

**NASA
Technical
Paper
2915**

1989

The Fault-Tree Compiler (FTC)

Program and Mathematics

Ricky W. Butler
*Langley Research Center
Hampton, Virginia*

Anna L. Martensen
*PRC Kentron, Inc.
Aerospace Technologies Division
Hampton, Virginia*



National Aeronautics and
Space Administration
Office of Management
Scientific and Technical
Information Division

Contents

Introduction	1
Fault-Tree Construction	1
The FTC User Interface	4
FTC Program Overview	4
Fault-Tree Definition Syntax	5
Lexical Details	6
Constant Definitions	6
Variable Definition	6
Expressions	7
Basic Event Definition	8
Gate Definition	8
Hierarchical Fault Trees	9
FTC Commands	10
EXIT	11
INPUT	11
PLOT	11
READ	11
RUN	11
SHOW	12
ACCURACY	12
CARE3	12
ECHO	12
LIST	12
POINTS	12
TIME	12
FTC Graphics	13
PLOT Command	13
PLOTINIT and PLOT+ Commands	13
Example FTC Sessions	13
Outline of a Typical Session	13
Examples	14
Example 1	14
Example 2	14
Example 3	15
Example 4	16
Example 5	19
Example 6	20
Mathematical Foundations of the FTC Program	22
Preliminaries	22
The Basic Approach	23

Notation	24
The FTC Algorithm	24
Basic Algorithm	25
Justification for the Basic Algorithm	27
Lemma 1	27
Lemma 2	27
Lemma 3	28
Pruning error bound	29
More Efficient Algorithm	30
Derivation of Error Bound for the More Efficient Algorithm	31
Domain of Efficiency	32
Concluding Remarks	32
Appendix—Error Messages	33
References	35

Introduction

Fault-tree analysis was first developed in 1961-62 by H. A. Watson of Bell Telephone Laboratories under an Air Force study contract for the Minuteman Launch Control System. The use of fault trees has since gained widespread support and is often used as a failure analysis tool by reliability engineers. Though conceptually simple, especially for those with a knowledge of basic circuit logic, the fault tree can be a useful tool. Although many computation techniques have been developed, a single superior algorithm has not been discovered. Some algorithms are superior for some problems but inferior for others (ref. 1). In this paper a new algorithm is presented which is tailored for the analysis of fault trees used to model fault-tolerant architectures—in particular, fault trees where the dominant failure modes contain a small number of basic events (e.g., 1, 2, or 3). This paper also presents a new program called the Fault-Tree Compiler (FTC) which is based on this new solution technique. The program provides the user with an expressive language for defining his fault tree and automatically calculates the probability of the top event in the tree. The program can perform a sensitivity analysis with respect to any specified parameter of the fault tree, such as a component failure rate or a specific event probability.

The motivation for the development of the Fault-Tree Compiler began with the observation that the Computer-Aided Reliability Estimation (CARE III) program (ref. 2) was often being used for the analysis of fault trees. Although CARE III can be used to solve fault trees, it was designed primarily to analyze complex reconfigurable systems where the fault-handling capabilities must be included in the reliability analysis. Therefore, it was not optimized for systems that can be described by a simple fault tree alone. The CARE III fault-tree code provided a minimal framework for the FTC mathematical solution technique. A more efficient solution technique which utilized an automatic pruning technique was developed and implemented in FTC. (The original CARE III algorithm required the user to manually prune the search space.) An error bound on the pruning technique has been derived which is presented in detail in this paper. The user interface to the program is a high-level language for describing fault trees. The improved solver and the Fault-Tree Compiler input language and sensitivity analysis capabilities provide a powerful fault-tree solver; in short,

1. The FTC program has a simple yet expressive input language
2. Automatic sensitivity analysis is provided
3. The mathematical solution technique can be used to obtain accuracy to within a user-specified number of digits
4. A hierarchical capability is provided which can simplify the preparation of the fault-tree input file and significantly reduce the program execution time
5. FTC is capable of handling common mode events, where the same event may appear more than once in the fault tree

The FTC solution algorithm was implemented in FORTRAN, and the user interface, in Pascal. The program executes on Digital Equipment Corporation (DEC) VAX computers running the VMS operating system and on computers with the Berkeley UNIX operating system.

This paper is organized as follows. First, the reader is given a brief introduction to fault trees. Next, the user interface to the FTC program is given along with several example sessions. Finally, the mathematical basis of the program is given, including the new algorithm and a theorem which provides an error bound for the algorithm.

A short tutorial on the construction of fault trees is provided. The tutorial will outline the basic gate types allowed by the FTC program and their use in describing an example system of interest.

Fault-Tree Construction

Fault trees are typically constructed by starting with the top event (usually representing some undesirable situation) and determining all possible ways to reach that event. This approach is often referred to as the *top down* or *backward* approach. An example of a *bottom up*

or *forward* approach is the failure modes and effect analysis (FMEA), where the analyst starts with the different failure modes of the system components and traces the effects of the failures.

An example fault-tree structure is shown in figure 1. The event of interest, referred to as the *top event*, appears as the top level in the tree. Only one top event is allowed. *Basic events* are the lowest level of the fault tree, and different combinations of basic events will result in the top event. In figure 1, the basic events are indicated by circles. The user associates a probability of occurrence with each basic event in the tree. Note that a basic event may appear more than once in the fault tree, in which case, it is referred to as a *common mode event*. A useful feature of the FTC program is its ability to handle these common mode events.

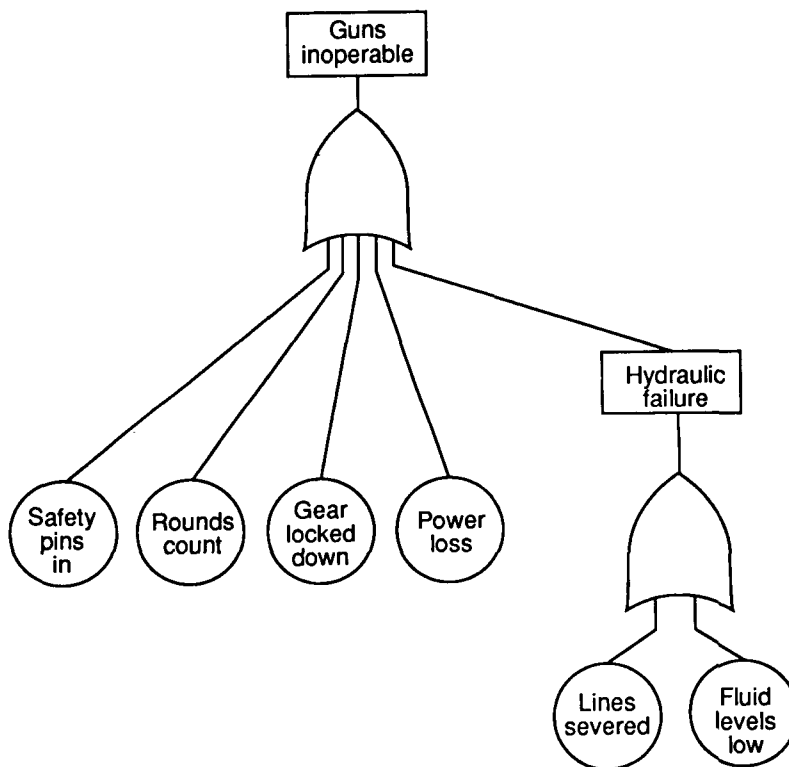
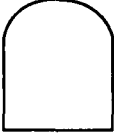
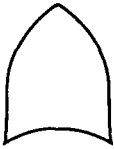
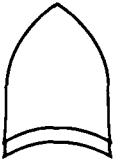
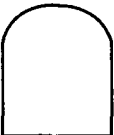



Figure 1. Example fault tree.

Events are combinations of basic events or other (lower) events. In figure 1 the output of the OR gate is an event. Typical fault-tree notation allows *comment* boxes to appear in the tree to describe an event. Though only two comment boxes appear in the example (to describe the hydraulic failure and the top event), boxes could have appeared above any event, basic or nonbasic. *Logic gates* delineate the causal relations which ultimately result in the top event. In the FTC program the following gates are allowed:

The following short example illustrates the top down process by which a fault tree is constructed:

The F-15C fighter has three primary weapon systems: heat-seeking missiles, radar missiles, and the gun. Occasionally the guns will be inoperable, due possibly to one or more separate events. The fault tree shown in figure 1 delineates the possible causes of in-flight gun no-fire. The preflight ground check includes the removal of several safety pins, including three pins which, once removed, will allow the gun to fire. A "rounds counter" on the plane determines the total number of rounds (bullets) to be fired. It is possible to completely restrict the firing of the gun with the proper rounds counter setting. The landing gear locked in the down position will also prevent the gun from firing. Additionally, loss of electric power to ignite the bullets or hydraulic power to rotate the barrels will completely inhibit the gun. Loss of hydraulic power may occur if the hydraulic lines are severed or the hydraulic fluid levels are low.

Gate symbol	Gate name	Result of gate
	AND	The output occurs when all input events occur simultaneously; an arbitrary number of input events are allowed
	OR	The output occurs when one or more of the input events occur; an arbitrary number of input events are allowed
	EXCLUSIVE OR	The output occurs when one, but not both, of the input events occurs; only two input events are allowed
	m OF n	The output occurs when m of n inputs occur; the number of inputs must be greater than or equal to m
	INVERT	This gate performs a complement (applying DeMorgan's Law) on the input; the number of inputs must equal one

It is not the goal of this paper to teach the construction of fault trees; however, this simple example illustrates several important elements of fault-tree modeling:

1. All basic events must be independent. In probability theory, two events, A and B , are independent if

$$P(AB) = P(A)P(B)$$

It is important to note that it is often very difficult to establish independence of events. In this example, it is assumed that the safety pin removal and setting of the rounds counter are independent events, even though both are performed during the preflight ground check.

2. Sequences of events cannot be modeled with the gates allowed by the FTC program. For many systems of interest, an event Z occurs if and only if event A occurs before event B . If event B occurs before event A , a different result is seen. At best, the analyst must define a basic event which is the result of some sequence of events and assign a probability to the basic event.
3. Mutually exclusive events must be handled with care. *Basic events cannot be mutually exclusive.* For example, basic event A cannot be defined as *Power on* and basic event B defined as *Power off*. However, basic event A may be defined as *Power on*, and an

INVERT gate (which performs the probabilistic complement of the input) with basic event *A* as input may define the event *Power not on*.

