NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

FAULT ISOLATOR TOOL FOR SOFTWARE FAULT TREE ANALYSIS

by

Russell William Mason

March 1995

Thesis Advisor: Timothy J. Shimeall

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

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

4. TITLE AND SUBTITLE FAULT ISOLATOR TOOL FOR SOFTWARE FAULT TREE ANALYSIS (U)	5. FUNDING NUMBERS
6. AUTHOR(S) Mason, Russell William	
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. A necessary element of a SFTA process is the construction of software fault trees based upon the syntactical structure of the software being analyzed. The specific problem addressed by this thesis is how can the process of generating software fault trees based upon the translation of Ada source code files be automated.

The approach taken to address this problem was to develop an automated tool that manipulates files created by the Automated Code Translation Tool (ACTT) [Ord 93 and Reid 94] developed earlier at the Naval Postgraduate School. The ACTT is an automated tool that translates Ada source code files into statement template tree structures that can be used to construct software fault trees.

This thesis presents the Fault Isolator Tool (FIT), an automated process for locating and isolating those parts of a statement template tree structure generated by the ACTT tool that are related to statements in Ada programs that the analyst selects for evaluation. The FIT tool then generates software fault trees in a form compatible with the Fault Tree Editor (FTE), an interactive graphical editor designed by Chuck Lombardo, a computer systems administrator at the Naval Postgraduate School. The FTE was developed for the display, editing, and evaluation of software fault trees.

14. SUBJECT TERMS Software Safety, Software Fault Tree Analysis	15. NUMBER OF PAGES 77
	16. PRICE CODE

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

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

FAULT ISOLATOR TOOL FOR SOFTWARE FAULT TREE ANALYSIS

by

Russell W. Mason
Lieutenant Commander, United States Navy
B.A., University of Texas Arlington, 1981

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

March 1995

Accesion For	
NTIS CRA&I	✓
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

Author: Russell W Mason

Russell W. Mason

Approved by: Timothy J. Shimeall

Timothy J. Shimeall, Thesis Advisor

[illegible]

John Daley, Second Reader

Ted Lewis

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. A necessary element of a SFTA process is the construction of software fault trees based upon the syntactical structure of the software being analyzed. The specific problem addressed by this thesis is how can the process of generating software fault trees based upon the translation of Ada source code files be automated.

The approach taken to address this problem was to develop an automated tool that manipulates files created by the Automated Code Translation Tool (ACTT) [Ord 93 and Reid 94] developed earlier at the Naval Postgraduate School. The ACTT is an automated tool that translates Ada source code files into statement template tree structures that can be used to construct software fault trees.

This thesis presents the Fault Isolator Tool (FIT), an automated process for locating and isolating those parts of a statement template tree structure generated by the ACTT tool that are related to statements in Ada programs that the analyst selects for evaluation. The FIT tool then generates software fault trees in a form compatible with the Fault Tree Editor (FTE), an interactive graphical editor designed by Chuck Lombardo, a computer systems administrator at the Naval Postgraduate School. The FTE was developed for the display, editing, and evaluation of software fault trees.

ACKNOWLEDGEMENTS

I would first of all like to express my appreciation to Dr. Timothy Shimeall for his support and guidance in completing this thesis. Despite his personal erratic and full schedule, he was always available for questions and lending a helpful hand when I needed it. I will always view the opportunity I had to work with him as an honor and a privilege. I would also like to express my appreciation to LCDR John Daley whose insightful and thorough editing were invaluable and contributed much to producing a higher quality result.

Most of all, I would like to thank my wife, Karen, for her patience, understanding, and support during our tour at the Naval Postgraduate School. As always, she handled the toughest aspects of our tour here without complaint, and with a grace I've come to admire.

The Fault Isolator Tool (FIT), Automated Code Translation Tool (ACTT), and the Fault Tree Editor (FTE) are all available via anonymous ftp. Contact shimeall@cs.nps.navy.mil for details.

TABLE OF CONTENTS

I. INTRODUCTION

The rapid transformation of society from the industrial age to the information age has brought about greater and greater dependence on computers throughout the world. The potential advantages of using computers is just now being fully realized. Computers and their related components permeate electro-mechanical devices, and the systems that employ these devices, influencing practically every aspect of our lives. Of particular importance is the realization that digital computers are now being given more and more control of functions previously performed only by human operators and/or proven analog methods [Lev 86]. Many of these functions that computers are taking over are as controlling and monitoring agents in large complex systems where failure of the computer component could result in death, injury, loss of property, or environmental damage [Lev 86]. The need for efficient and effective means of analyzing and assessing the degree of safety of computer elements to ensure that these catastrophic events never happen has never been greater. For a variety of reasons, effective and timely methods of analyzing the software element of computer units for flaws or conditions that could cause an unsafe state in a system have not been forthcoming. This situation could be greatly improved by the development of automated tools designed specifically to aid software engineers and system safety analysts in their ability to pinpoint potential errors in the design, development, and maintenance of safety-critical software.

A. SAFE SYSTEMS

Bailey defines a system as "... an entity that exists to carry out some purpose" [Bai 82]. Systems are composed of humans, machines, and other things that work together (interact) to accomplish some goal which the same components could not produce independently [San 93]. Therefore, any determination made about the safety of a system as a whole must include a thorough examination of all of the individual elements or parts of the system, including the environments in which the system may be required to operate.

More and more today, complex systems involving time-critical physical processes or mechanical devices are being built that include a computer component consisting of both computer hardware and computer software sub-components. One reason for this is the astonishing advances in computer hardware engineering resulting in computer parts that are smaller (miniaturized), faster, more powerful, more reliable, and cheaper. A secondary, but no less important reason for the integration of computers into large, complex and potentially hazardous systems may be the recognition by society of the vast potential and capabilities of computers. Examples of these potentially hazardous systems where the role of computers is becoming more and more important include transportation, energy, aerospace, basic industry, medicine, and defense systems [Lev 86]. The rate at which this integration is taking place is no less astonishing. For instance, a study by Griggs revealed that only 10 percent of our weapon systems required computer software in 1955, but by 1981 this figure was over 80 percent [Gri 81].

The U.S. Dept. of Defense *System Safety Program Requirements* manual defines a safe system as one that enjoys "...freedom from those conditions that can cause death, injury, occupational illnesses, or damage to or loss of equipment or property" [MIL 84]. Leveson points out that this definition is unrealistic because a strict application of this definition would result in labeling any system "...that presents an element of risk" as unsafe [Lev 86]. This would mean that practically any system ever designed for the benefit of man—past, present, or future, would be considered unsafe. For example, even something as simple and apparently harmless as a cotton ball would be labeled unsafe because of the impossibility of removing the potential for a person swallowing a cotton ball and choking to death. Air transportation is statistically one of the safest forms of transportation, and yet hundreds die every year in airplane accidents. Perhaps a more appropriate definition of a safe system would be a system that provides a maximum of freedom from those conditions that can cause death, injury, occupational illnesses, or damage to or loss of equipment or property while providing a maximum of beneficial service to the user(s) of the system.

Because any system inherits some element of risk by its very nature, the goal of system designers is not to eliminate all possible risks to make a safe system, but to weigh the various factors that affect the cost, practicality, and social acceptability of making a system as risk free as possible. For instance, to prevent someone from swallowing a cotton ball, the practice of producing cotton balls as large as basketballs would eliminate this risk, but is not practical and probably would not be very accepted by society. Therefore, the development of safe systems involves a series of trade-offs between providing the service or product that originated the production of the system in the first place, and the elimination of as many risks as possible that are cost-effective, practical, and socially acceptable.

No discussion about the definition or description of safe systems would be complete without discussing the weakest link in the safety issue loop—the human factor. Because of certain undeniable and sometimes puzzling traits of human nature, even the best engineered, most safely designed systems can exhibit hazardous and sometimes dangerous conditions. From conceptualization to use, there are certain principles or factors that all systems must address:

- **No system can entirely remove the human factor from the safety factor loop.** Human made systems are conceptualized, designed, developed, maintained, and used by humans for the benefit of humans. With the current technology today and the foreseeable future, the ideal that systems can be spawned by some ultra superior artificial intelligence, and then developed, maintained, and used entirely in the absence of any type of human intervention or regard is inconceivable. Even systems that are considered totally automated and perform all of their operational functions with little or no human intervention can totally exclude the human factor. Sanders and McCormick state that "All automated systems require humans to install, program, reprogram, and maintain them." [San 93].
- **Humans will not remove themselves from the safety factor loop, even at the expense of a less safe system.** One of the inherent characteristics of humans is that they want to be in control. So much so that they will often opt for a system with less safety features enforced to retain the control they desire. A good example of this is the automobile. The technology exists to enforce the occupants of a vehicle to wear seat belts, a factor that would reduce the amount of accidental car deaths by the thousands every year. And yet, the automobile industry refuses to install such a system knowing how unpopular this concept would be with their potential clientele. Even though

people know that wearing seatbelts greatly enhances the safety of operating a vehicle, they want the control of deciding whether or not to use that safety feature.

- **Humans have a natural mistrust of systems that exclude human decision making.** Whether justified or not, this human characteristic dictates that systems be designed to include humans in decision making processes, especially in systems involving critical and potentially hazardous consequences as a result of actions initiated by the system. Leveson addressed this issue when discussing human factors issues and whether or not systems would be less mishap prone if humans were "... removed from the loop" (decision making loop). The current evidence appears to be that, although humans do make mistakes, computers also make mistakes, and removing humans from the loop in favor of so-called expert systems or total computer control is probably not desirable [Lev 86].
- **Humans can and will intentionally and unintentionally circumvent the built-in safety features of a safe system.** This is perhaps the most frustrating of all human traits that those responsible for building and maintaining safe systems have to contend with. One need only observe the nightly news to see examples of this. Humans die needlessly in traffic accidents because of refusal to wear seatbelts; countless fires have been started in homes by humans placing copper pennies inside circuit breakers that have short circuited; traffic accidents are caused by drivers running red lights at intersections with traffic control lights in perfect working order; many airplane accidents are caused by human error; and the list goes on and on. Systems can be designed to minimize the risks to humans, but in the end, no system can be designed absolutely "human proof".
- **Safety is not always a high priority to humans.** Humans by nature crave the element of risk. The entertainment industry makes billions of dollars every year by providing nothing more than an element of danger to their customers in the form of high speed roller coaster rides in theme parks around the world. One would be hard pressed to find a beneficial service to man other than that of providing an extreme risk in a multitude of activities that man participates in such as bunji jumping, mountain climbing, sky diving, and so on. Some industries thrive on the fact that humans simply ignore safety when it comes to their health and well being. The tobacco industry capitalizes on this human trait.

Given the discussion above, one may wonder what type of measuring stick is used to categorize a system as a safe system. According to Leveson, "System safety is a subdiscipline of systems engineering that applies scientific, management, and engineering principles to ensure adequate safety, throughout the system life cycle, within the constraints of operational effectiveness, time, and cost." [Lev 86] Safety and safe systems are relative terms in that safe systems are not safe, only adequately safe as compared to unsafe systems which are not adequately safe. It would be hard to classify an automobile as a safe system given the number of yearly traffic accidents and fatalities; however, an automobile with safety belts is safer than one without, and an automobile with safety belts and airbags is more safe that an automobile with just seatbelts. Leveson also adds that:

> Safety is also relative in that nothing is completely safe under all conditions. There is always some case in which a relatively safe material or piece of equipment becomes hazardous.[Lev 86]

Therefore, the question that must be answered is not "Is the system safe?", but "Is the system adequately safe?". Answering this question is the primary function of the system safety analyst.

B. SOFTWARE SAFETY

Software safety relates to the safety of systems that contain one or more computer components, especially those systems where computer components either control or monitor time-critical physical processes or mechanical devices. As stated earlier, the computer component of a system contains both a hardware and software sub-component. However, because of astounding advances in computer hardware engineering, computer hardware has become so reliable that this component is no longer considered a major cause of system failures [Ord 93]. Software development on the other hand, has fallen years behind the technology of the hardware that it controls and has become the major "bottleneck" in system improvement [Cha 91]. In addition, software presents unique problems that render conventional procedures of system safety, like those used in analyzing hardware components, inappropriate or inapplicable.

In defining software safety, we have to examine the system which contains or relies upon the software that is being scrutinized. Leveson states "Software engineering techniques that do not consider the system as a whole, including the interactions between the hardware (computer and non-computer), software, and human operators, will have limited usefulness for real-time control software." [Lev 86] In this context, Mcdonald states that a generally accepted definition of software safety is found in the *Software Systems Safety Handbook* [Bon 85], which defines it as "The application of system safety engineering techniques to software development in order to insure and verify that software design takes positive measures to enhance the safety of the system and that errors that which could reduce the safety of the system have been eliminated or controlled." [McD 89] McDonald emphasizes the key phrase "... software design takes positive measures to enhance the safety of a system..." in the definition indicating that the "... objectives of a software safety program are changed from passive to active and from reactive to pro-active." In other words he states, for software safety:

1. The goal is that conditions hazardous to the system are eliminated or reduced to an acceptable level,
2. Safety requirement concerns are to make the system as mishap free as possible within the resource constraints of time, money, trade-offs with other requirements, and
3. Design implications are that the system not only does what it is supposed to do, but also is prevented from doing what it is not supposed to do. [McD 89]

As is the case with system safety in general, one cannot exclude the effects that human factor issues have on software safety either. In fact, Brown states that virtually all errors that occur in software may be attributed to humans. Furthermore, he claims that these errors can be categorized into four different groups: design, coding and testing, operator, and maintenance [Bro 88]. Because the advent of computers as an integral element of time-critical systems is relatively new, humans are being required to adjust to the proper use of these new high tech systems, often with catastrophic results. Neumann relates several incidents where "... the problem of people inappropriately trusting computers" has resulted in several recent disasters. He cites examples including the shootdown of the Iranian Airbus by the U.S.S. Vincennes; the Air France fly-by-wire Airbus crash; the Exxon Valdez oil spill; the British Midland Airways 737 crash; and others. Neumann states that "It is clear that computer systems (1) cannot ensure completely adequate system behavior and (2) cannot enforce completely adequate human behavior." [Neu 89]

C. SYSTEM SAFETY VS. SOFTWARE SAFETY ANALYSIS

Compared to system safety which became a concern in the late 1940s and was then defined as a separate discipline in the late 1950s [Lev 86], software safety is a relatively new engineering discipline being identified as such in the early 1980s [McD 89]. System safety analysis techniques have developed over the years to the point of almost being an exact science. This is reflected in the high reliability requirements, tending to range from 10^{-5} to 10^{-9} over a given period of time [Lev 86], that modern safety-critical systems demand, and which is more, can be satisfied by highly developed and effective system safety analysis techniques in use today. It would be convenient if the same techniques that work so well in system safety analysis could be applied to the discipline of software safety analysis. Unfortunately, many of the system safety engineering techniques "... do not seem to apply when computers are introduced." [McD 89] According to McDonald:

> Most of the system safety techniques that exist today were developed based on electromechanical (EM) systems, and there are significant differences between those EM systems and software-controlled systems. The EM system safety engineering techniques are designed to cope with random failures; as a result, they are based on:
>
> 1. Statistical quality control,
> 2. Probabilistic reliability approaches, and
> 3. Periodic maintenance and/or replacement.
> 4. Assumption that human design errors can be avoided completely, or located and removed prior to operation.
> 5. Error prevention through systematic design and validation and reuse of proven hardware modules, and
> 6. Reliance on historical usage information, including the results of accident investigations. [McD 89]

McDonald then explains why these techniques do not work when applied to software safety analysis:

1. Strictly software failures of operational software relative to safety are not random; they are deterministic due to being built in during specification, design, and/or implementation and they will occur with a probability of 1 whenever the necessary external conditions are satisfied. Hence, their control cannot be based on statistical or probabilistic approaches.
2. Software does not wear out; hence, safety is not increased through periodic maintenance or replacement. Periodic maintenance can lead to degradation of safety if changes are not properly designed and implemented.
3. Computers/microprocessors have greatly increased complexity and coupling of systems, so there has been a non-linear increase in human-error-induced faults. It is generally impossible to demonstrate that designs are correct and that failure mechanisms have been completely eliminated.
4. Error prevention relative to safety could be improved through systematic design and validation; however, that is what a true software safety program is about.
5. Real-time, process-control software modules are commonly designed and implemented specifically for the system into which they are to be incorporated. Hence, proven modules are not typically reused.
6. Since software modules are not normally reused, there is little if any historical usage information that is of value in software hazard analysis.
7. Software controls systems by initiating hardware actions through electron flow, so there is often little evidence at the scene of an accident to define the state of the software at the time the failure occurred. Given what is left to work with, hardware and/or human failures are the most common allocations for faults when accidents occur—especially when the operators are killed in the accident. [McD 89]

Given this, it is doubtful that software safety analysis techniques will ever achieve an assurance of the high degrees of reliability that system safety engineers can guarantee with strictly electro-mechanical systems using well developed analysis techniques that have evolved for these types of systems. In fact, it is questionable whether the reliability degree of software can even be measured [Lev 86]. Whether the introduction of computers into systems will improve safety as suggested by Rouse [Rou 81], or that replacing electro-mechanical devices with computers can actually impair safety [Lev 86], the fact remains that computers are being introduced to control some hazardous systems. Leveson states that "There are just too many good reasons for using them and too few practical alternatives." [Lev 86] Because of this, there is a need for emphasizing the development of effective analysis techniques and tools to aid the software safety analyst in achieving the goals of software safety. This is the focus of this thesis, and existing techniques and tools that will be applied to develop the target FIT tool are discussed in the next section.

