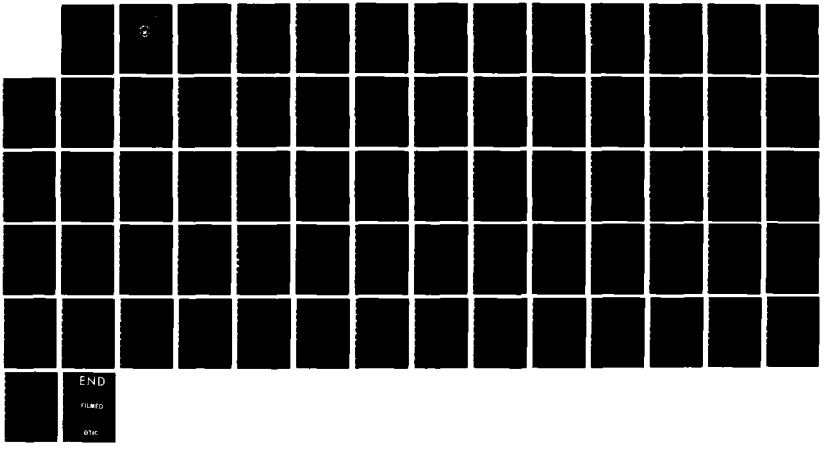
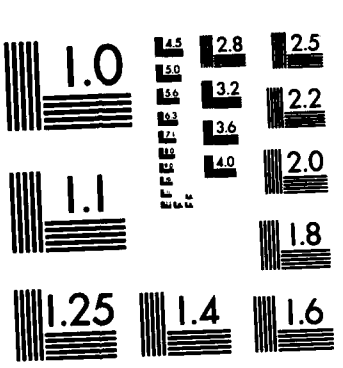
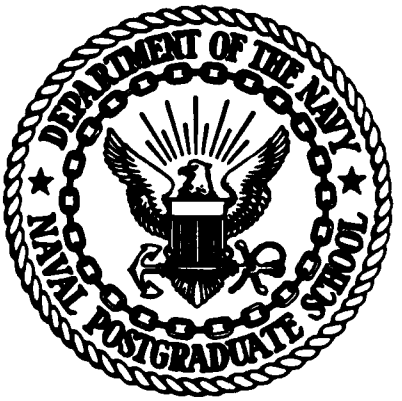
AD-A159 856 PROBABILISTIC ANALYSIS OF FAULT TREES USING PIVOTAL DECOMPOSITION(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA W T MCCULLERS JUN 85 1/1

UNCLASSIFIED F/G 12/1 NL





NAVAL POSTGRADUATE SCHOOL
Monterey, California



THESIS

PROBABILISTIC ANALYSIS OF FAULT TREES USING PIVOTAL DECOMPOSITION

by

William T. McCullers III

June 1985

Thesis Advisor: R. Kevin Wood

Approved for public release; distribution is unlimited

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A159 856	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Probabilistic Analysis of Fault Trees Using Pivotal Decomposition	5. TYPE OF REPORT & PERIOD COVERED Master's Thesis June 1985
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) William T. McCullers III	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943-5100	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943-5100	12. REPORT DATE June 1985
	13. NUMBER OF PAGES 68
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public rlease; distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Fault Tree; Pivotal Decomposition; System Failure Probability; Reliability; Binary Systems

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

An algorithm is presented for computing the exact failure probability for binary systems represented as fault trees. This algorithm does not rely on cut sets. Instead, it applies recursive pivotal decomposition together with probabilistic

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

#20 - ABSTRACT - (CONTINUED)

structural reductions and modularization directly to the fault tree. A further capability of the algorithm is the sequential printing of equations to form a function for a specific fault tree which computes system failure probability given the basic event probabilities.

Approved for public release; distribution is unlimited.

Probabilistic Analysis of Fault Trees Using Pivotal Decomposition

by

William T. McCullers III
Major, United States Marine Corps
A.B., Rutgers University, 1973

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN OPERATIONS RESEARCH

from the

NAVAL POSTGRADUATE SCHOOL
June 1985

Author: William T. McCullers III
William T. McCullers III

Approved by: R. Kevin Wood
R. Kevin Wood, Thesis Advisor

James D. Esary
James D. Esary, Second Reader

Alan R. Washburn, Chairman,
Department of Operations Research

Kneale T. Marshall
Kneale T. Marshall,
Dean of Information and Policy Sciences

ABSTRACT

In this thesis, an algorithm is presented for computing the exact failure probability for binary systems represented as fault trees. This algorithm does not rely on cut sets. Instead, it applies recursive pivotal decomposition together with probabilistic structural reductions and modularization directly to the fault tree. A further capability of the algorithm is the sequential printing of equations to form a function for a specific fault tree which computes system failure probability given the basic event probabilities.

Additional keywords: numerical analysis; computer programs; reliability; systems analysis.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

Many of the ideas explored by me in this thesis were originated by my thesis advisor, Professor R. Kevin Wood, and he deserves equal credit for this research. I would like to thank the Department of Computer Science at the Naval Postgraduate School for all of the technical assistance they provided. Thanks are also extended to Dr. Howard E. Lambert and to Professor Robert E. Ball of the Department of Aeronautical Engineering at the Naval Postgraduate School for providing some "interesting" fault trees for testing purposes. I thank my wife, Joanie, and two sons, Brendan and Sean, for their patience, support, and sacrifice during the months spent in this research.

I. INTRODUCTION

Fault trees are used in many fields of application to aid in assessing the probability of failure of a complex binary system as a result of sub-system or component failures. An algorithm is presented here for computing the exact failure probability for binary systems represented as fault trees. Due to the improved efficiency of this algorithm over those currently in use, reliability engineers and other users will find it useful for conducting fault tree analyses in which multiple computations of failure probabilities are needed.

Fault trees are commonly used models to represent failures in complex electrical, mechanical, and other systems. Their use originated in 1961 at Bell Telephone Laboratories in the safety assessment of the Minuteman Launch-Control System [Ref. 1]. Since then many other applications for fault trees have been found. Arnborg [Ref. 2] refers to their use in weapons effectiveness models, and Atkinson [Ref. 3] uses a fault tree model to analyze a naval weapons system. Ball [Ref. 4] uses fault trees to identify critical zones and components of aircraft subjected to anti-aircraft fire. Other areas in which fault tree models have been applied include nuclear power plant safety [Refs. 5,6,7,8], electrical systems [Ref. 9], computer hardware design [Ref. 10], and chemical processing [Ref. 11].

Efficient methods for computing the probability of system failure or, equivalently, system reliability are needed for users with large fault trees to analyze. One use for such computations is in obtaining importance measures for basic events or component failures. *Importance measures* are methods of assigning numerical values to basic events which in some way gauge how critical a component is to system reliability. These values are useful for sensitivity analysis. For example in an electrical circuit the failure of a component linked in series will be more critical to system reliability than will the same component linked in parallel. In a complex system such structural characteristics may not be so obvious. Importance measures will reflect the relative importance to the system resulting from system structure and component characteristics for each component. Lambert [Ref. 12] discusses four measures of event importance which can be computed exactly or approximately given a method for computing system reliability.

Needs exist for efficient system reliability computations for other uses. Mizukami [Ref. 13] and Derman, et al. [Ref. 14], discuss constrained problems of resource allocation with the objective of maximizing system reliability such as

$$\max \; h(\underline{p}(\underline{y}))$$

$$\text{s.t.} \quad \sum_i y_i \leq A$$

where y_i is the amount of resource allocated to component i, $\underline{p}(\underline{y})$ is an m-vector of failure probabilities of the components given $\underline{y}$, and $h(\underline{p}(\underline{y}))$ is the system reliability. Since $h(\underline{p}(\underline{y}))$ is nonlinear, this problem requires a solution using nonlinear programming techniques [Ref. 15]. Most of these techniques require computation of the objective function gradient at each iteration. Each component i in the gradient evaluated at $\underline{y}$ is given by

$$\frac{\partial h}{\partial y_i} = \sum_j \frac{\partial h}{\partial p_j}\frac{\partial p_j}{\partial y_i} = \sum_j (h(\underline{p}(\underline{y}) \mid p_j = 1) - h(\underline{p}(\underline{y}) \mid p_j = 0)) \frac{\partial p_j}{\partial y_i}$$

Thus each gradient computation requires 2m computations of $h(\underline{p}(\underline{y}))$.

In some binary systems the failures of some of the basic components are statistically dependent. In these cases, computation of system failure probability requires numerical integration. For instance, if components i, j, and k are dependent while all other component failure probabilities are independent, then system failure probability $g(\underline{p})$ can be found using

$$g(\underline{p}) = \int_0^1 \int_0^1 \int_0^1 g(\underline{p} \mid (p_i = x_i, p_j = x_j, p_k = x_k)) f(x_i, x_j, x_k) dx_i dx_j dx_k$$

where $g(\underline{p} \mid (p_i = x_i, p_j = x_j, p_k = x_k))$ is the system failure probability with the probabilities of components i, j, and k

fixed, and $f(x_i,x_j,x_k)$ the joint probability density function of components i, j, and k. Numerical integration of this function requires many computations of system failure probability. The more rapidly that $g(\underline{p}|(p_i = x_i, p_j = x_j, p_k = x_k))$ can be computed, the smaller the increments of numerical integration can be, and the more accurate $g(\underline{p})$ will be.

Many fault trees used in applications are quite large. Arnborg [Ref. 2] states that some of the military models used in practice require as many as 100,000 evaluations of fault trees containing as many as 1000 basic components to evaluate performance over different tactical situations. Reliability optimization, numerical integration, and importance determination cannot be performed on some of these larger fault trees given current methods. It is obvious that a need exists for more efficient methods to compute system failure probability for binary systems.

A. DEFINITIONS AND NOTATION

A fault tree is used to represent a binary system. A *binary system* is a system in which all components and the entire system are assumed to be either completely operational or completely failed. A binary system is denoted (C,Φ) where C is the set of *components* and Φ is a binary function of the component states. Let $x_i \in \{0,1\}$ represent the state of the ith component of a binary system with m components. The system state is given by $\Phi(\underline{x}) \in \{0,1\}$, where $\underline{x} = (x_1,x_2,\ldots,x_m)$ is the system *state vector*. If $x_i = 0$, then the state vector $\underline{x}$

in the number of replicated basic events. Reduced state enumeration enumerates the states of each replicated event e_i over any cut event e_j. Reduction is achieved since the states of all e_i below e_j can be replaced by the states of e_j in an expression for the states of some e_k above e_j. This method is only useful, however, when no prime F-modules of the fault tree contain a large number of replicated events.