4. Typically, fault trees are developed to demonstrate the probability of some undesirable top event. A typical top event might be *Catastrophic System Failure*. Generally, it is much faster to enumerate the ways that a system will fail than it is to enumerate the ways a system will succeed. Occasionally, however, it is more advantageous to create a *success* tree. The FTC program makes no distinction between the trees; it simply solves for the probability of the top event in a tree.
5. Basic events must be assigned a probability of occurrence. The FTC program also allows for failure rates to be assigned to basic events. The user must also supply a value for the mission time at which the probability of system failure will be evaluated. Parametric analysis is facilitated by allowing one basic event probability or rate to vary over a range of values. The syntax is described in the section "Fault-Tree Definition Syntax."

For more information on fault trees, reference 3 is recommended. The next sections discuss (1) the user interface for the FTC program, (2) example FTC sessions, and (3) the mathematical foundations of the solution technique used in version 2 of the program. In the appendix, the error messages generated by the program are explained.

The FTC User Interface

In this section the user interface to the FTC program is described. The interface consists of a tree-definition language and user commands. An overview of the interface is given followed by a detailed description of the tree-definition language and user commands.

FTC Program Overview

The user of the FTC program must define his fault tree with a simple language. There are two basic statements in the tree-definition language—the basic event definition statement and the gate definition statement. The basic event definition statement defines a fundamental event and associates a probability with this event. For example, the statement

```
X: 0.002;
```

defines a fundamental event which occurs with probability 0.002. The gate definition statement defines a gate of the fault tree by specifying the gate type and all its inputs. For example, the statement

```
G1: AND( Q12, V123, L12, E5);
```

defines an "AND-gate" with output G1 and inputs Q12, V123, L12, and E5. The basic events and gate-output identifiers may consist of letters, numbers, and underscores () but must begin with a letter. The following gates are allowed: AND, OR, XOR (EXCLUSIVE OR), INV (INVERT gate), and m OF (m out-of-n gate).

The input language is probably best explained by way of an example. A fault tree and the corresponding FTC input file are shown in figure 2. The first seven lines assign probabilities to the basic events E1, E2, E3, E4, E5, and E6. These probabilities have been defined in terms of a parameter P. This has been done to illustrate the expression syntax of the input language. The next four lines define four gates by specifying the inputs to a gate as arguments. For example, the first of these four define an "AND gate" whose inputs are the basic events E1 and E2. The output of this gate is then given the name G1. Other gates can be defined using basic events or previously defined gates. The top gate of the model is always named TOP. When the RUN command is issued to the FTC program, the probability of TOP is calculated.

The program also provides a capability for parameter sensitivity studies. For example, suppose one wishes to determine the impact of varying the value of P in the model above. The

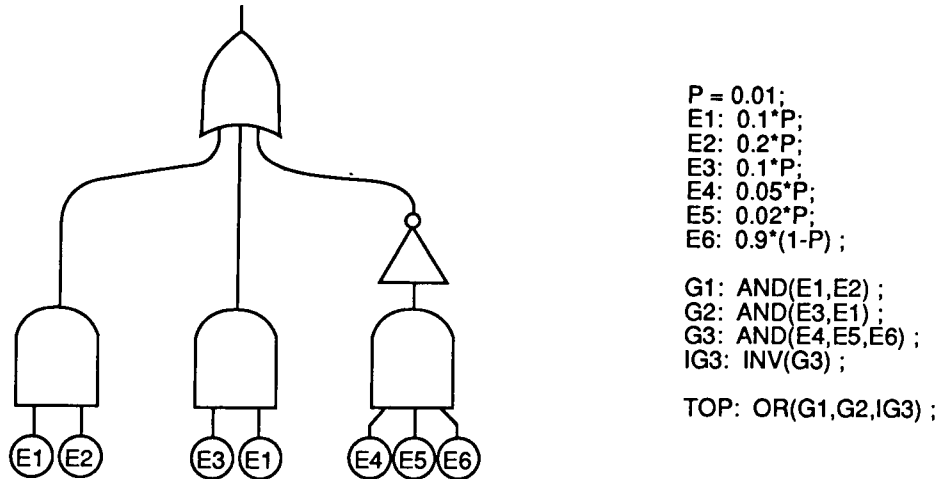


Figure 2. Fault tree and corresponding FTC input file.

user first determines a range of values, say 0.1 to 0.6, and then alters the first line of the above model as follows:

```
P = 0.1 TO 0.6;
```

The program will automatically calculate the probability of the TOP event as a function of P.

Finally, the program allows the user to specify his fault tree by using a hierarchical approach. This is done by defining subtrees whose top event probabilities can be used when defining other subtrees or the main tree. For example:

```

SUBTREE TREE_1;
X1: 0.1; X2: 0.2; X3: 0.1;
TOP: AND(X1,X2,X3);

SUBTREE TREE_2;
Y1: 0.2; Y2: 0.007; Y3: 0.02;
TOP: OR(Y1,Y2,Y3);

TREE MAIN;
E1: 0.1;
E2: TREE_1;
E3: TREE_2;
E4: TREE_1*(1-TREE_2);
E5: 2*TREE_1 + 0.3*TREE_2;
G1: AND(E4,E5);
TOP: OR(E1,E2,E3,G1);

```

This input defines a hierarchical fault tree which consists of a main tree and two subtrees. Note that the main tree references the names of the subtrees in its basic-event definitions. The meaning of this is simple. The program first calculates the "TOP" probability of the subtrees. The probabilities of each of these TOP events are stored in the names of the subtrees. Thus, event E2 in the main tree has a probability of occurrence equal to the TOP event of the first subtree.

Fault-Tree Definition Syntax

The basic-event definition statement and the gate definition statement are the only essential ingredients of the FTC input language. However, the flexibility of the FTC program has been increased by adding several features commonly seen in programming languages such as FORTRAN or Pascal. The details of the FTC language are described in the following subsections.

Lexical Details

The probabilities assigned to events or component failure rates are floating point numbers. The Pascal/FORTRAN real syntax is used for these numbers. Thus, all the following would be accepted by the FTC program:

```
0.001
12.34
1.2E-4
1E-5
```

The semicolon is used for statement termination. Therefore, more than one statement may be entered on a line. Comments may be included any place that blanks are allowed. The notation “(*)” indicates the beginning of a comment and “*)” indicates the termination of a comment. The following is an example of the use of a comment:

```
GYRO_F: 0.025;      (* PROBABILITY OF A GYRO FAILURE *)
```

If statements are entered from a terminal (as opposed to using the READ command described subsequently), then the carriage return is interpreted as a semicolon. Thus, interactive statements do not have to be terminated by an explicit semicolon unless more than one statement is entered on the line.

In interactive mode, the FTC system will prompt the user for input by a line number followed by a question mark. For example,

```
1?
```

The number is a count of the current line plus the number of syntactically correct lines entered into the system thus far.

Constant Definitions

The user may equate numbers to identifiers. Thereafter, these constant identifiers may be used instead of the numbers. For example,

```
LAMBDA = 0.0052;
RECOVER = 0.005;
```

Constants may also be defined in terms of previously defined constants:

```
GAMMA = 10*LAMBDA;
```

In general, the syntax is

```
<name> = <expression>;
```

where <name> is a string of up to eight letters, digits, and underscores (_) beginning with a letter, and <expression> is an arbitrary mathematical expression as described in a subsequent section entitled “Expressions.”

Variable Definition

In order to facilitate parametric analyses, a single variable may be defined. A range of values is given for this variable. The FTC system will compute the probability of the top event as a function of this variable. If the system is run in graphics mode (to be described later), then a plot of this function can be made. The following statement defines LAMBDA as a variable with range 0.001 to 0.009:

```
LAMBDA = 0.001 TO 0.009;
```

Only one such variable may be defined. A special constant, POINTS, defines the number of points over this range to be computed. This constant can be defined any time before the RUN command. For example,

```
POINTS = 25;
```

specifies that 25 values over the range of the variable should be computed. The method used to vary the variable over this range can be either geometric or arithmetic and is best explained by an example. Suppose POINTS = 4, then

Geometric:

XV = 1 TO* 1000;

yields XV values of 1, 10, 100, and 1000.

Arithmetic:

XV = 1 TO+ 1000;

yields XV values of 1, 333, 667, and 1000. The * following the TO implies a geometric range. A TO+ or simply TO implies an arithmetic range.

One additional option is available—the BY option. Adding the phrase BY <inc> to this syntax causes the program to start with the specified first value and determine the subsequent values of the variable by adding <inc> (if arithmetic) or multiplying by <inc> (if geometric). In this case, the value of POINTS is automatically calculated by the program. For example,

V = 1E-6 TO* 1E-2 BY 10;

sets POINTS equal to 5 and the values of V to 1E-6, 1E-5, 1E-4, 1E-3, and 1E-2. The statement

Q = 3 TO+ 5 BY 1;

sets POINTS equal to 3, and the values of Q to 3, 4, and 5.

In general, the syntax is

<var> = <expression> TO {<c>} <expression> {BY <inc>}

where <var> is a string of up to eight letters and digits beginning with a letter, <expression> is an arbitrary mathematical expression as described in the next section, and the optional <c> is a + or *. The BY clause is optional; if it is used, then <inc> is any arbitrary expression.

Expressions

When defining constants or an event probability, arbitrary functions of the constants and the variable may be used. The following operators may be used:

- + addition
- subtraction
- * multiplication
- / division
- ** exponentiation

The following standard functions may be used:

EXP(X)	exponential function
LN(X)	natural logarithm
SIN(X)	sine function
COS(X)	cosine function
ARCSIN(X)	arc sine function
ARCCOS(X)	arc cosine function
ARCTAN(X)	arc tangent function
SQRT(X)	square root

Both () and [] may be used for grouping in the expressions. The following are permissible expressions:

2E-4
1 - [EXP(-LAMBDA*TIME)]

Basic Event Definition

The fundamental events of the fault tree (i.e., events which are not the outputs of a gate in the tree) must be assigned probabilities. This is accomplished by using the basic-event definition statement. This statement has the following syntax:

<event-id> : <expression>;

where <event-id> is the name of the event and <expression> is an expression defining the probability of the event which must evaluate to a number between 0 and 1. Alternately, the user can specify the rate of an event. Rates are specified by using the following syntax:

<event-id> -> <rate-expression>;

where <event-id> is the name of the event and <rate-expression> is an expression defining the rate of the event. The probability of the event is calculated by the program with the following formula:

$$\text{Prob}[\text{event}] = 1.0 - \exp(-\langle\text{rate-expression}\rangle * \text{TIME})$$

where TIME is the value of the special constant TIME which defines the mission time. If TIME is not defined by the user, then the program uses 10 for the mission time. Note that this formula represents the standard exponential distribution function,

$$F(t) = 1 - e^{-\lambda t} \quad (t \geq 0)$$

(Note that the solution algorithm only manipulates probabilities. The rates and times are only used by the input language processor to derive an event probability. The solution algorithm is combinatorial and does not deal with the rates directly.)

