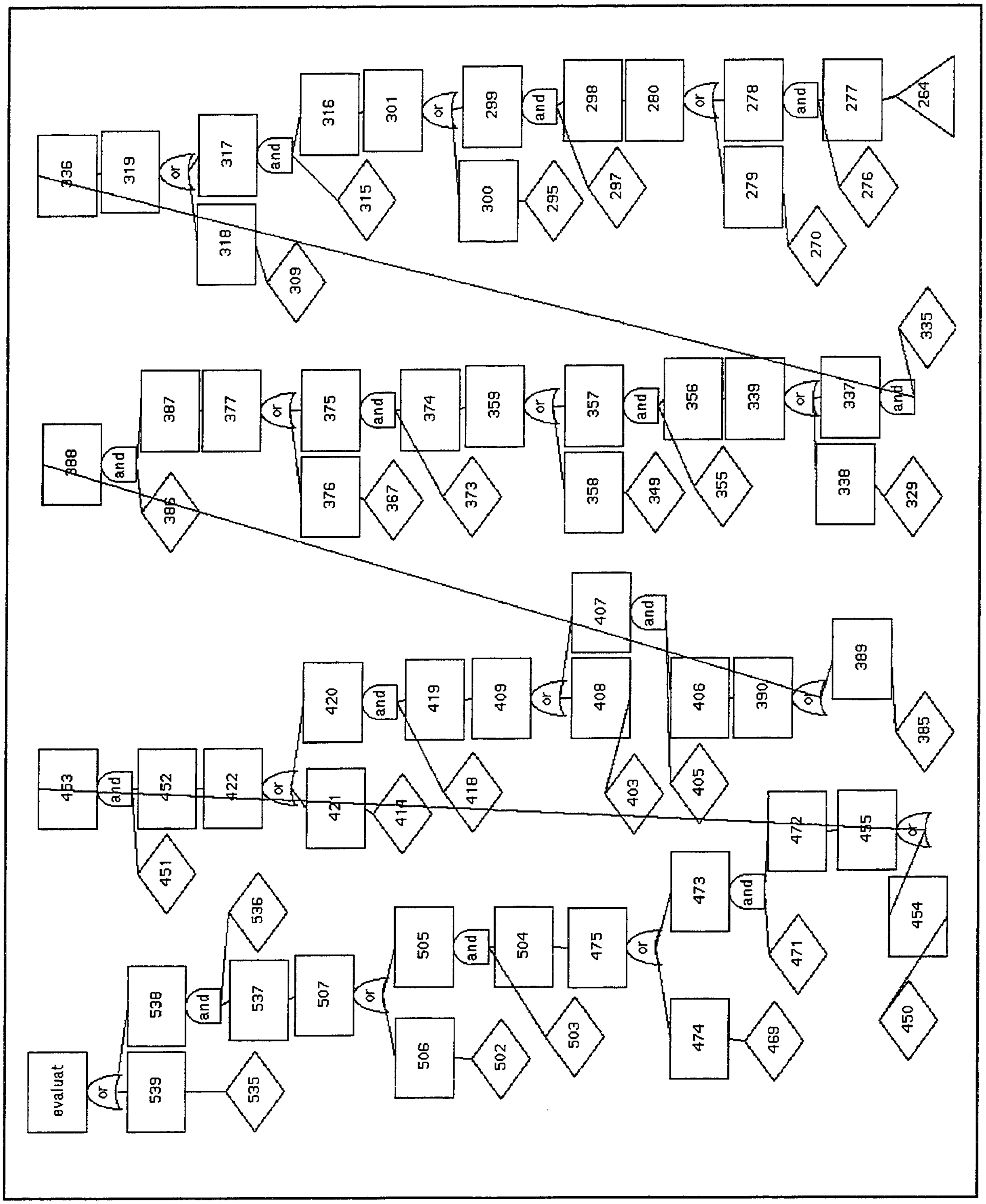


Figure 20: Evaluate Sub-Tree Level One

Node Label	Fault Description
evaluate	Sequence of statements causes Fault
539	Last statement causes Fault
535	If statement causes Fault
538	Previous statements causes Fault
536	Last Statement did not mask Fault
537	Sequence prior to last causes Fault
507	Sequence of statements causes Fault
506	Last statement causes Fault
502	If statement causes Fault
505	Previous statements causes Fault
503	Last Statement did not mask Fault
504	Sequence prior to last causes Fault
475	Sequence of statements causes Fault
474	Last statement causes Fault
469	Procedure call causes Fault
473	Previous statements causes Fault
471	Last Statement did not mask Fault
472	Sequence prior to last causes Fault
455	Sequence of statements causes Fault
454	Last statement causes Fault
450	If statement causes Fault
453	Previous statements causes Fault
451	Last Statement did not mask Fault

Table 19: Evaluate Sub-Tree Level One Fault Description Listing

Node Label	Fault Description
452	Sequence prior to last causes Fault
422	Sequence of statements causes Fault
421	Last statement causes Fault
414	Assignment Statement causes Fault
420	Previous statements causes Fault
418	Last Statement did not mask Fault
419	Sequence prior to last causes Fault
409	Sequence of statements causes Fault
408	Last statement causes Fault
403	Procedure call causes Fault
407	Previous statements causes Fault
405	Last Statement did not mask Fault
406	Sequence prior to last causes Fault
390	Sequence of statements causes Fault
389	Last statement causes Fault
385	Procedure call causes Fault
388	Previous statements causes Fault
386	Last Statement did not mask Fault
387	Sequence prior to last causes Fault
377	Sequence of statements causes Fault
376	Last statement causes Fault
367	Assignment Statement causes Fault
375	Previous statements causes Fault
373	Last Statement did not mask Fault
374	Sequence prior to last causes Fault

