

2

NAVAL POSTGRADUATE SCHOOL Monterey, California

AD-A231 859



DTIC
ELECTE
FEB 25 1991
S B D

THESIS

SAFETY ANALYSIS OF
HETEROGENEOUS-MULTIPROCESSOR
CONTROL SYSTEM SOFTWARE

by
Janet A. Gill

December, 1990

Thesis Advisor:

Timothy J. Shimeall

Approved for public release; distribution is unlimited

91 2 21 040

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified		1b. RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE			
4. PERFORMING ORGANIZATION REPORT NUMBER(S)		5. MONITORING ORGANIZATION REPORT NUMBER(S)	
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School	6b. OFFICE SYMBOL (If Applicable) 37	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School	
6c. ADDRESS (city, state, and ZIP code) Monterey, CA 93943-5000		7b. ADDRESS (city, state, and ZIP code) Monterey, CA 93943-5000	
8a. NAME OF FUNDING/SPONSORING ORGANIZATION	6b. OFFICE SYMBOL (If Applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS (city, state, and ZIP code)		10. SOURCE OF FUNDING NUMBERS	
		PROGRAM ELEMENT NO.	PROJECT NO.
		TASK NO.	WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) SAFETY ANALYSIS OF HETEROGENEOUS-MULTIPROCESSOR CONTROL SYSTEM SOFTWARE			
12. PERSONAL AUTHOR(S) Janet A Gill			
13a. TYPE OF REPORT Master's Thesis	13b. TIME COVERED FROM TO	14. DATE OF REPORT (year, month, day) December 1990	15. PAGE COUNT 63
16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
17. COSATI CODES		18. SUBJECT TERMS (continue on reverse if necessary and identify by block number)	
FIELD	GROUP	Software Safety, Petri Net, Fault Tree, Software Engineering, Integrated System Analysis	
	SUBGROUP		
19. ABSTRACT (Continue on reverse if necessary and identify by block number) Fault trees and Petri nets are two widely accepted graphical tools used in the safety analysis of software. Because some software is life and property critical, thorough analysis techniques are essential. Independently, Petri nets and fault trees serve limited evaluation purposes. This thesis presents a technique that converts and links Petri nets to fault trees and fault trees to Petri nets. It enjoys the combinational benefits of both analysis tools. Software Fault Tree Analysis and timed Petri nets facilitate software safety analysis in heterogeneous-multiprocessor control systems. Analysts use a Petri net to graphically organize the selected software. A fault tree supports a hazardous condition with subsequent leaf node paths that lead to the hazard. Through the combination of Petri nets and fault trees, an analyst can determine a software fault if he can reach an undesired Petri net state, comparable with the fault tree root fault, from an initial marking. All transitions leading to the undesired state from the initial marking must be enabled and the states must be marked that represent the leaf nodes of the fault tree path. (continued)			
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS		21. ABSTRACT SECURITY CLASSIFICATION Unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL Timothy J. Shimeall		22b. TELEPHONE (Include Area Code) (408) 646-2509	22c. OFFICE SYMBOL Code CS/Sm

19. It is not the intention of this thesis to suggest that an analyst be replaced by an automated tool. There must be analyst interaction focusing the analyst's insight and experience on the hazards of a system. This method is proposed only as a tool for evaluation during the overall safety analysis.

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	



Approved for public release; distribution is unlimited.

**Safety Analysis of Heterogeneous-Multiprocessor
Control System Software**

by

Janet A. Gill

Civilian, Naval Air Test Center, Patuxent River, Maryland
B.S., University of West Florida, 1985

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

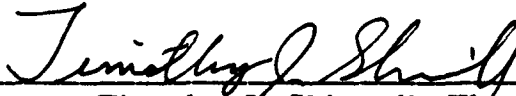
December, 1990

Author:

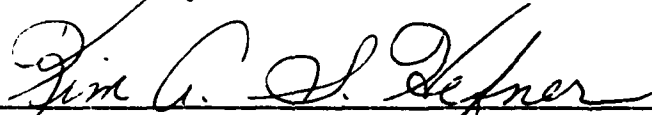


Janet A. Gill

Approved by:



Timothy J. Shimeall, Thesis Advisor



Kim A. S. Hefner, Second Reader



Robert B. McGhee, Chairman, Computer Science
Department

ABSTRACT

Fault trees and Petri nets are two widely accepted graphical tools used in the safety analysis of software. Because some software is life and property critical, thorough analysis techniques are essential. Independently, Petri nets and fault trees serve limited evaluation purposes. This thesis presents a technique that converts and links Petri nets to fault trees and fault trees to Petri nets. It enjoys the combinational benefits of both analysis tools.

Software Fault Tree Analysis and timed Petri nets facilitate software safety analysis in heterogeneous-multiprocessor control systems. Analysts use a Petri net to graphically organize the selected software. A fault tree supports a hazardous condition with subsequent leaf node paths that lead to the hazard. Through the combination of Petri nets and fault trees, an analyst can determine a software fault if he can reach an undesired Petri net state, comparable with the fault tree root fault, from an initial marking. All transitions leading to the undesired state from the initial marking must be enabled and the states must be marked that represent the leaf nodes of the fault tree path.

It is not the intention of this thesis to suggest that an analyst be replaced by an automated tool. There must be analyst interaction focusing the analyst's insight and experience on the hazards of a system. This method is proposed only as a tool for evaluation during the overall safety analysis.

TABLE OF CONTENTS

I.	INTRODUCTION	1
A.	SAFETY-CRITICAL HETEROGENEOUS SYSTEMS	1
B.	RISKS OF ERRONEOUS SOFTWARE.....	2
C.	EVALUATION OF SOFTWARE SAFETY.....	4
D.	SOFTWARE FAULT TREE ANALYSIS	5
E.	TIMED PETRI NETS	5
F.	SFTA AND PETRI NET INTEGRATION/TRANSITION	6
G.	SCOPE OF THESIS.....	8
II.	CURRENT PETRI NET AND FAULT TREE ANALYSIS TECHNIQUES	9
A.	PETRI NET APPLICATIONS AND DEVELOPMENT	9
B.	FAULT TREE APPLICATIONS AND DEVELOPMENT.....	18
C.	INTEGRATING ANALYSIS TECHNIQUES	24
III.	ANALYSIS TECHNIQUE INTEGRATION.....	28
A.	INTRODUCTION.....	28
B.	PETRI NET TO FAULT TREE CONVERSION AND INTEGRATION	30
1.	Petri Net to Fault Tree Conversion Initiation.....	32
2.	Petri Net to Fault Tree Starting Point.....	32

3.	Petri Net to Fault Tree Graphical and Tabular Linkage.....	3 3
4.	Complete Development of the Petri Net Fault Tree Linkage Table	3 5
5.	Remedy If No Path to the Unsafe Event is Exposed.....	3 5
C.	FAULT TREE TO PETRI NET CONVERSION AND INTEGRATION	3 8
1.	Fault Tree to Petri Net Starting Point.....	3 8
2.	Fault tree to Petri Net Graphical and Tabular Linkage...	3 9
3.	Petri Net Completion.....	4 1
IV.	SUMMARY AND CONCLUSIONS.....	4 3
A.	INTEGRATED ANALYSIS TECHNIQUE	4 3
B.	LESSONS LEARNED	4 4
C.	FUTURE WORK.....	4 4
D.	CONCLUSIONS.....	4 6
	LIST OF REFERENCES.....	4 7
	BIBLIOGRAPHY	4 9
	INITIAL DISTRIBUTION LIST	5 2

LIST OF FIGURES

Figure 2.1 Basic Petri Net Structure.....	12
Figure 2.2 Basic Petri Net Structure With Tokens Before Transition Firing.....	12
Figure 2.3 Basic Petri Net Structure With Tokens After Transition Firing	12
Figure 2.4 Ada Code for Traffic Light Controller.....	14
Figure 2.5 Timed Petri Net for Traffic Light Controller.....	15
Figure 2.6 Fault Tree Symbols.....	20
Figure 2.7 Possible Fault Tree for Traffic Controller.....	21
Figure 2.8 Fault Tree Reflecting Ada Code for Traffic Light Controller.....	23
Figure 3.1 Petri Net Representing OFP 240 High-Level Code Segment.....	29
Figure 3.2 General Petri Net And Fault Tree Graphical and Tabular Linkage.....	31
Figure 3.3 OFP Root Fault.....	32
Figure 3.4 OFP Petri Net to Fault Tree Cross Diagram.....	34
Figure 3.5 OFP Basic Petri Net Fault Tree Link Table Without Gates	36
Figure 3.6 OFP Basic Petri Net Fault Tree Link Table With Gates.....	36
Figure 3.7 OFP Fault Tree.....	37
Figure 3.8 Petri Net 'and' and 'or' Gates	39
Figure 3.9 Petri Net Creation from a Fault Tree.....	40

LIST OF TABLES

TABLE 1	TRANSITIONS LEADING TO HAZARDS IN THE TRAFFIC LIGHT EXAMPLE	16
TABLE 2	PETRI NET, FAULT TREE, AND SEMANTIC FORMAL DESCRIPTIONS	26

ACKNOWLEDGEMENTS

I would first and foremost like to thank Timothy J. Shimeall, my thesis advisor, for his unfaltering guidance. He was always accessible and never was at a loss for an appropriate answer, editing comment, or word of encouragement. Choosing software system safety as my research area, with Professor Shimeall as my advisor, has proven to be the best decision of my career at the Naval Postgraduate School (NPS).

Professor Kim Hefner, my second reader, and Commander Rachal Griffin were inspiring as instructors and will remain as my professional role models. They both were always willing to explain any concept without insult. The sign of genius is to make a difficult subject simple, not a simple subject difficult.

Commander Hoskins, the Computer Science Curricular Officer, and Professor McGhee, the Computer Science Department Head, were always willing to assist on any academic or professional issue and readily displayed civilian advocacy. Their respective secretaries, Jean and Shirley, were a tremendous support of my administrative needs, ranging from the trivial to the complex.

My peers, coming to Monterey from all parts of the United States and the world, gave me immeasurable friendship and intellectual support during my tour away from my civilian long-term home base. The comradery was extremely strong as we met the NPS challenges side-by-side.

I. INTRODUCTION

A. SAFETY-CRITICAL HETEROGENEOUS SYSTEMS

Many military systems require specialized multiprocessors. Weapon systems and aircraft control systems are prime examples. They have complex controlled system architectures that must operate under tight timing constraints requiring the use of multiple processors to execute independent control tasks. System developers may include multiple processors in the initial prototype design or add them during subsequent upgrades. The system may consist of many identical processors, but specific control needs require heterogeneous processors.

"Normally only software that exercises direct command and control over the condition or state of the hardware components or can monitor the state of the hardware components are considered critical from a safety viewpoint." [Ref. 2] Software that controls and monitors systems, such as missiles or aircraft, is safety-critical.

The use of software in safety-critical flight systems in multiprocessing environments such as the A6, F18, and the proposed P7 military aircraft leads to a need to analyze the safety of software. These intricate multiple-mission systems are susceptible to four different types of software faults. The first type is an undesired or unexpected event. The second type is an event occurring out of sequence. The third type is a specified event failing to occur. The last type is the magnitude or the direction of an event is wrong.