Gate Definition

Once all the fundamental events are defined, the gate definition statement may be used to define the structure of the fault tree. The syntax of this statement is

<output-id> : $\left\{ \begin{array}{l} \text{OR} \\ \text{INV} \\ \text{XOR} \\ \text{AND} \end{array} \right\} (\langle\text{input}\rangle, \langle\text{input}\rangle, \dots);$

or

<output-id> : <int> OF (<input>, <input>, ...);

The <output-id> is the name of the (nonbasic) event which is the output of the gate and <int> is a positive integer which is less than or equal to the number of arguments. The type of gate is indicated by the reserved words OR, AND, INV, XOR, or OF as follows:

- AND output probability is probability of all events occurring
- OR output probability is probability of one or more events occurring
- XOR output probability is probability of one of events, but not both, occurring (i.e., EXCLUSIVE OR gate)
- INV output probability is probabilistic complement of input (i.e., INVERT gate)
- m OF output probability is probability of m or more events occurring (i.e., m OF n gate)

Any number of input events may be included within the parentheses for the AND, OR, and m OF gates. The XOR gate takes two arguments and the INV gate takes one. The following gate definition statements are valid:

G1: AND(X, Y, Z);
G2: OR(A1, A2, A3);
G3: 2 OF (A1, A2, A3, A4);

```

CAT_DIES: 9 OF (HIT_BY_CAR, ATTACKED_BY_DOG, STARVED_TO_DEATH,
                DROPPED_400_FEET, FLAMBEED, CAUGHT_IN_FAN_BELT,
                DROWNED_IN_TOILET, SUFFOCATED, EXCESSIVE_DIARRHEA,
                LOST_IN_DESERT, FED_ANTIFREEZE, MICROWAVED,
                STEPPED_ON );

```

Hierarchical Fault Trees

Often a system consists of several identical independent subsystems. In order to preserve the independence, it is necessary to replicate the subsystem fault tree in the system model. For example, suppose we have a system which contains four identical independent subsystems. The system fails when three of the subsystems fail. Each subsystem consists of four components. If any component fails, the subsystem fails. The following fault tree describes the subsystem:

```

COMP_1: .01;
COMP_2: .02;
COMP_3: .03;
COMP_4: .05;
SUBSYSTEM_FAILS: OR(COMP_1, COMP_2, COMP_3, COMP_4);

```

The system fault tree is as follows:

```

SYSTEM_FAILS: 3 OF (SUBSYS_1_FAILS, SUBSYS_2_FAILS, SUBSYS_3_FAILS,
                   SUBSYS_4_FAILS);

```

To integrate these sections into one fault tree, the subsystem must be replicated four times using different event names in each replicate:

```

SUBSYS_1_COMP_1: .01;
SUBSYS_1_COMP_2: .02;
SUBSYS_1_COMP_3: .03;
SUBSYS_1_COMP_4: .05;
SUBSYS_1_FAILS: OR( SUBSYS_1_COMP_1, SUBSYS_1_COMP_2,
                   SUBSYS_1_COMP_3, SUBSYS_1_COMP_4 );

SUBSYS_2_COMP_1: .01;
SUBSYS_2_COMP_2: .02;
SUBSYS_2_COMP_3: .03;
SUBSYS_2_COMP_4: .05;
SUBSYS_2_FAILS: OR( SUBSYS_2_COMP_1, SUBSYS_2_COMP_2,
                   SUBSYS_2_COMP_3, SUBSYS_2_COMP_4 );

SUBSYS_3_COMP_1: .01;
SUBSYS_3_COMP_2: .02;
SUBSYS_3_COMP_3: .03;
SUBSYS_3_COMP_4: .05;
SUBSYS_3_FAILS: OR( SUBSYS_3_COMP_1, SUBSYS_3_COMP_2,
                   SUBSYS_3_COMP_3, SUBSYS_3_COMP_4 );

SUBSYS_4_COMP_1: .01;
SUBSYS_4_COMP_2: .02;
SUBSYS_4_COMP_3: .03;
SUBSYS_4_COMP_4: .05;
SUBSYS_4_FAILS: OR( SUBSYS_4_COMP_1, SUBSYS_4_COMP_2,
                   SUBSYS_4_COMP_3, SUBSYS_4_COMP_4 );

SYSTEM_FAILS: 3 OF (SUBSYS_1_FAILS, SUBSYS_2_FAILS, SUBSYS_3_FAILS,
                   SUBSYS_4_FAILS);

```

Obviously, this is a tedious process. Therefore, the FTC program provides the user with a hierarchical fault-tree capability. The following model is semantically equivalent to the previous fault tree:

```
SUBTREE SUBSYSTEM_FAILS;

    COMP_1: .01;
    COMP_2: .02;
    COMP_3: .03;
    COMP_4: .05;

    TOP: OR(COMP_1,COMP_2,COMP_3,COMP_4);

TREE SYSTEM_FAILS;

    SUBSYS_1_FAILS: SUBSYSTEM_FAILS;
    SUBSYS_2_FAILS: SUBSYSTEM_FAILS;
    SUBSYS_3_FAILS: SUBSYSTEM_FAILS;
    SUBSYS_4_FAILS: SUBSYSTEM_FAILS;

    TOP: 3 OF (SUBSYS_1_FAILS, SUBSYS_2_FAILS, SUBSYS_3_FAILS,
               SUBSYS_4_FAILS);
```

The model is defined in two sections. The first section defines a subtree which is named `SUBSYSTEM_FAILS`. This subtree is solved by the program and the probability of its top event is saved in the identifier `SUBSYSTEM_FAILS`. In subsequent trees or subtrees this identifier can be used. In this model, four events in the main tree are given the probability of the subsystem, that is, `SUBSYSTEM_FAILS`.

To simplify the analysis of the effect of a system parameter on the probability of the top event, global variables and constants may be used. These must be defined before any subtrees are defined. Global events cannot be defined.

The effect of the change in the failure probability of a component in the previous model could be investigated by using the following model:

```
FP = .01 TO .05 BY .01;

SUBTREE SUBSYSTEM_FAILS;

    COMP_1: FP;
    COMP_2: .02;    COMP_3: .03;    COMP_4: .05;

    TOP: OR(COMP1_,COMP_2,COMP_3,COMP_4);

TREE SYSTEM_FAILS:

    SUBSYS_1_FAILS: SUBSYSTEM_FAILS;    SUBSYS_2_FAILS: SUBSYSTEM_FAILS;
    SUBSYS_3_FAILS: SUBSYSTEM_FAILS;    SUBSYS_4_FAILS: SUBSYSTEM_FAILS;

    TOP: 3 OF (SUBSYS_1_FAILS, SUBSYS_2_FAILS, SUBSYS_3_FAILS,
               SUBSYS_4_FAILS);
```

FTC Commands

The FTC program is controlled by interactively entered commands. These commands can be used to read in model-description files, set various options, initiate the computation, plot results, etc. These commands will be described in detail in this section.

There are two types of commands in FTC. The first type of command is initiated by one of the following reserved words:

```
EXIT    INPUT    PLOT    READ    RUN    SHOW
```

The second type of command is invoked by setting one of the special constants
ACCURACY CARE3 ECHO LIST POINTS TIME
equal to one of its predefined values.

EXIT

The EXIT command causes termination of the FTC program.

INPUT

The INPUT command increases the flexibility of the READ command. Within the model description file created with a text editor, INPUT commands can be inserted that will prompt for values of specified constants while the model file is being processed by the READ command. For example, the command

```
INPUT LVAL;
```

will prompt the user for a number as follows:

```
LVAL?
```

This creates a new constant LVAL which is equal to the value input by the user. Several constants can be interactively defined with one statement, for example,

```
INPUT X, Y, Z;
```

PLOT

The PLOT command can be used to plot the output on a graphics display device. This command is described in detail in the next section, "FTC Graphics." This command is only available at installations which have the graphics library TEMPLATE.

READ

A sequence of FTC statements may be read from a disk file using the READ command. The following command reads FTC statements from a disk file named SIFT.TRE:

```
READ SIFT.TRE;
```

If no file name extension is given, the default extent .TRE is assumed. A user can build a model description file with a text editor and use this command to read the file into the FTC program.

RUN

After a fault tree has been input to the FTC program, the RUN command is used to initiate the computation:

```
RUN;
```

The output is displayed on the terminal according to the LIST option (described later). If the user wants the output written to a disk file instead, the following syntax is used:

```
RUN <outname>;
```

where the output file <outname> may be any permissible VAX VMS file name. Two positional parameters are available on the RUN command. These parameters enable the user to change the value of the special constants POINTS and LIST in the RUN command. For example,

```
RUN (30,2) OUTFILE.DAT;
```

is equivalent to the following sequence of commands:

```
POINTS = 30;  
LIST = 2;  
RUN OUTFILE.DAT
```

Each parameter is optional so the following are acceptable:

<code>RUN(10);</code>	change POINTS to 10 then run.
<code>RUN(,0);</code>	change LIST to 0 and run.
<code>RUN(20,1);</code>	change POINTS to 20 and LIST to 1 then run.

After the RUN command has completed, the fault tree and all symbol definitions are deleted. However, the results of the run are stored and available for plotting via the PLOT command.

SHOW

The value of the identifier <id> is displayed by the following command:

```
SHOW <id>;
```

This function cannot be used to obtain values for gate identifiers or identifiers which depend upon the variable.

ACCURACY

With the ACCURACY command, the user specifies the number of digits of accuracy he desires in the final answer. A significant decrease in execution time can be obtained if only a few digits accuracy is necessary. The default value is 6. This parameter is used by the program to reduce the number of vectors which must be searched. Also the program calculates an accumulated rounding error due to the imprecision of floating point arithmetic. If this error is large enough to influence the specified accuracy, the following warning message is given:

```
ROUND OFF ERROR => ONLY x DIGITS ACCURACY
```

CARE3

If set equal to 1, the program will generate a file containing the fault tree in the CARE III syntax. (See ref. 2.) The default value of 0 specifies that no CARE III file be written. The name of the generated file is CARE3.TRE. The first line of a CARE III input file must contain four numbers which give the first and last input event numbers and the first and last output event numbers. Note that the input range is completely specified, but that the upper value on the output range is specified by X. The user must edit the file, supplying the appropriate upper value for the output range and inserting the tree into an otherwise complete CARE III input file.