Table 19: Evaluate Sub-Tree Level One Fault Description Listing

Node Label	Fault Description
359	Sequence of statements causes Fault
358	Last statement causes Fault
349	Assignment Statement causes Fault
357	Previous statements causes Fault
355	Last Statement did not mask Fault
356	Sequence prior to last causes Fault
339	Sequence of statements causes Fault
338	Last statement causes Fault
329	Assignment Statement causes Fault
337	Previous statements causes Fault
335	Last Statement did not mask Fault
336	Sequence prior to last causes Fault
319	Sequence of statements causes Fault
318	Last statement causes Fault
309	Assignment Statement causes Fault
317	Previous statements causes Fault
315	Last Statement did not mask Fault
316	Sequence prior to last causes Fault
301	Sequence of statements causes Fault
300	Last statement causes Fault
295	Procedure call causes Fault
299	Previous statements causes Fault
297	Last Statement did not mask Fault
298	Sequence prior to last causes Fault
280	Sequence of statements causes Fault

Table 19: Evaluate Sub-Tree Level One Fault Description Listing

Node Label	Fault Description
279	Last statement causes Fault
270	Assignment Statement causes Fault
278	Previous statements causes Fault
276	Last Statement did not mask Fault
277	Sequence prior to last causes Fault
264	Sequence of statements causes Fault