D. BACKGROUND AND PREVIOUS WORK

Before software safety analysis can be applied to a system containing a computer element, analysts must first develop a thorough understanding of the complete system. A thorough understanding would include descriptions of:

- system hardware
- system software
- system users
- potential environments
- life cycle phases
- functional characteristics [Bas 89]

This is needed because of the fact that "... software cannot directly produce a hazard without first passing through hardware or human interfaces." [Bas 89]

Once a thorough understanding of the complete system has been attained, there are two basic techniques that are available for software safety analysis, Top-Down (or deductive) analysis and Bottom-Up (or inductive) analysis [Bas 89]. The tool being developed in this research is for safety analysts to use during a Top-Down analysis approach.

According to Bass, the Top-Down methodology of software safety analysis consists of seven steps:

1. Identify potential hardware and human decision hazards.

2. Evaluate the risk of each hazard (the probability of occurrence and potential severity of a mishap).
3. Identify hazard-related software elements.
4. Evaluate software elements for design or programming errors that may lead to a hazard.
5. Develop safe design and test recommendations for hazard elimination or severity reduction.
6. List safety-critical system elements for further analysis.
7. Track, verify, and validate hazard control recommendations and program development. [Bas 89]

The tool being developed in this research is being designed to assist software safety analysts during what Bass has identified as the fourth step in a Top-Down analysis methodology, which is evaluating software elements to determine how potential hazards might result from faults in program code. An implicit requirement of the use of this tool is that the first three steps of the Top-Down methodology have already been performed before using the tool, with special attention to step three which is identifying hazard-related software elements. Before an analyst can determine if software can cause an unsafe or hazardous condition, the analyst must know what are considered the hazardous conditions of the system. The major goal of performing the third step in this methodology is identifying those hazardous conditions of the system that could be caused by software faults. It is in the fourth step of the Top-Down methodology that analysts then evaluate software to determine if it can cause any of the hazardous conditions or states identified in step three. It is also implicit that the quality of performance done in step three can seriously impact the effectiveness and accuracy of the results in steps that are subsequently performed in the Top-Down methodology.

The approach to developing the tool that is the goal of this research is to build on techniques and tools that have already been identified as effective for the analysis of safety-critical software. Specifically, the tool provides an automated means of effectively applying the technique of Software Fault Tree Analysis (SFTA) [Cha 87]. To accomplish this, the tool processes files that are generated by the Automatic Code Translation Tool previously developed by Ordonio [Ord 93] and enhanced by Reid [Rei 94].

1. Software Fault Tree Analysis

Software Fault Tree Analysis (SFTA) was derived from Fault Tree Analysis (FTA), a system safety engineering technique developed in the 1960s to analyze the safety of electro-mechanical systems [Cha 87]. SFTA extends the same principles of FTA and applies them to systems containing computers as subcomponents. The basic concept behind SFTA is to specify unsafe states in a software program that could cause hazardous conditions in a system, and then verify that the program will never reach such a state. For example, for a traffic control system that involved software controlling the traffic light display at a four way intersection, we could specify that one unsafe state would be the traffic light simultaneously displaying green to traffic traveling on perpendicular paths. Obviously, this could lead to the hazardous condition of a collision between vehicles. A software safety analysts would then, using SFTA, attempt to verify that the controlling software could not reach such a state.

There is an important distinction between what can be accomplished through SFTA, and what SFTA cannot accomplish. The goal of SFTA is to prove or verify that software will not do something, not to verify that it does what it is supposed to do. Cha points out that "Most real-time embedded systems have two goals: (1) accomplishing a mission or function, while (2) not causing harm in the process." [Cha 87] SFTA is only concerned with verifying that the software element of a system does not prevent accomplishing the second goal.

A Software Fault Tree is a collection of symbols (See Figure 1), each with a distinct semantic meaning, that are connected by logical gates that graphically depicts the logical interrelationships of basic events or states in the software program that could lead to the primary event that is being evaluated, referred to as the root fault. The definition of what each symbol represents is as follows:

- **Rectangle**. The rectangle represents an event that will be further analyzed.
- **Diamond**. The diamond represents nonprimal events which are not developed further for lack of information or insufficient consequences.
- **Circle**. The circle indicates an elementary event or primary failure of a component where no further development is required.
- **Pentagon**. The pentagon, sometimes referred to as the "house" symbol, represents the continued operation of the component and is used for events that normally occur in the system.
- **Ellipse**. The ellipse is used to indicate a state of the system that permits a fault.
- **Triangle**. The triangle represents another sub-tree for the node which is not depicted on the current tree.
- **AND Gate**. The AND gate serves to indicate that all input events are required in order to cause the output event.
- **OR Gate**. The OR gate indicates that any one of the input events may produce the gated event. [Ord 93]

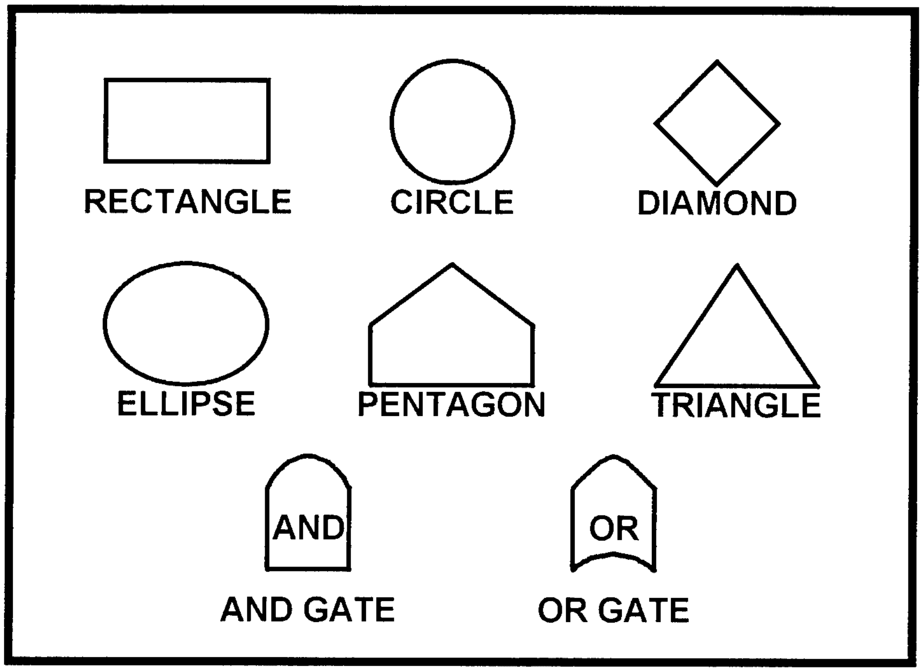


Figure 1. Software Fault Tree Symbols

The basic procedure for building a Software Fault Tree is similar to that of building a Fault Tree. An assumption is made that a hazardous condition has occurred, and then the analysts works backwards to determine all possible causes of the hazardous condition. The root of the tree is designated as the hazardous condition, and then all the necessary preconditions for the hazard to occur are described by the nodes at the next level of the tree with either an AND or an OR gate connector. The nodes at this level are then expanded in a similar fashion until all leaves are unable to be expanded any further in this fashion for some reason. The difference between a Fault Tree and a Software Fault Tree is that the Software Fault Tree generation is based upon statement-specific templates that reflect the semantics of the programming language that is being used.

The basic statement templates for the Ada programming language were first proposed by Cha, Leveson, and Shimeall in 1987 [Cha 87]. This set of templates was further refined and added on to by Ordonio and Reid in their thesis research in 1993 and 1994 respectively. A full listing of the statement templates for the Ada language is included in references [Ord 93 and Rei 94].

While SFTA can be an effective technique for evaluating safety-critical software, it has a serious drawback that can't be overlooked. The time and effort that would be required of an analyst to manually generate Software Fault Trees for even small programs can be prohibitive. The work involved would be very tedious, and would probably result in errors being made if done by hand. This is unacceptable when evaluating systems that could potentially result in the lost of human life or other serious consequences if faults in the system are not discovered in time. To overcome this problem, Ordonio developed (further refined by Reid [Rei 94] an automated tool that generates a collection of connected statement templates from parsing Ada source code programs [Ord 93].

2. Automated Code Translation Tool [Ord 93 and Rei 94]

The Automated Code Translation Tool (ACTT) is an automated tool that translates Ada programming language statements into collections of template structures which in turn can be used to construct Software Fault Trees that safety analysts can use in SFTA. The effect of the tool is a reduction in the amount of time involved and in the amount of errors that would result from manual translation of code to a form that could be used in SFTA. The ACTT consists of four main components:

- **Lexical Analyzer**. The lexical analyzer breaks up the character stream that makes up the Ada source code (a program is essentially a stream of text characters) into meaningful units called tokens. Tokens are identifiers, constants, reserved words, operators, and any other sequence of characters that are defined by the programming language (in this case, Ada) as valid primitive units. Identification of tokens is necessary for the next component of the ACTT, the parser.
- **Parser**. The parser is a process that determines whether a string of input, or more specifically, a series of tokens, constitutes a valid statement of a particular programming language. If so, it then determines the syntactic structure of the string as defined by the programming language. The primary purpose of the parser is to validate the string as a correct statement for a given language.
- **Template Generator**. The template generator works in concert with the parser by building the appropriate statement templates for structures that have been recognized by the parser.
- **File Generator**. The file generator transforms the collection of statement templates built by the template generator into a file that can then be opened by a Fault Tree Editor (FTE). An FTE is a program that allows an analyst to view the graphical representation of a Fault Tree or Software Fault Tree. Normally, the FTE also provides the means for manipulating Fault Trees such as inserting nodes, deleting nodes, and other node editing functions. The ACTT generates files that are compatible with the FTE developed at the Naval Postgraduate School by Chuck Lombardo.

The ACTT was originally developed by Ordonio at the Naval Postgraduate School [Ord 93]. Ordonio instituted most of the Ada programming language statement templates developed by Cha, Leveson, and Shimeall [Cha 87] with some alterations and also expansion to include some statement templates not identified earlier. The ACTT was further developed by Reid [Rei 94] to include those Ada language structures that were left out of the original ACTT, namely, structures dealing with concurrency and exception handling. The combined efforts of Ordonio and Reid resulted in a tool that accounts for all of the statement structures defined in the Ada language (before Ada-9X).

It is important to note that while the resulting files generated by the ACTT may resemble a Software Fault Tree, they are not in fact fault trees. In an earlier section it was emphasized that before constructing a Software Fault Tree, the process of identifying a specific hazard to evaluate is absolutely essential. The files generated by the ACTT do not specify any specific faults. Instead, they provide an aggregate of related statement templates as determined by the structure of the input Ada program with nodes containing generic fault descriptions that an analysts can select as a basis for constructing Software Fault Trees. The main benefit gained by the analysts in using the ACTT is that it frees him or her from constructing the template structure of the particular program they are analyzing by hand. As the size, complexity, and safety- criticalness of software programs increases, this benefit cannot be appreciated too much.

Figure 2 depicts a model of the ACTT as described by Ordonio. For a more detailed description of the ACTT, the reader is referred to [Ord 93] and [Rei 94].

E. GOAL OF AUTOMATED SFTA PROCESS

The goal of an automated SFTA process is to automate those procedures that: (1) can be automated, (2) should be automated, and (3) will enhance the ability of the analyst to perform effective and efficient SFTA. Not all the procedures involved with a SFTA process can be automated. For instance, making recommended changes to a software program at the end of a SFTA process is a procedure that cannot be automated with current technology, nor perhaps should it be even if the means to do so existed. A second proposition is that not all of the procedures involved in an SFTA process should be automated, in effect, removing the human element from critical decision making processes involved in SFTA. The reason being the same argument used to explain why computers should not replace human pilots. In safety-critical systems, human control and monitoring of system processes is absolutely essential. Considering the ramifications that faulty software safety assessment can have on the safety of systems, human guidance and judgment should always be considered irreplaceable elements in any software safety

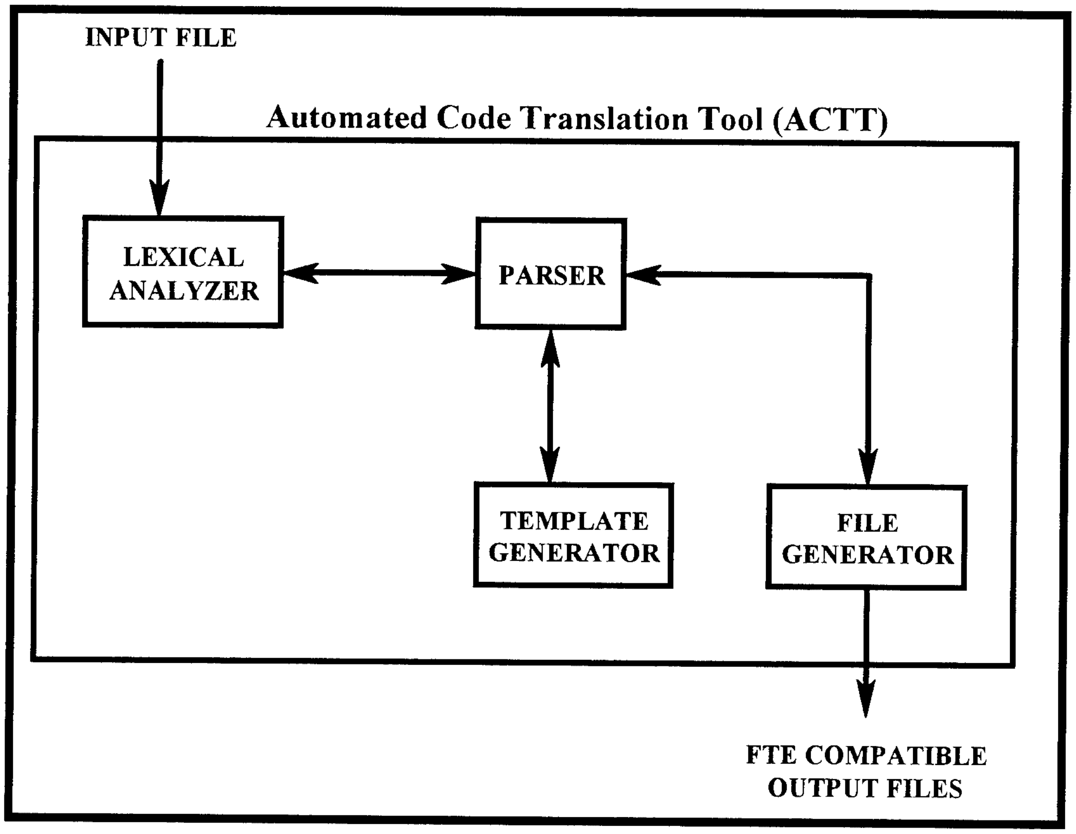

INPUT FILE
Automated Code Translation Tool (ACTT)
LEXICAL ANALYZER
PARSER
TEMPLATE GENERATOR
FILE GENERATOR
FTE COMPATIBLE OUTPUT FILES

Figure 2: Automated Code Translation Tool Structure [Ord 93]

analysis process. The third proposition regarding automation in SFTA is that an automated process should enhance the analyst's ability to effectively and efficiently perform SFTA in the safety assessment of software. Automated processes such as the construction of software fault trees from the interpretation of software programs are items that can aid a software safety analyst by saving the time required for manual generation of software fault trees, and by reducing the potential for errors that manual generation of software fault trees tends to induce into the process.

One other point that has to be made is that an automated SFTA process does not guarantee the accuracy of the final results of the safety evaluation of software. Like any other tool, the results are only as good or as bad as the person(s) using the tool. The automated tools described in the next chapter do not guarantee, nor can they, that mistakes will not be made in the employment of the tools. There is no guarantee that the analyst using the tools will not make mistakes in interpreting the significance of a software fault tree. Furthermore, no guarantees are made that the analyst chooses the right root faults to evaluate, nor are there any guarantees that the analyst chooses all of the right root faults to evaluate necessary in a complete examination of the software being scrutinized. These are skills that can only be attained by practice and experience, and ultimately it is the analyst who will have to decide how accurate and thorough the SFTA process has been.