ECHO

The ECHO constant can be used to turn off the echo when reading a disk file. The default value of ECHO is 1, which causes the model description to be listed as it is read. (See example 4 in the section "Example FTC Sessions.")

LIST

The amount of information output by the program is controlled by the LIST command. Three list modes are available as follows:

LIST = 0; No output is sent to terminal, but results can still be displayed using PLOT command.

LIST = 1; Output is sent to terminal; this is the default.

LIST = 2; Output sent to terminal contains more detailed information, e.g., number of vectors processed. See example 6.

POINTS

The POINTS constant specifies the number of points to be calculated over the range of the variable. The default value is 25. If no variable is defined, then this specification is ignored.

TIME

The TIME constant specifies the mission time when rates are used for events. The TIME constant has meaning only when the model includes failure rates, which depend upon time. For example, if the user sets TIME = 1.3, the program computes the probability of the top event

at mission time equal to 1.3. The default value of `TIME` is 10. If the default value of `TIME` is not to be used, `TIME` must be defined before any events are defined. (The program does not assume any particular units of time. The program assumes that the units used for `TIME` are the same as those used for the event rates.)

FTC Graphics

Although the FTC program is easily used without graphics output, many users desire the increased user friendliness of the tool when assisted by graphics. The graphics output module is written in FORTRAN but uses the graphics library `TEMPLATE` (available at the Langley Research Center). Thus, the FTC graphics capabilities will only be available at installations which have this library. Alternatively, this module can be rewritten by using another graphics library.

The FTC program can plot the probability of system failure as a function of any model parameter. The output from several FTC runs can be displayed together in the form of contour plots. Thus, the effect on system reliability of two model parameters can be illustrated on one plot.

PLOT Command

After a `RUN` command, the `PLOT` command can be used to plot the output on the graphics display. The syntax is

```
PLOT <op>, <op>, ... <op>
```

where <op> are plot options. Any `TEMPLATE` "USET" or "UPSET" parameter can be used, but the following are the most useful:

<code>XLOG</code>	plot <i>X</i> -axis using logarithmic scale
<code>YLOG</code>	plot <i>Y</i> -axis using logarithmic scale
<code>XYLOG</code>	plot both <i>X</i> - and <i>Y</i> -axes using logarithmic scales
<code>NOL0</code>	plot <i>X</i> - and <i>Y</i> -axes with linear scaling
<code>XLEN=5.0</code>	set <i>X</i> -axis length to 5.0 in.
<code>YLEN=8.0</code>	set <i>Y</i> -axis length to 8.0 in.
<code>XMIN=2.0</code>	set <i>X</i> -origin 2 in. from left side of screen
<code>YMIN=2.0</code>	set <i>Y</i> -origin 2 in. above bottom of screen

PLOTINIT and PLOT+ Commands

The `PLOTINIT` and `PLOT+` commands are used together to display multiple runs on one plot. A single run of FTC generates unreliability as a function of a single variable. To see the effect of a second variable (i.e., display contours of a three-dimensional surface) the `PLOT+` command is used. The `PLOTINIT` command should be called before performing the first FTC run. This command defines the second variable (i.e., the contour variable):

```
PLOTINIT BETA;
```

This defines `BETA` as the second independent variable. Next, the user must set `BETA` to its first value. After the run is complete, the output is plotted by using the `PLOT+` command. The parameters of this command are identical to the `PLOT` command. The only difference is that the data values are saved and can be displayed in conjunction with subsequent run data values. Next, `BETA` must be set to a second value, another FTC run made, and `PLOT+` must be called again. This time both outputs will be displayed together. Up to 10 such runs can be displayed together.

Example FTC Sessions

Outline of a Typical Session

The FTC program was designed for interactive use. The following method of use is recommended:

1. Using a text editor, create a file of FTC commands describing the fault tree to be analyzed.
2. Start the FTC program and use the READ command to retrieve the model information from this file.
3. Then, various commands may be used to change the values of the special constants, such as LIST, POINTS, as desired. Altering the value of a constant identifier does not affect any transitions entered previously even though they were defined with a different value for the constant. The range of the variable may be changed after transitions are entered.
4. Enter the RUN command to initiate the computation.
5. Issue PLOT command to plot the results.

Examples

The following examples illustrate interactive FTC sessions. For clarity, all user inputs are given in lowercase letters.

Example 1. This session illustrates direct interactive input and the type of error messages given by FTC:

```
$ ftc

FTC V2.4    NASA Langley Research Center

1?
2? time = 10.0;
3? lambda = 1e-4;
4?
5? X: 1.0 - exp( -lambda*time);
6? Y: 1.0 - exp( -lamda*time);
      ^ IDENTIFIER NOT DEFINED
6? Y: 1.0 - exp( -lambda*time);
7? top: or(X,Y);
8?

          Pr[ TOP EVENT ]
-----
          1.99800E-03

*** WARNING: SYNTAX ERRORS PRESENT BEFORE RUN
3 OUT OF THE 4 VECTORS ANALYZED CAUSED TOP EVENT
SMALLEST NUMBER OF EVENTS CAUSING TOP EVENT = 1
0.420 SECS. CPU TIME UTILIZED

9? exit
```

The warning message is simply informative. If a user receives this message, he should check his input file to make sure that the model description is correct. In this example, since the syntax error was corrected in the next line, the model was correct. A complete list of program-generated error messages is given in the appendix. The message "3 OUT OF THE 4 VECTORS ANALYZED CAUSED TOP EVENT" informs the user of the amount of processing performed by the program. The message "SMALLEST ..." indicates the least number of concurrent basic events which caused the TOP event.

Example 2. This example illustrates the recommended method of using FTC—creating a file with a text editor and issuing a READ command. Prior to initiating the FTC program, the text file SIMP.TRE was created by using an editor. The contents of this file are echoed as it is read in by the program:

```
$ ftc
FTC V2.4    NASA Langley Research Center

1? read SIMP.TRE
```

```

2. EVNT1: 0.001;
3. EVNT2: 0.002;
4. EVNT3: 0.003;
5. EVNT4: 0.0004;
6. EVNT5: 0.0005;
7. EVNT6: 0.0006;
8. TOP: 2 OF (EVNT1, EVNT2, EVNT3, EVNT4, EVNT5, EVNT6);

9? accuracy = 2;
10? run

```

MODEL FILE = SIMP.TRE

FTC V2.4 14-OCT-1988 09:00:20

```

-----
                Pr[TOP EVENT]      ERRORS/WARNINGS
-----
                2.06649E-05

```

16 OUT OF THE 24 VECTORS ANALYZED CAUSED TOP EVENT
SMALLEST NUMBER OF EVENTS CAUSING TOP EVENT = 2
0.590 SECS. CPU TIME UTILIZED

11? exit

The statement ACCURACY = 2 instructed the program to compute an answer with 2 digits accuracy. Consequently, the program only analyzed 24 out of the 64 possible vectors (i.e., six events implies 2^6 vectors). The exact answer is 2.06859E-05, which illustrates that the program produced an answer with more accuracy than 2 digits. The pruning algorithm used by FTC is conservative; consequently, the program often produces results with more accuracy than requested. For details on the pruning algorithm used by FTC, see the section "Mathematical Foundations of the FTC Program."

Example 3. This example demonstrates the use of the hierarchical fault-tree capability to partially describe an aircraft pitch control architecture. The proposed architecture is composed of four independent actuator subsystems and the supporting hydraulic and electronic systems. Each actuator subsystem is comprised of a pitch rate sensor, a computer, and the actuator. Two of the four actuator subsystems failing will result in loss of pitch control. Likewise, loss of either the hydraulic or electronic system will cause loss of pitch control.

\$ ftc

FTC V2.4 NASA Langley Research Center

```

1? read ex3.mod

2. SUBTREE ACT_SYS_FAIL;
3.
4. PITCH_RATE_SENSOR -> 1.8E-05;
5. COMPUTER          -> 4.4E-04;
6. ACTUATOR          -> 3.7E-05;
7.
8. TOP: OR(PITCH_RATE_SENSOR, COMPUTER, ACTUATOR);
9.
10. TREE LOSS_OF_PITCH_CONTROL;
11.
12. HYDRAULIC_FAILURE: 1.3E-06;      (* HYDRAULIC SYSTEM FAILURE RATE *)
13. ELECTRONICS_FAILURE: 5.0E-04;    (* AIRCRAFT ELECTRONICS FAILURE *)
14. ACT_SYS1_FAILS: ACT_SYS_FAILS;
15. ACT_SYS2_FAILS: ACT_SYS_FAILS;    (* THE ACTUATOR SYSTEM IS *)
16. ACT_SYS3_FAILS: ACT_SYS_FAILS;    (* COMPOSED OF FOUR INDE- *)
17. ACT_SYS4_FAILS: ACT_SYS_FAILS;    (* PENDENT SUBSYSTEMS. *)
18.

```

19. GATE1: 2 OF (ACT_SYS1_FAILS,ACT_SYS2_FAILS,ACT_SYS3_FAILS,ACT_SYS4_FAILS);
 20. TOP: OR (GATE1, HYDRAULIC_FAILURE, ELECTRONICS_FAILURE);
 21.

22? run

MODEL FILE = EX3.TRE

FTC V2.4 14-OCT-1988 09:00:25

Pr[ACT_SYS_]	ERRORS/WARNINGS
4.93777E-03	

7 OUT OF THE 8 VECTORS ANALYZED CAUSED TOP EVENT
 SMALLEST NUMBER OF EVENTS CAUSING TOP EVENT = 1

Pr[TOP EVENT]	ERRORS/WARNINGS
6.46555E-04	

43 OUT OF THE 48 VECTORS ANALYZED CAUSED TOP EVENT
 SMALLEST NUMBER OF EVENTS CAUSING TOP EVENT = 1
 1.340 SECS. CPU TIME UTILIZED

23? exit

Example 4. This example illustrates the use of the FTC program to process the fault tree used in the Integrated Airframe Propulsion Control System Architecture (IAPSA II) project to analyze surface control failures. (See ref. 4.)