Table 19: Evaluate Sub-Tree Level One Fault Description Listing

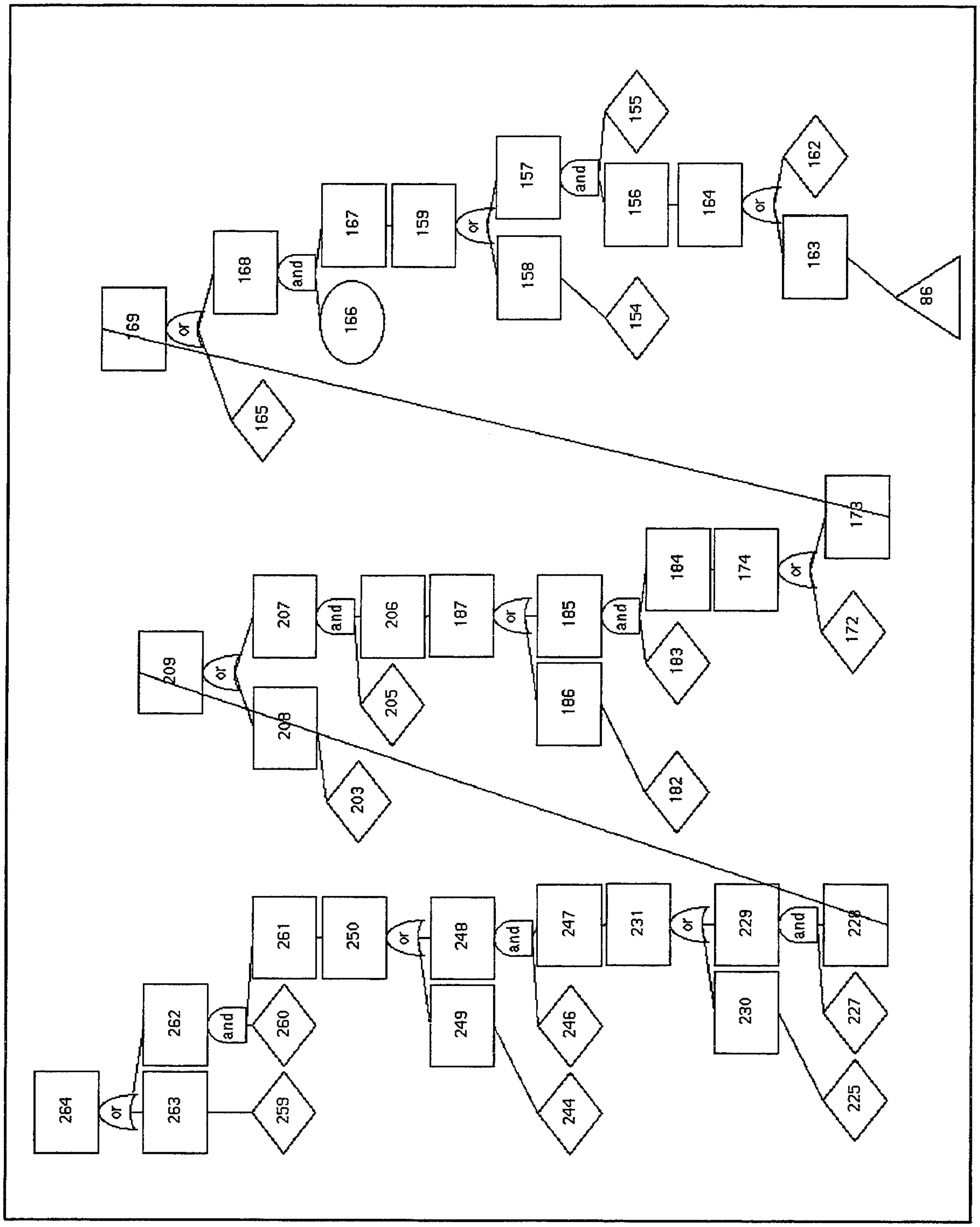


Figure 21: Evaluate Sub-Tree Level Two

Node Label	Fault Description
264	Sequence of statements causes Fault
263	Last statement causes Fault
259	Procedure call causes Fault
262	Previous statements causes Fault
260	Last Statement did not mask Fault
261	Sequence prior to last causes Fault
250	Sequence of statements causes Fault
249	Last statement causes Fault
244	Procedure call causes Fault
248	Previous statements causes Fault
246	Last Statement did not mask Fault
247	Sequence prior to last causes Fault
231	Sequence of statements causes Fault
230	Last statement causes Fault
225	Procedure call causes Fault
229	Previous statements causes Fault
227	Last Statement did not mask Fault
228	Sequence prior to last causes Fault
209	Sequence of statements causes Fault
208	Last statement causes Fault
203	Procedure call causes Fault
207	Previous statements causes Fault
205	Last Statement did not mask Fault

Table 20: Evaluate Sub-Tree Level Two Fault Description Listing

Node Label	Fault Description
206	Sequence prior to last causes Fault
187	Sequence of statements causes Fault
186	Last statement causes Fault
182	Procedure call causes Fault
185	Previous statements causes Fault
183	Last Statement did not mask Fault
184	Sequence prior to last causes Fault
174	Sequence of statements causes Fault
173	Last statement causes Fault
169	If statement causes Fault
165	Evaluation of condition causes Fault
168	Condition true and statements causes Fault
166	If condition true
167	Then statements causes Fault
159	Sequence of statements causes Fault
158	Last statement causes Fault
154	If statement causes Fault
157	Previous statements causes Fault
155	Last Statement did not mask Fault
156	Sequence prior to last causes Fault
164	Sequence of statements causes Fault
163	Last statement causes Fault
86	If statement causes Fault
162	Previous statements causes Fault
172	Previous statements causes Fault