Of the methods discussed above only PAFT F77 takes advantage of topological reductions and then only in a crude manner. This thesis applies probabilistic structural reductions to fault trees. Although theoretical complexity remains exponential in the number of replicated events, actual complexity will be reduced by these reductions.

2. Recursive Pivotal Decomposition

Let g(F) denote the system failure probability for a particular fault tree F. If F has no replicated events, g(F) may be computed by repeated application of simple reductions. When F is reduced to a single basic event e_j, $g(F) = p_j$. If, after all simple reductions have been made, F is not reduced to a single event, some replicated basic event e_i must remain. From the theorem of total probability, for any remaining basic event e_i

$$g(\underline{p}) = p_i g(1_i, \underline{p}) + (1 - p_i) g(0_i, \underline{p})$$

for a binary system. This is the equation for *pivotal decomposition*. For a fault tree the equation becomes

which use cut sets include "inclusion-exclusion" [Ref. 24: p. 98-101], "sum of disjoint products" [Refs. 25,26], and "$\Sigma\Pi$" [Ref. 27]. A common requirement of these methods is the enumeration and storage of all cut sets. The number of cut sets in a binary system can be exponential in the size of the system. Therefore, for a large system these methods may be limited to approximations for $g(\underline{p})$. Using the inclusion-exclusion and sum of disjoint products methods the generation of all terms needed for computation of $g(\underline{p})$ is exponential in the number of cut sets. Consequently, for both of these methods the complexity is exponential on an exponential function of the problem size. Most methods which depend on cut sets never take advantage of the structure of the systems they model, such as the presence of modules or other simplifying properties, and, consequently, are guaranteed to always require large amounts of time and space to compute $g(\underline{p})$. $\Sigma\Pi$, which locates independent blocks of cut sets and evaluates them separately, can achieve exponentially better efficiency than the sum of disjoint products methods.

Two methods which do not use cut sets are "PAFT F77" [Ref. 28] and "reduced state enumeration" [Ref. 2]. These methods are based on the fault tree model of a binary system. PAFT F77 removes all replicated basic events by conditioning and then uses simple reductions to compute $g(\underline{p})$. This method does not allow replicated intermediate events, and is guaranteed an actual complexity factor which is exponential

to compute g(p) in time bounded by a polynomial function of the number of events [Ref. 23: p. 113]. The best known upper bound on time for any algorithm to solve g(p) is an exponential function of the problem size. The best known bound on space, however, is polynomial.

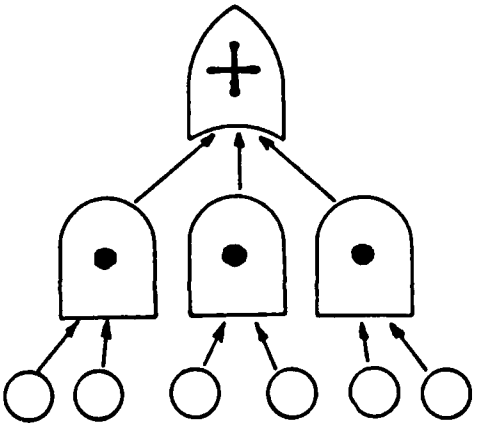
Despite the inherent exponential complexity of the problem, it is still possible to exactly compute g(p) for many moderate sized fault trees. It is the purpose of this study to take advantage of structural properties of fault trees to extend the range of problems for which exact probabilities can be computed. The method described for use in a fault tree with no replicated events will be useful as a subroutine in a more general algorithm.

C. COMPUTATIONAL METHODS

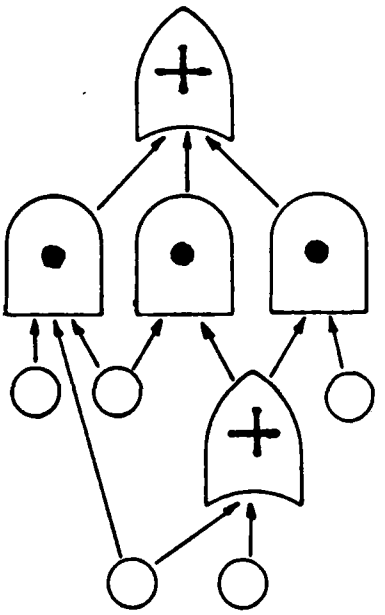
Several different exact and approximate methods for probabilistic analysis of fault trees have been developed for fault trees with replicated events. Most of these methods ignore the topological structure of the fault tree while relying on cut set enumeration to compute g(p). Because of the inefficiency of these methods, exact values of g(p) are not computable for large systems and must be approximated by use of upper and lower bounds or Monte Carlo simulation.

1. Existing Methods

Current methods for computing g(p) for binary systems represented as fault trees can be placed into two categories, those using cut sets and those not using cut sets. Methods



a. Without Replicated Events



b. With Replicated Events

Figure 1-2 Fault Trees

computing $g_j(\underline{p})$ at each logic event from the bottom of the fault tree to the top event. This procedure can be used in any fault tree without replicated events. Computation of top event probability for a fault tree in this case can be accomplished in time $O(|L|)$ in space $O(|L|)$. (Since H is assumed connected, $|L| \geq |E| - 1$, and $O(|E| + |L|)$ is effectively $O(|L|)$.) Referring to Figure 1-2a, F is searched from the top event downward, i.e., following $\overset{*}{H}$. When an intermediate event which has only basic input events is found, the probability of the intermediate event is computed, and it becomes a basic event. The search continues, gradually reducing all intermediate events to basic events in a backtracking procedure until the top event probability is computed. These reductions are *simple reductions*, and a formal algorithm to perform them is given in Chapter II.

The assumption of independence among input events which allows simple reductions cannot be made throughout a fault tree containing replicated events. Any two events e_i and e_j which are on separate directed paths from the same replicated event e_k cannot be assumed to be independent since the states of e_i and e_j both depend on e_k. Replicated events complicate the computation of top event probability. In fact, Rosenthal showed the problem of computing $g(\underline{p})$ for a fault tree F containing replicated events to be a member of the class of nondeterministic polynomial hard (NP hard) problems [Ref. 22]. Consequently, no algorithm exists or is likely to be developed

mission fails to pass the pre-flight safety checks and consequently cannot begin the mission.

Let $g(\underline{p})$ denote the probability of the top event in a fault tree, and let $g_i(\underline{p})$ denote the probability of occurrence of an intermediate event i. In a fault tree without replicated events, computation of $g(\underline{p})$ is easy. Since the top and intermediate events are represented by logic events, e_j, their probability can be computed directly if the events, e_i, for all i s.t. $(v_i, v_j) \in \vec{L}$ are all mutually independent and have known probabilities. The equations used to compute these probabilities are found in Table 1-2.

TABLE 1-2

Logic Event Probabilities

Event Type	Computation
AND	$g_j(\underline{p}) = \prod_i p_i$
OR	$g_j(\underline{p}) = 1 - \prod_i (1 - p_i)$
2-out-of-3	$g_j(\underline{p}) = p_1p_2p_3 + (1-p_1)p_2p_3 + p_1(1-p_2)p_3 + p_1p_2(1-p_3)$
NOT	$g_j(\underline{p}) = 1 - p_i$

Hwang [Ref. 20] and Shanthikumar [Ref. 21] provide recursive algorithms for general K-out-of-N systems which operate in polynomial time. Using these equations $g(\underline{p})$ can be found by

assumptions are valid for the three categories of systems described below.

The first category is the set of non-repairable systems. In this case $p_i = F_i(\tau)$ is the probability that component i has failed by time τ [Ref. 19]. System failure by time τ then is $g(\underline{F}(\tau))$. A tactical aircraft on a mission is an example of a non-repairable system where the interval $(0,\tau)$ represents the time span from takeoff to landing.

The second category is the set of systems for which component "up" and "down" times form independent renewal processes [Ref. 19]. Here, D_i is the component "down" time, and U_i is the component "up" time. The probability that component i is "down" or in a failed state at a given instant of time and the proportion of time that i will spend in a "down" state are both given by

$$p_i = \frac{E(D_i)}{E(U_i) + E(D_i)}$$

An example of this type of system is an electrical power generating station which runs continuously.

The final category of failures is point failures. *Point failures* are realized if a system fails to activate when its "on" switch is engaged. In this case p_i and $g(\underline{p})$ are simply the probabilities that component i and the system, respectively, fail to activate. Point failure is a fair assumption for modeling the probability that an aircraft to be flown on a

Computation of any problem on a digital computer requires time and storage. Let f be some function of the size of the fault tree such as f(|E|) or f(|L|). Then let O(f) be a known linear function of f which provides an upper bound on some requirement for the problem. O(f) is the *algorithmic complexity* of the problem for the specific requirement. If the requirement is *space*, then O(f) denotes the storage requirement in terms of the problem size, while if the requirement is *time*, it denotes the CPU time required in the same terms.

Although not utilized in this study, later reference will be made to other fault tree algorithms which utilize cut sets and path sets. A *cut set* is a set of basic events whose occurrence ensures occurrence of the top event. A cut set is *minimal* if no event can be removed while still ensuring occurrence of the top event. A *path set* is a set of basic events whose nonoccurrence ensures nonoccurrence of the top event. [Ref. 16: p. 9] (This terminology originates from network reliability.)

B. PROBLEM DEFINITION AND COMPLEXITY

The objective of this thesis is to develop an efficient algorithm to compute g(p), the probability of the top event of a fault tree. It is assumed that a probability p_i for each basic event in F is known. However, assignment of a probability p_i to a basic event is only correct when certain assumptions about the modeled system can be made. These

$H - v_j = \{H_0, H_1, H_2, \ldots, H_k\}$, where each H_i is connected for all i, but there is no connection between H_i and H_j for $i \neq j$, and where H_0 contains the vertex corresponding to the top event of F. Let $H_i = (V_i, L_i)$, and $E_i = \{e_\ell : e_\ell = (v_\ell, t_\ell)$ for all $v_\ell \in V_i\}$. Then, $F_i = (E_i + e_j, \vec{L}_i \cup \{\ell_{kj} \in \vec{L} : v_k \in V_i\})$ is an F-*module* for $i = 1,2,\ldots,k$ with cut event e_j. The non-null union of any combination of these F_i is also an F-module with cut event e_j.