The surface control system has three separate actuation channels, each consisting of an actuation stage and a disengage device stage. The actuation channels are brickwalled with force-sum voting at the control surface. Channel self-monitoring techniques are the primary method of fault detection and isolation. Each actuation channel contains two special devices for fault tolerance. The disengage device can deactivate a faulty channel. The surface can be controlled by one channel if the other two channels have been deactivated. Additionally, an override device in each channel allows two good channels to overpower a channel with a failed disengage device. Thus, surface failure (top event) can occur in two ways: (1) loss of all three actuation channels and (2) loss of two channels when one of the lost channels has a failed disengage device. The following tree describes these aspects of the system failure process:

TREE SURFACE_FAILURE;

```

CH1: CH_FAULT;          (* Channel 1 failure *)
CH2: CH_FAULT;          (* Channel 2 failure *)
CH3: CH_FAULT;          (* Channel 3 failure *)

DD1 -> 6.0E-6;          (* Channel 1 disengage device failure *)
DD2 -> 6.0E-6;          (* Channel 2 disengage device failure *)
DD3 -> 6.0E-6;          (* Channel 3 disengage device failure *)

```

LOSS_OF_ALL_CHANNELS: AND(CH1, CH2, CH3);

```

SF1A: AND(CH1, DD1);
SF1B: OR(CH2, CH3);
CHANNEL1_UNISOLATED: AND(SF1A, SF1B);

```

```

SF2A: AND(CH2, DD2);
SF2B: OR(CH1, CH3);
CHANNEL2_UNISOLATED: AND(SF2A, SF2B);

```

```

SF3A: AND(CH3, DD3);
SF3B: OR(CH1, CH2);
CHANNEL3_UNISOLATED: AND(SF3A, SF3B);

TOP: OR (LOSS_OF_ALL_CHANNELS, CHANNEL1_UNISOLATED,
        CHANNEL2_UNISOLATED, CHANNEL3_UNISOLATED);

```

Next, the failures leading to an actuation channel breakdown must be enumerated in a subtree. An actuation channel failure can occur because of the loss of the intersystem (I/S) bus, lack of two surface commands, or a fault in the actuation channel elements—the I/S bus terminal, the elevator processor, and the electrical and mechanical actuation hardware. Surface commands can be lost due to command generation faults or computer bus terminal faults. The following subtree describes actuation channel failure:

```

SUBTREE CH_FAULT;

C1_COMMAND: COMMAND_FAILURE;      (* Loss of command 1 *)
C2_COMMAND: COMMAND_FAILURE;      (* Loss of command 2 *)
C3_COMMAND: COMMAND_FAILURE;      (* Loss of command 3 *)
C4_COMMAND: COMMAND_FAILURE;      (* Loss of command 4 *)

C11 -> 1E-6;                       (* computer 1 bus terminal fault *)
C21 -> 1E-6;                       (* computer 2 bus terminal fault *)
C31 -> 1E-6;                       (* computer 3 bus terminal fault *)
C41 -> 1E-6;                       (* computer 4 bus terminal fault *)

ACT1 -> 90E-6;                     (* fault in actuation channel elements *)
B1 -> 20E-6;                       (* failure in I/S bus *)

AC1: OR(C1_COMMAND, C11);
AC2: OR(C2_COMMAND, C21);
AC3: OR(C3_COMMAND, C31);
AC4: OR(C4_COMMAND, C41);

LOSE_TWO_COMMANDS: 3 OF (AC1, AC2, AC3, AC4);
TOP: OR(ACT1, B1, LOSE_TWO_COMMANDS);

```

A command generation fault can occur due to lack of pilot control sensors (PCS) data, lack of inertial reference air data computer (IRADC) data, or computer failure. The loss of data can be due to data source failure, I/S bus failure, or computer bus terminal failure:

```

SUBTREE COMMAND_FAILURE;

PCS1 -> 11.0E-6;                   (* loss of channel 1 PCS data *)
B1 -> 20E-6;                       (* failure in channel 1 I/S bus *)
C11 -> 1E-6;                       (* bus terminal to channel 1 fault *)
IRADC1 -> 122.5E-6;               (* loss of channel 1 IRADC data *)

PCS2 -> 11.0E-6;                   (* loss of channel 2 PCS data *)
B2 -> 20E-6;                       (* failure in channel 2 I/S bus *)
C12 -> 1E-6;                       (* bus terminal to channel 2 fault *)
IRADC2 -> 122.5E-6;               (* loss of channel 2 IRADC data *)

PCS3 -> 11.0E-6;                   (* loss of channel 3 PCS data *)
B3 -> 20E-6;                       (* failure in channel 3 I/S bus *)
C13 -> 1E-6;                       (* bus terminal to channel 3 fault *)
IRADC3 -> 122.5E-6;               (* loss of channel 3 IRADC data *)

CLC1 -> 100.0E-6;                 (* computer failure rate *)

```

```

LPD1: OR(PCS1, B1, C11);
LPD2: OR(PCS2, B2, C12);
LPD3: OR(PCS3, B3, C13);
PCS_DATA_LOSS: AND(LPD1, LPD2, LPD3);

LID1: OR(IRADC1, B1, C11);
LID2: OR(IRADC2, B2, C12);
LID3: OR(IRADC3, B3, C13);
IRADC_DATA_LOSS: AND(LID1, LID2, LID3);

TOP: OR(PCS_DATA_LOSS, IRADC_DATA_LOSS, CLC1);

```

To simplify the discussion, the TREE section was presented first. In an input file to the FTC program, all subtrees must be placed before the TREE section. The above model was available in file IAPSA.TRE prior to the following interactive session:

```
$ ftc
```

```
FTC V2.4   NASA Langley Research Center
```

```
1? echo = 0
2? read iapsa
```

```
87? run
```

```
MODEL FILE = IAPSA.TRE
```

```
FTC V2.4 14-OCT-1988 09:00:31
```

```

----- Pr[COMMAND_]   ERRORS/WARNINGS -----
9.99503E-04

```

```
51 OUT OF THE 181 VECTORS ANALYZED CAUSED TOP EVENT
SMALLEST NUMBER OF EVENTS CAUSING TOP EVENT = 1
```

```

----- Pr[CH_FAULT]   ERRORS/WARNINGS -----
1.09940E-03

```

```
40 OUT OF THE 77 VECTORS ANALYZED CAUSED TOP EVENT
SMALLEST NUMBER OF EVENTS CAUSING TOP EVENT = 1
```

```

----- Pr[TOP EVENT]   ERRORS/WARNINGS -----
1.76344E-09

```

```
20 OUT OF THE 58 VECTORS ANALYZED CAUSED TOP EVENT
SMALLEST NUMBER OF EVENTS CAUSING TOP EVENT = 3
3.270 SECS. CPU TIME UTILIZED
```

```
88?
```

Example 5. This example illustrates the use of the program to investigate the sensitivity of a fault tree to a parameter.

FTC V2.4 NASA Langley Research Center

```

1? read ex5

2. V = 0 TO 1 BY .1;
3. G11: V;
4. G12: V/2;
5. G13: SQRT(V);
6. G14: 1 - V;
7. G15: 1 - V/2;
8. G16: 1 - SQRT(V);
9. G17: V*(1-V);
10. G18: (1-V)*(1-V*V)*(1-V**3);
11.
12. A21: AND(G11,G12,G13);
13. A22: OR(G12,G13);
14. A23: XOR(G13,G14);
15. A24: 3OF(G15,G16,G17,G18);
16. A25: AND(G16,G17);
17.
18. B31: OR(A21,A25);
19. B32: INV(A22);
20. B33: OR(A24,A22);
21.
22. C41: AND(B31,A23);
23. C42: AND(B33,B32,A22);
24.
25. TOP: 2OF(C41,C42,A25,A23);
26.

27? run

```

MODEL FILE = EX5.TRE

FTC V2.4 14-OCT-1988 09:00:34

V	Pr[TOP EVENT]	ERRORS/WARNINGS
0.00000E+00	0.00000E+00	.. WARNING: EVENT PROB. = 0 or 1
1.00000E-01	3.99655E-02	
2.00000E-01	4.86548E-02	
3.00000E-01	5.23680E-02	
4.00000E-01	6.02219E-02	
5.00000E-01	7.75698E-02	
6.00000E-01	1.09150E-01	
7.00000E-01	1.60335E-01	
8.00000E-01	2.37549E-01	
9.00000E-01	3.48165E-01	
1.00000E+00	5.00000E-01	.. WARNING: EVENT PROB. = 0 or 1

12 OUT OF THE 141 VECTORS ANALYZED CAUSED TOP EVENT
SMALLEST NUMBER OF EVENTS CAUSING TOP EVENT = 3
7.810 SECS. CPU TIME UTILIZED

```

28? plot
29? disp copy

```

(* See Figure 3 *)

The warning messages alert the user that some of the events in the model occur with probability 0 and/or 1. The FTC program gives correct answers for such models; however, sometimes this is indicative of a user error.

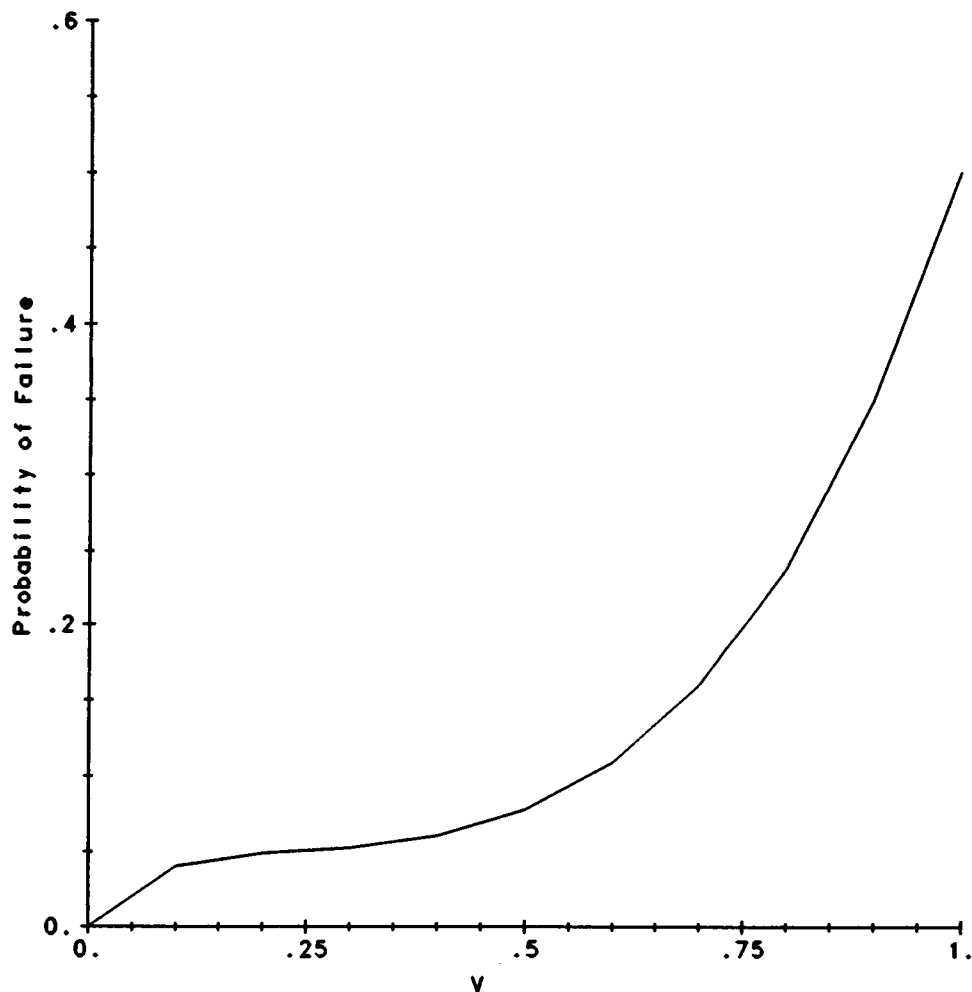


Figure 3. Plot generated by example 4.