Table 20: Evaluate Sub-Tree Level Two Fault Description Listing

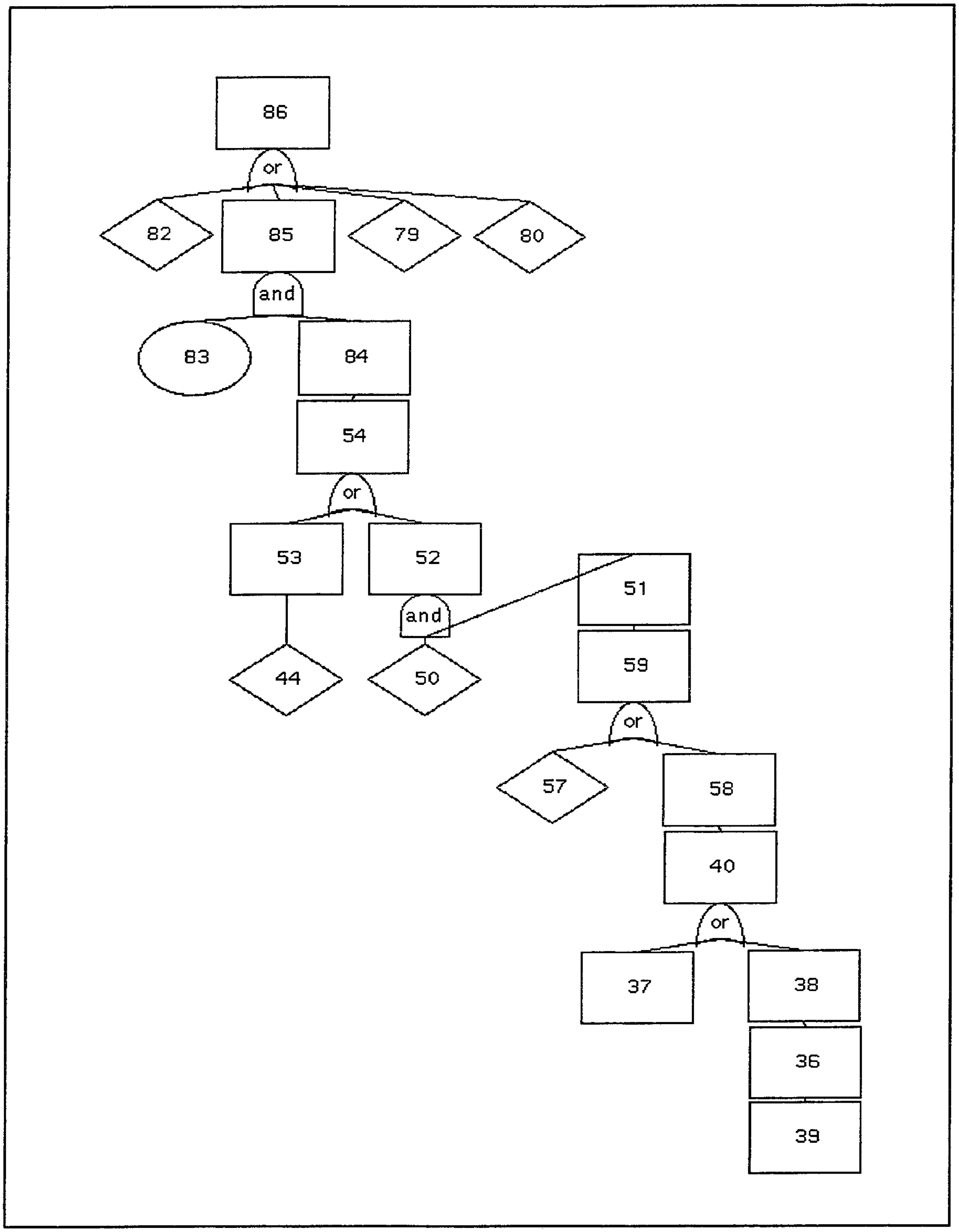


Figure 22: Evaluate Sub-Tree Level Three

Node Label	Fault Description
86	If statement causes Fault
82	Evaluation of condition causes Fault
85	Condition true and statements causes Fault
83	If condition true
84	Then statements causes Fault
54	Sequence of statements causes Fault
53	Last statement causes Fault
44	Assignment Statement causes Fault
52	Previous statements causes Fault
50	Last Statement did not mask Fault
51	Sequence prior to last causes Fault
59	Sequence of statements causes Fault
58	Last statement causes Fault
40	Procedure call causes Fault
37	Procedure elaboration causes Fault
38	Procedure body causes Fault
36	Broken_Encoder_Line
39	Procedure not found on table
57	Previous statements causes Fault
79	ELSE part causes Fault
80	Action by other task on variable causes Fault

Table 21: Evaluate Sub-Tree Level Three Fault Description Listing

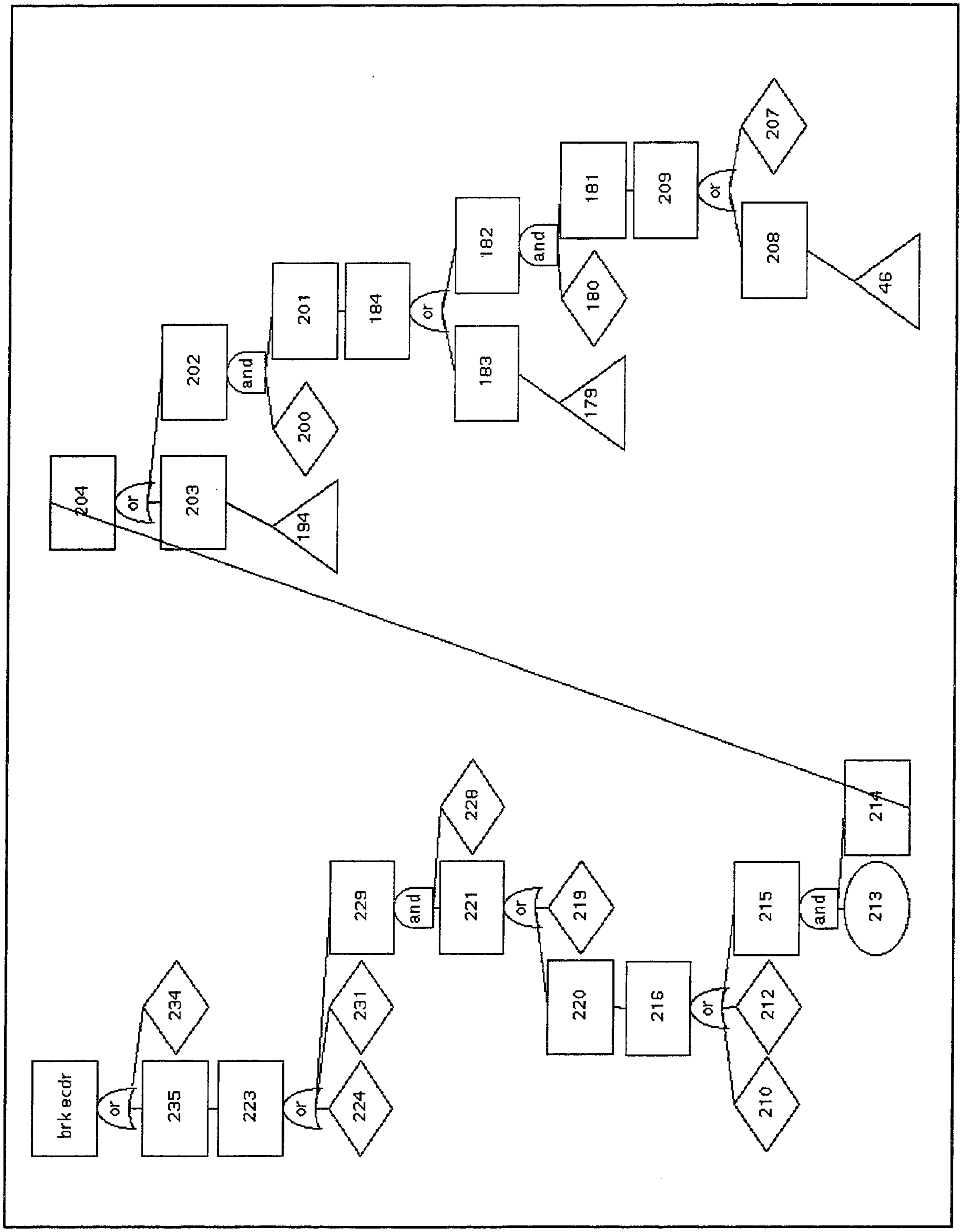


Figure 23: Brk Ecdr Root Sub-Tree

Node Label	Fault Description
brk ecdr	Sequence of statements causes Fault
235	Last statement causes Fault
223	Loop Statement causes Fault
224	Loop never executed
231	Loop condition evaluation causes Fault
229	Nth Iteration causes Fault
221	Sequence of statements causes Fault
220	Last statement causes Fault
216	If statement causes Fault
212	Evaluation of condition causes Fault
215	Condition true and statements causes Fault
213	If condition true
214	Then statements causes Fault
204	Sequence of statements causes Fault
203	Last statement causes Fault
194	Assignment statement causes fault
202	Previous statements causes Fault
200	Last Statement did not mask Fault
201	Sequence prior to last causes Fault
184	Sequence of statements causes Fault
183	Last statement causes Fault
179	If statement causes fault
182	Previous statements causes Fault