[Ref. 2] The use of incompletely developed and analyzed software in safety-critical systems may cause the loss of life, property, or environmental harm. Formal analysis methodologies executed at all stages of development, from requirements analysis through maintenance, help to reduce this loss.

For any given system task the flow of execution of the software controlling that task may span several processors [Ref. 1]. Flight systems execute many tasks simultaneously just to keep the craft airborne and proceeding in a pilot- or automatically-controlled direction.

B. RISKS OF ERRONEOUS SOFTWARE

Risk is the probability of an accident occurring of a specified magnitude over a given time period. When the task, like flight control, involves a risk of human life or property, the analyst executes an evaluation of the safety of the software in order to avoid an accident or mishap.

The term "mishap" denotes an unplanned event or series of events that result in death, injury, occupational illness, damage to or loss of equipment or property, or environmental. It includes both accidents and harmful exposures. "A mishap can be thought of as a set of events combining in random fashion or, alternatively, as a dynamic mechanism that begins with the activation of a hazard and flows through the system as a series of sequential and concurrent events in a logical sequence until the system is out of control and a loss is produced (the 'domino theory')." [Ref. 4]

Safety is a concern when systems are controlling or releasing energy, such as mechanical, electrical, or chemical. When software is used in such systems, safety must be insured so the risk to human life is minimum.

To ensure the safety of software is to prevent mishaps. Software faults may lead to hazards, and hazards may lead to mishaps; therefore, evaluation of safety-critical software events is vital.

Software alone is not hazardous, but when the software controls a system, it becomes as potentially hazardous as the total system. A failure, malfunction, or design error in the control of hardware components causes or allows a hazard to occur.

A fault is a software bug. This may lead to an error, a discrepancy between a computed, observed or measured value or condition and the true specified, or theoretically correct value or condition. An error, in turn, may lead to a failure: the termination of the ability of a functional unit to perform its required function.

[Ref. 5]

Software faults may result from incorrect or incomplete specifications and requirements, leading to incorrect or incomplete designs, incorrect programming or coding, or hardware-induced corruption. Hopefully, a thorough testing program would locate all faults. In reality, however, the many possible combinations of sequences makes total fault detection extremely difficult. Also, analysts only write tests against requirements, so they may overlook incorrect requirements.

In the past, analysts did not find many latent software faults until the prototype was out in the field. Safety-critical systems cannot afford this delay in fault discovery because once the system is out in the field, people, property, or the environment may be at risk. The analyst must execute thorough analysis in the stages before system delivery to help prevent risk.

A thorough safety analysis is possible because it does not need to consider all faults, just safety-critical faults. Safety analysis only evaluates the system for possible faults derived from the Preliminary Hazard Analysis (PHA).

C. EVALUATION OF SOFTWARE SAFETY

The evaluation of the software safety must trace the flow of software execution, analyzing the sequential and concurrent operations performed and determining if the system acts to prevent or reduce risks. Currently, analysts execute manual analysis using limited evaluation tools, with substantial cost and opportunity for analysis errors. Inaccurate results occur readily in systems where analysts base analysis and design on informal discussions between a software expert group and a system applications expert group. Many analysts depend too much on "corporate knowledge" and not enough on the use of proven methods of design, analysis, and testing. Life-, property-, and environment-critical systems urgently require thorough safety analysis in the software life cycle to avoid risk to life and property.

Leveson [Ref. 7] surveys software safety in terms of why, what, and how. "A fair conclusion might be that 'why' is well understood, 'what' is still subject to debate, and 'how' is completely up in the air." [Ref. 7]

Analysts may combine multiple analysis techniques to evaluate safety. This thesis presents one integration method of "how"--the combination of SFTA and Petri net analysis techniques--in this thesis.

D. SOFTWARE FAULT TREE ANALYSIS

Leveson and Harvey [Ref. 6] developed Software Fault Tree Analysis (SFTA). Hardware system analysts use Fault Tree Analysis (FTA) to analyze a system in the context of its environment and operation. They find credible sequences of events that can lead to a specified hazard. Leveson and Harvey derived SFTA from FTA to analyze systems containing software components. The fault tree is a graphic representation of parallel and sequential combinations of events and system states that result in the occurrence of the predefined hazard. The events and states can be associated with component failures, human errors, or any other pertinent events and states that can lead to the hazard. A fault tree represents the logical interrelationships of events and states that lead to the hazard.

E. TIMED PETRI NETS

Murata [Ref. 8] and Leveson/Stolzy [Ref. 9] state that timed Petri nets describe time-critical events in multiprocessor control

applications and determine if safety-critical states are reachable during normal execution.

The analyst models a system in terms of conditions and events with Petri nets. "If certain conditions hold, then an event or 'state transition' will take place resulting in other (or the same) conditions taking place." [Ref. 2]

In the past, analysts mainly used Petri nets to evaluate performance and correctness. Researchers are currently proposing that analysts can achieve timing more readily with Petri nets than with fault trees.

F. SFTA AND PETRI NET INTEGRATION/TRANSITION

SFTA and timed Petri nets are integrated in this thesis to facilitate software safety analysis in heterogeneous-multiprocessor control systems. These techniques are equivalent in expressive power; analysts can facilitate both techniques by allowing one technique to use the information about the system expressed by the other technique. [Ref. 1]

Analysts use a Petri net to graphically organize the selected software code. A fault tree explicitly supports a hazardous condition root node with subsequent leaf node paths that lead to the hazard. Through the combination of Petri nets and fault trees, an analyst can determine a software fault if each preconditional node in an entire fault tree path links to one or more Petri net transitions and states.

Drawing on the specific A-6E example developed in McGraw's thesis [Ref. 10], Shimeall, McGraw, and Gill [Ref. 1], describe a general

technique for integrating these two analysis techniques. It uses a semantic model for information sharing between the techniques during the analysis. This model consists of three classes of objects: states, transitions, and linkages. The states contain information on conditions existing during program execution. Transitions contain information on actions performed during program execution, and reference the states that lead to and result from the transitions. Transitions also include timing information, indicating the enabling and firing times of the actions, along with deadlines after which the action stops. The state references in the transitions allow for any combination of states. The linkages contain information on undesired events. In general, the states and transitions contain information for the generation of Petri nets, and the linkages contain additional information for the generation of fault trees.

The semantic model forms the basis for automated support of the safety analysis process by allowing the analyst to rapidly and easily shift between techniques. The analyst may use Petri nets to describe the system architecture, shift to fault trees to describe the hazards associated with the system and the events that may lead to the hazards, then shift back and forth between Petri nets and fault trees to analyze those events. Each analysis technique may easily use the results obtained by the other technique. [Ref. 1]

This thesis expands on the Shimeall, McGraw, and Gill work by delineating a stepwise methodology for converting Petri nets to fault trees and fault trees to Petri nets. Graphical, as well as tabular,

linkages describe the Petri net/fault tree relationship and multi-step conversions.

G. SCOPE OF THESIS

The main research and development for this thesis suggests a methodology for integrating timed Petri net analysis and SFTA to analyze software system safety. Chapter II provides an overview of the background information researched and synthesized with original thought to create the proposed integrated safety analysis technique. Chapter III delineates a step-by-step Petri net to fault tree and fault tree to Petri net conversion and linkage process. Chapter IV discusses a summary of the research executed to develop the proposed integrated technique, the technique itself, and an analysis of its effectiveness as a safety-analysis technique. Chapter IV also presents recommendations for further study in the field of safety-analysis-technique integration.

II. CURRENT PETRI NET AND FAULT TREE ANALYSIS TECHNIQUES

This chapter surveys Petri net and fault tree research. Analysts may use any of several analysis techniques for a safety evaluation of a software system. However, the author selected Petri nets and fault trees for the proposed integrated method of safety analysis because first, they are the most mature, and second, analysts have used them in analysis for a relatively long time. Researchers have focused a great deal on these two graphical representations. Also, the individual qualities of Petri nets and fault trees interleave well into a single, effective analysis technique.

This survey describes some possible application areas of Petri nets and fault trees, giving both the strong and weak points of each analysis technique and describing a detailed graphical and textual representation of general Petri nets and fault trees.

A. PETRI NET APPLICATIONS AND DEVELOPMENT

Carl Petri [Ref. 11] created Petri nets in the 1960s. Researchers have made many enhancements since then. Murata [Ref. 8] surveys current Petri net properties and techniques thoroughly. "Petri nets are a promising analysis tool for describing and studying information processing systems characterized as being concurrent, asynchronous, distributed, parallel, non-deterministic, and/or stochastic." [Ref 8]

Petri nets are a graphical and mathematical tool that apply to many software systems. Some possible areas of applications are modeling and analysis of distributed-software systems, concurrent and parallel programs, multiprocessor memory systems, asynchronous circuits and structures, compiler and operating systems, and other discrete-event systems. Additional interesting applications are local-area networks, neural networks, and decision models. [Ref. 8]

Analysts use Petri nets to graphically represent a system, explicitly reflecting the concurrent or parallel activities of the system.

A Petri net is graphically represented as a directed graph with two kinds of nodes: places and transitions. It is textually represented as a five-tuple. The set of places, textually represented as P [Ref. 8] and drawn as circles, indicates the conditions or values present during program execution. The set of transitions, textually represented as T [Ref. 8] and drawn as bars or boxes, indicates the events that occur during program execution. Arcs join the places and the transitions as shown in Figure 2-1 [Ref. 10]. The arcs leading to a transition represent a precondition or an input and the arcs leading from a transition represent a postcondition or an output. These arcs are textually represented as a flow relation, F [Ref. 8], with weight, W , indicating what flows must be present for a transition to occur.

A marking, the presence of tokens in a subset of the places in the net, indicates the current state of the system. The M_0 set in the five-tuple [Ref. 8] represents the system initial state.

When a transition fires, indicating a change in the system state, it consumes a token from each of the input arcs, and generates a token on each of the output places. A transition leading from one set of places to another is enabled to fire when all of the places leading into the transition contain tokens [Ref. 10] (See Figure 2-2 [Ref. 8]). Each transition potentially enables further transitions as shown in Figure 2-3. If more than one transition is enabled at once, a non-deterministic choice is made as to which transition fires [Ref. 1].

The importance of this non-deterministic firing representation is that in real life, events do not purely happen sequentially. A discrete event can occur at any time that all of its preconditions have been met. One example is driving a car from point A to point B. There are many mechanical and maneuvering events that can take place between point A and point B. Some events have a particular order. The driver must turn the ignition key before the engine starts. Many events can happen concurrently. The driver may apply pressure to the accelerator and turn the wheel at the same time. He may execute many non-deterministic decisions while getting from point A to point B and Petri nets can represent these decisions well. Using timed Petri nets allows the incorporation of timing information into the analysis. Real-time embedded-system analysis requires

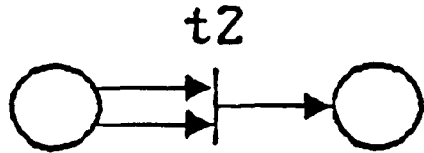
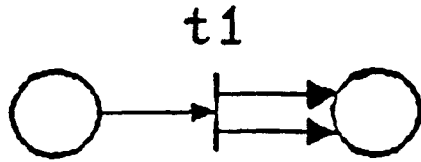


Figure 2-1 Basic Petri Net Structure

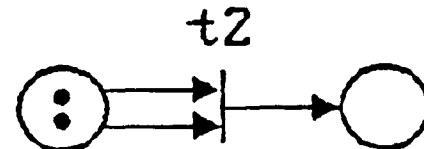
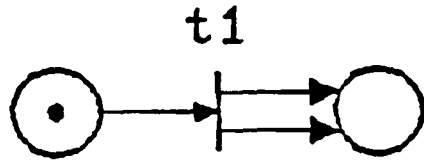


Figure 2-2 Basic Petri Net Structure With Tokens Before Transition

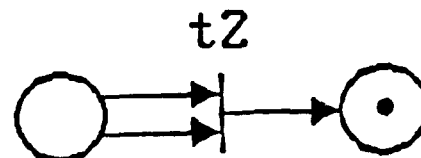
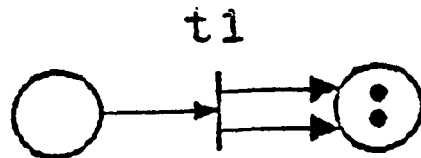


Figure 2-3 Basic Petri Net Structure With Tokens After Transition

timing analysis. Even basically correct software actions that occur too early or too late can lead to unsafe conditions. [Ref. 9]

Analysts add a minimum time function and maximum time function to the above five-tuple Petri net description to define the time frame boundaries within which a firing can occur.