Example 6. This example illustrates the use of the program to investigate the effect of variations in mission time on the probability of system failure. The model is run twice, once with LIST = 1 and once with LIST = 2, to illustrate the LIST options.

FTC V2.4 NASA Langley Research Center

```

1? read EX6

2. TIME = 0.1 TO* 1000 BY 10;
3.
4. E1 -> 1E-4;
5. E2 -> 2E-4;
6. E3 -> 3E-4;
7. E4 -> 4E-4;
8. E5 -> 5E-4;
9. E6 -> 6E-4;
10.
11. G1: AND(E1,E2);
12. G2: AND(E3,E4);
13. G3: AND(E5,E6);
14.
15. TOP: OR(G1,G2,G3);

```

16? list=1;
17? run

MODEL FILE = EX6.TRE

FTC V2.4 14-OCT-1988 09:00:45

TIME	Pr[TOP EVENT]	ERRORS/WARNINGS
1.00000E-01	4.39979E-09	
1.00000E+00	4.39790E-07	
1.00000E+01	4.37902E-05	
1.00000E+02	4.19197E-03	
1.00000E+03	2.60782E-01	

37 OUT OF THE 64 VECTORS ANALYZED CAUSED TOP EVENT
SMALLEST NUMBER OF EVENTS CAUSING TOP EVENT = 2
2.160 SECS. CPU TIME UTILIZED

18? read EX6

19. TIME = 0.1 TO* 1000 BY 10;
20.
21. E1 -> 1E-4;
22. E2 -> 2E-4;
23. E3 -> 3E-4;
24. E4 -> 4E-4;
25. E5 -> 5E-4;
26. E6 -> 6E-4;
27.
28. G1: AND(E1,E2);
29. G2: AND(E3,E4);
30. G3: AND(E5,E6);
31.
32. TOP: OR(G1,G2,G3);
33? list=2;
34? run

MODEL FILE = EX6.TRE

FTC V2.4 14-OCT-1988 09:00:49

TIME	Pr[TOP EVENT]	FV	NV	ERRORS/WARNINGS
1.00000E-01	4.3997900519443E-09	16	43	
1.00000E+00	4.3978987686773E-07	16	43	
1.00000E+01	4.3790174680721E-05	31	58	
1.00000E+02	4.1919723847700E-03	37	64	
1.00000E+03	2.6078188495868E-01	37	64	

37 OUT OF THE 64 VECTORS ANALYZED CAUSED TOP EVENT
SMALLEST NUMBER OF EVENTS CAUSING TOP EVENT = 2
2.160 SECS. CPU TIME UTILIZED

The columns FV and NV report the number of vectors which caused system failure and the total number of vectors processed, respectively.

Mathematical Foundations of the FTC Program

In this section, a new algorithm for solving fault trees is presented along with a proof that the algorithm produces an answer within a user-specified level of accuracy. Then, a more efficient form of the algorithm which is implemented in FTC is given and the proof is revised.

Preliminaries

The FTC solution technique relies upon three basic model assumptions:

1. System components, or basic events, fail independently,
2. Components are either failed or operational; an "in-between" state does not exist.
3. The system is either failed or operational; no "in-between" state exists.

In the following discussion, the fault tree is generalized to have n basic events and a probability of occurrence associated with each. Basic events will be referred to as *components* and a probability of *failure* will be attributed to each of the components in the system. We will assume that the components have been numbered in order of decreasing failure probability.

Let E_i represent the event that the i th component fails

Then E_i occurs with the i th largest probability and $P(E_i) \geq P(E_j)$ whenever $i \leq j$. Next, define an indicator variable e_i (for $1 \leq i \leq n$) as follows:

$$e_i = \begin{cases} 0 & \text{if event } i \text{ does not occur (i.e., component } i \text{ does not fail)} \\ 1 & \text{if event } i \text{ occurs (i.e., component } i \text{ fails)} \end{cases}$$

It is possible to enumerate all combinations of components failed and components operational describing the different possible states of the system. Each system state can be represented by an n dimensional "binary" vector v composed of 1's and 0's, where 1 indicates that the component has failed and 0 indicates that the component has not failed:

$$v = (e_1 \ e_2 \ e_3 \ \dots \ e_n)$$

Then, we have by independence of the n basic events,

$$P(v) = \prod_{i=1}^n (e_i P(E_i) + [1 - P(E_i)] [1 - e_i])$$

The event that all n components fail, for example, would be represented by the vector

$$(1 \ 1 \ 1 \ \dots \ 1).$$

A system composed of four basic events would generate the following binary vectors:

- | | | | |
|--------------|--------------|---------------|---------------|
| 1. (0 0 0 0) | 5. (0 0 0 1) | 9. (0 1 1 0) | 13. (1 1 0 1) |
| 2. (1 0 0 0) | 6. (1 1 0 0) | 10. (0 1 0 1) | 14. (1 0 1 1) |
| 3. (0 1 0 0) | 7. (1 0 1 0) | 11. (0 0 1 1) | 15. (0 1 1 1) |
| 4. (0 0 1 0) | 8. (1 0 0 1) | 12. (1 1 1 0) | 16. (1 1 1 1) |

For an n -component system there are 2^n possible binary vectors representing 2^n distinct system states. The j th system state is denoted by v_j , and its associated probability by $P(v_j)$. Note that the j th binary vector can be written in terms of the variables e_{ij} , ($e_{1j} \ e_{2j} \ \dots \ e_{nj}$), and that the probability of the j th system state is

$$P(v_j) = \prod_{i=1}^n (e_{ij} P(E_i) + [1 - P(E_i)] [1 - e_{ij}])$$

The sample space S is the set of all possible system states denoted by the 2^n binary vectors. By definition,

$$P(S) = 1.$$

Because the components are either failed or not failed, the 2^n binary vectors exhaustively describe all possible system states. Therefore,

$$P(v_1 + v_2 + \dots + v_{2^n}) = P(S) = 1.$$

Clearly, the system can be in only one state at any given time, indicating that the system states are mutually exclusive. By definition,

$$P(v_i v_j) = 0 \quad (\text{for every } i \text{ and } j \neq i)$$

and for the 2^n mutually exclusive system states,

$$P(v_1 + v_2 + \dots + v_{2^n}) = P(v_1) + P(v_2) + \dots + P(v_{2^n}).$$

To calculate the probability of system failure, the sample space S composed of 2^n system states is divided into two subsets, where one subset contains all the system failure states and the other subset contains all states where the system is operational. Because the system must be either failed or operational, these two subsets are clearly exhaustive and mutually exclusive. Furthermore the system states composing each of these two subsets are mutually exclusive, and the probability of either subset can be found by summing the appropriate individual system state probabilities. Therefore, the calculation of the total probability of system failure by simply summing the probabilities of the system configurations that represent system failure is exact.

The Basic Approach

The program reorders the basic events in the binary vector in order of decreasing probability. (A mapping from the new order to the user-defined names is maintained.)

The program then generates binary vectors in an orderly fashion, starting with binary vector

$$(0\ 0\ 0\ \dots\ 0_n).$$

The following is the sequence generated for a four-event tree:

```
(0 0 0 0)
(1 0 0 0)
(0 1 0 0)
(0 0 1 0)
(0 0 0 1)
(1 1 0 0)
(1 0 1 0)
(1 0 0 1)
(0 1 1 0)
(0 1 0 1)
(0 0 1 1)
(1 1 1 0)
(1 1 0 1)
(1 0 1 1)
(0 1 1 1)
(1 1 1 1)
```

Vectors are checked through the user-defined system tree. For each binary vector found to represent a system-fail configuration, its probability is added to a running total of binary vector probabilities, where all vector probabilities in the sum represent system-fail configurations. As shown above, the total number of binary vectors to be checked through the system tree is 2^n . For systems with many components, the number of fault vectors to check through the system tree can be very large. A simple and effective pruning technique has been developed to reduce the total number of fault vectors checked through the system tree. The pruning technique will not affect the FTC program answer; it will reduce run time and improve efficiency.

The FTC program uses two types of pruning. The first type of pruning is based on the concept that the probability of occurrence of a vector with a large number of 1's is typically much less than the probability of occurrence of a vector with just a few number of 1's. The first type of pruning is more easily explained by letting $\Psi(v) = \text{Number of 1's in vector } v$. The program determines a level γ such that all vectors v with $\Psi(v) \geq \gamma$ contribute a negligible amount to the final answer. The program continues processing vectors until the first vector v with $\Psi(v) \geq \gamma$ is encountered. Since vectors are processed in increasing order of $\Psi(v)$, only the negligible vectors are not processed. The second type of pruning skips all vectors with probabilities less than a computed threshold. The error bounding theorems presented subsequently demonstrate that both pruning methods together are conservative with respect to a user-specified level of accuracy.