Table 22: Brk Ecdr Root Sub-Tree Fault Description Listing

Node Label	Fault Description
180	Last Statement did not mask Fault
181	Sequence prior to last causes Fault
209	Sequence of statements causes Fault
208	Last statement causes Fault
46	Assignment Statement causes Fault
207	Previous statements causes Fault
210	Action by other task on variable causes Fault
219	Previous statements causes Fault
228	Condition true past n-1
234	Previous statements causes Fault

Table 22: Brk Ecdr Root Sub-Tree Fault Description Listing

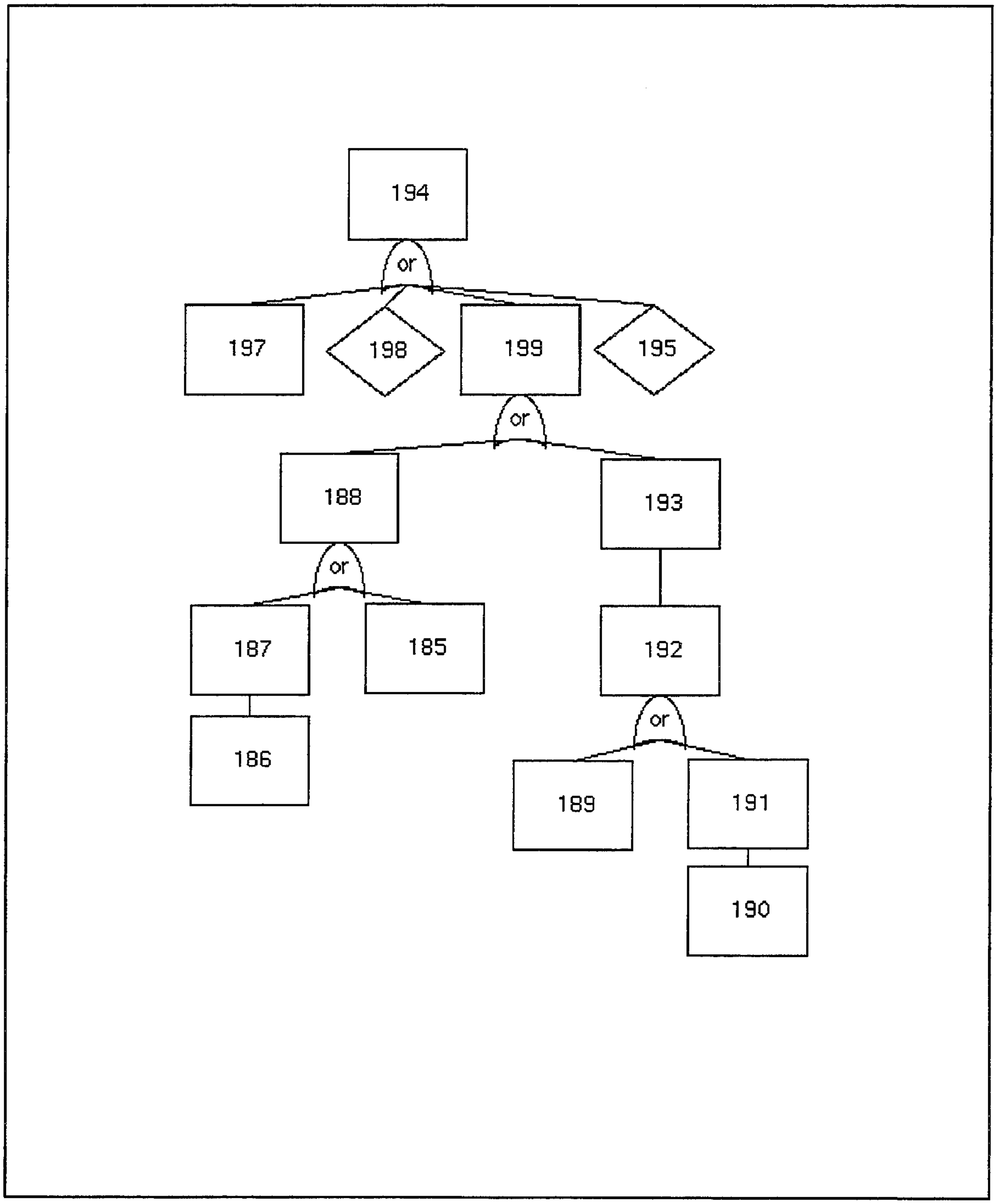


Figure 24: Brk Ecdr Node 194 Sub-Tree

Node Label	Fault Description
194	Assignment Statement causes Fault
197	Change in values causes Fault
198	Exception causes Fault
199	Operand Evaluation causes Fault
188	Indexed Component causes Fault
185	Encoders_3000ms_Ago
187	Relation causes Fault
186	Line_Number
193	Relation causes Fault
192	Indexed Component causes Fault
189	Encoders_Current
191	Relation causes Fault
190	Line_Number
195	Action by other task on variable causes Fault