F. STATEMENT OF PROBLEM

Because computers provide capabilities that are unequaled by other forms of technology, the trend toward using them as key elements in safety-critical systems is evident throughout society. As this trend continues, software safety concerns are assuming greater importance within the general area of system safety. The failure of adequate software safety assessment and the impact that this has had on systems resulting in loss of life and massive property damage is well documented [Lev 86]. The major contributor to inadequate software safety assessment has been a general lack of effective techniques and tools that software safety analysts need to perform their jobs.

Compared to general System Safety Engineering involving primarily the safety assessment of hardware components, the field of software safety is relatively new. Because of this, it is not surprising that the methods of operation in performing software safety analysis are not on an equal level with those of general system safety analysis. In fact, it has been argued that not only is it impossible for software safety analysts to guarantee that software can meet the same reliability levels that are common of hardware components, but that it is even questionable whether the reliability level of software can even be measured. This is because of the unique nature of software analysis involving any type of verification and validation. Software is far more complex than hardware and the means to evaluate all the possible states that even small programs can attain does not exist. Whereas hardware components that have been proven to meet certain safety requirements are usually mass produced and easily integrated into a variety of systems, software is normally very specialized to meet the unique demands required by the systems that depend on it. It is unlikely that software modules will ever reach the level of reusability that hardware components are able to. For these and other reasons, new approaches are needed to develop software safety analysis methodologies that are as equally effective as those being used to analyze the safety of hardware components.

One logical approach to develop effective software safety analysis methods is to extend as much as possible the principles, techniques, and tools of System Safety Engineering to this field. With some modification, many of the directives, current analysis techniques, and databases for hardware system safety are usable for software safety determination [Neu 89]. One technique used in System Safety Engineering that has great potential of being adapted to the field of Software Safety Engineering is Fault Tree Analysis.

The technique of Fault Tree Analysis adapted to software safety is called Software Fault Tree Analysis (SFTA). Some of the potential benefits of applying SFTA in analyzing software include: (1) determination of software safety requirements, (2) detection of software logic errors, (3) identification of multiple failure sequences involving different parts of the system that can lead to hazards, (4) provide guidance in the selection of critical run-time checks, and (5) provide guidance in testing [Lev 86]. However, the practical application of SFTA to software safety assessment presents problems that must be overcome before SFTA can be considered as an effective technique. The major problems to be overcome are that the manual generation of Software Fault Trees (SFT) necessary for SFTA is prohibitively expensive (in terms of both time and cost), and the laborious nature of manually creating SFTs tends to induce errors into the process.

This thesis presents a interactive tool that automates the process of generating SFTs for Ada source code programs. In addition, the tool provides functionalities that a software safety analyst can use to decrease the time needed to obtain a level of understanding of SFTs necessary in SFTA. To generate SFTs, the tool processes text files that are created by the Automated Code Translation Tool (ACTT), a tool that builds a collection of connected statement template structures by parsing Ada source code files [Ord 93 and Rei 94]. These processed files can then be input to a Fault Tree Editor (FTE), an interactive graphical editor designed for the study of SFTs. The interactive nature of the target tool allows the user to guide the construction of SFTs based on specific faults that he or she wishes to evaluate. The process involves locating nodes in statement template tree structures related to a source code line number and file name of an Ada source code file that the analyst selects, pruning off those parts of the statement template tree structure that are not relevant to the fault be evaluated, and then directing the results to an FTE-compatible output file. The tool also provides functionalities for the automated manipulation of the properties of individual SFT nodes that the analyst can use to increase his understanding of a particular SFT.

It is not the intent of this research to suggest that the process of software safety analysis should be fully automated even if it were possible to do so. There are certain steps in a full system safety analysis program where human intervention and control is absolutely essential, especially in the evaluation of safety-critical systems where mistakes can lead to endangering human life or can lead to extensive property damage. However, the development of automated tools that can assist safety analysts in the performance of their work is becoming more and more necessary as the complexity and safety-criticalness of systems increases. Automated tools are not designed to take control of safety assessment processes away from humans, but are designed to enable safety analysts to more efficiently and effectively control the processes employed to accomplish safety analysis.

G. SUMMARY OF CHAPTERS

Chapter II describes the target Fault Isolator Tool (FIT) that is the focus of this thesis. The chapter includes a detailed description of the main functions and features of the

FIT tool along with a description of the process involved with employing the target tool and the other automated tools mentioned above for performing SFTA in analyzing the level of safety in Ada software programs. Chapter III presents an example of a practical application using a small Ada program as the software to be evaluated, and showing how the FIT tool can be applied to aid a software safety analyst. Chapter IV contains a conclusion of the results of this research, recommendations for the use of the FIT tool, and a brief summary of possible future research.

II. FAULT ISOLATOR TOOL DESCRIPTION

The Fault Isolator Tool (FIT) is an automated tool designed to be used by programmers and software safety analysts that will allow a more efficient and effective means of applying Software Fault Tree Analysis (SFTA) in evaluating the safety of Ada software programs. SFTA can be an effective method of evaluating the logic of software for states or conditions that could cause hazardous situations in systems in which the software is a component. A necessary element for the application of SFTA is the construction, either manual or automated generation, of tree structures called Software Fault Trees (SFTs). However, manual construction of SFTs can be prohibitively time consuming and tedious work that has a tendency to introduce inaccuracies into the process of SFTA. Even one error in this process could potentially result in catastrophic events in safety-critical systems risking the loss of human life, serious injury, or extensive property damage. The FIT tool is designed to relieve those concerned with the safety assessment of Ada programs from constructing SFTs by hand, thereby contributing to the overall accuracy, effectiveness and efficiency of SFTA.

The FIT tool uses the output of the Automated Code Translation Tool (ACTT) [Ord 93 and Rei 94] previously developed at the Naval Postgraduate School. The ACTT tool processes Ada source code programs generating a collection of connected statement template structures. The ACTT tool does not create SFTs. A SFT consists of a specific hazard or event designated as the root fault, and then a set of logically connected conditions or software states that could cause the root fault. The ACTT tool does not designate any specific faults, but provides an aggregate of statement template structures based on the logic of the software program being evaluated from which SFTs can then be constructed. Statement templates are mini-tree structures consisting of nodes representing conditions and connecting logic gates that represent the failure modes of statements as defined by the Ada programming language. The output files created by the ACTT are text files that are compatible with the Fault Tree Editor (FTE) developed at the Naval Postgraduate School.

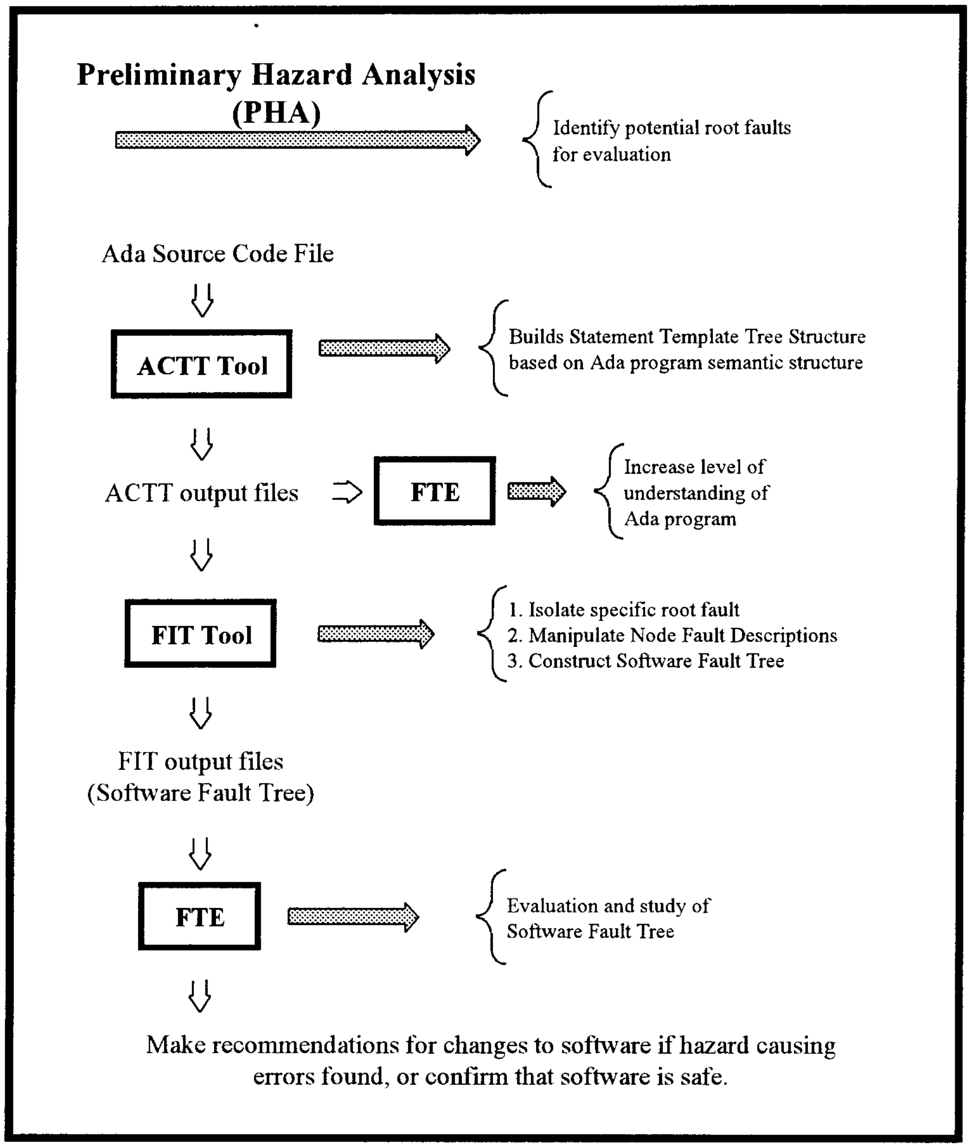
The FTE is an interactive tool designed for the graphical display and editing of SFTs. The FIT tool was developed to process the files created by the ACTT tool to construct SFTs for editing with the FTE. The basic process for applying SFTA to an Ada program using the tools mentioned above consists of:

- Identify specific potential hazards that could be caused by software logic errors (this step is normally done during Preliminary Hazard Analysis).
- Input the targeted Ada source code file to the ACTT tool.
- Input the files created by the ACTT tool to the FTE tool to increase understanding of the source code.
- Input the files created by the ACTT tool to the FIT tool to isolate a specific node related to a source code line number that is believed to be the cause of a specific software hazard (designated as the root fault). The FIT tool will automatically prune the original ACTT output file removing those parts of the tree that are unrelated to the designated root fault. The FIT tool will then create a new FTE-compatible file.
- Input the file created in the previous step to the FIT tool to manipulate the fault descriptions of the remaining nodes and save the results. Changing the fault descriptions of the nodes in the resultant SFT helps show the relationship between the nodes in the tree and the specific root fault being evaluated.
- Input the file created in the previous step to the FTE tool for evaluation. This file can also be input into the FIT tool for further refinement if desired by the safety analyst.
- Repeat this process for all potential hazards identified in the first step.
- Make recommendations for changes to software if logical algorithms of existing source code were shown to permit any of the designated root faults to occur, or confirm that existing source code did not permit any of the designated root faults to occur.

Figure 3 depicts a model of the process just described. The remaining sections in this chapter give a detailed description of the individual steps in the process described above along with a description of the main features and functions of the FIT tool.

A. IDENTIFYING ROOT FAULTS FOR EVALUATION

Before the technique of SFTA can be applied to assess the safety level of software, some method for determining which states in the software may constitute a hazard must be



Preliminary Hazard Analysis (PHA)
Identify potential root faults for evaluation
Ada Source Code File
ACTT Tool
Builds Statement Template Tree Structure based on Ada program semantic structure
ACTT output files
FTE
Increase level of understanding of Ada program
FIT Tool
1. Isolate specific root fault
2. Manipulate Node Fault Descriptions
3. Construct Software Fault Tree
FIT output files (Software Fault Tree)
FTE
Evaluation and study of Software Fault Tree
Make recommendations for changes to software if hazard causing errors found, or confirm that software is safe.

Figure 3: SFTA Process Using Automated Tools

performed. The method normally used is referred to as hazard analysis. Hazard analysis is a general term that encompasses many sub-procedures and processes. Hammer defines hazard analysis this way:

> Hazard analysis is the investigation and evaluation of:
>
> - The interrelationships of primary, initiating, and contributory hazards that may be present

- The circumstances, conditions, equipment, personnel, and other factors involved in safety of a product or the safety of the system and its operation
- The means of avoiding or eliminating any specific hazard by use of suitable designs, procedures, processes, or material
- The controls that may be required for possible hazards and the best methods for incorporating those controls in the product or system
- The possible damaging effects resulting from lack or loss of control of any hazard that cannot be avoided or eliminated
- The safeguards for preventing injury or damage if control of the hazard is lost [Ham 72]

While numerous types of hazard analyses are used, the types that affect the application of SFTA the most are:

- **Preliminary Hazard Analysis (PHA).** This involves an initial risk assessment whose purpose is to identify safety critical areas and functions, identify and evaluate hazards, and identify the safety design criteria to be used. The results of this type of analysis may be used in developing system safety requirements and in preparing performance and design specifications [Lev 86]. Hammer refers to this as predesign analysis which determines and evaluates those hazards that might be present in a system to be developed. Among other things, he states that this type of analysis "...may indicate undesirable characteristics, conditions, and practices to be avoided." [Ham 72, p. 87-88]
- **Subsystem Hazard Analysis (SSHA).** Leveson states that the purpose of this type of analysis is to "... identify hazards associated with the design of the subsystems, including component failure models, critical human error inputs, and hazards resulting from functional relationships between the components and equipment comprising each subsystem." [Lev 86] This type of analysis is used to determine how hazards associated with the operating or failure modes of subsystems or components can affect the overall safety of the system.
- **System Hazard Analysis (SHA).** The purpose of this type of hazard analysis is to recommend changes and controls and evaluate design responses to safety requirements. In contrast to SSHA, SHA "... determines how system operation or failure modes can affect the safety of the system and its subsystems." [Lev 86] Hammer refers to this type of analysis as postdesign

analysis, which is used to determine "...whether selected equipment and procedures meet the standards and criteria established as a result of the predesign analysis." [Ham 72]

A combination of all three of these types of hazard analysis may be relevant to the application of SFTA in a safety evaluation of software. A simplistic example would be the traffic control system mentioned in Chapter I. Applying PHA to this system, safety analysts may deduce that two vehicles on perpendicular paths in the intersection at the same time is a hazardous condition in the system (traffic control system). Then applying SSHA, the analysts would deduce that the software component responsible for the display of the traffic lights simultaneously displaying green lights to traffic on perpendicular paths would be one cause for the hazard identified during the PHA stage. This would also identify one potential root fault for the software safety analysts to isolate. A specification requirement that would result out of this would be that the software not display green lights simultaneously to traffic on perpendicular paths. Finally, during SHA or post-design analysis, the software safety analysts would construct a software tree based upon that part of the program associated with the timing of the display and the color of the display of lights to determine if the software meets the specification requirement as stated.

In selecting particular root faults to evaluate, it would be helpful if the software safety analyst has a good understanding of the Ada source code programs that he or she is evaluating. Naturally, this implies that the analyst has an adequate knowledge of the syntax and semantics of the Ada programming language itself. This also implies that good software engineering practices should be used in developing the software as the person(s) responsible for assessing the safety of the software may not be the same person(s) writing code. It is the analyst's job to locate those parts of the source code that may have an affect on causing or preventing the hazards identified by the analysis techniques mentioned above. With this done, the analyst selects those nodes in the statement template tree structures generated by the ACTT tool that correspond to the parts of the source code that the analysts has determined to be relevant to a particular hazard.

B. USE OF THE ACTT TOOL IN SFTA PROCESS

The ACTT tool generates the statement template tree structures that will be used to construct SFTs. Using the ACTT tool involves the simple procedure of typing in the filename of the Ada source code program that is being analyzed when prompted to do so. The ACTT tool will then automatically generate FTE compatible text files using the process described in Chapter I. The only requirements of the user is to ensure that the target file is in the current directory, and that the source code is free of all syntax errors. The ACTT will generate new files and name them automatically. The analyst should note which new files are created as more than one file may be created by the ACTT tool when processing an Ada source code file.

The files generated by the ACTT can then be input into the FTE tool for study to increase the analyst's understanding of the software program being evaluated. This will present to the analyst a graphical model of the semantics of the program. The analyst should be aware that the FTE tool as currently designed can only handle files of a limited size. Until the FTE tool is modified to handle larger size files, this limitation can be somewhat overcome using the FIT tool.