Safety analysis determines if the net can reach an unsafe state. A human can make an unsafe decision outside of the system, such as pulling out into ongoing traffic, but the hardware and software system should not allow a hazard to be reachable.

In the safety analysis [Ref. 9], if an unsafe event (represented by a transition) can be reached from the initial marking M_0 , corrections need to be made. If it is possible that a marking will eventually cause a transition to be enabled, the transition is reachable from the marking. Analyzing a system can be complex, but the Petri net aids in the visual understanding of event ordering and results.

Figure 2-4 [Ref. 12] contains Ada code for the control of a traffic light. A Petri net can represent all system states and transitions relating to the software algorithms. Figure 2-5 depicts the timed Petri net associated with the Ada code in Figure 2-4.

Table 1 describes the transitions representing events that could cause the hazard of both the East/West and the North/South cars entering the intersection at the same time. These descriptions reflect the erroneous activity of the East/West light while the North/South car may enter the intersection.

```

1 procedure traffic is
2   type direction is (east, west, south, north);
3   type color is (red, yellow, green);
4   type light_type is array (direction) of color;
5   lights : light_type := (green, green, red, red);
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;
27  task body controller is
28  begin
29    loop
30      accept notify (dir : in direction) do
31        case dir is
32          when east | west =>
33            lights := (green, green, red, red); delay 5.0;
34            lights := (yellow, yellow, red, red); delay 1.0;
35            lights := (red, red, green, green);
36          when south | north =>
37            lights := (red, red, green, green); delay 5.0;
38            lights := (red, red, yellow, yellow); delay 1.0;
39            lights := (green, green, red, red);
40        end case;
41      end notify;
42    end loop;
43  end controller;
44  begin
45    for dir in east..north loop
46      sensor(dir).initialize (dir);
47    end loop;
48  end traffic;

```

Figure 2-4 Ada Code for Traffic Light Controller

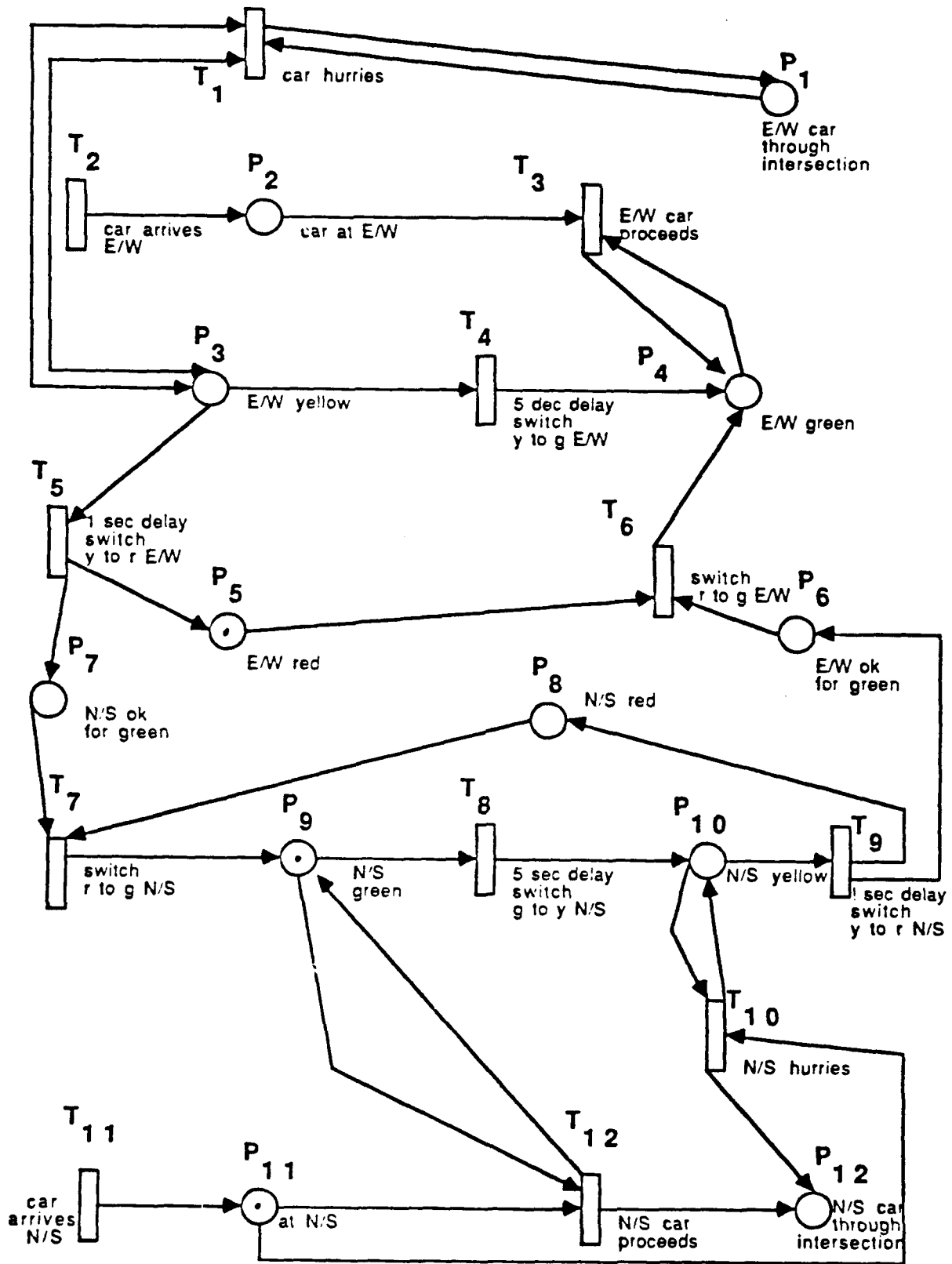


Figure 2-5 Timed Petri Net For Traffic Light Controller

Table 1 TRANSITIONS LEADING TO HAZARDS IN THE TRAFFIC LIGHT CONTROLLER

ORIGINATION	TRANSITION	DESTINATION
P ₂ Car at E/W	E/W car runs light	P ₁ E/W car through intersection
P ₃ E/W light yellow	Light broken- Stuck on yellow	P ₁ E/W car through intersection
P ₄ E/W light green	Light broken- Stuck on green	P ₁ E/W car through intersection
P ₅ E/W light red	Light broken- Stuck on red	E/W car runs light
P ₄ E/W light green	> 5 second delay from green to yellow	P ₃ E/W light yellow
P ₃ E/W light yellow	>1 second delay from yellow to red	P ₅ E/W light red
P ₄ E/W light green	>6 second delay from green to red	P ₅ E/W light red
P ₅ E/W light red	Light prematurely green	P ₄ E/W light green

If an analyst can follow a hazardous-condition path within the software code during Petri net analysis, the system will have to be modified. Software safety analysts only seek software controlled errors.

Time is critical in this specific example. Software developers determined the delay times for light changes by hard-coding these changes in lines 32 through 39 of the selected Ada code. If a light is functioning, but is delayed on green or yellow too long, or turns green prematurely, a hazard exists and the software code needs to be corrected.

Consider the last item in Table 1, the East/West light turning prematurely green and allowing the two perpendicular cars in the intersection at the same time. The initial markings for Figure 2-5 could be in first in p_5 , indicating the E/W light is red, second in p_9 , indicating the North/South light is green, and third in p_{11} , indicating a car is at the North/South intersection.

Because p_4 can be reached prematurely while the North/South light is going through its timing sequence, modifications must be made to the software system. One possible scenario that exposes a hazard is: 1) A North/South car approaches the intersection and its signaling turns the North/South light green. 2) An East/West car approaches the intersection, finds the East/West light red, signals for green, and waits. 3) A second North/South car approaches the intersection, checks the light, and sees green. This North/South car enters the intersection, but as it is entering, the East/West light turns

green. The undesired result is that one North/South car and one East/West car are in the intersection at the same time.

The drawback in using Petri nets for analysis is that they are difficult and time consuming to analyze. To generate the entire reachability graph from Petri nets consumes exponential space and time. [Ref. 10]. Integrating this analysis method with the more specific fault tree approach to analysis may make evaluation simpler and potentially save time and money while maintaining order, concurrency, and timing.

B. FAULT TREE APPLICATIONS AND DEVELOPMENT

Researchers created Fault Tree Analysis (FTA) in the 1960s for analyzing hardware. Electrical and/or mechanical systems needed a way to analyze safety. Researchers then created Software Fault Tree Analysis (SFTA) in the early 1980s to evaluate applications and systems software. [Ref. 6]

The safety analyst conducts a Preliminary Hazard Analysis (PHA) of the software first. PHA seeks to find potential hazards involved in the execution of a system. The analyst then places these hazards into severity categories He then employs SFTA to point out single-point failure modes and guide further design in the most fruitful direction for hazard elimination and reduction [Ref. 12]. Certification personnel may use SFTA to examine already-developed software.

An undesired event, or hazard, could occur due to environmental conditions, human error, or component failure. The fault tree depicts

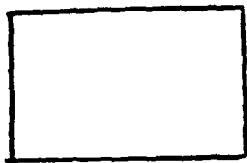
the logical interrelationship of these basic events that lead to the hazard. SFTA works backward and tries to prove that the hazard cannot be reached. [Ref. 6]

A fault tree developer takes the root node and abstractly establishes sets of possible conditions, or leaf nodes, that lead to the root node.

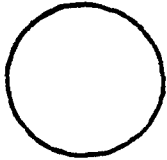
"Proof by contradiction is conveniently used in SFTA since the goal of the analysis is to prove that the software will not permit some event." [Ref. 12] If the analysis can prove a contradiction to the loss (root) event then the event cannot happen as a single-point failure within the software. If the initial software or system state starts a possible series of events that lead to the root event, then the system developers need to alter the software or system design to prevent the root event [Ref. 1].

Leveson and Harvey [Ref. 6] list and describe the relevant fault tree symbols used in SFTA. These graphical representations are shown in Figure 2-6. Fault trees are textually represented by N, the set of nodes, and G, the set of 'and' and 'or' gates.

One possible hazard description resulting from a PHA done on the high-level Ada code in Figure 2-4 [Ref. 12] is that both a car from the North/South direction and a car from the east/west direction can enter the intersection at the same time. Figure 2-7 portrays probable scenario nodes that could lead to this root node.