Consider the F-module $F' = (E', \vec{L}')$ in F. Let $e_i \in E$ be any event connected into the cut event e_j by links $\ell_{ij} \in \vec{L}$. If $e_i \in E'$ for all i, then e_j is an *F-module top*, and F' is a *simple F-module*. If separated from F, a simple F-module with an F-module top has the same properties as a fault tree. The cut event of a general F-module may have other e_i connected into it where $e_i \notin E'$, and therefore does not necessarily possess all the fault tree properties. F is always an F-module of F. Any other F-module in F is a *proper F-module*. An F-module is *trivial* if it contains only one or more unreplicated basic events plus the cut event. Any F, whose only proper F-modules are trivial, is a *prime F-module*.

In a graph H, if a maximal set of vertices $V_0 \subseteq V$ exists such that for every distinct subset of three vertices $\{v_i, v_j, v_k\} \subseteq V_0$ there exists a path between v_i and v_j not containing v_k, then v_0 is a *biconnected component* [Ref. 18: p. 179]. If all paths from any $v_i \in V_0$ to any $v_\ell \notin V_0$ must pass through the same vertex $v_j \in V_0$ for $i \neq j$, then v_j is a cut vertex of V_0.

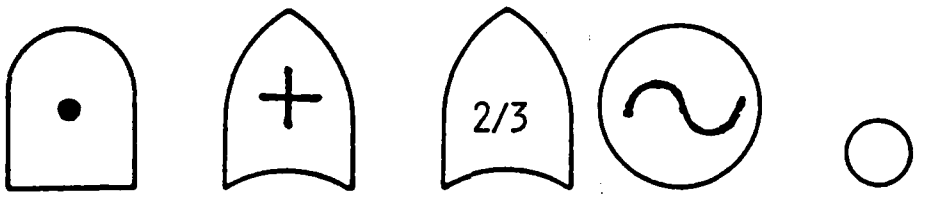


Figure 1-1 Logic Events

the most commonly encountered in fault tree models. In fact, all structure functions can be represented using only logic types AND, OR, and NOT. NOT events will always have an out-degree and in-degree of one, and their presence implies a noncoherent system. Figure 1-1 displays the symbols for events to be discussed in this thesis. This thesis will only consider these event types since they are the most common, and the algorithm developed using these event types can be easily extended to other types.

An *event tree* is a generalization of a fault tree in which system operation or failure can be represented. Event trees representing failures are usually referred to as fault trees. There are no structural or computational differences between fault trees and event trees, and the term "fault tree" is used throughout this thesis. Another representation of a binary system which is used is the *reliability network*. This representation is not considered here since it does not lend itself to modeling general binary systems [Ref. 17].

A *module* is a set of basic events which behave as one event. Consider a binary system (C,Φ) with $A \subseteq C$, and let $\underline{x} = (\underline{x}_A, \underline{x}_{\overline{A}})$. If $\Phi(\underline{x}) = \Phi'(\Phi''(\underline{x}_A)\underline{x}_{\overline{A}})$, for structure functions Φ' and Φ'', then (A,Φ'') is a module [Ref. 16: p. 16].

A module in a binary system can often be directly recognized in a fault tree. Consider the graph H derived from F and a specified vertex v_j. If H is connected, and $H-v_j$ is disconnected, then v_j is a *cut vertex*, and e_j is a *cut event*.

tree. Failures in basic components cause failures in intermediate components which may ultimately lead to occurrence of the top event, aircraft failure.

In the fault tree each event has a type, $t_i \in T$. For the top and intermediate events, t_i denotes a logic type, e.g., AND, OR, while for basic events, t_i is type BASIC. Any event with an out-degree greater than one represents a *replicated event*. The number of replicated events in the fault tree is denoted by r.

Table 1-1 shows the logical operations performed at e_j on the events e_i linked into e_j by the links ℓ_{ij}.

TABLE 1-1

Logical Operations

Logic Event	Input	Output
AND	x_i for all i s.t. $(v_i,v_j) \in \vec{L}$	$\prod_i x_i$
OR	x_i for all i s.t. $(v_i,v_j) \in \vec{L}$	$1 - \prod_i (1 - x_i)$
K-out-of-N	x_i for all i s.t. $(v_i,v_j) \in \vec{L}$	$\begin{cases} 1 \text{ for } \sum_i x_i \geq k \\ 0 \text{ for } \sum_i x_i < k \end{cases}$
NOT	x_i	$1 - x_i$

Logic types included in T are AND, OR, NOT, and (at least) K-out-of-N. Other logic types are possible, but these are

$\overleftarrow{H} = (V,\overleftarrow{L})$ is similar to $\vec{H}$ but with its links directed in the opposite, i.e., "downward", direction; and $H = (V,L)$ is an undirected graph where L is $\vec{L}$ taken as an unordered set.

A further requirement for F to be a fault tree is that $\vec{H}$ be acyclic and possess a unique vertex $v_j > v_i$ for all $v_i \neq v_j$ in any acyclic ordering of V. In the graph $\vec{H}$, v_j corresponds to the *top event* e_j of F. The state of the top event is the system state $\Phi(\underline{x})$. The top event is dependent on intermediate and basic events and has out-degree zero. *Intermediate events* (or *logic events*) are any events with out-degrees and in-degrees both greater than zero. A *basic event* represents a system component, and has in-degree zero. The number of basic events is m. For now, it is assumed that all basic events are statistically independent, randomly occurring events.

For examples of fault tree event types consider a model of a complex tactical aircraft. This aircraft is composed of many basic components such as electrical generators, hydraulic pumps, flight control cables, and others for which failures can be assumed to be statistically independent. (For this aircraft assume that these components are independently powered.) The failures of these basic components are represented in a fault tree by basic events. Each of these components is a part of a greater system, i.e., electrical, hydraulic, and flight controls, respectively. Failures of these sub-systems become the intermediate events of the fault

is written $(0_i,\underline{x})$ where x_j is arbitrary for $j \neq i$. Setting $x_i = 1$ yields a state vector of $(1_i,\underline{x})$. Likewise if every basic component i is assigned a probability p_i, then $\underline{p} = (p_1,p_2,\ldots,p_m)$ is a vector of given probabilities. The probability of a system failure is given by $g(\underline{p})$, and system reliability is given by $h(\underline{p}) = 1 - g(\underline{p})$. If $p_i = 0$, then the vector $\underline{p}$ is denoted $(0_i,\underline{p})$ where p_j maintains its original value for all $j \neq i$. Similarly, setting $p_i = 1$ yields the vector $(1_i,\underline{p})$.

A binary system can be coherent or noncoherent. A system is *coherent* if Φ is monotonically increasing, and all components are relevant. Component i is *relevant* if $\Phi(1_i,x) \neq \Phi(0_i,\underline{x})$ for some value of the state vector $\underline{x}$. If the system state is constant in x_i for all values of $\underline{x}$, then component i is *irrelevant* [Ref. 16: p. 6].

Fault trees are the most commonly used models of binary systems. A fault tree is denoted $F = (E,\vec{L})$ where E is the set of *events*, and $\vec{L}$ is the set of *links*. An event $e_i \in E$ is a pair $e_i = (v_i,t_i)$ where $v_i \in V$ is the *event vertex* and $t_i \in T$ is the *event type* . Events are connected by links $\ell_{ij} = (v_i,v_j) \in \vec{L}$ where the ordered pair (v_i,v_j) denotes a directed link from e_i to e_j. Link ℓ_{ij} transmits the output from event e_i to the input of event e_j. The *out-degree* of e_i is the number of j such that $(v_i,v_j) \in \vec{L}$. The *in-degree* of e_j is the number of i such that $(v_i,v_j) \in \vec{L}$.

Three graphs derived from F will be useful. $\vec{H} = (V,\vec{L})$ is a directed graph with links directed "upward" as in F;

$$g(F) = p_i g(F|x_i = 1) + (1 - p_i) g(F|x_i = 0)$$

$$= p_i g(F_1) + (1 - p_i) g(F_0)$$

where F_1 is a fault tree derived from F given that e_i has occurred, and F_0 is a fault tree derived from F given that e_i has not occurred. If simple reductions completely reduce F_1 and F_0, then $g(F_1)$ and $g(F_0)$ are computed, and g(F) can then be computed. If not, events in F_1 and/or F_0 are selected for conditioning, and the procedure is repeated recursively until all failure probabilities can be computed through simple reductions or until conditioning implies $g(F_k|x_i) = 0$ or 1. Figure 1-3 shows a recursive decomposition of a fault tree F.

Recursive pivotal decomposition is further enhanced by identification of proper F-modules. If simple reductions fail to reduce F to a basic event e_j, then F may contain a non-trivial F-module F'. If F' is a simple, proper F-module with module top e_j, then pivotal decomposition may be applied to compute g(F'). F can then be replaced by $F - F' + e_j$ where t_j = BASIC, and p_j = g(F'). Using this *modularization* an exponential reduction in computation can be achieved, especially when repeated on recursively produced fault trees.

For small fault trees pivotal decomposition may be repeated quickly to compute g(F) for different values of $\underline{p}$ when necessary as in the constrained reliability maximization problem. For moderate to large-sized fault trees it may be