The text files that are generated by the ACTT are essentially a listing of all the nodes in the tree structure which contain information essential to reconstructing the tree when input to both the FTE and FIT tools. The reader will recall that a node in the statement template tree structure represents an event or state in the software program. Each node is described by a series of four consecutive lines that contain various data about the node. A breakdown of what each line represents is as follows:

- **Line 1.** A character string representing the label of the node. This label will be displayed by the FTE tool with the node it is associated with. the ACTT tool generates numeric strings for node labels.
- **Line 2.** A character string representing the fault description of the node.
- **Line 3.** A character string that is the filename of the source code file processed by the ACTT tool.
- **Line 4.** A series of seven integer fields separated by white space. The first and second integers are the starting and ending source code line numbers that

the node is associated with. The third and fourth integers are the X and Y coordinates of the node within the FTE graphical display area. The fifth integer is the type of node (1 - Rectangle, 2 - Circle, 3 - Diamond, 4 - Ellipse, 5 - Pentagon, 6 - Triangle), the sixth integer is the type of gate if any attaching the node to its children nodes (0 - no gate, 1 - AND gate, 2 - OR gate), and the seventh integer is the number of children of the node (See Figure 4 for a node listing description example).

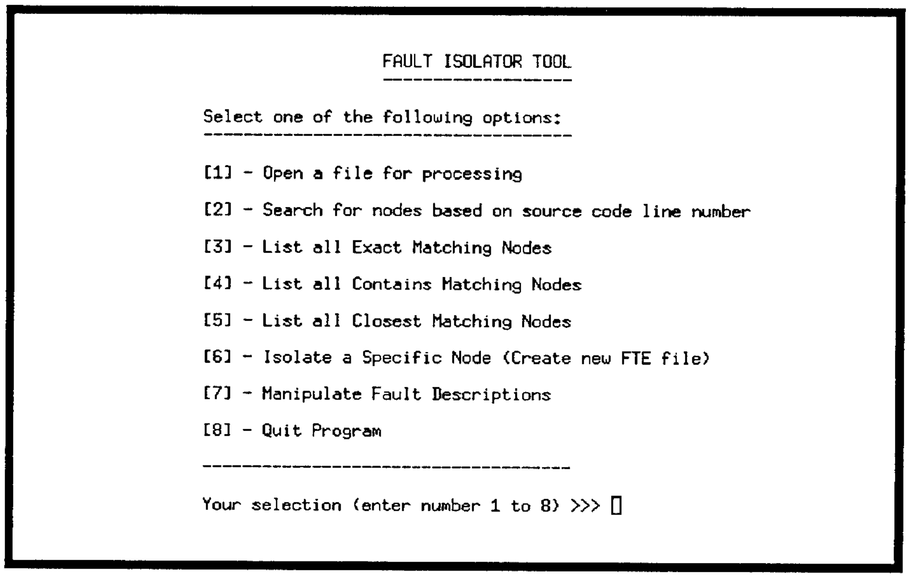
The order in which the nodes are written to the output files is determined by a depth first search or pre-ordering traversal of the entire tree structure. The FTE and FIT tools reconstruct the tree by reversing the preorder mechanism to assign appropriate pointer values to each node that point to their descendant nodes.

```
-- Example Node Values

Label:            "Top"
Fault:            "Node caused fault"
File:             "Example.a"
Start Line:       1
End Line:         55
X Coordinate:     200
Y Coordinate:     300
Type Node:        1
Type Gate:        0
No. Children:     0

-- Example Node Listing in FTE input file

Top
Node caused fault
Example.a
1 55 200 300 1 0 0
```

Figure 4: Example of Node Listing in FTE Compatible File

C. USE OF THE FIT TOOL IN SFTA PROCESS

The FIT tool is a menu driven automated process to assist software safety analysts and programmers in the construction of software fault trees. The user is guided through the process by a series of menus from which several options are available. The Main Menu is the first menu displayed to the user when executing the FIT tool program, and the menu that will be redisplayed upon completion of all other tasks except for exiting the program (See Figure 5). A detailed description of the various options presented in the Main Menu are given in the following sub-sections.[1]



```
                    FAULT ISOLATOR TOOL
                    -------------------
            Select one of the following options:
            ------------------------------------
            [1] - Open a file for processing
            [2] - Search for nodes based on source code line number
            [3] - List all Exact Matching Nodes
            [4] - List all Contains Matching Nodes
            [5] - List all Closest Matching Nodes
            [6] - Isolate a Specific Node (Create new FTE file)
            [7] - Manipulate Fault Descriptions
            [8] - Quit Program
            ------------------------------------
            Your selection (enter number 1 to 8) >>> ▯
```

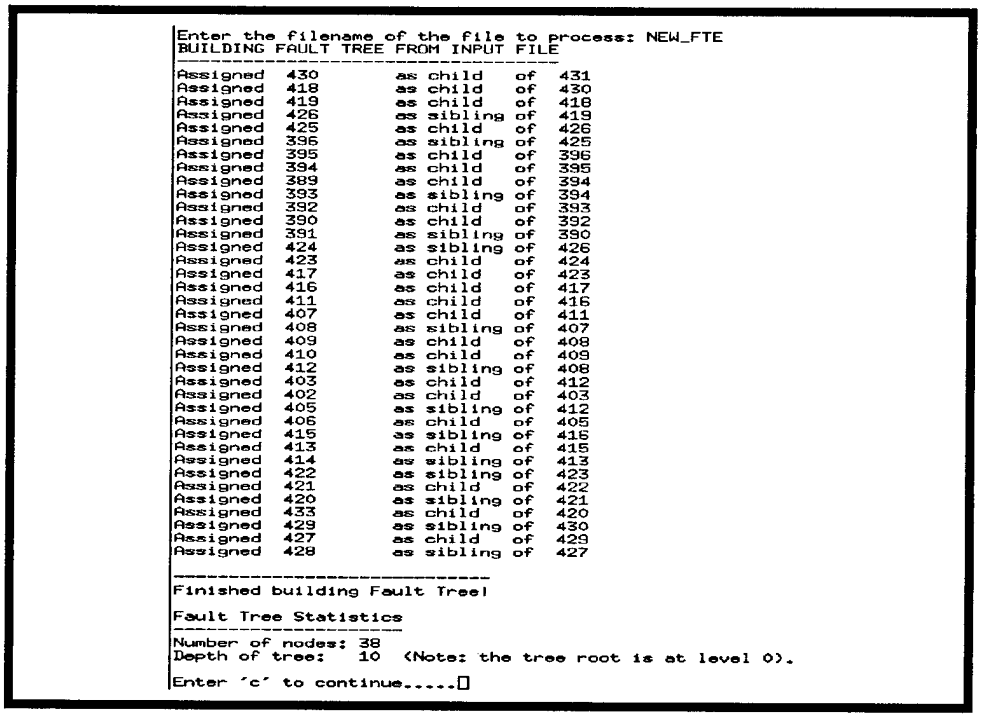
Figure 5: Main Menu Display of Fault Isolator Tool

1. Open a File for Processing

The FIT tool is designed to manipulate statement template tree structure files generated by the ACTT tool. This must be the first option chosen when the program is first executed. Once selected, this option prompts the user for the file name of an FTE compatible file (a file generated by the ACTT tool). Once a valid file name is given, the tool reconstructs the tree structure by assigning appropriate data and pointer values to each node's data structure. The tool provides feedback to the user by displaying the assignment of pointer values. This display may be useful in locating errors in the input file in the event the FIT tool is unable to properly reconstruct the tree (a corrupted file). Upon completion,

1. The "Quit Program" option is not covered for obvious reasons.

the tool will display statistics about the constructed tree including the total number of nodes contained in the tree and the depth of the tree (See Figure 6). This option may also be used to open different files for processing at a single session.



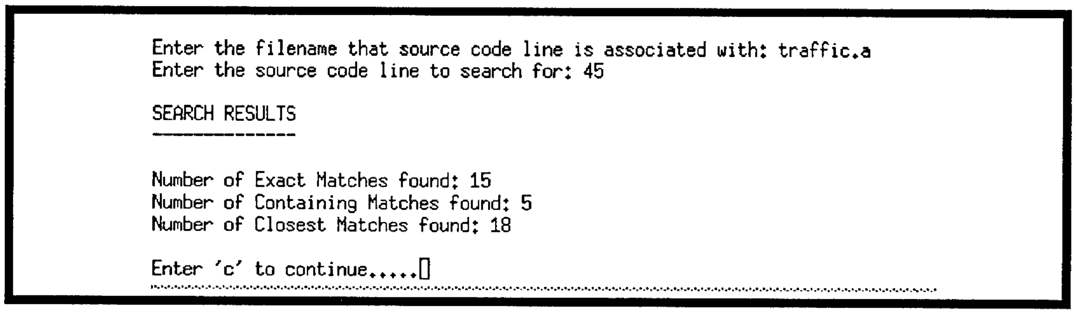
```
Enter the filename of the file to process: NEW_FTE
BUILDING FAULT TREE FROM INPUT FILE
-----------------------------------
Assigned  430        as child   of  431
Assigned  418        as child   of  430
Assigned  419        as child   of  418
Assigned  426        as sibling of  419
Assigned  425        as child   of  426
Assigned  396        as sibling of  425
Assigned  395        as child   of  396
Assigned  394        as child   of  395
Assigned  389        as child   of  394
Assigned  393        as sibling of  394
Assigned  392        as child   of  393
Assigned  390        as child   of  392
Assigned  391        as sibling of  390
Assigned  424        as sibling of  426
Assigned  423        as child   of  424
Assigned  417        as child   of  423
Assigned  416        as child   of  417
Assigned  411        as child   of  416
Assigned  407        as child   of  411
Assigned  408        as sibling of  407
Assigned  409        as child   of  408
Assigned  410        as child   of  409
Assigned  412        as sibling of  408
Assigned  403        as child   of  412
Assigned  402        as child   of  403
Assigned  405        as sibling of  412
Assigned  406        as child   of  405
Assigned  415        as sibling of  416
Assigned  413        as child   of  415
Assigned  414        as sibling of  413
Assigned  422        as sibling of  423
Assigned  421        as child   of  422
Assigned  420        as sibling of  421
Assigned  433        as child   of  420
Assigned  429        as sibling of  430
Assigned  427        as child   of  429
Assigned  428        as sibling of  427

----------------------------
Finished building Fault Tree!

Fault Tree Statistics
---------------------
Number of nodes: 38
Depth of tree:    10  (Note: the tree root is at level 0).

Enter 'c' to continue.....□
```

Figure 6: Example Tree Reconstruction Display

2. Search for Nodes Based on Source Code Line Number

The purpose of this option is to help locate nodes in the tree that correspond to statements at specific line numbers in the source code. The statement template trees generated by the ACTT tool can be very large, and attempting to locate specific nodes that relate to a statement in the source code can be very time consuming and tedious work. Once this option is chosen, the program prompts the user for a file name that the source code is associated with, and the line number in the source code of the statement that the analyst wants located. The tool will then traverse the entire tree structure comparing the start and end line data of each node to the input source code line number, and the node's file name data to the input file name. It uses the results of these comparisons to build three lists:

(1) an Exact Matches List, (2) a Containing Matches List, and (3) a Closest Matches List. The meaning and purpose of each of these three lists is explained below. Upon completion, the tool displays a summary of the the search results listing the number of nodes in each list that was found (See Figure 7).

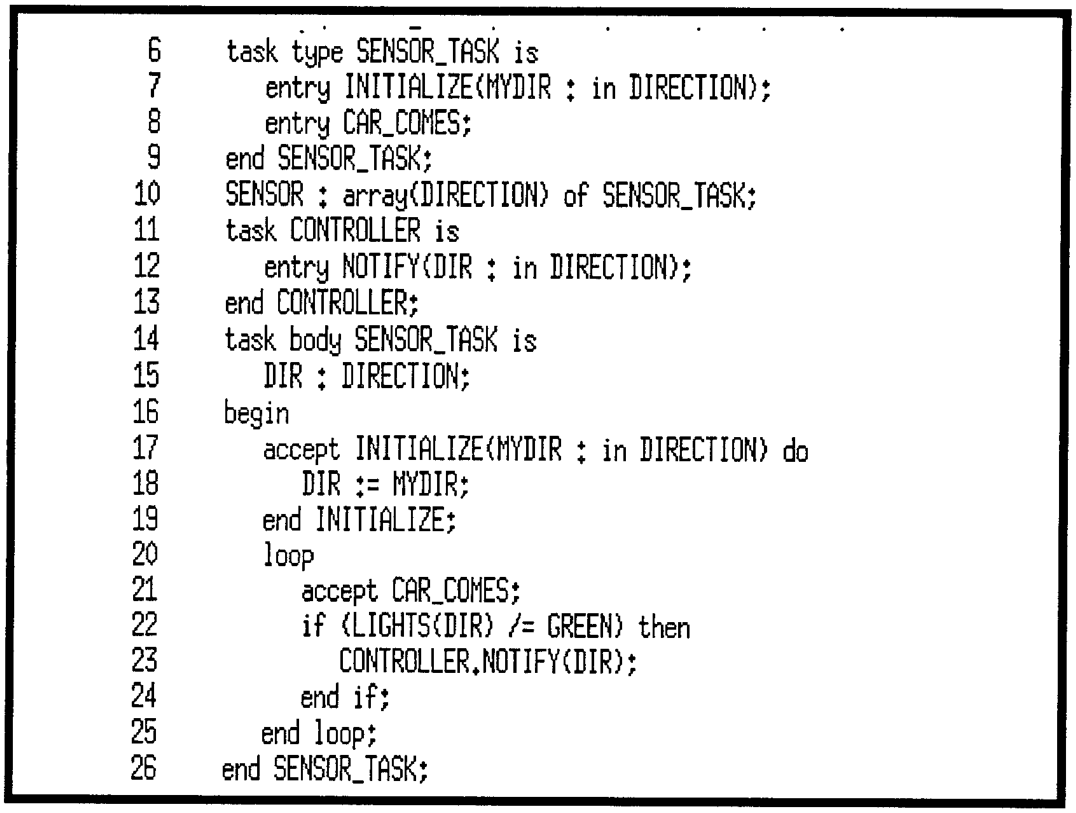


```
Enter the filename that source code line is associated with: traffic.a
Enter the source code line to search for: 45

SEARCH RESULTS
--------------

Number of Exact Matches found: 15
Number of Containing Matches found: 5
Number of Closest Matches found: 18

Enter 'c' to continue.....
```

Figure 7: An Example of the Results of Searching for Nodes that Correspond to a Source Code Line Number in a Specific File

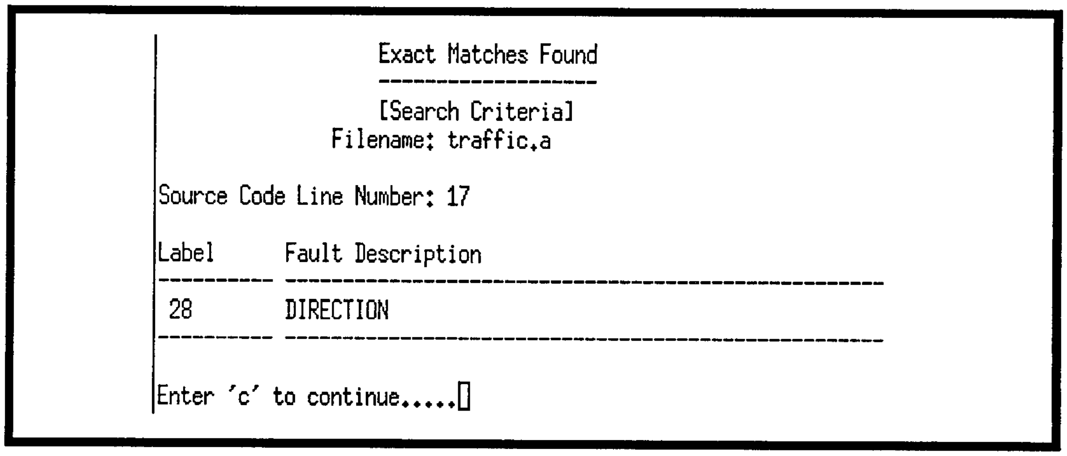
3. List all Exact Matching Nodes

This list is built as a result of a search as described above. The purpose of this option is to display those nodes in the tree structure that satisfy the definition of an Exact Match. The definition of an Exact Match is a node whose start line and end line equal the input source code line number as determined by the template generator component of the ACTT tool. These are usually those statements in the Ada language whose conditions are not dependent on other statements. For example, a simple "put" statement on a single line would meet this criteria. By contrast, an "if" statement usually spans several lines because of the sequence of statements between the "then" reserved word and the "end if;" designating the end of the "if" statement. Figure 8 depicts a portion of an Ada program to use as an example. Figure 9 depicts the resulting Exact Match List from conducting a search for nodes matching source code line number 17. The analyst may then use the results of this listing to isolate a specific node, or simply to locate its position in the statement template tree structure.



```
6     task type SENSOR_TASK is
7        entry INITIALIZE(MYDIR : in DIRECTION);
8        entry CAR_COMES;
9     end SENSOR_TASK;
10    SENSOR : array(DIRECTION) of SENSOR_TASK;
11    task CONTROLLER is
12       entry NOTIFY(DIR : in DIRECTION);
13    end CONTROLLER;
14    task body SENSOR_TASK is
15       DIR : DIRECTION;
16    begin
17       accept INITIALIZE(MYDIR : in DIRECTION) do
18          DIR := MYDIR;
19       end INITIALIZE;
20       loop
21          accept CAR_COMES;
22          if (LIGHTS(DIR) /= GREEN) then
23             CONTROLLER.NOTIFY(DIR);
24          end if;
25       end loop;
26    end SENSOR_TASK;
```