Notation

E_i	event that i th component fails
e_i	= 1 if event E_i occurs, otherwise 0
p_i	probability event E_i occurs (i.e., $P(e_i = 1)$)
$(e_1 e_2 e_3 \dots e_n)$	system state vector
V_k	vector with k 1's followed by $n-k$ 0's: $(\underbrace{1\ 1\ 1\ \dots\ 1}_k\ 0\ 0\ \dots\ 0)$
$P(V_k)$	probability that vector V_k occurs
$\Psi(v)$	number of 1's in vector
N_f	number of vectors which caused system failure thus far
N_a	total number of vectors which have been analyzed thus far
S_f	current estimate of system failure probability
d	user-specified number of digits accuracy
n	number of basic events in fault tree

The FTC Algorithm

In this section, two versions of the fault-tree solution algorithm are presented. First, the basic algorithm is discussed along with a proof of its error bound. Second, a slightly more efficient algorithm (which is used in FTC V2) is given along with its error bound. In order to facilitate the presentation of the FTC algorithm, some special notation is defined. Let V_k represent a vector with k 1's followed by $n-k$ 0's:

$$V_k = (\underbrace{1\ 1\ 1\ \dots\ 1}_k\ 0\ 0\ \dots\ 0)$$

and let

$$p_i = \text{Probability event } E_i \text{ occurs (i.e., } P(e_i = 1))$$

$P(V_k)$ = Probability that vector V_k occurs

$$= \prod_{i=1}^k p_i \prod_{j=k+1}^n (1 - p_j)$$

$\Psi(v)$ = number of 1's in a vector

N_f = number of vectors which caused system failure thus far

N_a = total number of vectors which have been analyzed thus far

S_f = current estimate of system failure probability

d = user-specified number digits accuracy

n = number of basic events in fault tree

Basic Algorithm

The program processes vectors until the first vector that causes system failure is encountered. Using the probability of failure of this vector as an estimate of the probability of system failure P_{sys} , the program calculates parameters γ and Ω . (The details of this calculation are explained subsequently.) The program continues processing vectors until a vector v is encountered with γ 1's or more (i.e., $\Psi(v) \geq \gamma$.) A cumulative sum of the probabilities of occurrence of the vectors which cause system failure is stored in the variable S_f . The program also skips over all vectors whose probability of occurrence is less than Ω . The algorithm is as follows:

$v = (0 \ 0 \ 0 \ \dots \ 0)$; $S_f = 0$; $N_f = 0$

REPEAT

 IF v causes system failure THEN

$N_f = N_f + 1$

 calculate $P(v)$

$S_f = S_f + P(v)$

 IF $N_f = 1$ THEN

 CALL CUTLEVEL(S_f, γ, Ω)

 ENDIF

 IF $P(v) < \Omega$ THEN CALL PRUNER(v) ENDIF

 ENDIF

 IF $(\Psi(v) \geq \gamma)$ THEN GO TO 100 ENDIF

 LAST $v = v$

$v = \text{NEXTvector}$

UNTIL LAST $v = (1 \ 1 \ \dots \ 1)$

100: $P_{\text{sys}} = S_f$

SUBROUTINE CUTLEVEL(S_f, γ, Ω)

$k = n$

 SUM = 0.0

 BOUND = $S_f[0.5 \times 10^{-d}]$ (* d = desired # digits accuracy*)

 REPEAT

 ERRGAM = SUM

 SUM = SUM + $\binom{n}{k} P(V_k)$

$k = k - 1$

 UNTIL SUM > BOUND

$$\gamma = k + 2$$

$$C_\gamma = \sum_{k=0}^{\gamma-1} \binom{n}{k}$$

$$\Omega = \frac{\text{BOUND} - \text{ERRGAM}}{(C_\gamma - N_a)}$$

END

SUBROUTINE PRUNER(v)

IF v is of FORM $(x_1 \ x_2 \ \dots \ x_j \ \underbrace{0 \ 1 \ 1 \ \dots \ 1}_{k} \ \underbrace{0 \ 0 \ \dots \ 0}_z)$ THEN

$$v = (x_1 \ x_2 \ \dots \ x_j \ \underbrace{0 \ 0 \ 0 \ \dots \ 0}_{z+1} \ \underbrace{0 \ 1 \ 1 \ \dots \ 1 \ 1}_k)$$

ENDIF

END

Subroutine CUTLEVEL calculates the parameters γ and Ω . The calculation of γ is based on the fact that the probability $P(V_k) \geq$ the probability of occurrence of any other vector with k 1's. Since there are $\binom{n}{k}$ vectors with k 1's, the total error in ignoring the contribution of vectors with γ or more 1's is

$$\prod_{k=\gamma}^n \binom{n}{k} P(V_k)$$

CUTLEVEL determines the smallest value for γ such that this error is negligible. CUTLEVEL also determines a probability Ω that is small enough such that all vectors whose probability of occurrence is less than it can be ignored.

Subroutine PRUNER just moves the rightmost cluster of 1-bits (i.e., a contiguous section of 1's) all the way to the right. For example, suppose that FTC is being used to solve a model with 6 basic events. The following is the order (down the columns) that the vectors would be generated (starting at $(1 \ 1 \ 0 \ 0 \ 0 \ 0)$):

(1 1 0 0 0 0)	(0 0 1 1 0 0)	(1 1 0 0 0 1)	(0 1 1 0 0 1)
(1 0 1 0 0 0)	(0 0 1 0 1 0)	(1 0 1 1 0 0)	(0 1 0 1 0 1)
(1 0 0 1 0 0)	(0 0 1 0 0 1)	(1 0 1 0 1 0)	(0 1 0 0 1 1)
(1 0 0 0 1 0)	(0 0 0 1 1 0)	(1 0 1 0 0 1)	(0 0 1 1 1 0)
(1 0 0 0 0 1)	(0 0 0 1 0 1)	(1 0 0 1 1 0)	
(0 1 1 0 0 0)	(0 0 0 0 1 1)	(1 0 0 1 0 1)	.
(0 1 0 1 0 0)	(1 1 1 0 0 0)	(1 0 0 0 1 1)	.
(0 1 0 0 1 0)	(1 1 0 1 0 0)	(0 1 1 1 0 0)	.
(0 1 0 0 0 1)	(1 1 0 0 1 0)	(0 1 1 0 1 0)	

Subroutine PRUNER

skips from $(1 \ 0 \ 0 \ 1 \ 0 \ 0)$ to $(1 \ 0 \ 0 \ 0 \ 0 \ 1)$
 $(1 \ 0 \ 0 \ 1 \ 1 \ 0)$ to $(1 \ 0 \ 0 \ 0 \ 1 \ 1)$
 $(0 \ 0 \ 1 \ 0 \ 1 \ 0)$ to $(0 \ 0 \ 1 \ 0 \ 0 \ 1)$

It is demonstrated subsequently that subroutine PRUNER only skips over vectors with probability of occurrence less than the argument v . Since PRUNER is only called when $P(v) < \Omega$, PRUNER only skips vectors with probability less than Ω .

Function NEXTvector generates the next vector Nv given the current vector v by the following algorithm:

FUNCTION NEXTvector(v)

IF right-most bit of v is a 0 THEN

FROM $v = (x_1 x_2 \dots x_j 1 0 \dots 0)$ generate

$$Nv = (x_1 x_2 \dots x_j 0 1 \dots 0)$$

ELSE IF all 1-bits are all the way to the right THEN

$Nv = (1 1 \dots 1 0 \dots 0)$ with 1 more 1-bit than v

ELSE

k = number of 1-bits in rightmost cluster of 1's in v

z = number of 0-bits preceding the rightmost cluster of 1-bits in v

FROM $v = (x_1 x_2 \dots x_j \underbrace{1 0 \dots 0}_z \underbrace{1 1 \dots 1}_k)$ generate

$$Nv = (x_1 x_2 \dots x_j \underbrace{0 1 1 \dots 1}_{k+1} \underbrace{0 \dots 0}_{z-1})$$

ENDIF

NEXTvector = Nv

END

Thus, NEXTvector generates the next vector by moving the rightmost 1-bit to the right until it reaches the last position. If all the 1-bits are all the way to the right, then a new vector is created with one more 1-bit than the previous vector. Otherwise, it then alters the order of the lower bits and then continues to move the rightmost bit to the right.

Justification for the Basic Algorithm

In this section a bound on the error due to pruning is derived. Several lemmas are first given which simplify the error bound analysis.

Lemma 1. If $p_\alpha > 0$ and $1 > p_\beta \geq 0$ then

$$p_\alpha \geq p_\beta \text{ implies } \frac{p_\beta(1-p_\alpha)}{p_\alpha(1-p_\beta)} \leq 1$$

Proof:

$$p_\alpha \geq p_\beta$$

$$1 - p_\beta \geq 1 - p_\alpha$$

$$p_\alpha(1 - p_\beta) \geq p_\beta(1 - p_\alpha)$$

$$1 \geq \frac{p_\beta(1 - p_\alpha)}{p_\alpha(1 - p_\beta)}$$

Lemma 2. If $\Psi(v) = k$ then $P(V_k) \geq P(v)$.

Proof: Suppose that v differs from V_k in only one place; that is, suppose that the α 1-bit in V_k is located at β in v :

$$\begin{aligned}
P(v) &= \prod_{\substack{i=1 \\ i \neq \alpha}}^k p_i \prod_{\substack{j=k+1 \\ j \neq \beta}}^n (1-p_j) p_\beta (1-p_\alpha) \\
&= \frac{\prod_{i=1}^k p_i \prod_{j=k+1}^n (1-p_j)}{p_\alpha (1-p_\beta)} p_\beta (1-p_\alpha) \\
&= \frac{p_\beta (1-p_\alpha)}{p_\alpha (1-p_\beta)} P(V_k) \\
&\leq P(V_k) \quad (\text{By lemma 1})
\end{aligned}$$

The argument is easily generalized for more than one displaced 1-bit.

Lemma 3. Subroutine PRUNER only skips vectors with probability $< \Omega$.

Proof: As shown earlier, the program generates vectors in a specific order. If v is of form $(x_1 x_2 \dots x_j \underbrace{011\dots 1}_{k} \underbrace{1100\dots 0}_z)$, then PRUNER generates the next vector Nv as follows:

$$Nv = (x_1 x_2 \dots x_j \underbrace{000\dots 0}_{z+1} \underbrace{011\dots 11}_k)$$

From the function NEXTvector it can be seen that all the vectors between v and Nv are of the following form:

$$Nv = (x_1 x_2 \dots x_j 0 y_1 \dots y_{n-j-1})$$

where k of the y -bits are 1's and z of the y -bits are 0's. (This can be seen by noting that NEXTvector has three branches in the IF statement. The first and third branches only generate vectors of the form above. The second branch is only executed when all the 1's are already all the way to the right, so it does not apply.) It is obvious that any vector with the Nv -form can be derived from v by merely moving bits to the right. Thus, subroutine PRUNER skips vectors which can be generated from the current one by merely moving 1-bits to the right.