Table 23: Brk Ecdr Node 194 Sub-Tree Fault Description Listing

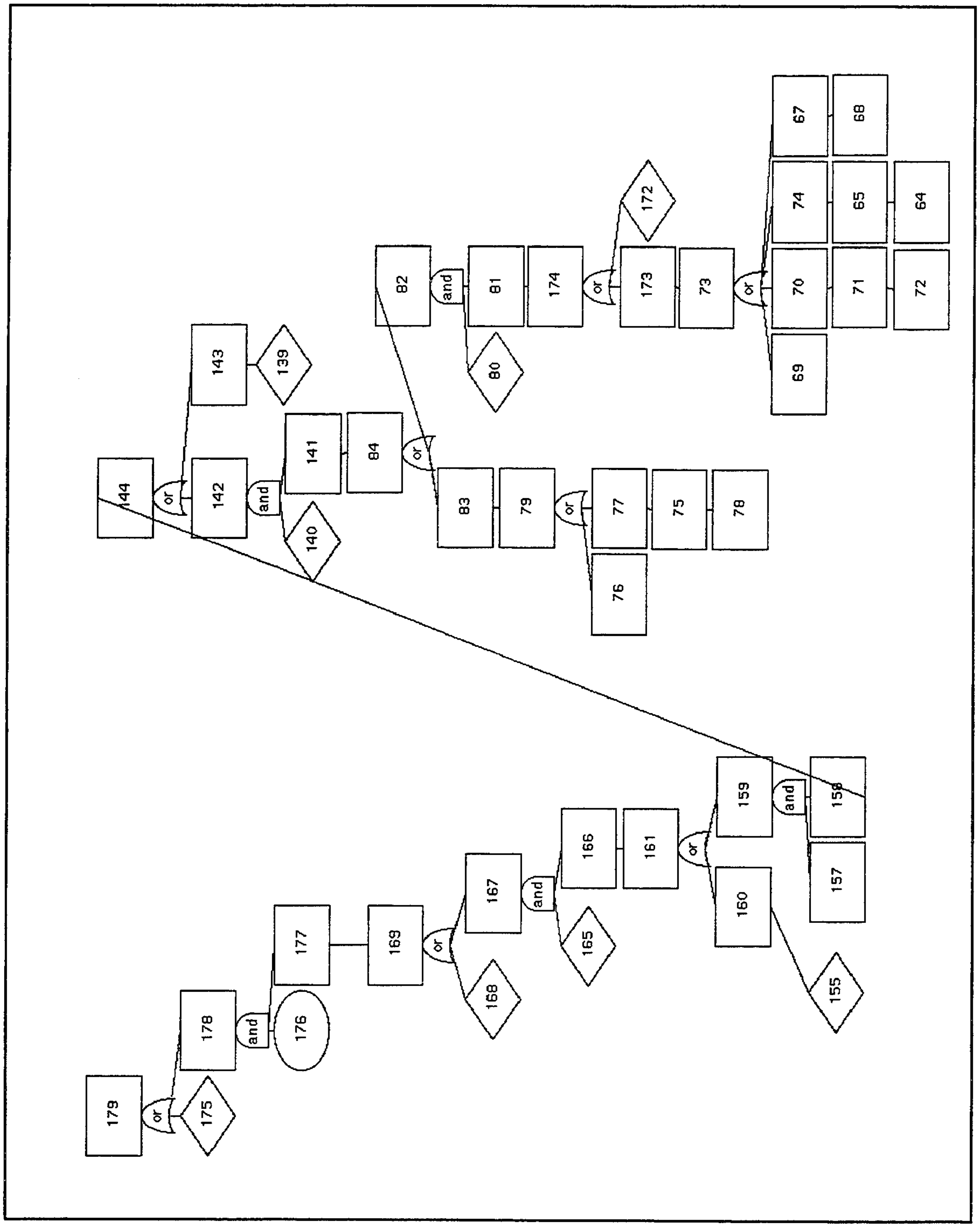


Figure 25: Brk Ecdr Node 179 Sub-Tree

Node Label	Fault Description
179	If statement causes Fault
175	Evaluation of condition causes Fault
178	Condition true and statements causes Fault
176	If condition true
177	Then statements causes Fault
169	Sequence of statements causes Fault
168	Last statement causes Fault
167	Previous statements causes Fault
165	Last Statement did not mask Fault
166	Sequence prior to last causes Fault
161	Sequence of statements causes Fault
160	Last statement causes Fault
155	Procedure call causes Fault
159	Previous statements causes Fault
157	Last Statement did not mask Fault
158	Sequence prior to last causes Fault
144	Sequence of statements causes Fault
143	Last statement causes Fault
139	If statement causes Fault
142	Previous statements causes Fault
140	Last Statement did not mask Fault
141	Sequence prior to last causes Fault
84	Sequence of statements causes Fault