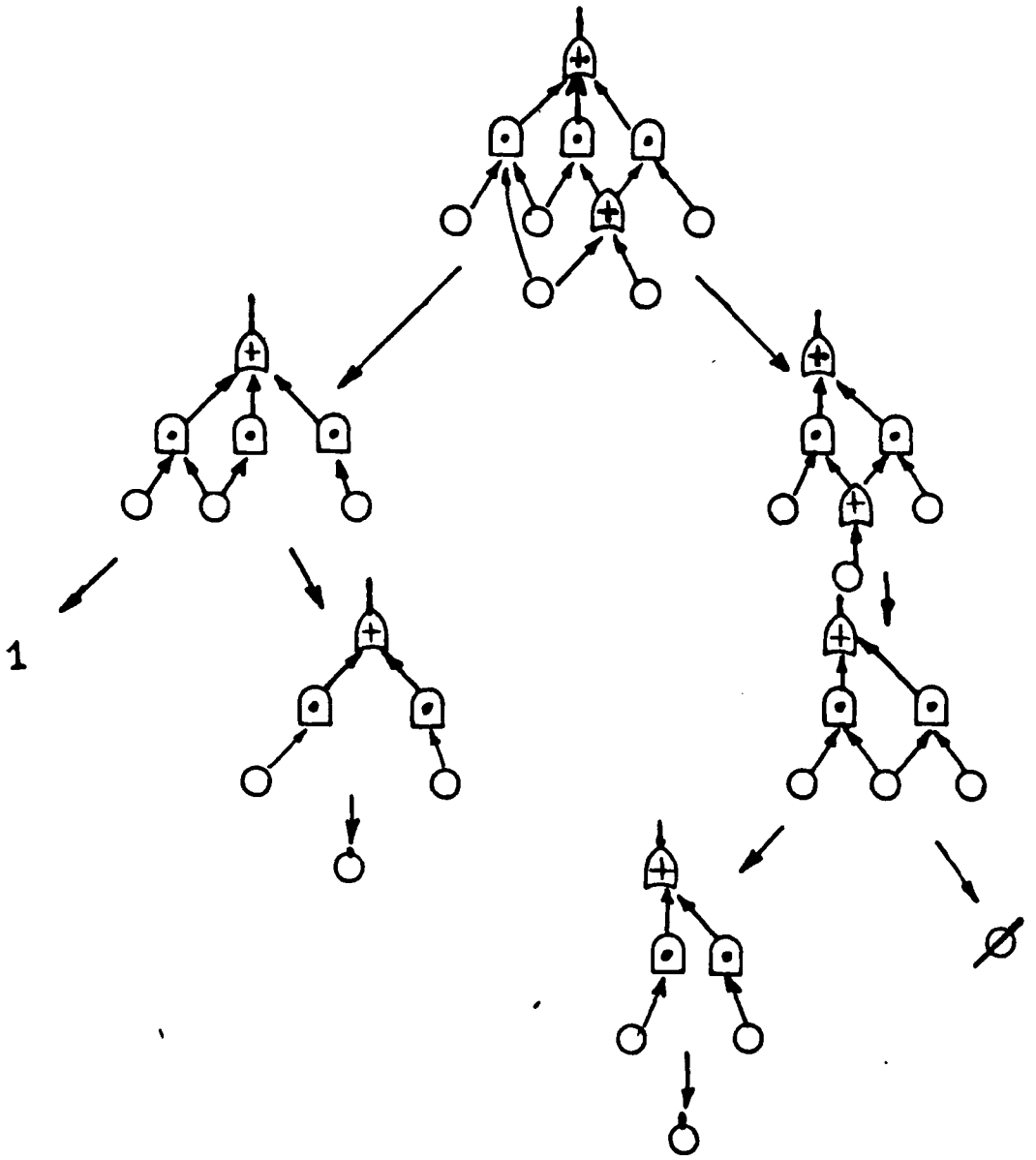


Figure 1-3 Pivotal Decomposition

possible to use pivotal decomposition to compute g(F) once in a reasonable amount of time but not multiple times. In this case it is possible to perform the simple reductions and pivotal decomposition on F without actually computing the probabilities in the process but, instead, saving each equation which would have been used to compute probabilities. When F has been completely reduced, the saved equations form an expression for g(p). This expression may now be used for rapid recomputations of g(p) without much of the work associated with the original fault tree algorithm.

Assuming that only replicated events are conditioned, time complexity for pivotal decomposition combined with simple reductions is $O(2^r|L|)$ for g(F). This is true since r is the greatest recursion level ever required to condition r replicated events. The time complexity of the expression g(p) will be identical to that of g(F) since g(p) will merely execute the computations produced in equational form by g(F). Actual time savings will, however, be realized by execution of the expression g(p) since building, storing, and reducing the structure of F is unnecessary. The space complexity of storing one fault tree is $O(|L|)$. For each step of conditioning, two different reductions must be performed on the same fault tree. To do this a copy of the current fault tree must be created and stored until it has been completely reduced. At the rth level of recursion, r copies of the fault tree are being stored. Consequently, the space

complexity for g(F) is $O(r|L|)$. Space complexity for storage of the expression g(p) is proportional to the time complexity of g(F).

Improvement of the actual time required to compute probabilities over existing methods will be attempted by taking advantage of fault tree structure, modularizing when possible, and exploring the use of some heuristics for intelligent conditioning.

II. ALGORITHMS

The main algorithm performs recursive pivotal decomposition combined with simple reductions on a fault tree. The main features of this algorithm and its supporting elements are presented in this chapter. F will be used to denote a fault tree with a probability assigned to each basic event. For notational simplicity let $|F|$ denote $|E|$ for $F = (E,\vec{L})$.

A. FAULTTREE

Faulttree is the primary algorithm used in this thesis. (See Figure 2-1.) The argument F is a simple F-module. In the first call to Faulttree, F is the original fault tree, but in all subsequent calls it is an F-module. (It will not necessarily be a proper F-module.) Faulttree receives F as an argument and returns the F-module top and its probability.

Sreduce performs all possible simple reductions on F, and if it reduces F to a basic event, Faulttree is finished. Otherwise, Faulttree will carry out further reductions using recursive pivotal decomposition. Findmodule searches for and returns a simple F-module F_0 in F. Also returned is e_j, the F-module top. If no proper, simple F-modules exist, $F_0 = F$. F_1, a copy of F_0, is produced so that two fault trees can be conditioned. At the end of the "if" block F_0 remains in F but as a basic event with probability given by the pivotal decomposition computation. The comments "{dummy 1}" and

```
algorithm Faulttree (F);
input:  A fault tree or simple F-module F with associated
        basic event probabilities
output:  The top event of F-module top e_j of F and its
         probability

begin
     While (|F| >1) do
           begin
           (F,p) ← Sreduce (F);
           if (|F| =1) then Return (F,p)
           else
                 begin
                 (F_0,e_j) ← Findmodule (F);
                 e_i ← Select (F_0) s.t. t_i = BASIC;
                 F_1 ← Copy (F_0);
                 (F_1,p_1) ← Condition (F_1,e_i,1);
                 if (|F_1| >1) then (e_j,p_1) ← Faulttree (F_1);
                 {dummy 1};
                 (F_0,p_0) ← Condition (F_0,e_i,0);
                 if (|F_0| >1) then (e_j,p_0) ← Faulttree (F_0);
                 p_j ← p_i p_1 + (1 - p_i) p_0;
                 {dummy 2};
                 t_j ← BASIC;
                 F ← F - F_0 + e_j;
                 end
           end;
     Return (F,p_j)
end;
```

Figure 2-1 Faulttree

"{dummy 2}" mark the spots where equation print statements can be inserted. This cycle of Sreduce, Findmodule, and pivotal decomposition on an F-module is continued until all F-modules are completely reduced.

Significant reductions in actual run times should be realized through the use of modularization. If a simple F-module can be located with s replicated events in a fault tree with r replicated events, then reduction methods can be applied to the F-module alone. After reducing

the F-module to a basic event, reductions continue on the remainder of the fault tree. Using these methods the original complexity factor of 2^r reduces to $2^s + 2^{r-s}$. By searching for F-modules and independently reducing each one, much time is saved.

Actual storage requirements can be expected to be well below the upper bound of $O(r|L|)$. Actual storage could only be this large if at each level of recursion during pivotal decomposition a copy of the original fault tree must be made. This cannot happen since at least one and frequently many events are removed at each conditioning step, thus gradually reducing the size of the fault tree as the level of recursion increases. Additionally, these operations are being performed on F-modules. Whenever a proper F-module is found, the size of the copy to be produced and stored is reduced.

1. Sreduce

This algorithm is sufficient for completely reducing F if it contains no replicated events. Sreduce is shown in Figure 2-2. Sreduce does a depth first search in $\overleftarrow{H}$ to find any event e_j with only unreplicated, basic events directly below. When such an e_j is found it is reduced to a basic event, $g_j(\underline{p})$ is computed, and all of the unreplicated, basic events can be disposed. As the algorithm backtracks to the top event, each F-module which has no replicated events is reduced to a single basic event. Upon leaving Sreduce, the only remaining, non-trivial F-modules in F contain replicated

```
algorithm Sreduce (F);
input:  A simple F-module F with associated basic event
        probabilities
output:  If fully reduced, the F-module top with its proba-
         bility.  Else, a partially reduced F

begin
      for all e_i ∈ E mark e_i "reducible";
      put module top of F on stack;
      while stack not empty do
            begin
            let e_j be the top element of the stack;
            For each untraversed ℓ_ji ∈ L_f do
                  begin
                  traverse ℓ_ji;
                  if e_i replicated then mark e_j "irreducible";
                  if e_i "reducible" and not BASIC then put e_i
                        on stack and let e_j ← e_i;
                  end;
            remove e_j from stack;
            if e_j "reducible" then
                  begin
                  p_j ← g_j(p) and mark e_j BASIC;
                  {dummy 3};
                  end;
            else mark top element of stack "irreducible";
            end;
      if (|F| = 1) then Return ({e_j,φ},p_j)
      else Return (F,undefined)
end.
```

Figure 2-2 Sreduce

events. "{dummy 3}" is a marker for inserting the print statements for $g_j(\underline{p})$. The time complexity of a call to Sreduce is $O(|L|)$.

2. Findmodule

This algorithm is a modification of Hopcroft's [Ref. 18:p. 185] depth first search for biconnected components. The search for biconnected components is effectively carried out in $H - V_u$ where H is derived from F, after performing all possible simple reductions, and V_u is the set of unreplicated

basic event vertices. As a result only F-modules containing at least one replicated event are found. Although Findmodule locates any such F-module, it returns only simple F-modules to Faulttree. If a located F-module is not simple, Findmodule will restructure it into a simple F-module with an F-module top or perform some other type of restructuring before returning it to Faulttree. These special restructuring procedures are described in Section C of this chapter. The time complexity of this routine is $O(|L|)$. Findmodule terminates as soon as an F-module is located.

3. Conditioning

Great reductions in computation can be obtained by selective conditioning in Faulttree. After locating an F-module F, a replicated basic event e_i is selected for conditioning. "Condition" is a procedure for making the associated reductions in F and is shown in Figure 2-3.