Figure 8: Example Portion of Ada Source Code Program

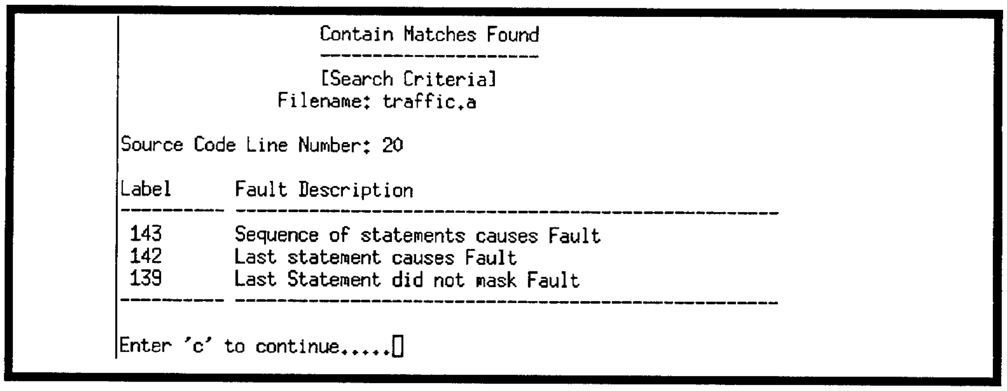


```
                     Exact Matches Found
                     -------------------

                     [Search Criteria]
                 Filename: traffic.a

Source Code Line Number: 17

Label      Fault Description
---------- -------------------------------------------------
 28        DIRECTION
---------- -------------------------------------------------

Enter 'c' to continue.....[]
```

Figure 9: Results from Searching for Exact Matches to Source Code Line Number 17

4. List all Contains Matching Nodes

This list is built as a result of a search described above. The purpose of this option is to display those nodes in the tree structure that satisfy the definition of an Containing Match. The definition of a Containing Match is a node which does not qualify as an Exact Match (start line not equal to end line), and whose start line data is less than or equal to the target source code line number, and whose end line data is greater than or equal to the target source code line number. These nodes may represent statements that encompass the statement at the target source code line number. An analyst may use this list to locate nodes that are relevant to the statement at the target source code line number if no Exact Matching nodes were found, or to use as candidates as root faults for a software fault tree that is more broad than a software fault tree if the statement at the target source code line number were used as the root fault. Using the same portion of Ada source code in Figure 8, the resulting Contains Matching List is displayed in Figure 10 after a search was conducted for the statement at line number 20. The start line data of the three resulting nodes is equal to line 17, corresponding to the "begin" block located at line 16.



```
                        Contain Matches Found
                        ---------------------
                        [Search Criteria]
                     Filename: traffic.a

Source Code Line Number: 20

Label       Fault Description
---------   ------------------------------------------------
 143        Sequence of statements causes Fault
 142        Last statement causes Fault
 139        Last Statement did not mask Fault
---------   ------------------------------------------------

Enter 'c' to continue.....[]
```

Figure 10: Results from Searching for Containing Matches to Source Code Line Number 20

5. List all Closest Matching Nodes

This list is also a result of conducting a search as described earlier. The purpose of this option is to display those nodes in the tree structure that satisfy the definition of an

Closest Match. The definition of a Closest Match is a node whose start line data is greater than the target source code line number but not greater than any other node's start line data that satisfies the first criteria. For example, if the target source code line number is 18, and there are several nodes whose start line data is 19, then all those nodes whose start line is 19 would go on the Closest Match List. This list is necessary because of the manner in which the ACTT tool generates start line data for the nodes in a statement template tree structure. Some of the Ada statements such as the "if" and "loop" statements will have starting lines that reflect the line number of the first statement subsequent to the "if" and "loop" statement lines. For example, if the analyst conducted a search for nodes corresponding to the the "loop" statement on line 20 of Figure 8, the resultant Exact Match List would be empty. In this case, the analysts should then display the Closest Matching List (See Figure 11) for a list of nodes relevant to the "loop" statement on line 20. By examining the resultant list, by studying the data of each of the nodes on the list, and by observing the location of each of the nodes in the list within the statement template tree structure, the analyst should be able to deduce which nodes he thinks most relevant for isolating or some other purpose.

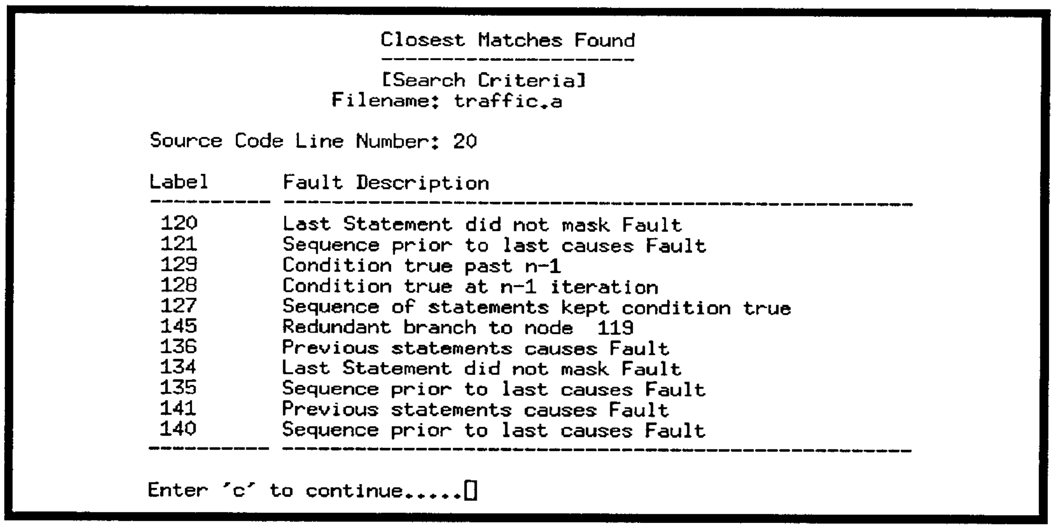
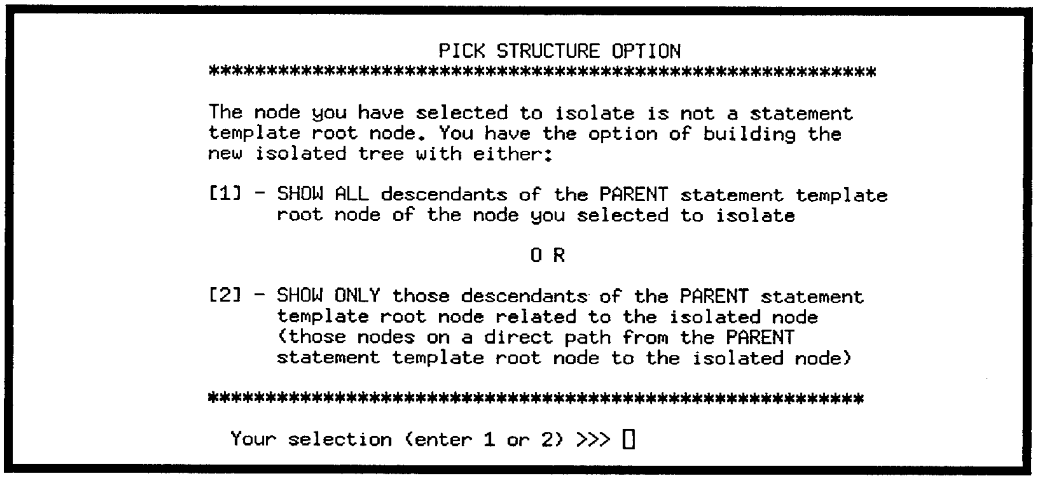

```
                         Closest Matches Found
                         ---------------------
                          [Search Criteria]
                       Filename: traffic.a

  Source Code Line Number: 20

  Label       Fault Description
  ----------  ------------------------------------------------
   120        Last Statement did not mask Fault
   121        Sequence prior to last causes Fault
   129        Condition true past n-1
   128        Condition true at n-1 iteration
   127        Sequence of statements kept condition true
   145        Redundant branch to node  119
   136        Previous statements causes Fault
   134        Last Statement did not mask Fault
   135        Sequence prior to last causes Fault
   141        Previous statements causes Fault
   140        Sequence prior to last causes Fault
  ----------  ------------------------------------------------

  Enter 'c' to continue.....▯
```

Figure 11: Results from Searching for Closest Matches to Source Code Line Number 20

6. Isolate a Specific Node (Create new FTE file)

The purpose of this option is to generate a software fault tree with the root fault of the software tree being a node that the analyst selects. The tool will remove those parts of the statement template tree structure that are not relevant to the selected root fault, and will adjust the X and Y coordinates of the remaining nodes to display the software fault tree starting at the top left of the FTE display area. Once this option has been selected, the program will prompt the user for the label of a node he wishes to isolate. Depending upon the type of node that the analyst has decided to isolate (identified by the fault description of the node given it by the template generator component of the ACTT tool), the program will prompt the user to select a structure option for the new software tree. If the target isolate node is not a statement template root node, the program will display the screen shown in Figure 12.



```
                    PICK STRUCTURE OPTION
**********************************************************

The node you have selected to isolate is not a statement
template root node. You have the option of building the
new isolated tree with either:

[1] - SHOW ALL descendants of the PARENT statement template
      root node of the node you selected to isolate

                           O R

[2] - SHOW ONLY those descendants of the PARENT statement
      template root node related to the isolated node
      (those nodes on a direct path from the PARENT
      statement template root node to the isolated node)

**********************************************************

  Your selection (enter 1 or 2) >>> ▯
```

Figure 12: Choose Structure Option Display

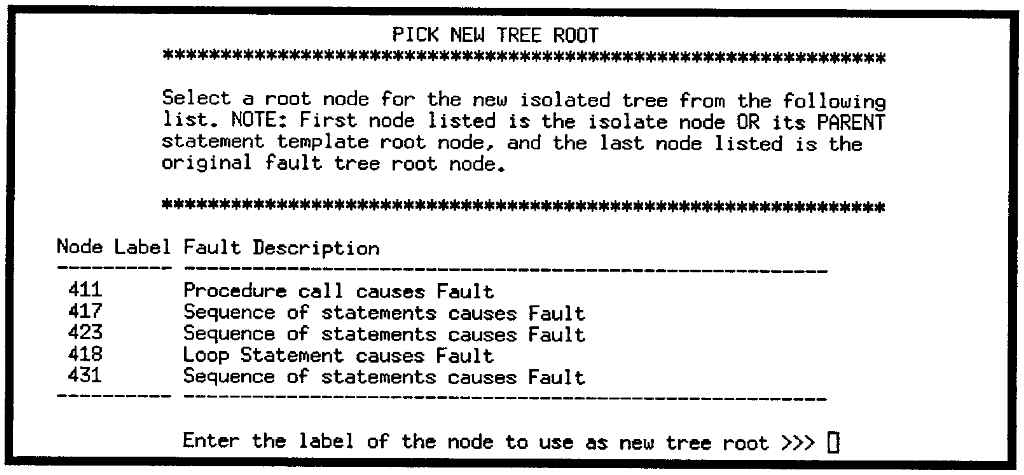
A statement template root node is the top node or root node of the following statement templates:

- Assignment Statement,
- If Statement
- Sequence of Statements
- Loop Statement

- Case Statement
- Procedure Call·Statement
- Block Statement
- Function Call Statement
- Raise Statement
- Delay Statement
- Entry Call Statement
- Abort Statement
- Selective Wait Statement
- Rendezvous Statement

If the targeted isolate node is a statement template root node, this option defaults to showing all descendants of the selected node.

After the structure option has been determined, the tool lists the statement template root nodes that are on a direct path from the target isolate node (or the parent statement template node of the targeted isolate node) back to the original root of the statement template tree structure. The analysts may choose among this list the node to be used as the root node of the software tree (See Figure 13 for an example). The program constructs the software tree dependent upon the chosen options and prompts the user for a file name to save the results in. This file may then be input in to the FTE tool for display and evaluation.



```
                         PICK NEW TREE ROOT
         ************************************************************

         Select a root node for the new isolated tree from the following
         list. NOTE: First node listed is the isolate node OR its PARENT
         statement template root node, and the last node listed is the
         original fault tree root node.

         ************************************************************

Node Label Fault Description
---------- ------------------------------------------------------
 411       Procedure call causes Fault
 417       Sequence of statements causes Fault
 423       Sequence of statements causes Fault
 418       Loop Statement causes Fault
 431       Sequence of statements causes Fault
---------- ------------------------------------------------------

           Enter the label of the node to use as new tree root >>> []
```

Figure 13: Example of Pick New Tree Root Node Display

7. Manipulate Fault Descriptions

The purpose of this option is to provide an automated process for changing the fault descriptions of the nodes in a software fault tree. Changing the fault descriptions of the nodes from the generic fault descriptions given them by the template generator component of the ACTT to more meaningful fault descriptions selected by the analyst will enhance the analyst's performance in studying and evaluating the resultant software fault tree. While the FTE tool provides the capability to change fault descriptions a node at a time, this can be time consuming and therefore better managed by an automated process.

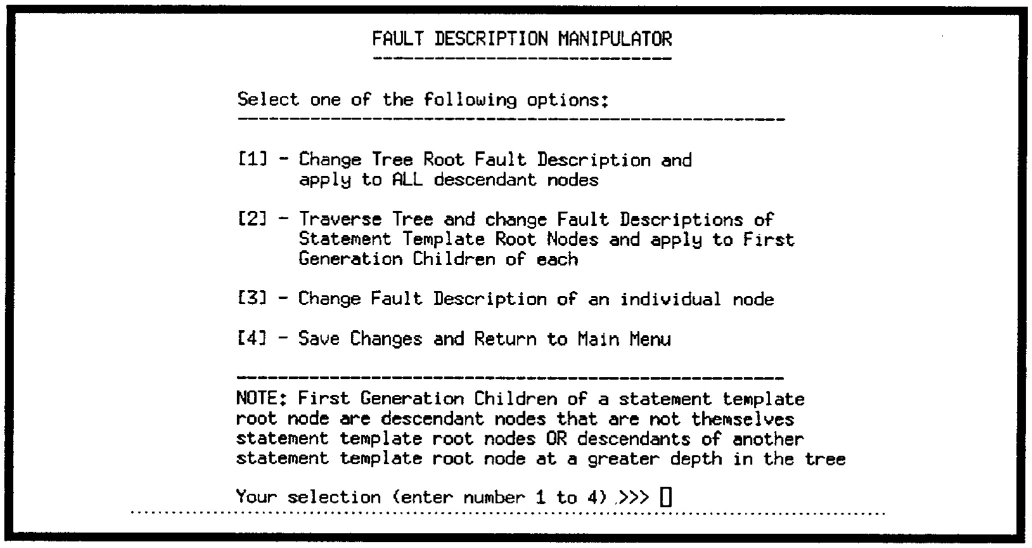
Once this option is selected from the Main Menu, another menu is displayed providing the analyst several options in manipulating the fault descriptions (See Figure 14). All options are self explanatory except for perhaps the second option on the menu. This option causes the program to traverse the tree, stopping at each statement template root node it encounters, prompting the user for a replacement expression for the word "Fault" in the generic fault description given it by the template generator component of the ACTT tool, and applying it to all of its first generation children nodes. This means that it searches for the word "Fault" in the fault description of each of its descendant nodes that are not themselves statement template root nodes or children of other statement template root nodes at a greater depth in the tree, and if found replaces the word "Fault" with the expression provided for the parent statement template root node by the analyst. If the word "Fault" is not found in the fault description of a child node, its fault description remains unchanged. Upon completion, the program prompts the user for a file name to save the results to, and then returns to the Main Menu.

D. FAULT ISOLATOR TOOL INTERNAL STRUCTURE

The FIT tool was implemented in the Ada programming language and consists of six modules:

- the main driver module (contains the main procedure **fi_driver**)
- a package for handling most of the main user interface display features **(inout_pkg)**

- a package that performs most of the work of the FIT tool (**fault_isolator_pkg**)
- a generic package that provides for the construction and manipulation of stack data structures (**stack_pkg**)
- a generic package that provides for the construction and manipulation of linked list data structures containing user defined data (**linklist**)
- and another generic package that provides for the construction and manipulation of linked list data structures containing strictly pointer values (**ptr_linklist**)



```
                    FAULT DESCRIPTION MANIPULATOR
                    -----------------------------

        Select one of the following options:
        ---------------------------------------------------

        [1] - Change Tree Root Fault Description and
              apply to ALL descendant nodes

        [2] - Traverse Tree and change Fault Descriptions of
              Statement Template Root Nodes and apply to First
              Generation Children of each

        [3] - Change Fault Description of an individual node

        [4] - Save Changes and Return to Main Menu

        ---------------------------------------------------
        NOTE: First Generation Children of a statement template
        root node are descendant nodes that are not themselves
        statement template root nodes OR descendants of another
        statement template root node at a greater depth in the tree

        Your selection (enter number 1 to 4) >>> []
```

Figure 14: Fault Description Manipulator Menu

Figure 15 on the next page presents a block structure model of the internal structure of the FIT tool showing the relationships between the various components. The large round edged rectangles represent the different modules, and the square edged rectangles represent the functions and/or procedures contained within each module. The lines from one module to another represent function and/or procedure calls. Those functions and/or procedures internal to a module and not used by another module are not depicted on the diagram. A more detailed description of the various modules, their functions and procedures, and the interface between modules is given in the next few sections.