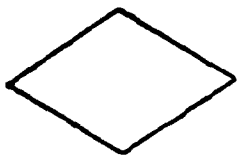
The *rectangle* indicates an event to be analyzed further.



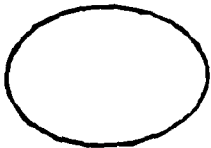
The *circle* indicates a basic fault event or primary failure of a component. It requires no further development, and its probability of occurrence is derived from the generic rate of the part.



The *house* is used for events which normally occur in the system. It represents the continued operation of the component, and its probability is the reliability of the part.



The *diamond* is used for non-primal events which are not developed further for lack of information or insufficient consequence.



The *oval* is used to indicate a condition. It defines the state of the system that permits a fault sequence to occur. It may be normal or result from failures.



The *AND* gate serves to indicate that all input events are required in order to cause the output event.



The *OR* gate indicates that one or more of the input events are required to produce the gated events.

Figure 2-6 Fault Tree Symbols

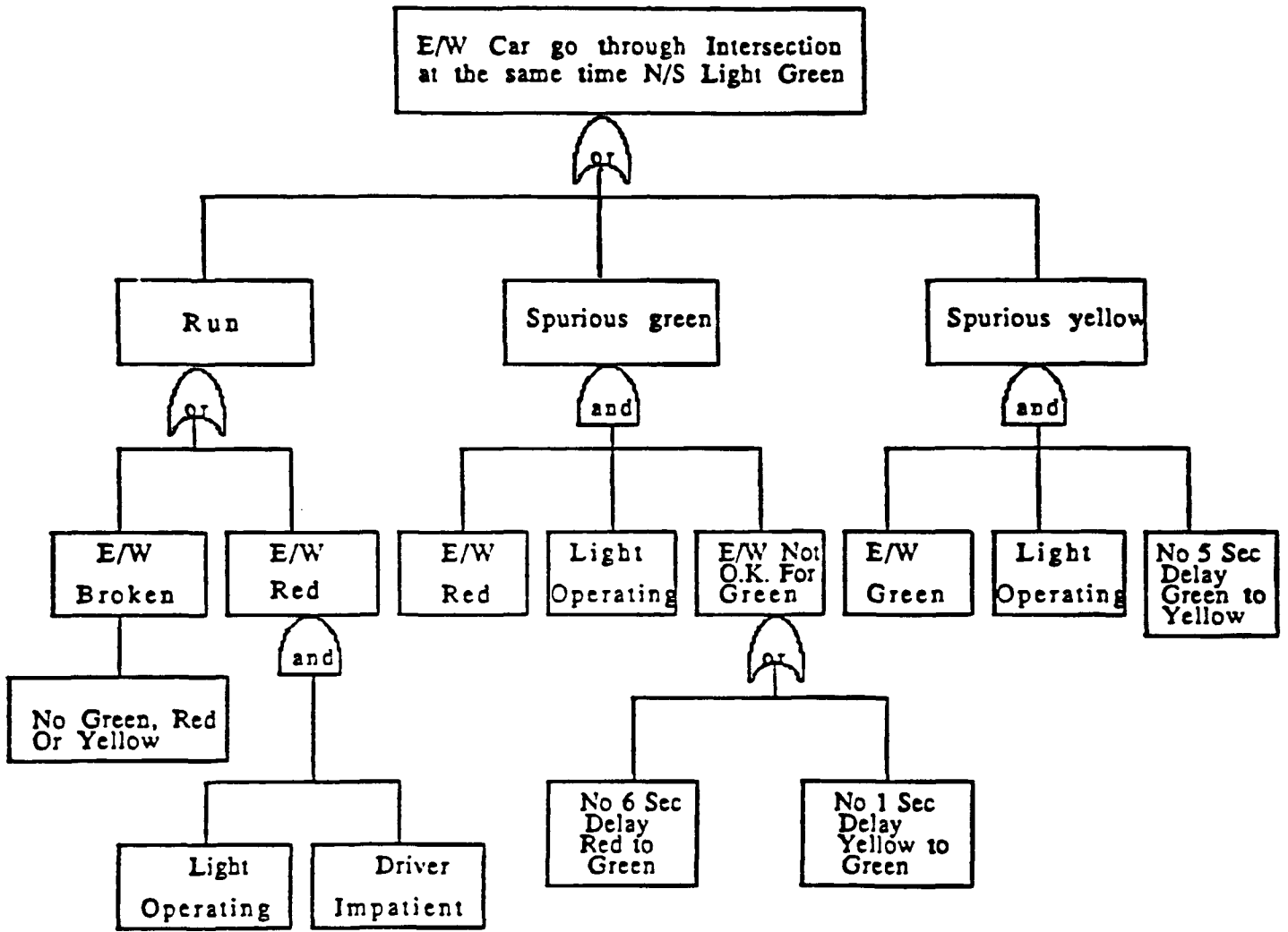


Figure 2-7 Possible Fault Tree for Traffic Light Controller

For fault tree development and evaluation for other hazards, the analyst should expand on the remainder of the root nodes derived from the PHA in order of priority.

A spurious yellow or green light being displayed in the North/South direction while a green light is displayed in the East/West direction are two conditions that would support the possibility of two cars coming from perpendicular directions being in the intersection at the same time.

Figure 2-8 [Ref. 12] expands on Figure 2-7 by detailing and tailoring the fault tree to examine the traffic-light program for the presence of spurious light conditions that enable two perpendicular cars to be in an intersection at the same time. The selected Ada code determines the timing of the light changes in lines 32 through 39. System developers hard-coded delay times for light changing.

Because the high-level code allows at least one path from the initial system state to the hazard, the Ada code supports the fault tree and is hazardous.

"The above fault tree analysis demonstrates that the above Ada code could contribute to the hazard (i.e., two cars, traveling at right angles to one another, are present in the intersection at the same time) if two successive rendezvous occur with the east or west sensor tasks and the north sensor task checks the state of the lights immediately prior to the second rendezvous". [Ref. 11] Eliminating this hazard requires changes to the code or controller.

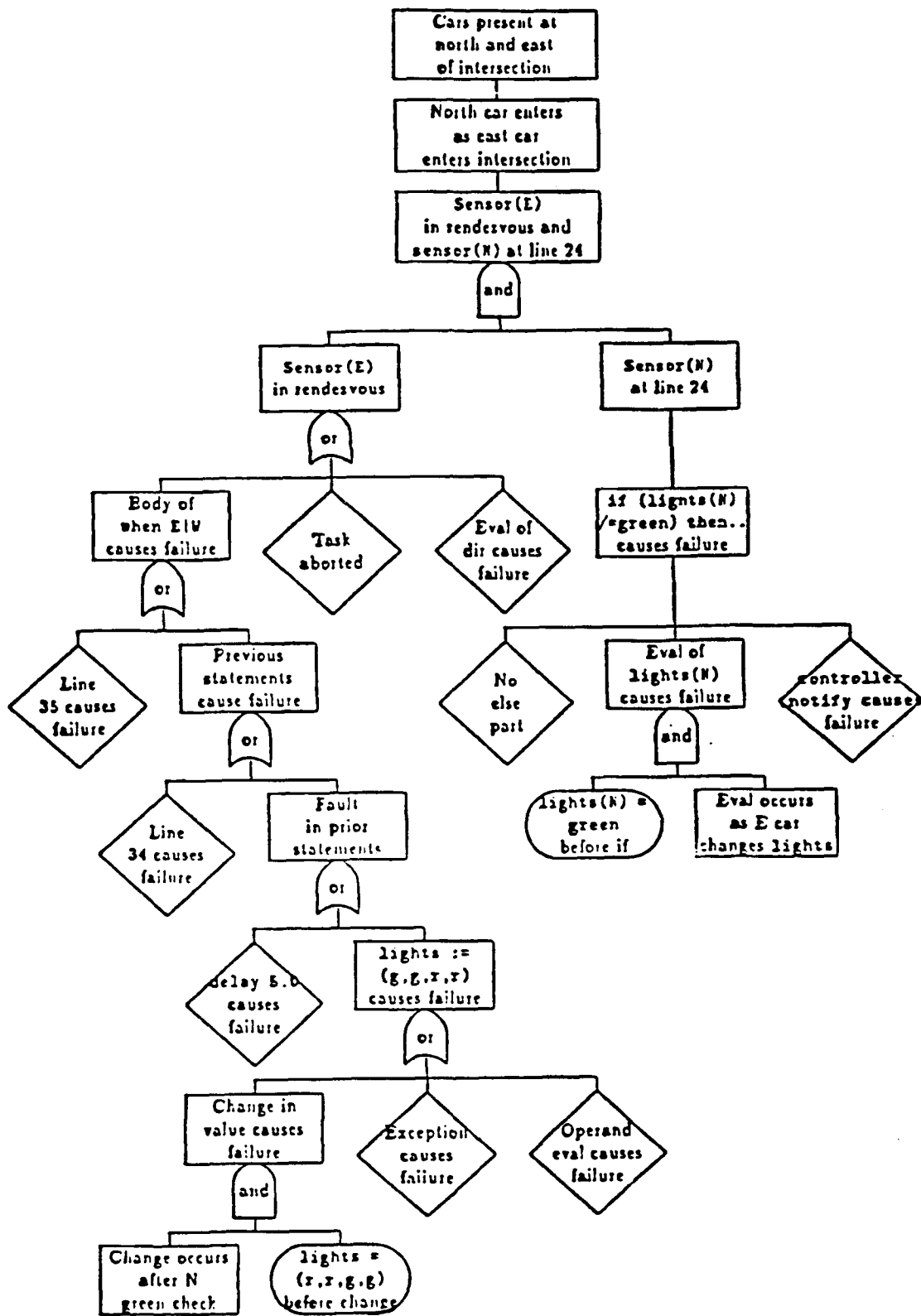


Figure 2-8 Fault Tree Reflecting Ada Code for Traffic Light Controller

SFTAs are limited as the only tool used for analyzing the software safety of a system. Fault trees are a static analysis technique. Timing analysis requires dynamic analysis. [Ref. 7] "They can, however, detect software logic errors and multiple failure sequences that may have essential information that can be shared with Petri net analysis". [Ref. 10]

C. INTEGRATING ANALYSIS TECHNIQUES

The software system safety research community has established integrating multiple system reliability analysis techniques over the past few years. Researchers developed one technique, the Hybrid Automated Reliability Predictor (HARP) [Ref. 13] in 1986. It integrates fault tree notation with the Markov Chain. "HARP converts the dynamic fault tree model notation into a Markov Chain and solves the Markov Chain using a standard well known numerical integration algorithm." [Ref. 14]

For many applications, analysts may use either software fault trees or timed Petri nets to evaluate safety critical behaviors of the control software [Ref 1]. Petri nets explicitly model the structure of a control system and the events during the execution of the control system. The semantics of those events and the resulting conditions are only represented abstractly. In fault trees, the semantics of the conditions and events are explicitly described, but the structure that gives rise to those events is dealt with abstractly [Ref. 1].

Shimeall, McGraw, and Gill [Ref. 1] argue for an integration of these two analysis techniques that gives the analyst different views

of the system. Analysts should not consider Petri nets and fault trees to be alternate techniques, but complimentary to one another.