Table 24: Brk Ecdr Node 179 Sub-Tree Fault Description Listing

Node Label	Fault Description
83	Last statement causes Fault
79	Procedure call causes Fault
76	Procedure elaboration causes Fault
77	Procedure body causes Fault
75	Stop_Motion
78	Procedure not found on table
82	Previous statements causes Fault
80	Last Statement did not mask Fault
81	Sequence prior to last causes Fault
174	Sequence of statements causes Fault
173	Last statement causes Fault
73	Procedure call causes Fault
69	Procedure elaboration causes Fault
70	Procedure body causes Fault
71	Off
72	Procedure not found on table
74	Parameter evaluation causes Fault
65	Relation causes Fault
64	Line_Number
67	Action by other task causes Fault
68	Line_Number
172	Previous statements causes Fault

Table 24: Brk Ecdr Node 179 Sub-Tree Fault Description Listing

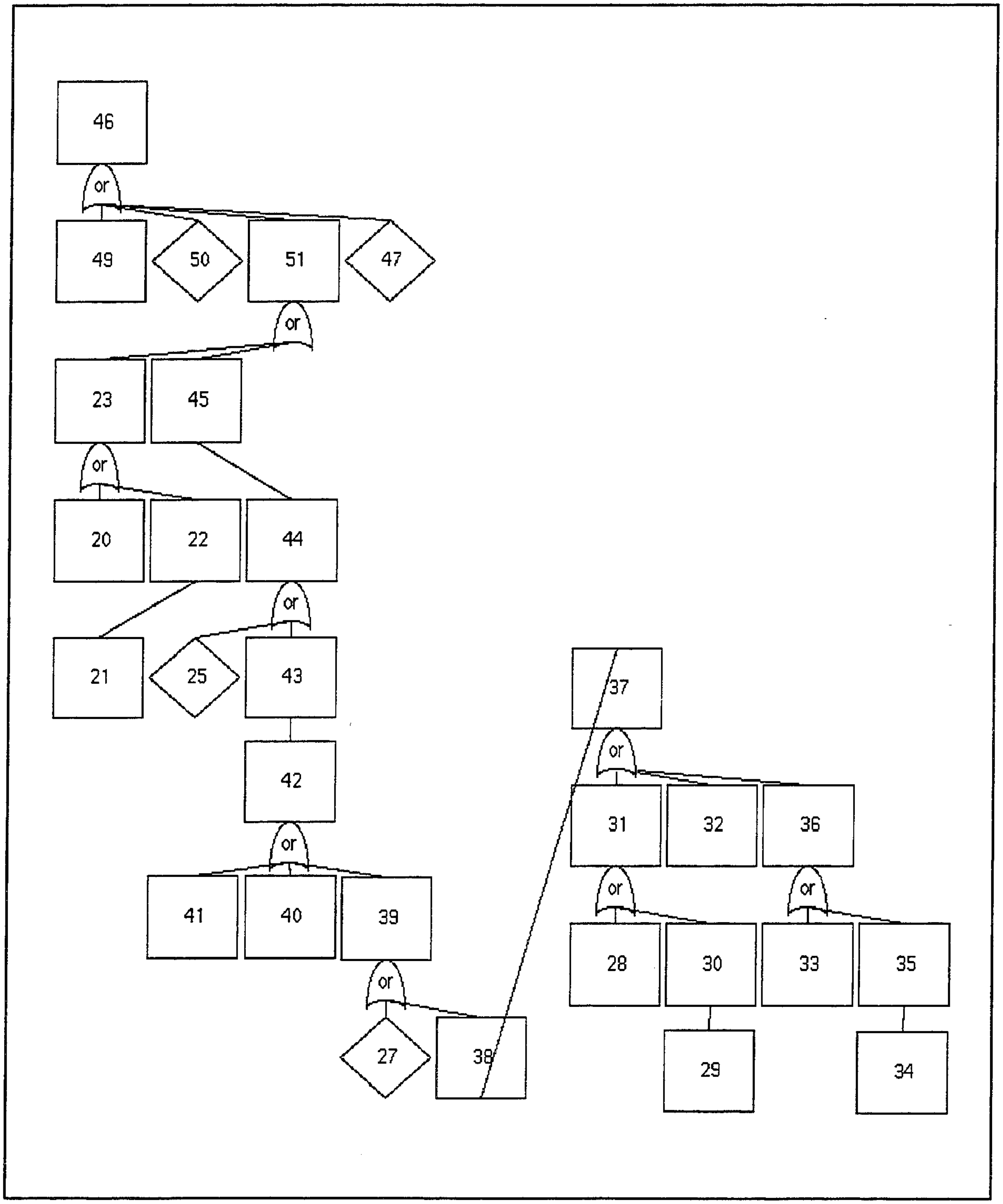


Figure 26: Brk Ecdr Node 46 Sub-Tree

Node Label	Fault Description
46	Assignment Statement causes Fault
49	Change in values causes Fault
50	Exception causes Fault
51	Operand Evaluation causes Fault
23	Indexed Component causes Fault
20	Encoder_Speed
22	Relation causes Fault
21	Line_Number
45	Relation causes Fault
44	Indexed Component causes Fault
25	Counts_Unsigned
43	Relation causes Fault
42	Division or Multiplication causes Fault
39	Indexed Component causes Fault
27	Real
38	Relation causes Fault
37	Addition or Subtraction causes Fault
31	Indexed Component causes Fault
28	Encoders_3000ms_Ago
30	Relation causes Fault
29	Line_Number
32	Subtraction causes Fault
36	Indexed Component causes Fault

Table 25: Brk Ecdr Node 46 Sub-Tree Fault Description Listing

Node Label	Fault Description
33	Encoders_Current
35	Relation causes Fault
34	Line_Number
41	Enc_Line_Error_Time
40	Division By Zero causes Fault
47	Action by other task on variable causes Fault

Table 25: Brk Ecdr Node 46 Sub-Tree Fault Description Listing

LIST OF REFERENCES

1. Leveson, Nancy G., "Software Safety: Why, What and How", *ACM Computing Surveys*, Volume. 18, No.2, pp. 125-163, June 1986.