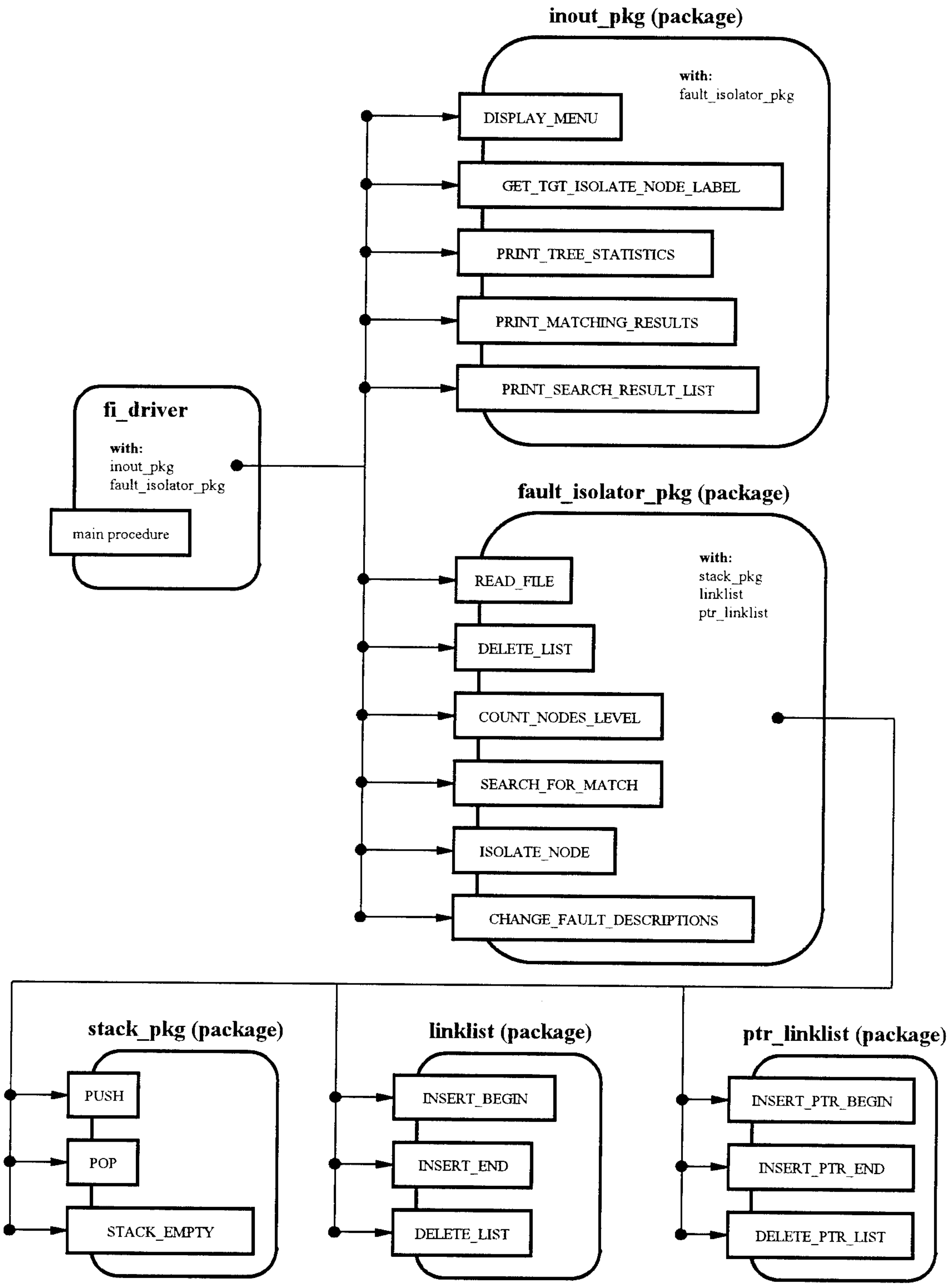


Figure 15: Block Diagram of FIT Tool Internal Structure

1. The Main Driver Module (procedure fi_driver)

Procedure **fi_driver** is the only procedure or function contained in this module. Its main purpose is to keep prompting the user for tasks that the user wants done until the user requests to exit the program. It uses data structures defined in the **inout_pkg** and **fault_isolator_pkg** modules for some of the parameters needed to make function and/or procedure calls to those modules. It also keeps track of the Exact Match, Contains Match, and Closest Match lists described in the previous section whenever the user conducts a search for nodes relevant to a source code line number in an Ada source code file. Once the procedure has prompted the user for a task request and received a reply, it makes the appropriate calls to the functions and/or procedures contained in the **inout_pkg** (for prompt or result displays) or the **fault_isolator_pkg** (to accomplish a specific task).

2. The inout_pkg Module

This module contains functions and procedures for presenting displays to the user for the purpose of either prompting the user for input, or for displaying the results from accomplishing a specific task. It uses ("withs") the **fault_isolator_pkg** for some of the data structures it needs as parameters for some of its functions and procedures. Specifically, it uses the pointer type defined in the **fault_isolator_pkg** that points to a tree node (FT_node_ptr), and it also uses the pointer type defined in the **fault_isolator_pkg** that points to a linked list element (FT_linklist.link). Its only interface is with the **fi_driver** module which calls the following functions and/or procedures:

- function DISPLAY_MENU. This function displays the main menu depicted in Figure 5 and prompts the user for his choice. It has no parameters and returns an integer corresponding to the user's choice.
- function GET_TGT_ISOLATE_NODE_LABEL. This function prompts the user to enter the label of the particular node he desires to isolate. It has no parameters and returns a string value.
- procedure PRINT_TREE_STATISTICS. This procedure prints out the statistics for a tree after reading in an input file and reconstructing the connecting pointer values (bottom of Figure 6). It takes two natural (integer) value parameters corresponding to the number of nodes in the tree and the

depth of the tree and does not return any parameters (parameters are "in" parameters only).

- procedure PRINT_MATCHING_RESULTS. This procedure prints out the results from conducting a search for all nodes relevant to a source code line number (See Figure 7). It has three "in" parameters that are linked list pointer values (defined in the fault_isolator_pkg) that represent the first nodes in the Exact Match, Contains Match, and Closest Match lists described earlier.
- procedure PRINT_SEARCH_RESULT_LIST. This procedure prints one of the three lists depicted in Figure 9, Figure 10, or Figure 11. Its parameters are an enumeration type that serves as a flag indicating which type of list is to be displayed (Exact Match, Contains Match, or Closest Match list), a string value indicating the filename that the target source code line number is associated with, a natural (integer) value indicating the target source code line number, and a linked list pointer value that is the first (head) node in the corresponding list.

3. The fault_isolator_pkg Module

This module contains the functions and procedures that accomplish most of the significant work of the FIT tool. This is where the primary data structures used by other modules are defined such as FT_Node and FT_Node_Ptr (the basic node elements that make up a software tree). This module also instantiates the generic **stack_pkg, linklist**, and **ptr_linklist** packages. Some of the data structures defined in this module are used by both the **fi_driver** module and the **inout_pkg** module. This module interfaces with the **fi_driver** module by providing the following procedures called by that module:

- procedure READ_FILE. This procedure prompts the user for the file-name of an FTE-compatible file (the kind generated by the ACTT tool), opens the input file if able, and then reconstructs the tree structure by assigning appropriate pointer values to the Child and Sibling pointers of each of the individual nodes in the tree. Its only parameter is an "in out" FT_Node_Ptr value that will point to the root node of the constructed tree.
- procedure DELETE_LIST. This procedure is an instantiation of the DELETE_LIST procedure defined in the generic **linklist** package. Its purpose is to delete a linked list by releasing the memory storage used to keep track of each of the elements in a linked list. Its parameter is an "in out" linked list pointer value that will be returned as a null value.

- procedure COUNT_NODES_LEVEL. This procedure traverses an entire tree structure counting the number of nodes in the tree and calculating the maximum depth. It has one "in" parameter that is of type FT_Node_Ptr (a pointer to the root node of the tree), and has two natural (integer) "out" parameters that correspond to the number of nodes and the max depth of the tree.
- procedure SEARCH_FOR_MATCH. This procedure traverses an entire tree structure and builds the Exact Match, Contains Match, and Closest Match lists described earlier. It has three "in out" linklist pointer value parameters that will be passed back to the calling module with pointers to the head nodes in each of the three lists, an "in" FT_Node_Ptr parameter corresponding to the root node of the tree, an "out" string parameter that corresponds to the file-name of the file that the target source code line number is associated with, and an "out" natural (integer) parameter that corresponds to the target source code line number.
- procedure ISOLATE_NODE. This procedure is responsible for constructing a new software fault tree by isolating a node that the user selects which is contained in a statement template tree structure. This procedure removes or "prunes" those parts of the original statement template tree structure that are not relevant to the selected targeted isolate node and saves the results to a file. It has two parameters, an "in out" string parameter corresponding to the label of the node that the user wants to isolate, and an "in" FT_Node_Ptr parameter that corresponds to the root node of the original statement template tree structure.
- procedure CHANGE_FAULT_DESCRIPTIONS. This procedure provides for the fault description manipulation functionality described earlier (See Figure 14). Its only parameter is an "in" FT_Node_Ptr parameter corresponding to the root node of the tree.

4. The stack_pkg Module

This module is a generic package that provides those functions and procedures necessary for stack data structure operations. Most of the tree traversal operations used by many of the functions and procedures in the **fault_isolator_pkg** involve a stack data structure algorithm. The generic package defined in this module is instantiated by the **fault_isolator_pkg** module, and the **fault_isolator_pkg** module is the only module that interfaces with this module. This module provides the following stack functions and procedures:

- procedure PUSH. This procedure pushes an object onto a stack. The two parameters are an "in out" Stack_Obj_Ptr (private access type that corresponds to a pointer to the stack), and an "in" Object_Type Parameter (a generic private type defined when instantiated). This procedure pushes the Object_Type onto the stack pointed to by Stack_Obj_Ptr, and returns the pointer to the stack.
- procedure POP. This procedure takes off the top item on a stack and returns the item to the calling routine. The two parameters are an "in out" Stack_Obj_Ptr (a pointer to the stack), and an "out" Object_Type (the item that is returned).
- function STACK_EMPTY. This function returns true if the stack pointed to by the "in" parameter of Stack_Obj_Ptr is empty (nothing in the stack), and false otherwise.

5. The linklist Module

This module provides the procedures necessary for linked list data structure operations. The FIT tool relies on several link lists data structures to accomplish many of its functions. Some of these linked lists have already been described such as the Exact Match, Contains Match, and the Closest Match lists. This module provides the means of linked list operations through the following procedures:

- procedure INSERT_END. This procedure adds a link list item to the end of a linked list. Its two parameters are a pointer to the first node of a linked list (the list head), and the item to be added to the end of the linked list.
- procedure INSERT_BEGIN. This procedure operates similar to the INSERT_END procedure just described except that it adds a data item to the beginning of a linked list. The parameters are the same.
- procedure DELETE_LIST. This procedure deletes an entire linked list by returning the memory storage keeping track of the data items (or nodes) in the linked list back to free memory. Its only parameter is a pointer to the first node in the linked list.

6. The ptr_linklist Module

This module is almost exactly like the linklist module described above except that the data items contained in the linked list are exclusively pointer values. This module was necessary for some of the operations performed when isolating a node as described in the section on the **fault_isolator_pkg** module such as building a linked list of pointer values representing the path from the root node of a statement template tree structure to the targeted isolate node. The same procedures and their corresponding parameters were used for this module as was used for the **linklist** module.

E. USE OF THE FTE TOOL IN SFTA PROCESS

The FTE is an interactive graphical editor designed specifically for the study and evaluation of software fault trees. It was designed by Charles P. Lombardo, a computer systems programmer at the Naval Postgraduate School in Monterey, California. FTE was developed using the C programming language and XView, an OPEN LOOK toolkit for the X11 Windowing system.

Although FTE provides an semi-automated process for constructing software fault trees through functions that will draw the various components and structures that make up a software fault tree using a basic point and click mouse interface, this is still considered a manual means for producing software fault trees. The time consuming element of constructing software trees is translating the semantic structure of a software program into a software fault tree. Furthermore, the interpretive nature of translating source code statements into software fault tree constructs tends to introduce errors and inconsistencies into the process. This translation process is better achieved through automated means which not only performs faster, but also produces more accurate and consistent results. This is the primary function of the ACTT and FIT tools.

However, the files generated by the ACTT and FIT tools would be useless to software safety analysts if the end result was not a graphical software fault tree that an analyst could study and evaluate. This is why the FTE tool is such an essential part of an automated SFTA process. It provides the graphical display of the logical interrelationships of the programming language structures that form a software program. It is this graphical display that the analyst will study to assess the degree of safety of software.

III. PRACTICAL APPLICATION EXAMPLE

To help explain the processes and tools described in the previous chapter, we will demonstrate how they might be applied in evaluating a small Ada program for faults that could lead to hazardous conditions. For sake of brevity, we will select only one particular fault to evaluate. This will suffice to demonstrate the principles and procedures described earlier, and in particular how the FIT tool might be used to aid programmers and software safety analysts.

A. THE TARGET SOFTWARE PROGRAM

The software program that we will be evaluating is one written by Cha, Leveson, and Shimeall in their paper on the application of Fault Tree Analysis to Ada programs [Cha 87]. The program that is to be evaluated is part of a traffic control system. Cha, Leveson, and Shimeall described the system as:

> A traffic light control system at an intersection consists of four (identical) sensors and a central controller. The sensors in each direction detect cars approaching the intersection. If the traffic light currently is not green, the sensor notifies the controller so that the light will be changed. A car is expected to stop and wait for a green light. If the light is green already, the car may pass the intersection without stopping. The controller accepts change requests from the four sensors and arbitrates the traffic light changes. Once the controller changes the light in one direction (east-west or south-north) to green, it maintains the green signal for five seconds so that other cars in the same direction may pass the intersection without stopping. Before the green light in any direction becomes red, it should remain in yellow for one second so that any car present in the intersection may clear. The light then turns to red while the light in the opposite direction turns green.

The Ada program used to implement the traffic control system is shown in Figure 16. Note that the program consists of two tasking elements (task type SENSOR_TASK and task CONTROLLER) contained within the main procedure TRAFFIC. An initialization (lines 17 through 19 and 45 through 47) is needed to assign a direction to each sensor because of the "... asymmetric nature of the Ada rendezvous (e.g., the called task does not know the identity of the calling task)" [Cha 87]. When a sensor requests the controller to change the lights, it passes the direction of the sensor to the controller. Note that the

```
procedure TRAFFIC is
  type DIRECTION is (EAST, WEST, SOUTH, NORTH);
  type COLOR is (RED, YELLOW, GREEN);
  type LIGHT_TYPE is array(DIRECTION) of COLOR;
  LIGHTS : LIGHT_TYPE := (GREEN, GREEN, RED, RED);
  task type SENSOR_TASK is
    entry INITIALIZE(MYDI R : in DIRECTION);
    entry CAR_COMES;
  end SENSOR_TASK;
  SENSOR : array(DIRECTION) of SENSOR_TASK;
  task CONTROLLER is
    entry NOTIFY(DIR : in DIRECTION);
  end CONTROLLER;
  task body SENSOR_TASK is
    DIR : DIRECTION;
  begin
    accept INITIALIZE(MYDIR : in DIRECTION) do
      DIR := MYDIR;
    end INITIALIZE;
    loop
      accept CAR_COMES;
      if (LIGHTS(DIR) /= GREEN) then
        CONTROLLER.NOTIFY(DIR);
      end if;
    end loop;
  end SENSOR_TASK;
  task body CONTROLLER is
  begin
    loop
      accept NOTIFY(DIR : in DIRECTION) do
        case DIR is
          when EAST | WEST =>
           LIGHTS := (GREEN, GREEN, RED, RED); delay 5.0;
           LIGHTS := (YELLOW, YELLOW, RED, RED); delay 1.0;
           LIGHTS := (RED, RED, GREEN, GREEN);
          when SOUTH | NORTH =>
           LIGHTS := (RED, RED, GREEN, GREEN); delay 5.0;
           LIGHTS := (RED, RED, YELLOW, YELLOW); delay 1.0;
           LIGHTS := (GREEN, GREEN, RED, RED);
          end case;
        end NOTIFY;
      end loop;
  end CONTROLLER;
 begin
   for DIR in EAST .. NORTH loop
     SENSOR(DIR).INITIALIZE(DIR);
   end loop;
 end TRAFFIC;
```

Figure 16: Ada Implementation of Traffic Light [Cha 87]

detection of an oncoming car by a sensor causes the execution of line 21 and that the actual passing of the car through the intersection begins when the program execution passes line 24 of the sensor task. This program serves as a simple example of the use of software as a controlling agent in systems where faults in the software could lead to hazardous conditions.