Leveson and Stolzy [Ref. 9] use Petri nets and FTA in conjunction with one another. Developing a complete reachability graph from the Petri nets representing a system is time consuming and difficult; however, the evaluation is simpler when Petri nets only describe the key portions of a system. [Ref. 9] The safety analyst only needs to consider life-, property-, or environment-critical portions of the high-level code in the analysis.

The SFTA system state information, represented as the preconditional leaf nodes to a specific root fault discovered in the PHA reduces the massive nature of Petri net analysis [Ref. 10]. Fault trees address one specific undesired event at a time, breaking the system analysis down into multiple discrete safety issues.

McGraw [Ref. 10] analyzes a real-time software example that is the upgrade of the A-6E operational flight program (OFP 240). China Lake controls and directs the new program, OFP 250. McGraw represents a key portion of this system with a Petri net and a fault tree.

Shimeall, McGraw, and Gill [Ref. 1] define the semantics of a linkage relation for a Petri net and a fault tree. They construct the semantic model shown in Table 2 by joining the separate formal descriptions presented previously and adding the linkage relation. This linkage represents the logical relationship between fault tree nodes with Petri net places and transitions. Textually there are three

Table 2 PETRI NET, FAULT TREE, AND SEMANTIC FORMAL DESCRIPTIONS

Timed Petri Nets:

$tpn = \langle P, T, F, W, E, D, M_0 \rangle$	
$P = \{p_1, p_2, \dots, p_m\}$	places
$T = \{t_1, t_2, \dots, t_k\}$	transitions
$F \subseteq (P \times T) \cup (T \times P)$	flow relation
$W : F \rightarrow \{1, 2, 3, \dots\}$	weight (tokens on each flow)
$E = \{e_1, e_2, \dots, e_k\}$	enabling times
$D = \{d_1, d_2, \dots, d_k\}$	deadline times
$M_0 : P \rightarrow \{1, 2, 3, \dots\}$	initial marking

Fault Trees:

$ft = \langle N, G, S, C, R \rangle$	
$N = \{n_1, n_2, \dots, n_j\}$	nodes (fault/failure statements)
$G = \{g_1, g_2, \dots, g_j\}$	gates (logical connections)
$S = \{s_1, s_2, \dots, s_j\}$	shapes (analysis role)
$C \subseteq (N \times N)$	child relation
$R \in N$	root node

Semantic Model

$sm = \langle L, tpm, ft \rangle$	
$L \subseteq (P \cup T) \times C \times N$	linkage relation

Constraints

$P \cap T = \emptyset, P \cup T \neq \emptyset$
$\forall i, 1 \leq i \leq k, e_i \geq 0 \wedge d_i \geq 0 \wedge d_i \geq e_i$
$\forall i, 1 \leq i \leq j, g_i \in \{\text{and, or, null}\}$
$\forall i, 1 \leq i \leq j, s_i \in \{\text{box, house, diamond, circle, oval}\}$
$ C = j - 1$
$\forall i, 1 \leq i \leq j,$
$(n_i \neq R \Rightarrow \{(n_q, n_i) \in C \text{ s.t. } 1 \leq q \leq j \wedge q \neq i\} = 1) \wedge$
$(n_i = R \Rightarrow \{(n_q, n_i) \in C \text{ s.t. } 1 \leq q \leq j \wedge q \neq i\} = 0)$
$\forall y, y \in (P \cup T), \forall n, n \in N,$
$(\exists g, g \in \{\text{and, or, null}\}, (y, g, n) \in L) \Rightarrow$
$ \{g \in \{\text{and, or, null}\} \text{ s.t. } (y, g, n) \in L\} = 1$

elements: 1) the union of Petri net P , places, and T , transitions, 2) G , the linkage containing 'and' , 'or', or 'null' gates, and 3) the fault tree N , nodes that define a Petri net fault tree linkage.

Chapter III expands on this work by describing the conversion from a Petri net to a fault tree, the conversion from a fault tree to a Petri net, and the formal linkage relationship between the two. Once the analyst identifies the places and transitions that may lead to the root event, he incorporates them into the linkage relation [Ref. 1].

Integrating, converting, and linking Petri net and fault tree analysis techniques in order to evaluate selected software code may reduce the development and maintenance efforts by reducing redundant analysis.

III. ANALYSIS TECHNIQUE INTEGRATION

A. INTRODUCTION

This thesis presents an analysis technique that integrates software fault tree analysis (SFTA) and timed Petri nets to facilitate software safety analysis in heterogeneous-multiprocessor control systems. The combination of Petri nets and fault trees in software safety analysis provides a greater convenience than using either individually.

The purpose for integrating Petri nets and fault trees is to enhance software safety analysis. The integrated technique melds the use of the Petri net representation of system events and the explicit fault representation and diagnosis in fault trees for a synergistic effect.

The example analyzed here proves that the design of a change in the flight control system of the A-6 fighter/bomber prevents an important hazard, inadvertent missile launch during practice [Ref. 10]. The Petri net in Figure 3-1 (taken from McGraw [Ref. 10]) represents a high level control flow of the proposed upgrade to the Grumman A-6E Operational Flight Program (OFP 240).

The analyst must make a decision whether to begin the safety analysis with Petri net or fault tree development. The analyst creates the Petri net to organize and partially order the events that occur during system execution. This visual representation of events, via transitions and states, simplifies the the analyst's understanding

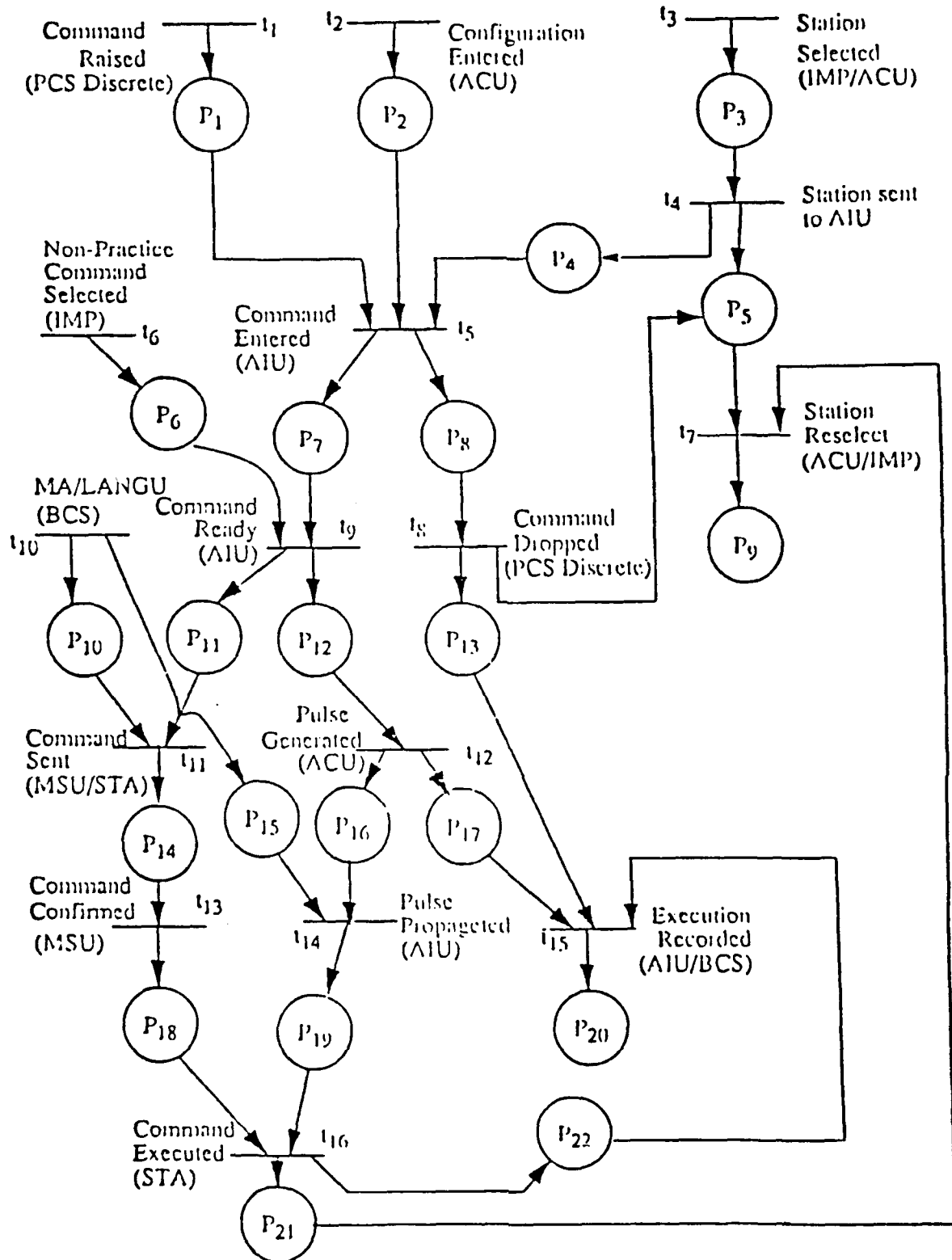


Figure 3-1 Petri Net Representing OFP 240 High-Level Code Segment

of individual software algorithms and their interaction for those system evaluators inexperienced or experienced with computer programming. When analysts pictorially organize activity paths, it is easier to point out system problems that may lead to hazards. This is especially true when there is concurrency or timing of action. Leveson and Stolzy [Ref. 9] explain Petri net development. A conversion from a Petri net to a fault tree and a linkage between the two is described in Section B. The analyst establishes a fault tree to Petri net graphical and tabular linkage, as in Figure 3-2, while he is creating the fault tree from a Petri net.

The analyst executes fault tree development first in the proposed conversion and integration method if the system chosen for analysis is independent, synchronous, localized, serial, deterministic, or non-stochastic. The analyst tailors a fault tree to its top event that corresponds to some particular system hazard, detailing the events and conditions that lead to the hazard. [Ref. 1]

B. PETRI NET TO FAULT TREE CONVERSION AND INTEGRATION

The dynamics of Petri nets aids the analyst in the understanding and evaluation of complex systems. However, Petri nets only abstractly represent the semantics of the events and conditions. Should the analysis require explicit representation of the semantics, the analyst may choose to convert the Petri net to a fault tree and to pursue the analysis utilizing fault tree techniques. This section details the steps used in Petri net to fault free conversion.