Condition also uses a depth first search, but from the replicated event outward, transmitting the effect of conditioning on the replicated event to other events in F. The search is conducted in $(E,\vec{L} \cup \overleftarrow{L})$ since other events both above and below an event to be removed may also be determined to be removable. Condition is configured for AND, OR, NOT, and 2-out-of-3 gates. However, addition of other types is easy. Any event to be removed from F is placed into the stack. When event e_i is removed from the stack, an outward search is conducted to find any other events to remove from

```
procedure condition (F,e_i,x);
input:  A simple F-module F, a basic event e_i to condition, the
state of the condition x
output:  If fully reduced, the F-module top and the state of
        the top event.  Else, a partially reduced F

begin
        put e_i on stack;
        while stack not empty do
              begin
              remove e_i from stack;
              for all e_j s.t. ℓ_ij ∈ L→ do
                      begin
                      if ((in-degree (e_j) =1) or ((t_j =OR)
                              and (x =1)) or ((t_j =AND) and
                              (x =0))) then
                              begin
                              if (e_j = module top of F) then
                              Return ({e_j,φ},x);
                              put e_j on stack;
                              if t_j = NOT then x ← 1-x;
                              end
                      else
                              begin
                              dispose ℓ_ij
                              if (t_j = 2-out-of-3) then
                                      if (x =1) then t_j = OR
                                      else t_j = AND;
                              end
                      end
              for all e_j s.t. ℓ_ij ∈ L← do
                      begin
                      if e_j unreplicated than put e_j on stack;
                      else dispose ℓ_ij
                      end
              if t_i = NOT then x ← 1-x;
              dispose e_i
              end
        Return (F,undefined)
end.
```

Figure 2-3 Condition

F. If events are not to be removed, their links to e_i are disposed. An event which is unreplicated and connected into e_i from below will be placed into the stack for removal from

F. The search looks upward from e_i to events e_j for all $\ell_{ij} \in \vec{L}$ and performs logic checks. For example, if the state variable x = 1, and t_j = OR, then e_j is placed into the stack. NOT events change x to 1-x. 2-out-of-3 events are transformed into AND or OR events depending on the current value of x. If the search reaches the F-module top of F, F is returned as a basic event with p = 0 or 1. If the F-module top is not reached in the search, F is returned, partially reduced from the form of the original argument. The time complexity of this search is $O(|L|)$.

4. The Select Procedure

Printed equations can be used for multiple executions of top event probability computations. In this case, conditioning on basic events so as to minimize the number of equations written will enhance efficiency even if the running time of Faulttree is increased. One way to do this is to develop a "good" procedure for selecting a replicated event e_i to condition. Various heuristics are possible such as choosing the e_i with greatest out-degree or the greatest or least distance from the cut event. These qualities can be determined with a routine in $O(|L|)$ time. A theoretically stronger heuristic is

$$\min_{e_i \in E_R} (\max_{j \in J} |R_j|)$$

where E_R is the set of replicated basic events in F, J the set of biconnected components remaining in the two fault trees

after conditioning e_i, and R_j the set of replicated events in biconnected component j. A "select procedure" was implemented to perform this. The procedure conditions on e_i using the algorithm Condition and creates the two fault trees F_{0i} and F_{1i}. Next, a depth first search is conducted in F_{0i} and F_{1i}, counting the replicated events $|R_j|$ in each biconnected component j. The biconnected components of $H|x_i$ correspond to prime F-modules in $F|x_i$ and to components which will become prime F-modules after recursively reducing current F-modules. The maximum $|R_j|$ found in the two depth first searches of F_{0i} and F_{1i} is saved for each e_i. These steps are repeated for all $e_i \in E_R$, and that e_i that minimizes $|R_j|$ is chosen for conditioning. This heuristic myopically minimizes the upper bound factor $\max_{j \in J} 2^{|R_j|}$ over all F-modules and components which will become F-modules.

B. FAILURE PROBABILITY FUNCTION

A second version of Faulttree was modified to print a set of equations which represent the failure probability function $g(\underline{p})$. All algorithms remain the same except that probability computations are replaced with "print statements." These statements are inserted in Faulttree and Sreduce in the spots marked by "dummy" comments. Since numerical computations are correctly ordered, so must be the printing of the equations. Faulttree must create an extra variable and print an equation for storing the probability of the top event for F_1 since its normal storage space will be overwritten

by the probability of the top event for F_0. "Dummy 1" is replaced by a statement to print the equation which stores the conditional probability in this extra variable. The pivotal decomposition equation is printed by a statement in the line marked by "Dummy 2." Table 2 shows the statements to be substituted for "Dummy 1" and "Dummy 2" in Faulttree.

TABLE 2

Printing Equations

Block	Statement
Dummy 1	XP[j]:=P[j];
Dummy 2	P[j]:=P[i]*XP[j] + (1 -P[i])*P[j];

In the table, j is the index of the F-module top while i is the index of the event conditioned. In Sreduce "Dummy 3" is replaced by a statement giving the equation for $g_j(\underline{p})$. In this case, the printed statement assigns a value to "P[j]" by writing on the right hand side of the equation a function of the basic, unreplicated events. The function to be printed is dependent on t_j and is taken from Table 1-2.

Although execution of $g(\underline{p})$ is $O(2^r|L|)$ just like the computation of g(F), actual time should be much less. Storage is also $O(2^r|L|)$, an increase from the storage required for direct computation of g(F). Storage of variables in $g(\underline{p})$ is only O(r+N). Recall that r is the number of replicated

events which also yields the maximum level of recursion, and N is the total number of events in the fault tree. The r term results from creating an extra variable at each level of recursion to store conditional, top event probabilities. The number of equations written is directly related to the time complexity of computing g(F). The total storage requirements are therefore of the same order as the time complexity of Faulttree, i.e., exponential. In practice, it is hoped that the number of equations produced is small enough that they can be evaluated efficiently.

C. ENHANCEMENTS

Proper application of Faulttree requires that F, whether an F-module or a fault tree, possess the properties of a fault tree. A general F-module does not necessarily meet this requirement while a simple F-module always does. Two enhancements to Findmodule, "event splitting" and "reconfiguration," are methods of dealing with non-simple F-modules. Event splitting can be applied to an F-module with a cut event of type AND or OR while reconfiguration is used for a cut event of type 2-out-of-3. The last enhancement reduces the number of equations produced by handling some simple reductions implicitly.

1. Event Splitting

When Findmodule locates a simple F-module F' with its F-module top e_k, F' and e_k are returned immediately to Faulttree. If F' is not simple, and t_k = AND or OR, then *event*

be compiled and executed. FTE reads from the same data file that Faulttree reads but only extracts the values for $\underline{p}$ in the process. FTE outputs the probability of the top event but can be usefully configured to compute event importance or perform other computations which require $g(\underline{p})$.

D. PROGRAM TESTING

Faulttree was tested on four fault trees, two of which are hypothetical, "Exampl" and "Examp2," and two of which are actual models of systems used in practice. One system, "Aircraft," represents the combat attrition of a single aircraft while another, "Nuke," represents a nuclear reactor accident. Input data files were created for the four fault trees, and Faulttree was executed for each to directly compute $g(F)$. Faulttree was again executed for each data file to produce four versions of FTE. Descriptions of the fault trees and data from test runs are given in Table 3-3.

TABLE 3-3

Test Runs

	Exampl	Examp2	Aircraft	Nuke
events	64	79	105	339
rep. events	7	15	4	59
CPU time	0.001	0.371	0.001	
events stored	112	330	178	2586
FTE equations	36	102	51	153,733
FTE CPU time	0.000	0.033	0.000	

```
8                9
1                3               3
2                3               4
2                2               2
5                6
3                2               2
6                7
4                2               2
7                9
5                1               0
0.001
6                1               0
0.001
7                1               0
0.002
9                1               0
0.002
```

Figure 3-2 Sample Input Data File

other dummy events which may have been created during event splitting or reconfiguration. The secondary array is used in pivotal decomposition to store the conditional probability for an event while a probability is computed for the same event given the opposite condition. The size of this array is no greater than the deepest recursion level of Faulttree. The heading is printed after TEP since array sizes for FTE are not available in Faulttree until F has been completely reduced. FTE-main is a routine which reads $\underline{p}$ from the input data file and invokes TEP to compute $g(\underline{p})$. When FTE-heading, TEP, and FTE-main are combined to create FTE, FTE is ready to

types to be represented in the fault tree. These blocks will make it easy to modify this program for use of other specific event types by insertion of the proper blocks of code.

C. INPUT AND OUTPUT

The input for Faulttree is a data file describing F. The first line of the data gives integer values for the number of events and the highest event identification number. The remainder of the file gives the detailed event data. Each event occupies two lines of the file. The first line gives three integers: event identification, event type, and number of events directly below. The second line lists the events below by identification or gives event probability for a basic event. Figure 3-2 is a sample input data file.

Faulttree outputs either the system failure probability or a set of equations forming an expression for g(p). This expression is in the form of a three part Pascal program "FTE" (Fault Tree Expression). Faulttree prints the heading "FTE-heading" and a subroutine "TEP" (Top Event Probability) for FTE while the main program "FTE-main" is kept permanently on file. TEP contains the equations which are printed by Faulttree in reducing F. It is configured to receive the argument p from FTE-main and return g(p). TEP and FTE-main use variables and arrays declared in FTE-heading. FTE-heading is printed by Faulttree after reductions on F are complete. Two arrays are declared in the heading. The primary array has a component for each event in F plus any

minimize storage and time concurrently, two arrays were created at the beginning of the program, one to store event records and the other to store link records. All records needed for the entire program are created and placed into these arrays. Records are re-used from these arrays by saving the index of the last record currently in use. Whenever a new record is needed it can be taken from the next point in the array beyond the index. Prior to making a copy of F in Faulttree, the current value of the index is saved in another variable. This copy of F is then produced, increasing the index value. The copy is passed as an argument to Faulttree. Upon return from Faulttree the copy is no longer needed, and the index can be reset to its prior value. Meanwhile, as reductions are made in Sreduce and Condition, the program effectively "burns bridges" by setting pointers to nil where events beyond these pointers are to be removed.

F-modules are dealt with directly without being disconnected or removed from F. Faulttree and its subroutines pass arguments in the form of F-modules. This is actually accomplished in the program by passing a variable containing a pointer to the F-module top. The subroutines treat the F-module as a fault tree by never searching above the F-module top.

In the subroutines Sreduce and Condition, some sections of the code were written in block format. That is, sections of code can be removed or inserted depending on the event

Because of this data structure, it is easy to change the fault tree during a search. Reductions can be made by deleting a link and reconnecting the links on either end of it, or by setting pointers to "nil." Event types or identifications can be changed or newly computed basic event probabilities stored. (Probabilities only need to be stored in event records when direct computation of system failure probability is performed.)

B. PROGRAMMING

Another feature of Pascal which was useful was its ability to call procedures recursively. This capability was used for pivotal decomposition so that recursive calls could be made in the program Faulttree until F was reduced completely. Although recursion could have been used in some subroutines, it uses more time and storage [Ref. 29: p. 300] than non-recursion and therefore was used only for pivotal decomposition.