B. DETERMINING A ROOT FAULT TO EVALUATE

As the previous chapter explained, the first stage in a SFTA process consists of determining which conditions in the system are considered hazardous, and more specifically, which of these conditions could come about as a result of faults in the software. For the simple system being used in this example, one obvious hazardous condition would be the event of two (or more) vehicles traveling on perpendicular paths in the intersection at the same time. For instance, a car traveling north and a car traveling east in the intersection simultaneously would certainly constitute a hazard as anyone who has ever been in a car collision could attest.

A non-trivial matter in identifying potential root faults for evaluation is that of determining which hazardous conditions can be directly attributed to a failure in the software as opposed to other elements (such as hardware or human factors) in the system. For example, we have identified two cars on opposing (perpendicular) paths in the intersection at the same time as a hazardous condition in the system. However, this condition may or may not come about as a direct result of faults in the software. If the placement of the traffic lights inhibits visibility (hardware or physical element), or if drivers run red lights (human factor), this could also cause the hazardous condition identified above, but is not something that can be attributed to faulty software. In this example, we can identify the event of a simultaneous display of green lights to opposing traffic as a cause for the identified hazardous condition that can be directly attributed to the software since the software is responsible for controlling the display of the colored lights.

Another point that must be considered is that the software may have faults due to other non-software characteristics of the system. Clearly, the established speed limit of the road system which the traffic control system is a part of will have an influence on the display length of a yellow light. The amount of time that a yellow light is displayed that is adequate for slow traffic may not be adequate for traffic traveling at higher speeds. In determining hazardous conditions that could be caused by faults in the software, safety analysts must consider the entire system in which the software operates.

For the purpose of this example, the focus of this evaluation will be narrowed to the event that two cars approaching from the north and east are in the intersection at the same time. Specifically, this evaluation will analyze the software to see if it could cause two cars to be in the intersection at the same time if a car approaching from the north observes a green light as it approaches the intersection as a car traveling east desires to enter the intersection. This is the same example condition evaluated initially in the [Cha 87] reference without any automated tools, and further elaborated by Reid [Reid 94] in the context of the use of the ACTT tool.

C. USE OF ACTT TOOL WITH TRAFFIC CONTROL EXAMPLE

After identifying a particular root fault to evaluate, the next step in the automated SFTA process is to generate statement template tree structures by using the ACTT. This is a simple procedure of executing the ACTT program and typing in the file name of the Ada source code program. More than one statement template tree structure file may be created by the ACTT tool. In this example, the ACTT tool generates three separate files after parsing the input Ada code: (1) NEW_FTE, (2) TASK_BODYA, and (3) TASK_BODYB.

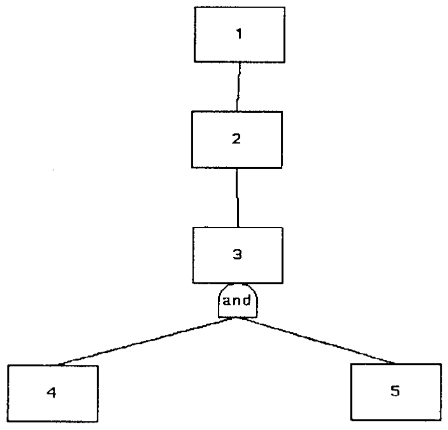
At this stage, the analyst is able to input the files generated by the ACTT tool into the FTE tool. This will present to the user a graphical display of the statement template tree structures associated with the Ada program. This could help increase the analyst's understanding of the logical relationships between the various Ada statements in the source code. Furthermore, by observing the start and end line values of the nodes within the different statement template tree structures, the analyst may be able to discern which of the three files to use as input to the FIT tool. A study of the graphical tree structures of the three files using the FTE and the text form of the three files reveals that the file NEW_FTE is associated with the source code line statements from line 45 to line 47, TASK_BODYA is associated with those statements from line 17 to line 27, and that TASK_BODYB is associated with those statements from line 30 to line 43. This information will be of use when deciding which file to open to isolate a specific node.

D. USE OF FIT TOOL WITH TRAFFIC CONTROL EXAMPLE

The root fault to evaluate is the event that two cars are in the intersection when a car approaching the intersection from the north observes a green light and enters the intersection without stopping and a car approaching the intersection from the east desires to enter the intersection. This implies that the **sensor(east)** task is in rendezvous with the controller task (source code line number 23) and that the **sensor(north)** task bypassed the rendezvous point (source code line number 25 because of the null else). Figure 17 depicts the top level fault tree for this condition. Note that the tree indicates that the top event could happen if and only if both sub-events occur [Cha 87]. The events **sensor(east)** in rendezvous (node 4 in the top level fault tree) and **sensor(north)** at line 24 (node 5 in the top level fault tree) can now be evaluated separately to see how these two events could occur.

1. Evaluating the Sensor(north) at Line 24 Event

Looking at the **sensor(north)** at line 24 event, it is necessary to trace the program backward from line 24 to see how the **sensor(north)** task bypasses the rendezvous point. An examination of the source code reveals that the **if** statement located at line 22 would be the best candidate for the root fault of this event to look at. The TASK_BODYA file generated by the ACTT tool can be input to the FIT tool to isolate the node associated with this statement. TASK_BODYA generates a statement template tree structure which is too large to be displayed here. To identify which node in the tree structure is associated with the **if** statement located at line number 22, a search is conducted choosing option two of the FIT tool main menu. Examining the resulting Exact Match, Contains Match, and Closest Match lists to reveals which node represents the **if** statement template associated with this

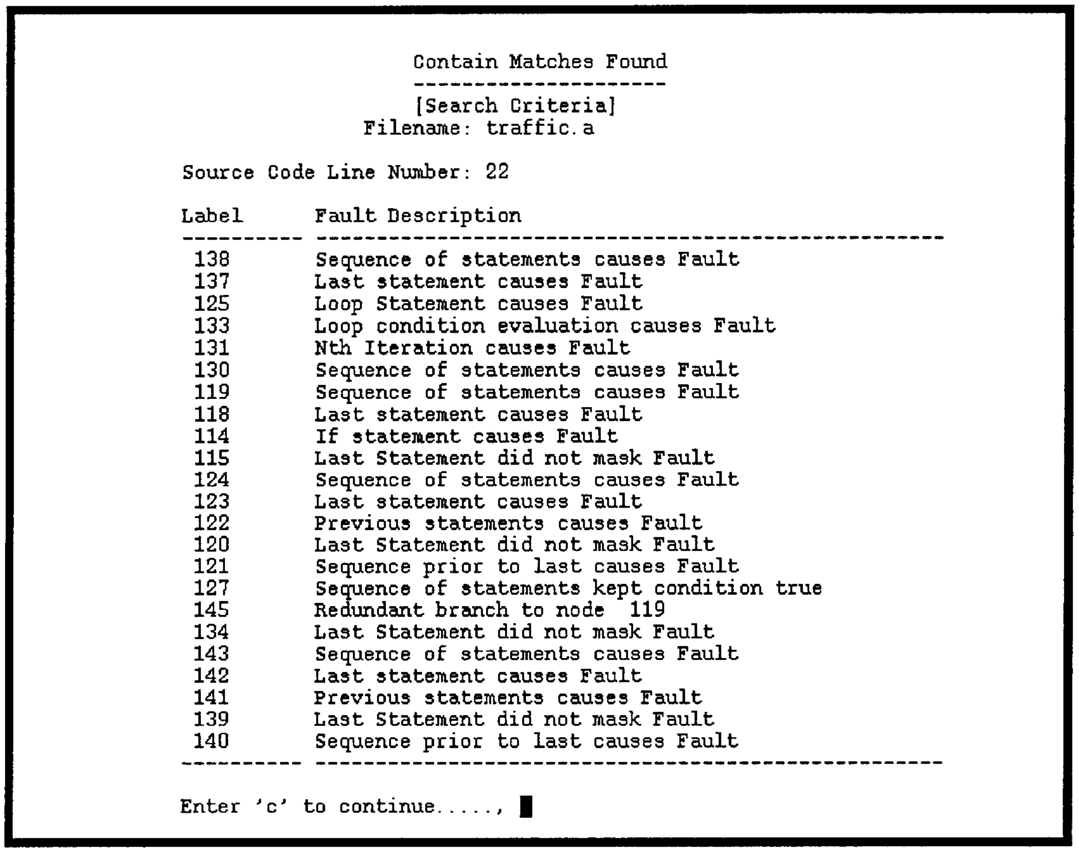


Node	Fault Description
1	Cars present at north and east of intersection
2	North car enters as east car enters intersection
3	Sensor(E) in rendezvous and sensor(N) at line 24
4	Sensor(E) in rendezvous
5	Sensor(N) at line 24

Figure 17: Top Level Fault Tree [Cha 87]

statement. It can be identified by the fault description: "If statement causes Fault". Figure 18 shows the resulting Contains Match list from conducting the search. The observation made is that the fault description for node 114 identifies it as the **if** statement template root node, and therefore is the node that will isolated. Choosing option six from the FIT tool main menu and then entering 114 as the node to isolate results in the software fault tree depicted in Figure 19.

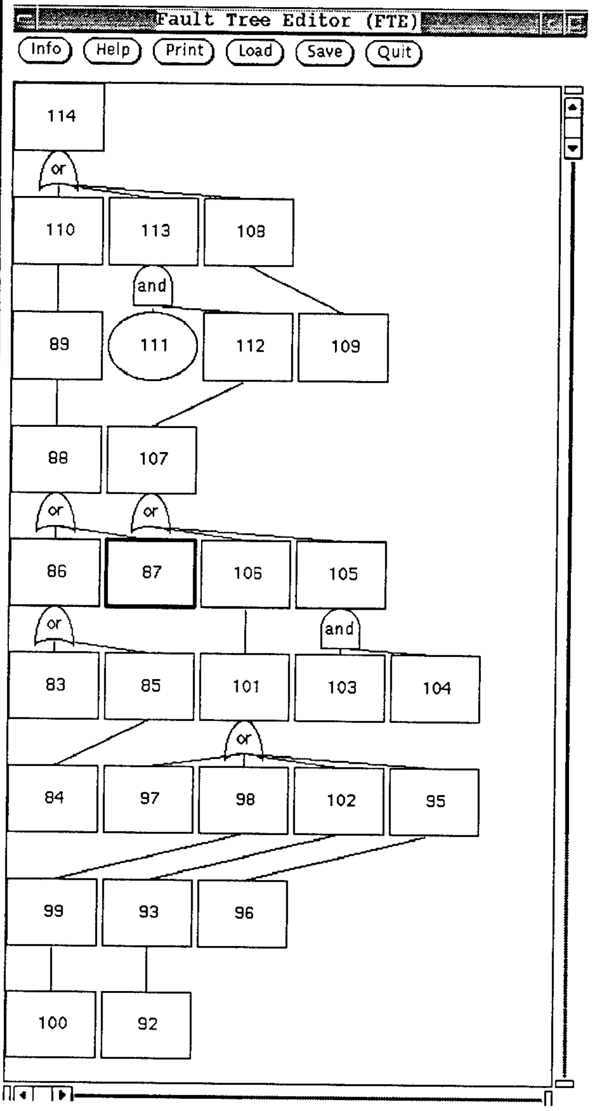
Next, using the FTE tool, the tree generated by isolating node 114 is further refined. It is known that the **sensor(north)** task bypasses the rendezvous with the controller (the condition given to evaluate was that the light was green for the car approaching from the north), therefore any further analysis of node 113 and its subtree can be terminated. This is indicated by replacing node 113's current symbol (a rectangle) with a diamond indicating an undeveloped event. By the same reasoning, the rectangle shape of node 86 is replaced with a diamond because it is known that the indexed component of the **if** statement does not cause the root fault being evaluated—**sensor(north)** at line 24. After some manipulation of the fault descriptions of the remaining nodes in the software fault tree, the resultant software fault tree is shown in Figure 20.



```
                          Contain Matches Found
                          ---------------------
                          [Search Criteria]
                        Filename: traffic.a

Source Code Line Number: 22

Label      Fault Description
---------- ----------------------------------------------------
 138       Sequence of statements causes Fault
 137       Last statement causes Fault
 125       Loop Statement causes Fault
 133       Loop condition evaluation causes Fault
 131       Nth Iteration causes Fault
 130       Sequence of statements causes Fault
 119       Sequence of statements causes Fault
 118       Last statement causes Fault
 114       If statement causes Fault
 115       Last Statement did not mask Fault
 124       Sequence of statements causes Fault
 123       Last statement causes Fault
 122       Previous statements causes Fault
 120       Last Statement did not mask Fault
 121       Sequence prior to last causes Fault
 127       Sequence of statements kept condition true
 145       Redundant branch to node  119
 134       Last Statement did not mask Fault
 143       Sequence of statements causes Fault
 142       Last statement causes Fault
 141       Previous statements causes Fault
 139       Last Statement did not mask Fault
 140       Sequence prior to last causes Fault
---------- ----------------------------------------------------

Enter 'c' to continue....., █
```

Figure 18: Contains Match List from Search for Line Number 22

By examining the resultant SFT, the analyst can deduce that for the **sensor(north)** task to bypass the rendezvous, **lights(north)** must be green (indicated by node 87). Node 108 and its subtree indicates that a race condition will occur if the **north(sensor)** is checking this as the **controller** task changes lights at the request of the **sensor(east)** task. This requires that the analyst examine the behavior of the **sensor(east)** task in rendezvous with the **controller** task to determine if a race condition is possible.

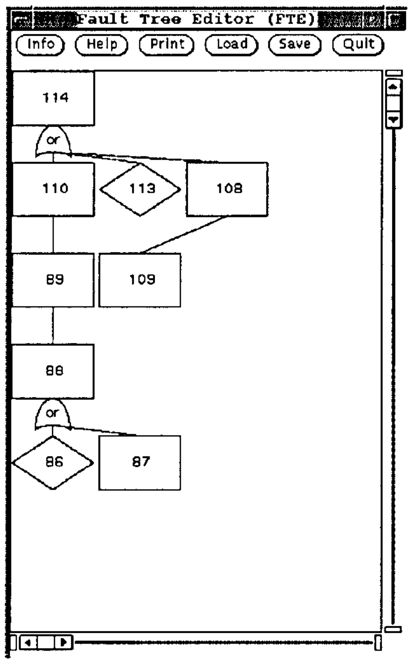


Node	Fault Description
114	If statement causes Fault
110	Evaluation of condition causes Fault
113	Condition true and statements causes Fault
108	Action by other task on variable causes Fault
89	Relation causes Fault
111	If condition true
112	Then statements causes Fault
109	DIR
88	Relation causes Fault
107	Sequence of statements causes Fault
86	Indexed Component causes Fault
87	GREEN
106	Last statement causes Fault
105	Previous statements causes Fault
83	Condition true and statements causes Fault
85	Relation causes Fault
101	Procedure call causes Fault
103	Last statement did not mask Fault
104	Sequence prior to last causes Fault
84	DIR
97	Procedure elaboration causes Fault
98	Procedure body causes Fault
102	Parameter evaluation causes Fault
95	Action by other task causes Fault
99	NOTIFY
93	Relation causes Fault
96	DIR
100	Procedure not found on table
92	DIR

Figure 19: Resultant Software Fault Tree From Isolating Node 114

2. Evaluating the Sensor(east) in Rendezvous Event

To isolate the node to build the SFT for the rendezvous event, it will be necessary to load the TASK_BODYB file (generated by the ACTT tool) into the FIT tool. This statement template tree structure is also too large to be displayed here. Next, a search is conducted for nodes in the tree corresponding to source code line number 30. Within this

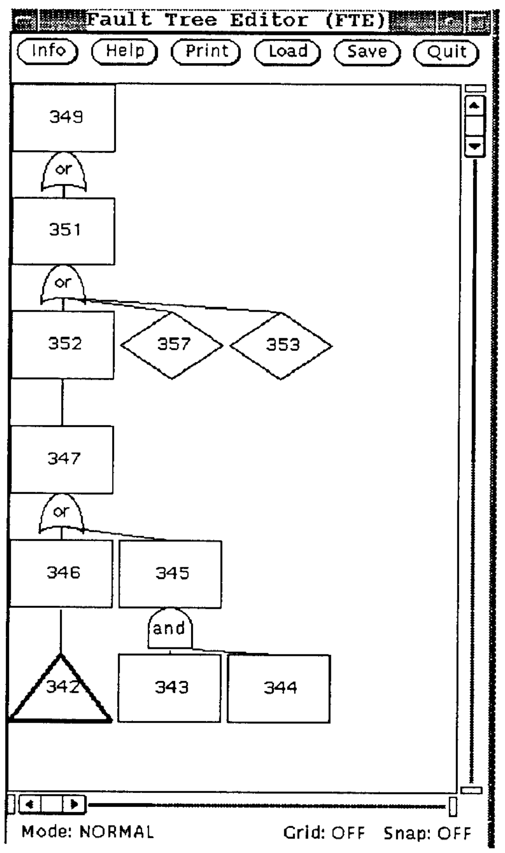


Node	Fault Description
114	If statement causes sensor(N) at line 24
110	Evaluation of condition causes sensor(N) at line 24
113	undeveloped event
108	Action by other task on variable causes sensor(N) at line 24
89	Relation causes sensor(N) at line 24
109	DIR
88	Relation causes sensor(N) at line 24
86	undeveloped event
87	GREEN causes sensor(N) at line 24

Figure 20: Resultant Software Fault Tree for If Statement at Line 22

list a search is conducted for the node that represents the rendezvous event statement template root node. This can be identified by the fault description: “Rendezvous causes Fault”. Node 349 is the only node found with this fault description in the Contains Match List after conducting the search. Node 349 is then selected as the node to be isolated, and after some manipulation with the FTE tool the resultant tree is shown in Figure 21.