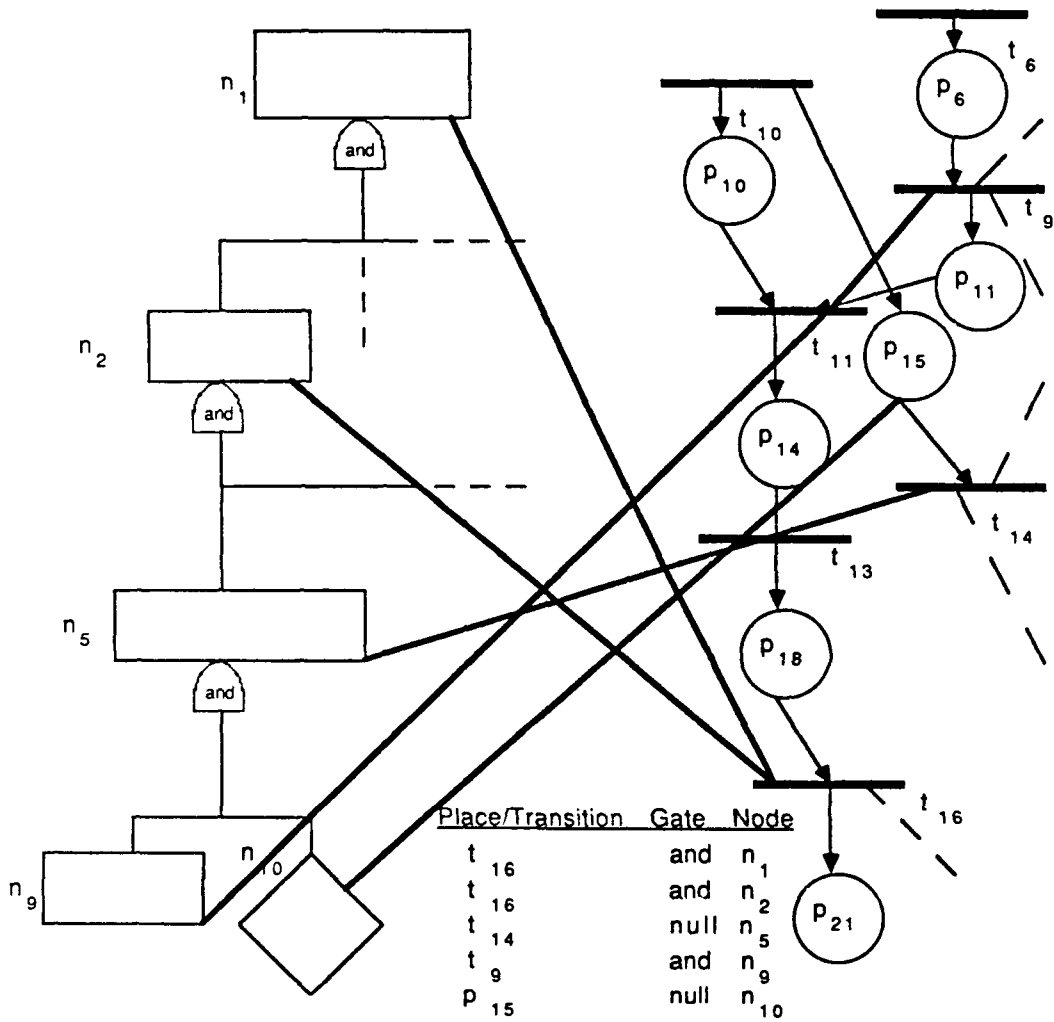


Figure 3-2 General Petri Net and Fault Tree Graphical and Tabular Linkage

1. Petri Net to Fault Tree Conversion Initiation

Initiate a conversion from a Petri net to a fault tree by choosing a root fault derived from the Preliminary Hazard Analysis (PHA) of the high level code. The example root fault, n_1 (Practice Command Causes Actual Effect), in Figure 3-3, is one of several root faults that could result from a PHA on OFP 240.

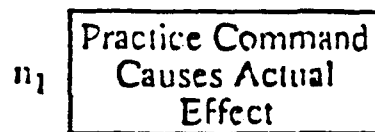


Figure 3-3 OFP Root Fault

2. Petri Net and Fault Tree Starting Point Link

The selected root fault provides a starting point for the link between the Petri net and fault tree. Once the analyst has selected the root fault, he associates it with the set of Petri net places and transitions that may immediately lead to the root fault. For the OFP 240 example the fault tree root fault condition node, n_1 (Practice Command Causes Actual Effect) in Figure 3-2 corresponds to the resultant transition, t_{16} (Command Executed), in Figure 3-1. The effect being analyzed is the firing of the weapon prompted by the executing command. While working backward in the Petri net, the analyst develops and supports subsequent fault tree nodes by working downward in the fault tree, as delineated in the next step.

3. Petri Net to Fault Tree Graphical and Tabular Linkage

Link the fault tree to the Petri net by working backward in the Petri net. This is done to see if the initial marking of the Petri net is reached by linking new nodes in the fault tree to the Petri net places and transitions they relate to. Figure 3-4 exemplifies a graphical cross linking of a portion of the related existing Petri net and newly created fault tree segments respectively. Note that the fault tree node n_2 (AIU Executes Live Command) also relates to the Petri net transition t_{16} (Command Executed(STA)). This node and transition both indicate the execution of the command to fire the weapon. The later fault tree node n_5 (Command Sent From ACU) corresponds directly to the Petri net transition t_{14} (Pulse Propagated) as the analyst works back up the Petri net. Two fault tree nodes, n_9 (Command Enabled) and n_{10} (Arm Switch Signal On) are preconditions to n_5 . The fault tree node n_9 links to the Petri net transition t_9 (Command Ready). The other node required before a command is sent, n_{10} (Arm Switch Signal On), is analogous to the PN_I place p_{15} . p_{15} indicates that a firing pulse is ready to for propagation due to the prerequisite condition of the landing gear being up.

This linkage can also be represented in tabular notation as shown in Figure 3-5, reflecting the Petri net to the fault tree relationship. Once a Petri net and fault tree are initially created and

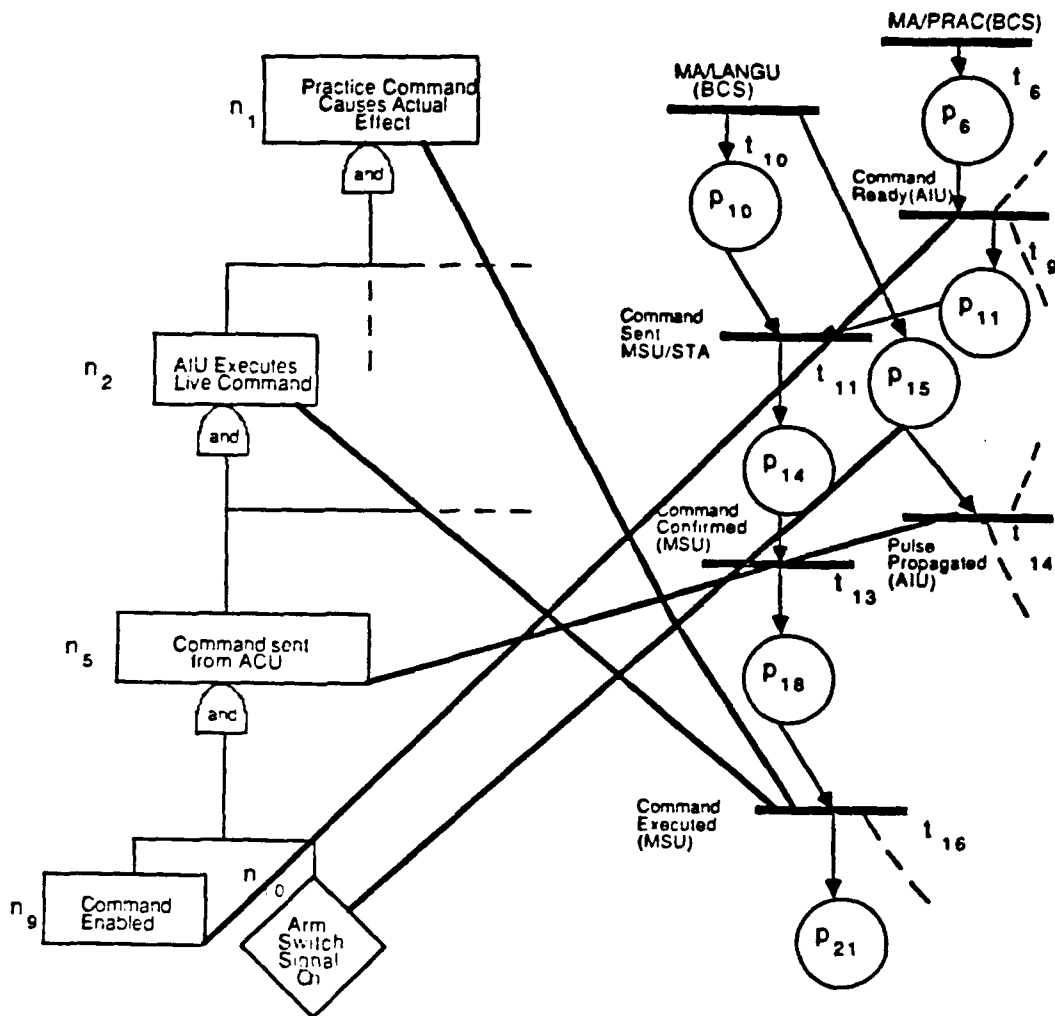


Figure 3-4 OFP Petri Net to Fault Tree Cross Diagram

linked, each fault tree has one or more linked corresponding portions of the Petri net identified in the Petri net fault tree (PNFT) Linkage.

4. Complete Development of the Petri Net Fault Tree

Linkage Table

Indicate the appropriate type of gate 'and', 'or', or 'null' in the PNFT Linkage table. Use 'and' or 'or' when two or more Petri net places and transitions relate to a single fault tree node and a null when a single Petri net place or transition relates to a single fault tree node. Figure 3-6 shows the gate relationships for the example segment. Sometimes the table does not reflect a complete linkage.

Figure 3-7 represents the fully developed fault tree.

5. Remedy If No Path to the Unsafe Event is Exposed.

If it is not possible to work back to the beginning of the Petri net, exposing a potential path to the unsafe event, do further analysis and take at least one modification measure. Execute this modification by expanding the fault tree, adding conditions, or making an assumption and seeing where the assumption leads to in the Petri net.

The software design itself may prevent a thorough backtrack, reaching the root fault, from occurring even if there is a single-component failure. This prevention may be determined by comparing the conditions expressed in the fault tree with those represented by the initial marking of the Petri net.