2. Reid Jr., William S., Software Fault Tree Analysis of Concurrent Ada Processes, M.S. Thesis, Naval Postgraduate School, Monterey, CA, September 1994.

3. Kopetz, H., "Software Reliability", Springer-Verlag New York Inc., 1979.

4. Heimerdinger, W. and Weinstock, C., "A Conceptual Framework for System Fault Tolerance", *Technical Report CMU/SEI-92-TR-33 Carnegie Mellon University*, October 1992.

5. Nelson, V., "Fault-Tolerant Computing: Fundamental Concepts", *IEEE Computer*, Volume. 23, No.7, pp. 19-25, July 1990.

6. Anderson, T. and Lee, P., "Fault Tolerant Principles and Practices", Princeton Hall Books, 1981.

7. Casey, Steven, "Set Phasers on Stun and Other Tales of Design, Technology and Human Error", Aegean Publishing Company, 1993.

8. Place, Patrick R.H., and Kang, Kyo C., "Safety-Critical Software: Status Report and Annotated Bibliography", *Technical Report CMU/SEI-92-TR-5 Carnegie Mellon University*, June 1993.

9. Ordonio, Robert R., An Automated Tool to Facilitate Code Translation for Software Fault Tree Analysis, M.S. Thesis, Naval Postgraduate School, Monterey, CA, September 1993.

10. Mason, Russell W., Fault Isolator Tool for Software Fault Tree Analysis, M.S. Thesis, Naval Postgraduate School, Monterey, CA, March 1995.

11. Leveson, Nancy G. and Stolzy, J.L., "Safety Analysis Using Petri Nets", *IEEE Transactions on Software Engineering*, Volume SE-13, No.3, pp.386-397, March 1987.

12. Harel, David, "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming Paper*, July 1986.

13. Noble, W. B., "Developing Safe Software for Critical Airborne Applications", *Proceedings of the IEEE 6th Digital Avionics Systems Conference*, December 1984, Baltimore, MD.

14. MESA Development Team, Software Development Plan (SDP) for Missile Engagement Simulation Arena (MESA) Control Software, Naval Air Warfare Center Weapons Division, China Lake, CA, August 1993.

15. MESA Development Team, Software Requirements Specification (SRS) for Missile Engagement Simulation Arena (MESA) Control Software, Naval Air Warfare Center Weapons Division, China Lake, CA, August 1993.

16. McIntee, J. W. Jr., "Fault Tree Techniques as Applied to Software (Soft Tree)", *Technical Report, United States Air Force,* March 1983.

17. Leveson, Nancy G. and Harvey, P. R., "Analyzing Software Safety", *IEEE Transactions on Software Engineering*, Volume SE-9, No.5, pp.569-579, September 1983.

18. Taylor, J. R., "Fault Tree and Cause Consequence Analysis for Control Software Validation", RISO National Laboratory, DK-4000 Doskilde, Denmark, pp. 5-17, January 1982.

19. Cha, Stephen S., "A Safety Critical Software Design and Verification Technique", Ph.D. Dissertation, *Technical Report 91-62, University of California, Irvine,* 1991.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
Cameron Station
Alexandria, VA 22304-6145

2. Library, Code 013 2
Naval Postgraduate School
Monterey, CA 93943-5101

3. Chairman, Code CS 1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

4. Professor Timothy J. Shimeall, Code CS/Sm 4
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

5. Lt. Col. David A. Gaitros 4
AFCES/CEOA
139 Barnes Drive, Suite 1
Tyndall AFB, FL 32403-5319

6. Dr. Robert M. Winter 1
Physics Department
Shippensburg University
Shippensburg, PA 17257

7. Mrs. Therese F. Giles 1
Software Project Manager
1490 West Lakeland Drive
Mechanicsville, MD 20659

8. Mr. Robert Parchetta 1
71 Blue Spruce Lane
Ballston Lake, NY 12019

9. Mr. Robert Westbrook 1
Naval Air Warfare Center - Weapons
Code 45F000D
China Lake, CA 93555-6001

10. Mr. Kenneth Wetzel 1
Naval Air Warfare Center - Weapons
Code 471310D
China Lake, CA 93555-6001

11. Mr. Tom Roseman 1
Naval Air Warfare Center - Weapons
Code 471310D
China Lake, CA 93555-6001

12. CDR Michael J. Holden 1
Computer Science Department (Code CS/Hm)
Naval Postgraduate School
Monterey, CA 93943

13. CDR Mark Barber 1
Computer Science Department (Code CS/Bm)
Naval Postgraduate School
Monterey, CA 93943

14. Mrs. Mary C. Winter 1
6700 NE 182nd Street
Seattle, WA 98155

15. Dr. Mario Ghezzo. 1
General Electric Research and Development
1 River Road
Building 40-4
Schenectady, NY 12345

16. LCDR Mathias W. Winter 2
121 East Burd Street
Shippensburg, PA 17257