In Pascal, records may be created and destroyed over the course of a program so that storage is only used when needed. This can be accomplished by use of the embedded functions "new" and "dispose." Some conservation of storage must be utilized in Faulttree when solving any large problems. Using new when making a copy of F and dispose during the reductions on F is one way to conserve storage. This way is time consuming, however, since invoking new, slows the program, and extra searches which would otherwise be unnecessary are required to reach all events and links for disposals. To

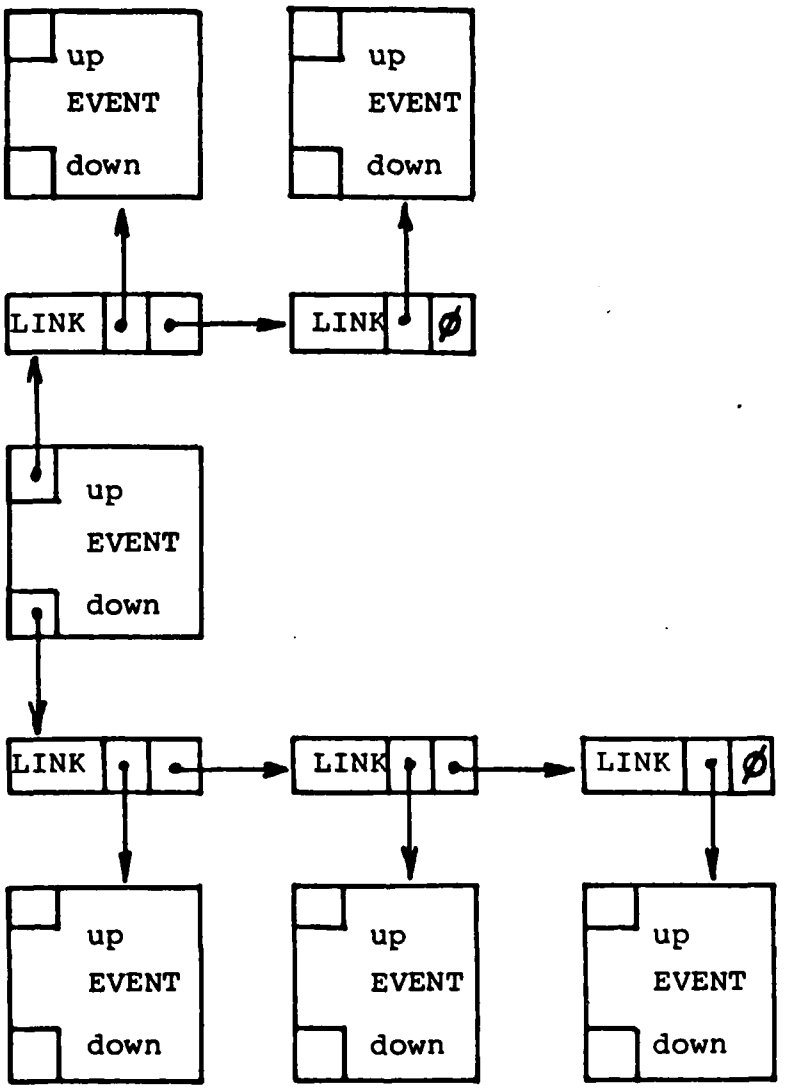


Figure 3-1 Linking of Events

TABLE 3-1

Event Record

Variable	Data Type
identity	integer
type	integer
up pointer	pointer to link record
down pointer	pointer to link record
probability (optional)	real

TABLE 3-2

Link Record

Variable	Data Type
event pointer	pointer to event record
next link	pointer to link record

link by the variable *next link*. Every link in the data structure points to an event record via the variable *event pointer*. The event records pointed to represent the e_j which are linked from e_i by $\{\ell_{ij}: \ell_{ij} \in \vec{L}\}$. The down pointer points to the first link of a set of links equal in number to the in-degree of e_i. These links are joined to one another in the same way, and each points to an event record representing an e_k which is linked into e_i by $\{\ell_{ki}: \ell_{ki} \in \vec{L}\}$. Figure 3-1 gives a visual representation of this structure.

convenient in Condition. This allowed a depth first search to remove events by starting at the basic event being conditioned rather than beginning the search at the top event which would require more time.

Because pivotal decomposition and other algorithms used deal with dynamic fault trees by restructuring and making reductions, the internal data structure for the computer program should facilitate changes to F. This facilitation was accomplished by the use of linked lists to represent the events and links of F. Two features available in Pascal which were useful for storing these linked lists are "records" and "pointers." Two types of records were designated *event records* and *link records*. A record allows the storage of different data types within a single entity. Integers, reals, arrays, and other types can be stored simultaneously in each record. Two pointer types were designated *event record pointers* and *link record pointers*. The pointers were used to connect events and links in the computer representation of the fault tree, and were also used to move from one event to another during searches through F.

Tables 3-1 and 3-2 list the information stored in event and link records.

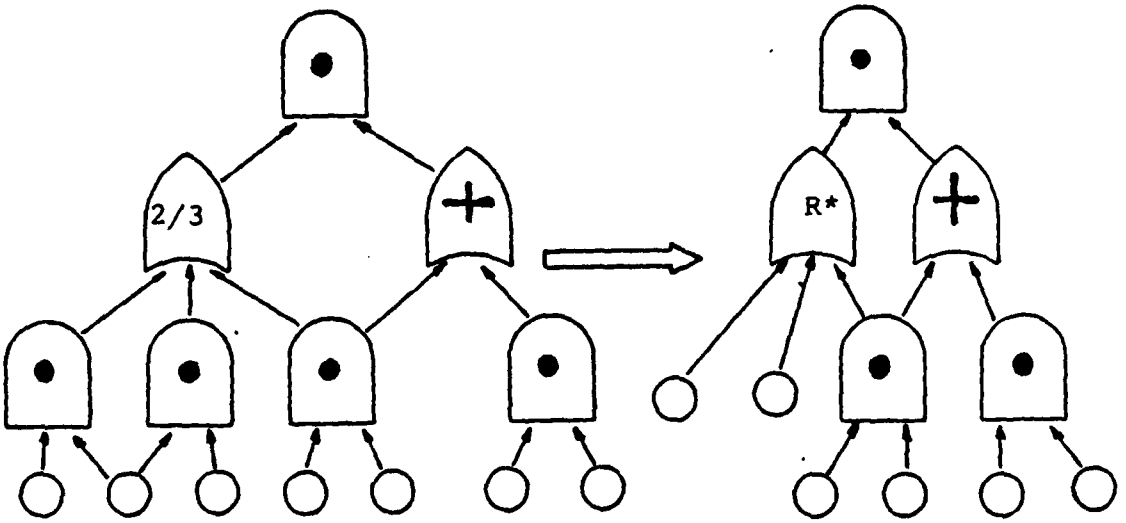
An event record is created for each e_i in F. Each event record has an *up pointer* and a *down pointer*. The up pointer points to the first link of a set of links equal in number to the out-degree of e_i. Each link is connected to the next

III. IMPLEMENTATION AND COMPUTATIONAL RESULTS

The computer codes for all programs are written in Berkeley 3.0 Pascal to take advantage of the recursive feature of this language. All tests on these programs were conducted on a VAX 11/780 computer under the Berkeley 4.0 Unix operating system. The main algorithm of the previous chapter was transformed into the dual purpose program "Faulttree" which can be used to directly compute g(F) or produce a subroutine containing the equations for g(p).

A. DATA STRUCTURES

The data structure used to represent the fault tree is effectively $(E,\vec{L} \cup \overleftarrow{L})$. That is, both upward and downward pointing links are maintained out of each event. Some storage could have been saved using only $(E,\overleftarrow{L})$ and creating $\vec{L}$ when needed, but this would have greatly increased the complexity of the program. Maintaining both $\vec{L}$ and $\overleftarrow{L}$ allowed flexibility for the various types of searches conducted in F during reductions and other operations. A depth first search using $(V,\overleftarrow{L})$ is performed in the simple reduction subroutine "Sreduce," a depth first search using $(V,\vec{L} \cup \overleftarrow{L})$ is performed in the subroutine "Condition," and a depth first search using (V,L) is performed in the subroutine "Findmodule" where $(\vec{L} \cup \overleftarrow{L})$ is used to simulate L. The use of $(V,\vec{L})$ was especially



*R denotes reconfigured event

Figure 2-5 Reconfiguration

$F = F - F' + e_k + e_{\ell_1} + e_{\ell_2}$. Future computation for $g_k(\underline{p})$ will use

$$g_k(\underline{p}) = p_{\ell_1} + (1-p_{\ell_1})p_{\ell_2}g_{i_3}(\underline{p})$$

Figure 2-5 exhibits the resulting structural modification to the fault tree.

3. Replacement

Another enhancement made was a change to Sreduce. Instead of computing $g_j(\underline{p})$ for a logic gate e_j with only a single basic event e_i below, e_j can simply be replaced by e_i, i.e., $e_j \leftarrow e_i$, $p_j \leftarrow p_i$, and dispose e_i. This is especially helpful in forming the expression for $g(\underline{p})$ since one equation is eliminated each time this *replacement* is made.