If it is necessary to extend the fault tree, integrate the nodes into an expanded fault tree FT_E . An existing fault tree leaf may become a

Place/Transition	Node
t_{16}^2	n_1
t_6^3	n_1
t_{16}^2	n_2
t_6^3	n_3
p_{14}^2	n_4
t_{14}^2	n_5
t_5^2	n_6
p_{15}^2	n_7
t_{11}^2	n_8
t_9^2	n_9
t_9^3	n_9
p_{15}^2	n_{10}
t_{10}^2	n_{11}
t_6^3	n_{12}
p_{15}^2	n_{13}

Figure 3-5 OFF Basic PNFT Link Table Without Gates

Place/Transition	Gate	Node
t_{16}^2	and	n_1
t_6^3	and	n_1
t_{16}^2	and	n_2
t_6^3	null	n_3
p_{14}^2	null	n_4
t_{14}^2	null	n_5
t_5^2	null	n_6
p_{15}^2	null	n_7
t_{11}^2	null	n_8
t_9^2	and	n_9
t_9^3	and	n_9
p_{15}^2	null	n_{10}
t_{10}^2	null	n_{11}
t_6^3	null	n_{12}
p_{15}^2	null	n_{13}

Figure 3-6 OFF Basic PNFT Link Table With Gates

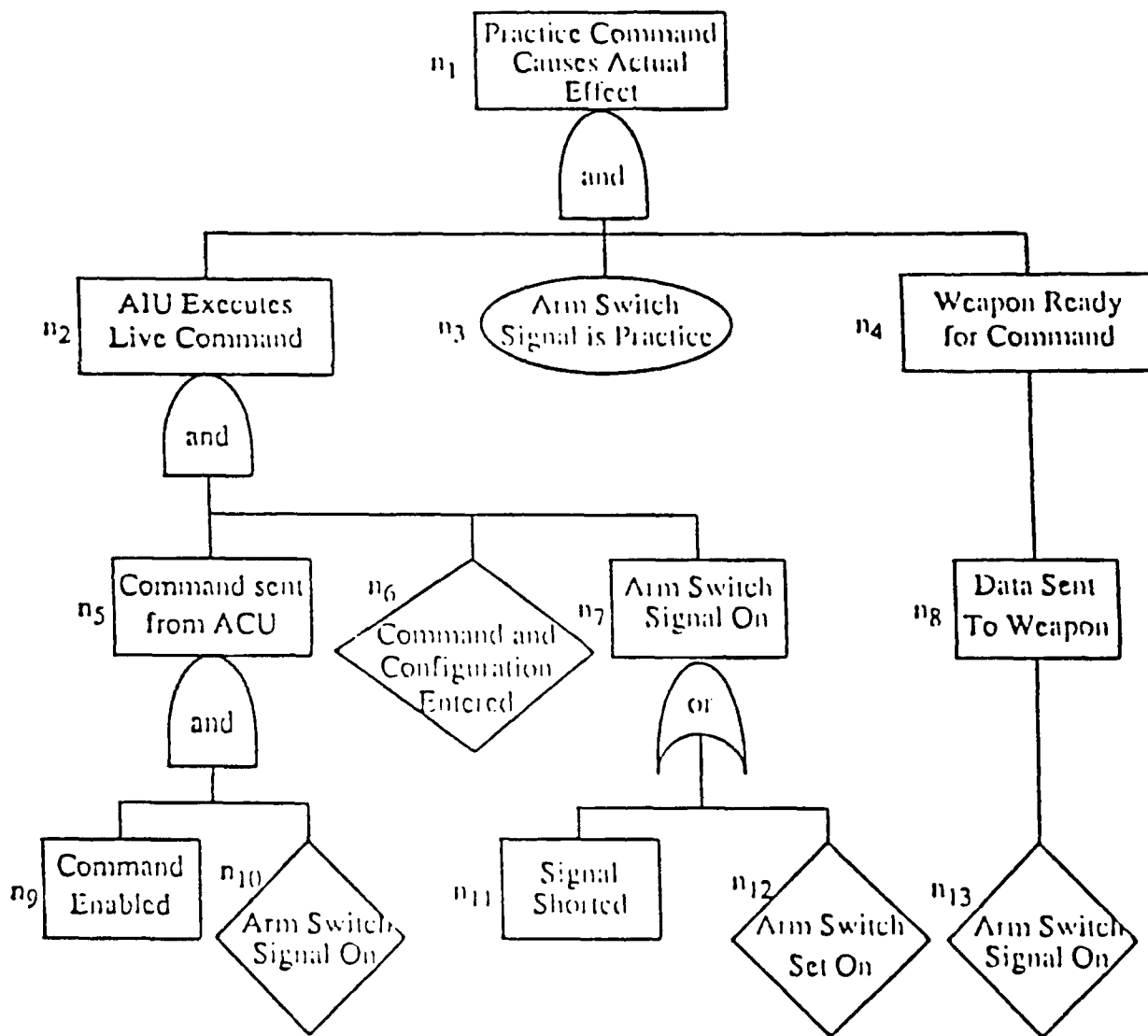


Figure 3-7 OFP Fault Tree

FT_E predecessor to the newly created nodes or a new expanded node lineage stemming from the root node may develop as the analysis proceeds.

The evolved FT_E thus becomes the basis for a FT_E to execute Petri net PN_E conversion. Execute the conversion from the FT_E to a PN_E by linking the additional FT_E nodes to their associated PN_E places and transitions.

If the Petri net has not changed except to reflect the FT_E , merely collate the links. Otherwise, establish the links to the new places or transitions in the Petri net and collate the links.

Simultaneously create an expanded Petri net fault tree $PNFT_E$ Linkage reflecting the relationships between the FT_E and the PN_E . Determine their associated type of gates 'and', 'or', or 'null'. Execute this cyclic conversion as many times as necessary during analysis.

C. FAULT TREE TO PETRI NET CONVERSION AND INTEGRATION

Fault trees are well suited to initially describe a system that is ordered or deterministic. During the analysis, should concurrency or timing issues arise, the analyst may elect to proceed using Petri net techniques. This section details the steps involved in converting fault trees to Petri nets.

Figure 3-8 shows the 'and' and 'or' relationships of transitions and places in Petri net analysis.

1. Fault Tree and Petri Net Starting Point Link

The root fault of the fault tree provides a starting point for the development of a partial Petri net and link between the fault tree and the Petri net. Figure 3-9 depicts the fault tree root fault condition node, n_1 (Practice Command Causes Actual Effect) as creating p_1 of the Petri net. The hazard being analyzed is the firing of a weapon during a simulated weapon use. While working top-down in the fault tree develop subsequent transitions and places in the Petri net, as delineated in the next step.

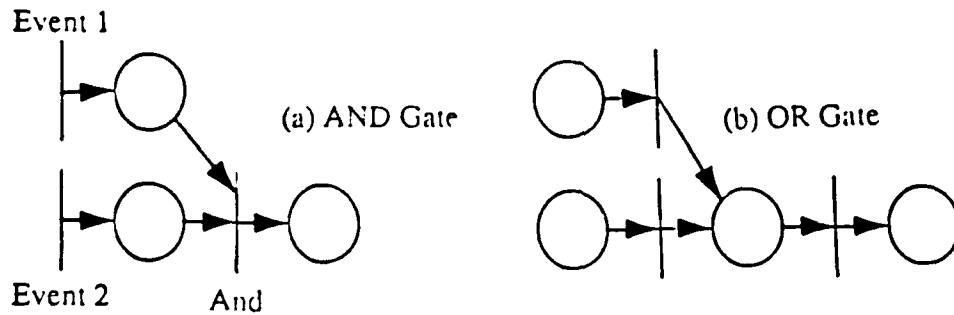


Figure 3-8 Petri Net 'and' and 'or' Gates

2. Fault Tree to Petri Net Graphical and Tabular Linkage

Work downward in the fault tree to create the Petri net. For each node in the tree, create a place in the Petri net to represent the condition. Use transitions to represent the combinations indicated by the gates in the fault tree. Create linkage elements with null gates to represent the relationships. In Figure 3-9 the fault tree node n_2 (AIU Executes Live Command) creates p_2 . This condition indicates the execution of the command to fire the weapon. Fault tree node n_5

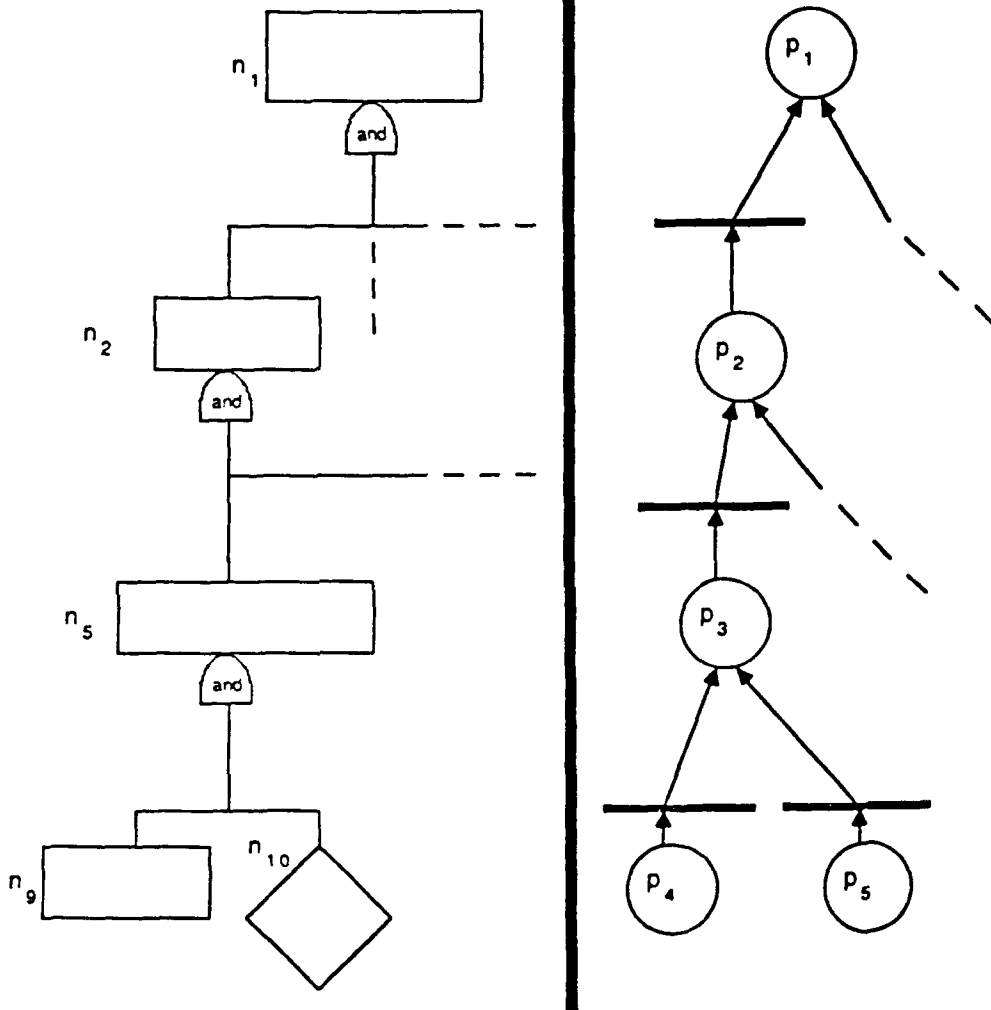


Figure 3-9 Petri Net Creation from a Fault Tree

(Command Sent From ACU) creates p_3 , the propagation of a pulse. Two fault tree nodes, n_9 (Command Enabled) and n_{10} (Arm Switch Signal On) are preconditions to n_5 . Fault tree node n_9 (Command Enabled) creates p_4 , command ready state. The other node required before a command is sent, fault tree node n_{10} (Arm Switch Signal On).creates p_5 . P_5 indicates that a firing pulse is ready for propagation due to the prerequisite of the landing gear being up.

Once the analyst creates a Petri net and links it to the fault tree, the net must be augmented before any analysis may be performed. The next step guides this augmentation.

3. Petri Net Completion

The initial Petri net generated from the fault tree will be incomplete. Fault trees only contain events and conditions specifically related to the root fault and omit all other functional behavior. This omitted behavior must be added to the Petri net before Petri net safety analysis can proceed.

Leveson and Stolzy [Ref. 9] describe how a Petri net may be derived from a software system. In this case, the derivation is focused on the gaps in the initial Petri net. As the Petri net is completed, the analyst may need to combine and rearrange nodes generated from the fault tree. As this is done, the linkage relation must be modified to reflect the change. Further inspection of analysis may reveal that the added portions of the Petri net relate to portions of the fault tree. In that case, the analyst must include

these connections in the linkage relation, modifying 'null' gates to 'and' or 'or' gates as needed.

The derived and expanded Petri net may then be used for safety analysis as described by Leveson and Stolzy [Ref. 9]. The results of this analysis lead to the generation of new fault tree nodes as described in Section B.