This software fault tree is then altered to reflect the event being analyzed. Node 357 was excluded as a possible cause of the root event because examination of the code does not indicate that there is a task abortion. Likewise for node 353 as the source code does not indicate that parameter evaluation caused the root event either although this is a decision that is at the discretion of the analyst. Because the tree structure generated from isolating node 349 is still very large (189 nodes and a depth of 29), the shape of node 342 was switched to a triangle indicating a subtree needing further evaluation. At this point the analyst could isolate node 342 using the FIT tool. However, examining the code and the top



Mode: NORMAL Grid: OFF Snap: OFF

Node	Fault Description
349	Rendezvous causes Fault
351	Fault in Rendezvous
352	Accept Body causes Fault
357	Task aborted (changed to undeveloped event)
353	Parameters cause Fault (changed to undeveloped event)
347	Sequence of statements causes Fault
346	Last statement causes Fault
345	Previous statements causes Fault
342	Case statement causes Fault (changed to subtree for further evaluation)
343	Last Statement did not mask Fault
344	Sequence prior to last causes Fault

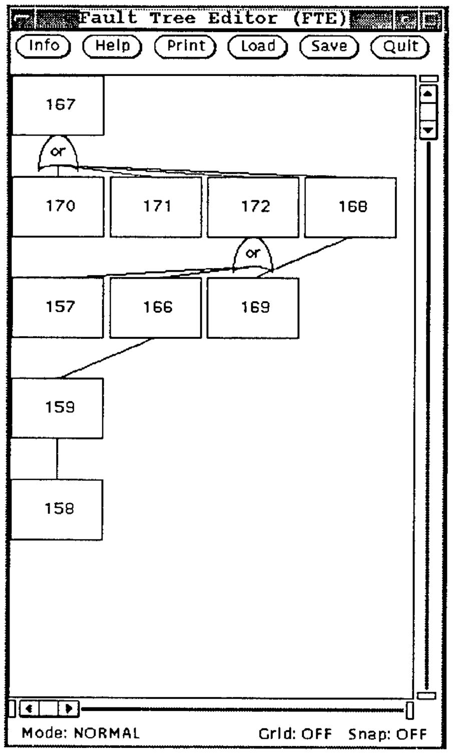
Figure 21: Sensor(east) in Rendezvous Software Fault Tree

event being evaluated provides a shortcut to the statement template root node that will eventually need to be examined. Because node 342 is the statement template root node for the case statement at line 31, and because the top event specifies that the direction is east, it can be deduced that the fault, if any, would have to occur in the body of the rendezvous represented in lines 33 through 35. However, line 34 can be excluded because the specification allows for the light to be yellow for one second for cars to clear the intersection. Using the same logic excludes the delay statement on line 33 as well. Therefore, the only statement left to be analyzed is the assignment statement on line 33. The statement template root node within the TASK_BODYB tree structure that corresponds to the assignment statement on line 33 can be found and isolated using the FIT tool (See Figure 22).

Examining the assignment statement template it was determined that node 172 ("Operand Evaluation causes Fault"), node 171 ("Exception causes Fault"), and node 168 ("Action by other task on variable causes Fault") are not applicable in this case. That leaves only node 170 ("Change in values causes Fault") that needs to be considered. One of the requirements in the specification was that no light change from green to red is allowed without a one second yellow light display in between. However, if the immediately preceding rendezvous was with the east or west sensor tasks, line 35 would have set the north and south light display to green, and this rendezvous would have set the north and south light display to red without an intervening yellow light. As expected, this is the same conclusion reached in reference [Cha 87] working without the benefits of automated tools.

E. PRACTICAL APPLICATION CONCLUSIONS

The question may be asked "If the results were the same, what is the purpose of using automated tools in analyzing software?". One answer to this question would be that the time necessary to analyze software using the automated tools presented in this paper will be considerably less than the time necessary using purely manual techniques of software fault tree construction. This becomes an important issue when it is considered that this example only evaluated one possible hazardous event out of numerous possible hazardous events associated with this Ada program. Each evaluation of the many hazardous events that would need to be evaluated to arrive at a determination of the safety degree of this software would require construction of the various software fault trees associated with each event being analyzed. The time required to construct relevant software fault trees for each event being analyzed may be too prohibitive for SFTA to be an effective method for software safety analysis, even for the small Ada program used in this example.



Node	Description
167	Assignment Statement causes Fault
170	Change in values causes Fault
171	Exception causes Fault
172	Operand Evaluation causes Fault
168	Action by other task on variable causes Fault
157	LIGHTS
166	Relation causes Fault
169	LIGHTS
159	Relation causes Fault
158	GREEN

Figure 22: Assignment Statement Template on Line 33

Another answer to the question proposed above is that by automating the process of translating Ada source code into its corresponding template tree structures, the potential for human error is greatly reduced, thereby increasing the likelihood that the final results will be more accurate and thorough. Considering that humans are apt to make mistakes when it comes to laborious and repetitive tasks, the generation of software fault tree structures is best handled by automated means resulting in faster, more consistent, and more accurate results.

IV. CONCLUSION

This thesis presented an automated tool (FIT tool) that in conjunction with existing tools (the ACTT and FTE tools) can be used to construct and manipulate software fault trees. Also given was a proposed method for the the application of the existing tools and the FIT tool in an automated Software Fault Tree Analysis process. To complete the description of the automated SFTA process, this paper presented an example of how the tools delineated in this research were applied to analyze a small Ada software program. This chapter gives a summary of the research results, recommendations for the use of the FIT tool, and some proposals for future research related to the subjects covered in this thesis.

A. RESEARCH SUMMARY

One outcome from this research was a recognition that there is a need for more efficient and effective techniques and tools for analyzing software for faults that could lead to hazardous conditions. The well developed and highly effective methods used in analyzing safety characteristics of hardware components cannot be directly applied to the discipline of software safety analysis. Software presents unique problems for safety analysts that dictate the necessity for the development of techniques, methods, and tools that address the unique nature of software safety analysis. Furthermore, the demand for effective software safety analysis tools will continue to rise as more and more potentially hazardous systems employ software as integral sub-components.

In the course of this research, two existing tools developed to assist software safety analysts were reviewed for the advantages they provide and for their limitations. The ACTT tool provides analysts a convenient method for translating Ada source code programs into statement template tree structures from which software fault trees can be generated. However, the structures generated by the ACTT tool are not true software fault trees, and must be further processed before an analyst can employ SFTA to analyze Ada programs. In addition, the enormous size of the tree structures generated by the ACTT tool when translating even small Ada source code files can limit their usefulness in the absence of additional refinement processes. The FTE tool provides a convenient method for analysts to construct graphical representations of software fault trees. However, the time consuming and labor intensive nature of manually translating source code files to software fault trees, even with the aid of the functionality that the FTE tool provides, precludes the FTE tool being used as a stand alone means for effective software safety analysis.

As a result of the research conducted in this thesis, the FIT tool was developed to resolve some of the limitations of the ACTT and FTE tools in order to support a more effective and efficient SFTA process. By providing functionality for locating and isolating specific nodes in the statement template tree structures generated by the ACTT tool, the FIT tool provides a means to the analyst for constructing software fault trees representing specific faults in the software that he or she chooses to evaluate. In the process of doing so, the FIT tool will automatically remove those parts of the statement template tree structure that are not relevant to the selected fault to be evaluated resulting in a more compact and manageable structure. An advantageous side effect of this process is that the limiting capability of the FTE to display structures of only a certain size can be overcome. In short, the FIT tool that was developed in the course of this research can be viewed as an intermediate process between the ACTT tool and the FTE tool. The FIT tool manipulates the files generated by the ACTT tool in order to create files that may be effectively used by the FTE tool. The result is an automated means that analysts can employ to more effectively apply SFTA in analyzing the safety level of software.

One conclusion reached in the course of this research is that there are limits to what sub-processes can be automated in an overall SFTA process. The ultimate goal of SFTA is to attain a thorough and accurate understanding of the semantic nature of software. This implies that there are determinations and interpretations that must be made by the human analyst. The goal of the automated tools described in this research is to relieve the analysts from performing some of the tedious and time consuming sub-processes of the overall process that distract him or her from concentrating on the ultimate goal. The accuracy of the results from conducting SFTA using automated tools in analyzing software is dependent upon the skill and experience of the analyst. It is the human analyst who must retain control of the safety analysis process, and be responsible for accurately interpreting the final results. Automated tools that handle some of the detail processing can allow the analyst to do so more effectively.

B. RECOMMENDATIONS

The tools described in this thesis were designed to be used by programmers, researchers, and analysts interested in the safety features of software. Specifically, these tools were designed to be used during a SFTA process. It is not expected that anyone who is unfamiliar with SFTA will be able to effectively employ these tools, or accurately interpret the final results that can be attained by using these tools.

This thesis makes no guaranteed claims about the accuracy or effectiveness of the tools described in this paper. More research is necessary before any such claims can be made. An assessment of the accuracy and the effectiveness of these tools can only be attained through extensive testing and use of these tools on practical applications.

This thesis does, however, contend that the automated tools described here have great potential for aiding analysts in applying SFTA in analyzing software for faults that could potentially lead to hazardous conditions. These tools can be used to supplement existing software safety analysis procedures where they exist, or provide a means for performing software safety analysis where no procedures for doing so are currently in place. In addition, the automated tools described in this paper may serve as a basis for future research related to the development of tools needed in software safety analysis. These tools should be viewed as first steps in the quest for development of techniques and tools that lend themselves to aiding software safety analysts in their all important goal of being able to guarantee the level of safety of software that modern systems demand.

C. FUTURE RESEARCH

Some areas of future research related to the techniques and tools mentioned in this paper have already been alluded to. Research is needed to assess the accuracy, effectiveness, and usefulness of these tools. This can perhaps be best accomplished by using these tools to analyze typical "real world" applications. The goal of such research would be, in part, to answer the following questions:

- Do the tools produce accurate results?
- Are the tools useful, and if not, what modifications would make them useful?
- Are the tools effective in analyzing software for potentially hazard causing faults?
- Are there better or easier methods to analyze software than an automated SFTA process?
- Are there other sub-processes in an overall SFTA process that can or should be automated?

In addition, research of this nature may reveal other pertinent questions regarding automated SFTA that have not been considered in this paper.

Modifications to some of the tools covered in this paper would also be useful. The current limitation of the FTE tool to display tree structures only up to a certain size needs to be overcome if this tool is to be truly effective in analyzing large software projects that are typical of modern systems. This limitation may be overcome through additional scrolling functionality that will allow users to scroll up or down the entire depth of a given tree structure. In addition, a scaling function would be useful for display and printout of tree structures that would enhance the analyst's ability to examine and evaluate software fault trees. Another tool modification worth consideration is the addition of a Graphical User Interface (GUI) to the FIT tool in order to enhance its usefulness to programmers and analysts. This modification may reveal additional desired functionality to be incorporated in to the existing FIT tool that will increase its usefulness.

Another area of possible future research would be developing tools that work by manipulating the semantic structure of the software being analyzed. By contrast, the ACTT and FIT tools presented in this thesis work basically by manipulation of the syntactical structure of software. Development of a tool that would generate a software fault tree(s) based upon a description of a semantic fault to be evaluated would further advance the discipline of software safety analysis. As a simplistic example, using the traffic control example application from the previous chapter, a tool that would generate all software fault trees related to the “changing of the variable LIGHTS causes Fault” description has the potential to be of even greater help to software safety analysts. The manner of describing semantic faults to be used as input, what processes would be used to generate all relevant software fault trees, and the characteristics of the output results are all questions that would need to be addressed by future research.

LIST OF REFERENCES

[Bai 82] Bailey, R., *Human Performance Engineering: A Guide for Systems Designers*, Englewood Cliffs, NJ: Prentice-Hall, 1982.

[Bas 89] Bass, L. and Martin, D. L., "Cost-Effective Software Safety Analysis", *Proceedings of the Annual Reliability and Maintainability Symposium*, Atlanta, GA., 1989.

[Bon 85] Bonnett, N., "System Safety Handbook, AFISC SSH 1-1: Software System Safety", Headquarters Air Force Inspection and Safety Center, September 1985.

[Bro 88] Brown, M. L., "Software Systems Safety and Human Errors", *Proceedings of IEEE Compass '88*, 1988.

[Cha 91] Cha, Stephen S., *A Safety-Critical Software Design and Verification Technique*, Ph. D. Dissertation, Department of Information and Computer Science, University of California, Irvine, Irvine, CA 1991.

[Cha 87] Cha, S. S., Leveson, N. G., and Shimeall, T. J., "Fault Tree Analysis Applied to Ada", *Proceedings of Tenth International Conference on Software Engineering*, Singapore, 1988.

[Gri 81] Griggs, J. G., "A Method of Software Safety Analysis", *Proceedings of the Safety Conference*, vol. 1, part 1 System Safety Soc., Newport Beach, Calif., 1981.

[Ham 72] Hammer, W., *Handbook of System and Product Safety*, Prentice-Hall, Inc., Englewood Cliffs, N. J., 1972.

[Lev 86] Leveson, N. G., "Software Safety: Why, What, and How", *ACM Computing Surveys,* June 1986.

[McD 89] McDonald, G. W., "Why There Is a Need for a Software-Safety Program", *Proceedings of the Annual Reliability and Maintainability Symposium*, Atlanta, GA., 1989.

[MIL 84] -------, "MIL-STD-882B Military Standard: System Safety Program Requirements", Headquarters Air Force Systems Command, March 1984.

[Neu 89] Neumann, P. G., "The Computer-Related Risk of the Year: Misplaced Trust in Computer Systems", *Proceedings of IEEE Compass '89*, 1989.

[Ord 93] Ordonio, R. R., *An Automated Tool to Facilitate Code Translation for Software Fault Tree Analysis*, M.S. Thesis, Naval Postgraduate School, Monterey, CA, September 1993.

[Rei 94] Reid, W. S., *Software Fault Tree Analysis of Concurrent Ada Processes*, M.S. Thesis, Naval Postgraduate School, Monterey, CA, September 1994.

[Rou 81] Rouse, W. B., "Human-Computer Interaction in the Control of Dynamic Systems", *ACM Computing Survey*, 1981.

[San 93] Sanders, M. S., and McCormick, E. J., *Human Factors in Engineering and Design*, McGraw-Hill, Inc., 1993.

INITIAL DISTRIBUTION LIST

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

Dudley Knox Library 2
Code 052
Naval Postgraduate School
Monterey, CA 93943-5101

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

Professor Timothy Shimeall, Code CS/sm 6
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Dr. Michael Friedman 1
18950 Mt. Castle Circle
Fountain Valley, CA 92708

Sung Deok Cha 1
Computer Science Deptartment, KAIST
373-1 Kusong-Dong, Yusong-Gu, Taejon 305-701, Korea

Mr. Robert F. Westbrook (Code 31) 1
Naval Air Warfare Center
Weapons Division
China Lake, CA 93555

Ms. Eileen T. Takach 1
Naval Air Warfare Center
Aircraft Division, Indianapolis
DP304N MS-31
6000 E. 21st Street
Indianapolis, IN 46219-2189

LCDR John A. Daley, USN Code CS/da Computer Science Department Naval Postgraduate School Monterey, CA 93943	1
LCDR Russell W. Mason c/o Commanding Officer Strike Fighter Squadron 106 NAS Cecil Field Jacksonville, FL 32215-0173	2