basic events) with cut vertex v_k. If F' is not simple, then since $v_k \in H'$ exactly two of the $e_i \in E'$, leaving one $e_i \notin E'$. Let the two events in E' be denoted e_{i_1} and e_{i_2} and let $e_i \notin E'$ be denoted e_{i_3}. The possible states of the pair $\{e_{i_1}, e_{i_2}\}$ are (1,1), (1,0), (0,1), and (0,0) of which (1,0) and (0,1) are indistinguishable to e_k. F' will be replaced by e_k and two basic events which will give an equivalent representation of the probability information stored in F'. To compute the needed probabilities a new top event e_j independent of F is created. The links $\ell_{i_1 k}$ and $\ell_{i_2 k}$ are removed, disconnecting $F' - e_k$ from F. Links $\ell_{i_1 j}$ and $\ell_{i_2 j}$ are formed to connect $F' - e_k$ to e_j via the pair $\{e_{i_1}, e_{i_2}\}$ forming the new fault tree $\hat{F}$. For $e_j \in \hat{E}$ let t_j = AND and call Faulttree to obtain

$$P(1,1) = (g_j(\underline{p}) \mid t_j = \text{AND})$$

Let t_j = OR and call Faulttree to obtain

$$P((1,1) \cup (1,0) \cup (0,1)) = (g_j(\underline{p}) \mid t_j = \text{OR})$$

e_k is given a new event type which denotes a "reconfigured" event with nonhomogeneous inputs. Two new basic events e_{ℓ_1} and e_{ℓ_2} are attached into e_k by $\ell_{\ell_1 k}, \ell_{\ell_2 k} \in \vec{L}$. $p_{\ell_1} = P(1,1)$ while $p_{\ell_2} = P((1,0) \cup (0,1))$ given by

$$P((1,0) \cup (0,1)) = P((1,1) \cup (1,0) \cup (0,1)) - P(1,1)$$

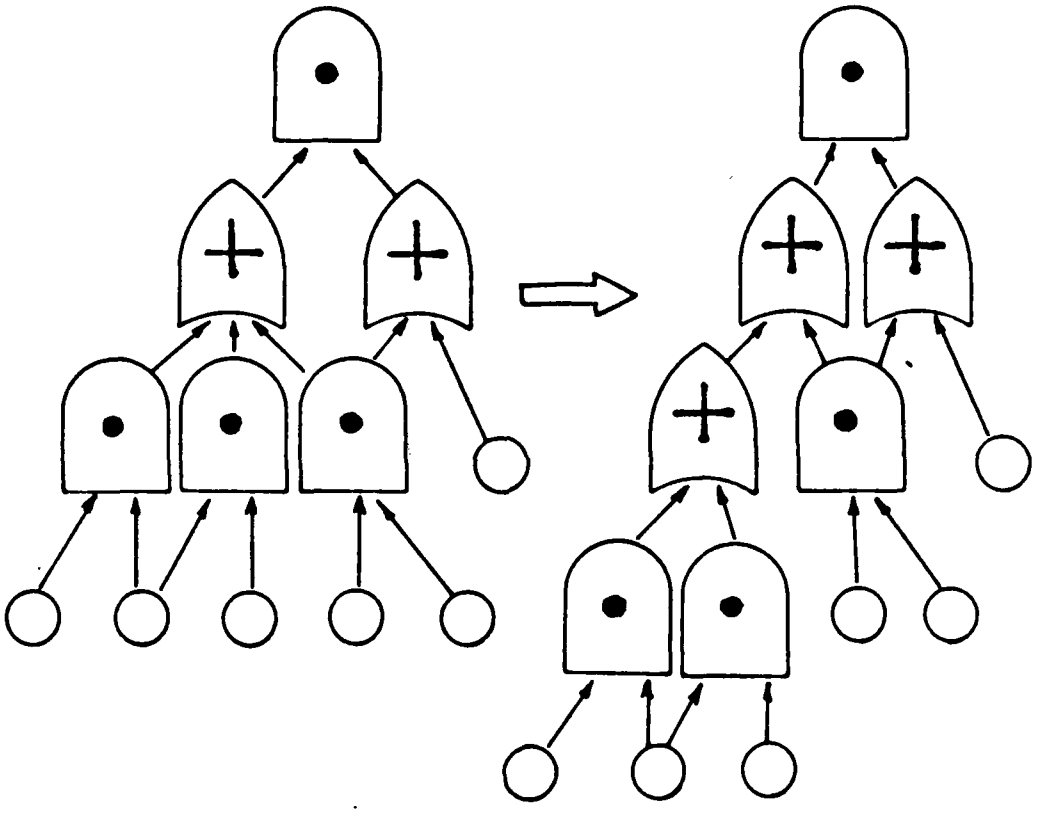


Figure 2-4 Event Splitting

splitting may be applied. Since F' is not simple, $e_i \notin E'$ must be linked into e_k by $\ell_{ik} \notin \vec{L}'$. "Split" e_k into two events e_{k_1} and e_{k_2} such that $t_{k_1} = t_{k_2} = t_k$, $\{\ell_{ik_1}\} = \{\ell_{ik}: \ell_{ik} \in \vec{L}'\}$, and $\{\ell_{ik_2}\} = \{\ell_{ik}: \ell_{ik} \notin \vec{L}'\} + \ell_{k_1k_2}$. A simple F-module $\hat{F}$ is formed by $\hat{F} = F' - e_k + e_{k_1}$ where e_{k_1} is the F-module top. Findmodule returns $\hat{F}$ to Faulttree. Event splitting works since

$$x_1 \cap x_2 \cap, \ldots, \cap x_n = x_0 \cap (x_{k+1} \cap x_{k+2} \cap, \ldots, \cap x_n)$$

for

$$x_0 = x_1 \cap x_2 \cap, \ldots, \cap x_k$$

and since

$$x_1 \cup x_2 \cup, \ldots, \cup x_n = x_0 \cup (x_{k+1} \cup x_{k+2} \cup, \ldots, \cup x_n)$$

for

$$x_0 = x_1 \cup x_2 \cup, \ldots, \cup x_k$$

Figure 2-4 shows the structural changes made to the fault tree by event splitting.

2. Reconfiguration

For a cut event e_k of F-module F' with t_k = 2-out-of-3, three events e_i are linked into the cut event e_k of F'. H' is a biconnected component of H (ignoring unreplicated

Nuke, described in the table, is actually a revised version of the original data. The original data contained 345 events of which 65 were replicated. Further explanation of the modification of this data is given below.

The table gives CPU time in seconds. All CPU times reported in this thesis exclude time required for input/output. As a measure of storage the maximum number of event records needed to compute each problem is included as "events stored." Also, the number of equations printed into FTE is listed. For all of the fault trees except Nuke, FTE was successfully compiled and executed, computing the system failure probability in less time than required by Faulttree. The times for execution of FTE are given in Table 3-3 in the row denoted FTE CPU time.

Initial tests on Nuke were made using the original data file. The first solution attempt for direct computation of g(F) required more than five hours of clock time for Faulttree during a low utilization period on the VAX. Exact CPU time was not determined. When Faulttree was reexecuted to produce FTE, over 600,000 equations were printed into TEP. This subroutine was too large to be compiled. Further tests were conducted with this data alone with the objective of reducing the number of equations being printed. First, data was generated from Faulttree to see what size modules were being located and to determine the extent of the reductions being accomplished by pivotal decomposition. It was found

that after the first call to Sreduce, which removed only six events, the fault tree was a prime F-module with all 65 replicated events and 339 of the original events still intact.

Several successful and unsuccessful techniques were implemented for reducing the size of TEP. The replacement procedure was implemented in Sreduce, and output was reduced to about 425,000 lines. Up to this point, replicated events for conditioning had been selected randomly. This worked satisfactorily for small problems. Various heuristics for choosing replicated events e_i for conditioning were tested with Nuke. Three of these which required linear time complexity were choosing e_i with (a) the greatest out-degree, (b) the least distance in links from the top event, and (c) the greatest distance in links from the top event. Implementation of heuristic (a) reduced output to about 417,000 lines while (b) and (c) increased the amount of output. Next, the reconfiguration procedure was developed, and it reduced the output to about 415,000 lines. The heuristic for computing $\min_{e_i \in E_R} (\max_{j \in J} |R_j|)$ for all replicated basic events was then added. This enhancement reduced output to 225,000 lines of output. Finally a crude graphical representation of the fault tree was produced with the hope that some visual clue might aid selective conditioning. Two sets of four replicated basic events were found. Every event in each set was linked to the same two intermediate events of four intermediate events total. The eight basic events were

replaced in the input data file by two basic events after hand-computing probabilities for the two new basic events based on the union of the four events each one replaced. With this revised data, Faulttree produced only 153,733 equations.

IV. RESULTS AND CONCLUSIONS

Pivotal decomposition has been shown to be a good method for computing system failure probabilities in fault trees, at least for the problems analyzed here. The basic algorithm in conjunction with several enhancements has computed exact probability for a fairly large fault tree having 345 events with 65 of them replicated. Some of these enhancements were key factors in reducing the amount of computation required by the basic algorithm. If other methods of reducing this computation can be applied to the computer code developed in this thesis, this program will be capable of being used as a tool in analysis of even larger fault trees.

A. FINDINGS

Space complexity was not a limiting factor in solving any of these fault trees. The greatest use of storage occurred in computing g(F) for Nuke. The total number of event records created was less than eight times the amount needed to store the original fault tree alone. Since the recursion level was noted to exceed 43 at some points during execution, the factor of eight is less than might be expected. The system storage requirements for a high recursion level such as this are probably more significant than the storage of problem data. The greatest limiting factor for computing probabilities in large fault trees is the time complexity

$O(2^r|L|)$ which also gives the complexity for the length of TEP. In this complexity figure, the factor $|L|$ is insignificant. Efforts to reduce complexity must be directed toward the factor 2^r. The fault tree aspects which most influence this factor are the number of replicated events and the structural characteristics of the fault tree which allow or make difficult its modularization. Even a fault tree with a large r value should not be difficult for Faulttree to reduce if it has one of the following three properties: (a) No prime F-modules contain a large r, (b) r is greatly reduced after a few recursions of pivotal decomposition, or (c) non-complex F-modules (low r per F-module) begin to form after a few recursions of pivotal decomposition.

Faulttree and FTE have been shown to be useful for the three fault trees Exampl, Examp2, and Aircraft. Faulttree computed top event probability in a fraction of a second, and FTE used less time. As a test of applicability FTE-main was modified to compute Birnbaum importances for every basic event in a given fault tree. For each basic event this requires two computations of top event probability by TEP. The number of basic events and time in seconds to compute all their Birnbaum importances are shown in Table 4 for the three fault trees.

Examp2 is the most complex fault tree of the three as evidenced by comparing the numbers of replicated events and the CPU time required by g(F) for the three fault trees.

TABLE 4

Time to Compute Birnbaum Importances for All Basic Events

	Examp1	Examp2	Aircraft
basic events	34	36	61
CPU time	0.017	0.067	0.017

(See Table 3-3.) For Examp2, 72 computations of g(p) are made in about one-fifth of the amount of time required to compute g(F) directly.

FTE was unable to be tested on Nuke due to the size of the subroutine TEP produced by Faulttree. Direct computation of g(F) was successful, although it required much CPU time. The structure of this fault tree impeded the formation of proper F-modules after reductions from conditioning. In fact, following as many as five conditionings, no replicated events are eliminated except for the one conditioned, and no proper F-modules are created.