IV. SUMMARY AND CONCLUSIONS

A. INTEGRATED ANALYSIS TECHNIQUE

The work described in this thesis integrates Software Fault Tree Analysis (SFTA) and timed Petri nets to facilitate software-safety analysis in heterogeneous-multiprocessor control systems. The integrated technique uses Petri nets for ordering events and conditions across multiple processors, explicitly representing concurrent and sequential events. This allows for simpler fault tree analysis, or use of techniques such as those of Leveson and Stolzy [Ref. 9], who describe Petri net analysis for concurrency errors and time-related errors.

The recording of analysis logic, explicitly representing the semantics of the events and conditions that lead to a hazard, requires SFTA. Petri nets explicitly show combinations of events and conditions but the semantics behind those events and conditions are abstracted away. The integrated technique uses fault trees for analyzing non-sequential (or non-local) sequences of conditions that lead to a hazard, where several parts of the system must coordinate for the hazard to occur. This part of the analysis is derived from that of Leveson and Harvey.

A stepwise methodology is presented for converting a fault tree into a Petri net or converting a Petri net into a fault tree, then linking the two together for an integrated analysis. The order of technique

usage is dependent on the particular software being analyzed. During Petri net to fault tree conversion, the analyst establishes a fault to a Petri net linkage while he is creating the fault tree from the Petri net. Inversely, he establishes a Petri net to fault tree linkage while he is creating the Petri net from the fault tree during fault tree to Petri net conversion.

The resulting integrated linkage relation eases the iterative conversion between the two analysis techniques. This relation links information in one representation to fields of the other representation.

B. LESSONS LEARNED

This thesis uses a formal basis to the integrated analysis technique. This formal basis is useful as it clarifies information content in the unions representations and provides a vocabulary to discuss conversion and linkage. An initial, informal, conversion sketch is created, then formalized and restructured around the formalization.

Multiple views of analysis information are stressed. The multiplicity of views gives the analyst a broader view of the analysis process, allowing him to both detect more subtle problems in the software and to identify problems in the analysis itself. The multiplicity occurs across two dimensions: presentation of technique (graphic and textual) and technique usage (Petri nets or fault trees).

Graphic and textual views are essentially equivalent and complimentary. The graphic view provides detail and connection focus.

The integration of SFTA and Petri net analysis also gives the analyst two complementary views of the system. The use of the system's organizational representations in the Petri net and the explicit fault representation and diagnosis in the fault tree are melded for a synergistic effect. The Petri net provides organization and emulation, whereas the fault tree provides combination and logic.

C. FUTURE WORK

The first recommendation stemming from the work done in this thesis is to automate the integrated software-safety analysis method presented. An analyst should not be replaced by an automated tool. An analyst interaction focusing the analysts insight and experience on the hazards of a system. This method is proposed only as a tool for evaluation during the overall safety analysis. Tools exist for Petri net (P-Nut) and fault tree (SFTAT) analysis. A means to tie these tools together is suggested in this thesis.

Other modeling techniques that lend themselves well to integration need to be explored (such as PHA, Markov Chains, and FMEA). Individually, safety-analysis techniques have weaknesses that may be better addressed by other techniques. The integration of two or more analysis methods may well help to reduce life,

property, and environmental losses due to hazard-inducing software by utilizing their combined advantages.

D. CONCLUSIONS

Software faults must not cause hazards after a system is in the field. Loss of life or property can result from hazardous software used in real-life environments. Analysts should make every effort to find all safety-critical software faults before the developer delivers the system to the users. Integrated analysis methods should enhance early fault discovery by focusing on the key safety-critical portions of the software and avoiding redundant analysis.

Further research in all areas of software-safety is required to prevent hazards as early as possible in the development and maintenance life-cycle of people-, property-, or environment-critical software.

LIST OF REFERENCES

1. Shimeall T. J., McGraw, R. J., and Gill, J. A., "Software Safety Analysis in Heterogeneous Multiprocessor Control Systems", *1991 Proceedings Annual Reliability and Maintainability Symposium*, Orlando, Fl, January 1991.
2. *Software Safety Handbook*, H. Q. AFISH/SSH 1-1, Norton Air Force Base, CA, 5 September 1985.
3. Petersen, D., *Techniques of Safety Management*, McGraw-Hill, New York, 1971.
4. Malasky, S. W., *System Safety Technology and Application*, Garland STPM Press, New York, 1982.
5. *Software Engineering Standards*, The Institute of Electrical and Electronics Engineers, New York, 1984.
6. Leveson, N. G., and Harvey, P. R., "Analyzing Software Safety", *IEEE Transaction Software Engineering*, SE-9, pp. 569-579, 5 September 1983.
7. Leveson, N. G., "Software Safety: Why, What, and How", *Computing Surveys*, vol. 18 no. 2, pp. 125-163, June 1986.
8. Murata, T., "Petri Nets: Properties, Analysis and Applications", *Proceedings of the IEEE*, Vol. 77, No. 4, pp. 541-580, April 1989.
9. Leveson, N. G. and Stolzy, J. L., "Safety Analysis Using Petri Nets", *IEEE Transactions on Software Engineering*, vol. SE-13, no. 3, March 1987.
10. McGraw, Richard J., *Petri Net and Fault Tree Analysis: Combining Two Techniques for a Software Safety Analysis on an Embedded Military Application*, M.S. Thesis, Naval Postgraduate School, Monterey, CA, December 1989.

11. Peterson, J. L., *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1981.
12. Cha, Stephen S., Leveson Nancy G., and Shimeall, Timothy J., "Fault Tree Analysis Applied to Ada", *Proceedings of the Tenth International Conference on Software Engineering*, Singapore, 1988.
13. Dugan, Joanne B., Trivedi, Kishor S., Smotheman, Mark K., and Geist, Robert M., "The Hybrid Automated Reliability Predictor", *Journal of Guidance, Control, and Dynamics*, vol. 9, no. 3, May-June 1986.
14. Howell, Sandra V., Bavuso, Salvatore J., and Haley, Pamela J., "A Graphical Language for Reliability Model Generation", *1990 Proceedings Annual Reliability and Maintainability Symposium*, Atlanta, GA, January 1990.

BIBLIOGRAPHY

Connolly, Brain, "Software Safety Goal Verification Using Fault Tree Techniques: A Critically Ill Patient Monitor Example", *Proceedings of IEEE Compass '89*, Gaithersburg, MD, pp. 18-29, June 19-23, 1989.

Griggs, J. G., "A Method of Software Safety Analysis", *Proceedings of the Safety Conference* (Denver, CO), vol. 1, part 1, System Safety Society, Newport Beach, CA, pp. III-D-1 to III-D-18, 1981.

Ericson, C. A., "Software and System Safety", *Proceedings of the 5th International System Safety Conference* (Denver, CO), vol. 1, part 1, System Safety Society, Newport Beach, CA, pp. III-B-1 to III-B-1, 1981.

Gloss, D. S., and Wardle, M. G., *Introduction to Safety Engineering*, Wiley, NY, 1984.

Hammer, W., *Handbook of System and Product Safety*, Prentice-Hall, Inc., 1972.

Harvey, Peter Randall, "Fault Tree Analysis of Software", M. S. Thesis, University of California, Irvine, CA, 1982.

Konakovsky, R., "Safety Evaluation of Computer Hardware and Software", *Proceedings of COMPAC '78*, IEEE, NY, pp. 559-564, 1978.

Leveson, Nancy G., *Building Safe Software*, Computer Science Technical Report NO. 86-14, University of California, Irvine, CA, February 1986.

Leveson, N. G., and Stolzy, J. L., "Using Fault Trees to Find Design Errors in Real Time Software", *AIAA 21st Aerospace Science Meeting*, Reno, NV, January 10-13, 1983.

McIntee, J. W., *Fault Tree Techniques As Applied to Software (Soft Tree)*, Technical Report, USAF, March 1983.

McKinlay, Archibald, "Software Safety Handbook", *Proceedings of IEEE Compass '89*, Gaithersburg, MD, pp. 14-19, 19-23 June 1989.

Merlin, P. M. and Farber, D. J., "Recoverability of Communication Protocols Implications of a Theoretical Study", *IEEE Transaction on Communications*, Vol. COM-24, pp. 1036-1043, September 1976.

Neumann, Peter G., "The Computer-Related Risk of the Year: Misplaced Trust in Computer Systems", *Proceedings of IEEE Compass '89*, Gaithersburg, MD, pp. 9-13, June 19-23, 1989.

Petri, C., *Kommunikation mit Automaten*, Ph.D. dissertation, University of Bonn, West Germany, 1962.

Razouk, R. R., *A Guided Tour of P-NUT*, Technical Report Number 86-05, Department of Information and Computer Science, University of California, Irvine, March 1986.

Roland, H. E. and Moariarity, B., *System Safety Engineering and Management*, Wiley, NY, 1983.

Rolandelli, C., Shimeall, T. J., Genung, C., and Leveson, N., *Software Fault Tree User's Manual*, Technical Report 86-06, University of California, Irvine, February 1986.

Taylor, J. R., *Logical Validation of Safety and Control System Specifications Against Plant Models*, Technical Report RISO-M-2292, Risoe National Laboratory, Roskilde Denmark, May 1981.

Thomas, Jeffery C., and Leveson, Nancy G., *Applying Existing Safety Design Techniques to Software Safety*, Computer Science Technical Report, University of California, Irvine, CA, September 1981.

Vesely, W. E., Goldberg, F. F., Roberts, N. H., and Haasl, D. F., *Fault Tree Handbook*, NURREG-0492, U. S. Nuclear Regulatory Commission, January 1981.

Weik, Martin, "Computer Dictionary", *IEEE Computer Society Standards Committee*, ed., IEEE Computer Society, 1976.

A-6E Operational Flight Program E 250, Mini-PPS, Revision B, Naval Weapons Center, China Lake, CA, 93555, 23 June, 1989.

A-6E 4 PI Developmental Flight Program, E 544.06, Math Flows Draft, Naval Weapons Center, China Lake, CA, 93555, 17 May , 1989.

MIL-STD-1553B (DOD), Specification Control , Missile C-11524/A(AIU), Program Performance Specification (PPS) for A-250, NWC-2478-250 Revision C, Naval Weapons Center, China Lake, 1 June 1989.

MIL-STD-1679 (DOD), Control Missile C-11524/A (AIY) and Ballistics Computer Set, CP-1391/ASQ-155A (BCS), Interface Design Specification (IDS), NWC-2482-250, Revision A. Naval Weapons Center, China Lake, CA, 93555, 1 August 1989.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2. Library, Code 52 Naval Postgraduate School Monterey, California 93943	2
3. Department Chairman, Code CS Computer Science Department Naval Postgraduate School Monterey, California 93943	1
4. Professor Timothy J. Shimeall, Code CS/Sm Computer Science Department Naval Postgraduate School Monterey, California 93943	2
5. Professor Kim A. S. Hefner, Code MA/Hk Math Department Naval Postgraduate School Monterey, California 93943	1
6. Mr. Bob F. Westbrook (Code 31) Naval Weapons Center China Lake, California 93555	1
7. Mr. Werner Hueber (Code 3104) Naval Weapons Center China Lake, California 93555	1
8. MS Janet A. Gill (Code FW532JG) Naval Air Test Center Patuxent River, Maryland 20670	5