Although the version of TEP produced with Nuke is presently too large to compile and use, it was reduced in size by more than 75 percent from the first execution by several innovations which were discussed in Chapter III. The large reductions accomplished by the implementation of replacement show that there are many instances of intermediate events with only one unreplicated basic event below. Although this technique was trivially easy to use, it was highly significant in

reducing the size of TEP. The addition of reconfiguration to the program reduced TEP by less than one percent. This may seem insignificant; however, Nuke only has three 2-out-of-3 events. Of the three, one is reduced and disposed in the first call to Sreduce leaving only two in the fault tree for pivotal decomposition. Before implementing reconfiguration if the cut vertex of an F-module $F' \subseteq F$ was a 2-out-of-3 event, and one of the events connected into the cut vertex was not in F', then F' could not be used but instead served to complicate F and impede the computational process. It is believed that reconfiguration will significantly reduce the actual complexity of any fault tree with many 2-out-of-3 events.

The heuristic for selecting events to condition reduced the size of TEP by 45 percent. Although this heuristic results in increased time complexity for Faulttree, the great reduction in the size of TEP is worthwhile.

It is hoped that pivotal decomposition, combined with techniques discussed in this thesis and other techniques, will be useful in the analysis of large fault trees. More methods of making reductions and locating F-modules exist. However, time limitations preclude their application in this thesis. It is believed that the addition of some of these other methods to Faulttree would greatly increase the range of solvable problems.

B. SUGGESTED FURTHER RESEARCH

There are many further enhancements to the pivotal decomposition method of fault tree probability computation which could increase the usability of Faulttree.

This thesis used the 2-out-of-3 event to demonstrate how techniques for K-out-of-N events can be applied. Specific K-out-of-N events would be easy to implement in the existing program. Other possible enhancements could be the addition of algorithms to compute probabilities of a general K-out-of-N event during simple reductions. To be of any practical use, this algorithm must handle a set of input events with unequal probabilities. In conjunction with this there should be a method for reconfiguration of an F-module with a general K-out-of-N cut vertex.

There exist other methods of locating F-modules and generalizations of F-modules that can locate more useful structures which are overlooked by the depth first search method applied here. The method used in this thesis only locates an F-module which is attached to the fault tree by a cut vertex. Wood [Ref. 30] uses a search for tri-connected components in solving network reliability problems, and this method could be used to locate F-modules connected by separating pairs. Applied to this algorithm for fault trees, additional F-modules would be located which aren't being located by the present method. For example, the two sets of four replicated events which were reduced to two

replicated events by hand computation were both examples of tri-connected components which would have been detected and reduced as F-modules thus reducing the overall problem complexity.

It may be sufficient in many applications to compute g(F) approximately or to obtain upper and lower bounds on g(F). Corynen [Ref. 26] is able to solve large problems and obtains accurate bounds without considering all branches of the backtrack search structure. In Faulttree, lower bounding could be accomplished by saving the product P_k of the probabilities of all events which have been conditioned up to recursion level k. The most recent value of P_k for all k is saved so that it is available during backtracking and further recursion. When $P_k < \delta$ for some small $\delta > 0$, then further recursions are unnecessary since the term in the pivotal decomposition algorithm is approaching zero. The algorithm can backtrack, and the term associated with the current recursion need not be added into the computation of g(F). If used, this method removes Faulttree from the realm of exact methods, and it might be risky to use the resulting expression for computation of system failure probability when the p_i values vary over a wide range.

There is surely a lower bound on the number of equations which must be written to give an expression for $g(\underline{p})$ for a particular fault tree. For some large fault trees the lower bound will be too large thus preventing the compilation of

the subroutine TEP. In this case TEP can be subdivided into multiple subroutines to be compiled separately and linked for execution.

By including some of these suggested additions to the work already accomplished, it is believed that Faulttree and FTE will be useful tools for fault tree analysis.

LIST OF REFERENCES

1. Chatterjee, P., Fault Tree Analysis: Reliability Theory and Systems Safety Analysis, Operations Research Center, University of California, Berkeley, 1974.

2. Arnborg, S., "Reduced State Enumeration--Another Algorithm for Reliability Evaluation," IEEE Transactions on Reliability, R-27, 1978, pp. 101-105.

3. Atkinson, J.H., "The Set Equation Transformation System Used in Analysis of a Typical Naval Weapon System," Reliability and Fault Tree Analysis, editors, Barlow, R.E., Fussell, J.B., Singpurwalla, N.D., SIAM, 1975, pp. 187-202.

4. Ball, R.E., The Fundamentals of Aircraft Combat Survivability: Analysis and Design, To be published by AIAA, 1985.

5. Cummings, G.E., "Application of the Fault Tree Technique to a Nuclear Reactor Containment System," Reliability and Fault Tree Analysis, editors, Barlow, R.E., Fussell, J.B., Singpurwalla, N.D., SIAM, 1975, pp. 805-825.

6. Hannum, W.H., Gavigan, F.X., Emon, D.E., "Reliability and Safety Analysis Methodology in the Nuclear Programs of ERDA," IEEE Transactions on Reliability, R-25, 1976, pp. 140-146.

7. Vesely, W.E., "Reliability Quantification Techniques Used in the Rasmussen Study," Reliability and Fault Tree Analysis, editors, Barlow, R.E., Fussell, J.B., Singpurwalla, N.D., SIAM, 1975, pp. 775-803.

8. Thatcher, R.M., "Evaluating the Effects of Seismic Events on Systems in a Nuclear Facility Using the $\Sigma\Pi$ Method," Lawrence Livermore Laboratory, University of California, Livermore, 1983.

9. Fussell, J.B., "Computer Aided Fault Tree Construction for Electrical Systems," Reliability and Fault Tree Analysis, editors, Barlow, R.E., Fussell, J.B., Singpurwalla, N.D., SIAM, 1975, pp. 37-56.

10. Goldberg, J., "A Survey of the Design and Analysis of Fault-Tolerant Computers," Reliability and Fault Tree Analysis, editors, Barlow, R.E., Fussell, J.B., Singpurwalla, N.D., SIAM, 1975, pp. 687-731.

11. Powers, G.J., Tompkins, F.C., Lapp, S.A., "A Safety Simulation Language for Chemical Processes: A Procedure for Fault Tree Analysis," Reliability and Fault Tree Analysis, editors, Barlow, R.E., Fussell, J.B., Singpurwalla, N.D., SIAM, 1975, pp. 57-75.

12. Lambert, H.E., "Measures of Importance of Events and Cut Sets in Fault Trees," Reliability and Fault Tree Analysis, editors, Barlow, R.E., Fussell, J.B., Singpurwalla, N.D., SIAM, 1975, pp. 77-100.

13. Mizukami, K., "Optimum Redundancy for Maximum System Reliability by the Method of Convex and Integer Programming," Operations Research, V-16, 1968, pp. 392-406.

14. Derman, C., Lieberman, G.J., Ross, S.M., "Optimal Allocation of Resources in Systems," Reliability and Fault Tree Analysis, editors, Barlow, R.E., Fussell, J.B., Singpurwalla, N.D., SIAM, 1975, pp. 307-324.

15. Luenberger, D.G., Linear and Nonlinear Programming, Addison-Wesley Publishing Company, Inc., 1984.

16. Barlow, R.E., Proschan, F., Statistical Theory of Reliability and Life Testing, To Begin With, 1981.

17. Agrawal, A., Barlow, R.E., "A Survey of Network Reliability," Operations Research, V-32, 1984, pp. 478-492.

18. Aho, A.V., Hopcroft, J.E., Ullman, J.D., The Design and Analysis of Computer Algorithms, Addison-Wesley Publishing Company, 1974.

19. Ross, S.M., "On the Calculation of Asymptotic System Reliability Characteristics," Reliability and Fault Tree Analysis, editors, Barlow, R.E., Fussell, J.B., Singpurwalla, N.D., SIAM, 1975, pp. 331-350.

20. Hwang, F.K., "Fast Solutions for Consecutive-k-out-of-n: F System," IEEE Transactions on Reliability, R-31, 1982, pp. 447-448.

21. Shanthikumar, J.G., "Recursive Algorithm to Evaluate the Reliability of a Consecutive-k-out-of-n: F System," IEEE Transactions on Reliability, R-31, 1982, pp. 442-443.

22. Rosenthal, A., "A Computer Scientist Looks at Reliability Computations," Reliability and Fault Tree Analysis, editors, Barlow, R.E., Fussell, J.B., Singpurwalla, N.D., SIAM, 1975, pp. 133-152.

23. Garey, M.R., Johnson, D.S., Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman and Company, 1979.

24. Feller, W., An Introduction to Probability Theory and Its Applications, John Wiley and Sons, 1968.

25. Nakazawa, H., "A Decomposition Method for Computing System Reliability by a Boolean Expression," IEEE Transactions on Reliability, R-26, 1977, pp. 250-252.

26. Abraham, J.A., "An Improved Algorithm for Network Reliability," IEEE Transactions on Reliability, R-28, 1979, pp. 58-61.

27. Corynen, G.C., "Evaluating the Reponse of Complex Systems to Environmental Threats: The ΣΠ Method," Lawrence Livermore Laboratory, University of California, Livermore, 1983.

28. Feo, T.A., "PAFT F77: Program for the Analysis of Fault Trees," Operations Research Center, University of California, Berkeley, 1983.

29. Schneider, G.M., Weingart, S.W., Perlman, D.M., Programming and Problem Solving with Pascal, John Wiley & Sons, Inc., 1982.

30. Wood, R.K., "A Factoring Algorithm Using Polygon-to-Chain Reductions for Computing K-Terminal Network Reliability," Networks, V-15, 1985, pp. 173-190.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5100	2
3.	Department Chairman, Code 55 Department of Operations Research Naval Postgraduate School Monterey, California 93943-5100	1
4.	Professor R. Kevin Wood, Code 55Wd Department of Operations Research Naval Postgraduate School Monterey, California 93943-5100	14
5.	Professor James D. Esary, Code 55Ey Department of Operations Research Naval Postgraduate School Monterey, California 93943-5100	1
6.	Dr. Howard Lambert 3728 Brunell Drive Oakland, California 94602	1
7.	Professor Richard E. Barlow IEOR Dept. Etcheverry Hall University of California, Berkeley Berkeley, California 94720	1
8.	Dr. Richard M. Thatcher Lawrence Livermore National Laboratory University of California, Livermore Livermore, California 94550	1
9.	Professor Robert E. Ball, Code 67Bp Department of Aeronautical Engineering Naval Postgraduate School Monterey, California 93943-5100	1

10. Department Chairman, Code 52Ml — 1
Department of Computer Science
Naval Postgraduate School
Monterey, California 93943-5100

11. Major William T. McCullers III, — 5
Code 0309
Office of the U.S. Marine Corps
Representative
Naval Postgraduate School
Monterey, California 93943-5100

END

FILMED

11-85

DTIC