To see that moving a 1-bit to the right results in a vector with lower probability, let v represent the original vector and v' represent the new vector. Let α be the location of the 1-bit which will be moved to location β which is to the right of α (i.e., $\beta > \alpha$):

$$\begin{aligned}
P(v) &= \prod_{i=1}^n (e_i P(E_i) + [1 - P(E_i)][1 - e_i]) \\
P(v) &= \prod_{\substack{i=1 \\ i \neq \alpha \\ i \neq \beta}}^n (e_i P(E_i) + [1 - P(E_i)][1 - e_i]) P(E_\alpha) [1 - P(E_\beta)] \\
P(v') &= \prod_{\substack{i=1 \\ i \neq \alpha \\ i \neq \beta}}^n (e_i P(E_i) + [1 - P(E_i)][1 - e_i]) P(E_\beta) [1 - P(E_\alpha)]
\end{aligned}$$

Thus,

$$\begin{aligned}
P(v') &= P(v) \frac{P(E_\beta)[1 - P(E_\alpha)]}{P(E_\alpha)[1 - P(E_\beta)]} \\
&\leq P(v)
\end{aligned}$$

The last step follows from $P(E_\alpha) \geq P(E_\beta)$ and lemma 1. Since PRUNER is only called when the initial vector v has probability of occurrence $< \Omega$, all the vectors skipped have probability $< \Omega$.

Pruning error bound. The algorithm utilizes two forms of pruning. The first form throws away all vectors v such that $\Psi(v) > \gamma$. A bound on the error which can result from such pruning, ERR_γ , is easily obtained. From lemma 2 and the fact that there are $\binom{n}{k}$ vectors with k 1-bits, we have

$$\sum_{v \in S} P(v) \leq \text{ERR}_\gamma = \sum_{k=\gamma}^n \binom{n}{k} P(V_k)$$

where $S = \{v | \Psi(v) > \gamma\}$.
But γ is chosen such that

$$\sum_{k=\gamma}^n \binom{n}{k} P(V_k) < S_f [0.5 \times 10^{-d}]$$

(In fact γ is the smallest positive integer which satisfies this equation.) Thus,

$$\text{ERR}_\gamma < S_f [0.5 \times 10^{-d}]$$

The second form of pruning eliminates small vectors before $\Psi(v)$ reaches γ . First Ω is either zero in which case no additional pruning is done or

$$\Omega = \frac{\text{BOUND} - \text{ERRGAM}}{C_\gamma - N_a}$$

where

$$C_\gamma = \sum_{k=0}^{\gamma-1} \binom{n}{k}$$

$$\text{BOUND} = S_f [0.5 \times 10^{-d}]$$

$$\text{ERRGAM} = \sum_{k=\gamma}^n \binom{n}{k} P(V_k) = \text{ERR}_\gamma$$

Since $C_\gamma =$ Total number of vectors with $\Psi(v) < \gamma$, and N_a is the number of vectors already analyzed, $(C_\gamma - N_a)$ is an upper bound on the number of vectors which can be thrown away. Lemma 3 demonstrated that subroutine PRUNER only skips vectors with probability of occurrence $< \Omega$. Thus, a bound on the error due to the second type of pruning, ERR_Ω , is

$$\begin{aligned} \text{ERR}_\Omega &= (C_\gamma - N_a) * \Omega \\ &= \text{BOUND} - \text{ERRGAM} \\ &= \text{BOUND} - \text{ERR}_\gamma \end{aligned}$$

The total error due to pruning ϵ is bounded by $\text{ERR}_\gamma + \text{ERR}_\Omega$:

$$\epsilon \leq \text{ERR}_\gamma + \text{ERR}_\Omega = \text{BOUND} = S_f [0.5 \times 10^{-d}]$$

Since $S_f \leq P_{\text{sys}}$,

$$\epsilon < P_{\text{sys}} [0.5 \times 10^{-d}]$$

or

$$\frac{\epsilon}{P_{\text{sys}}} < 0.5 \times 10^{-d}$$

Thus, there are d digits accuracy.

More Efficient Algorithm

The FTC V2.1 algorithm is more efficient than the basic algorithm presented in the previous section. The increased efficiency is obtained by recalculating γ and Ω periodically. Since both γ and Ω are functions of P_{sys} and N_a , improved values can be obtained as the computation proceeds. However, this must be done carefully in order to guarantee d digits accuracy.

```
v = (0 0 0 ... 0); Sf = 0; Nf = 0; Na = 0;
NEXTCUT = 1; PRUNED_SOME = FALSE; ERROMG = 0;
TOTAL_THROWN_AWAY = 0;
REPEAT
  IF v causes system failure THEN
    Nf = Nf + 1
    calculate P(v)
    Sf = Sf + P(v)
    IF Nf = NEXTCUT THEN
      CALL CUTLEVEL(Sf, γ, Ω)
      NEXTCUT = NEXTCUT*10
    ENDIF
    IF P(v) < Ω THEN CALL PRUNER(v, Ω) ENDIF
  ENDIF
  IF (Ψ(v) ≥ γ) THEN GO TO 100 ENDIF
  LASTv = v
  v = NEXT vector
UNTIL LASTv = (1 1 ... 1)
100: Psys = Sf

FUNCTION CUTLEVEL(Sf, γ, Ω)
  IF PRUNED_SOME THEN
    TOTAL_THROWN_AWAY = TOTAL_THROWN_AWAY + ERROMG
  ENDIF
  k = n
  PF = 0.0
  SUM = 0.0
  BOUND = Sf[0.5 × 10-d] { d = desired # digits accuracy }
  IF BOUND ≥ TOTAL_THROWN_AWAY THEN
    ROOMLEFT = BOUND - TOTAL_THROWN_AWAY
  ELSE
    RETURN
  ENDIF
  REPEAT
    ERRGAM = SUM
    SUM = SUM +  $\binom{n}{k}$  P(Vk)
    k = k - 1
  UNTIL SUM > ROOMLEFT
  γ = k + 2
```

$$C_\gamma = \sum_{k=0}^{\gamma-1} \binom{n}{k}$$

$$\Omega = \frac{\text{ROOMLEFT} - \text{ERRGAM}}{C_\gamma - N_a}$$

$$\text{ERROMG} = (C_\gamma - N_a) * \Omega$$

END
SUBROUTINE PRUNER(v, Ω)
PRUNED_SOME = TRUE
IF v is of FORM (0 0 ... 0 1 1 ... 1 1 0 0 ...) THEN
 $v = (0 0 \dots 0 1 1 1 1 1)$
ELSE
SKIP_A_FEW(v)
ENDIF
END

Derivation of Error Bound for the More Efficient Algorithm

The only difference between the original algorithm and the more efficient algorithm is that the more efficient algorithm recalculates γ and Ω as the computation proceeds. The variable BOUND represents the total amount of probability that can be thrown away and still have d digits accuracy. In the basic algorithm, the difference between BOUND and ERR_γ represents the amount of extra probability that is not used by the γ -type pruning. This extra probability enabled the additional Ω -type pruning. When the program recalculates γ and Ω , it is necessary that the error due to previous pruning be taken into consideration. (Clearly, since γ -type pruning terminates the program, only the previous Ω -type pruning must be considered.) To illustrate, let γ' and Ω' represent the first values and γ and Ω represent the new values to be calculated. Also let $\text{BOUND}(i)$ be the value of BOUND used for the i th calculation:

$$\text{BOUND}(i) = S_f * [0.5 \times 10^{-d}]$$

Clearly,

$$\text{BOUND} = P_{\text{sys}}[0.5 \times 10^{-d}] \geq \text{BOUND}(i) \quad (\text{for all } i)$$

If no pruning was done prior to the second calculation, then the original method would work fine. But, if pruning was done prior to the second calculation of γ and Ω , then there are three sources of error which must be accommodated. The following must hold:

$$\text{ERR}_\gamma + \text{ERR}_\Omega + \text{ERR}_{\Omega'} \leq P_{\text{sys}}[0.5 \times 10^{-d}]$$

In general, if $\text{ERR}_\Omega^{(i)}$ represents the error bound on the amount of pruning due to the i th value of Ω , say $\Omega^{(i)}$, and there were τ calculations of Ω (i.e., τ calls to subroutine CUTLEVEL), then the following must hold:

$$\text{ERR}_\gamma + \sum_{i=1}^{\tau} \text{ERR}_\Omega^{(i)} \leq P_{\text{sys}}[0.5 \times 10^{-d}]$$

The program maintains this summation $\sum \text{ERR}_\Omega^{(i)}$ in the variable TOTAL_THROWN_AWAY(τ). The parameter $\Omega^{(i)}$ is calculated as follows:

$$\begin{aligned} \Omega^{(i)} &= \frac{\text{ROOMLEFT} - \text{ERR}_\gamma}{C_\gamma - N_a} \\ &= \frac{\text{BOUND}(i) - \text{TOTAL_THROWN_AWAY}(i-1) - \text{ERR}_\gamma}{C_\gamma - N_a} \end{aligned}$$

Since $\text{BOUND}(i) \leq \text{BOUND}$:

$$\Omega^{(i)} \leq \frac{\text{BOUND} - \text{ERR}_\gamma - \text{TOTAL_THROWN_AWAY}(i-1)}{C_\gamma - N_a}$$

Thus, a bound on the error due to $\Omega^{(i)}$ -pruning denoted $\text{ERR}_\Omega^{(i)}$ is:

$$\text{ERR}_\Omega^{(i)} = \Omega^{(i)}(C_\gamma - N_a) = [\text{BOUND} - \text{ERR}_\gamma - \text{TOTAL_THROWN_AWAY}(i-1)]$$

Rearranging terms:

$$\text{ERR}_\Omega^{(i)} + \text{TOTAL_THROWN_AWAY}(i-1) = \text{BOUND} - \text{ERR}_\gamma$$

or

$$\text{ERR}_\gamma + \sum_{j=1}^i \text{ERR}_\Omega^{(j)} = \text{BOUND}$$

This equation is true for all i , in particular $i = \tau$:

$$\text{ERR}_\gamma + \sum_{j=1}^{\tau} \text{ERR}_\Omega^{(j)} = \text{BOUND}$$

The left side is the sum of the error bounds for all the pruning that could possibly take place. Thus, the total error in pruning ϵ is bounded:

$$\epsilon \leq \text{BOUND} = P_{\text{sys}}[0.5 \times 10^{-d}]$$

or

$$\frac{\epsilon}{P_{\text{sys}}} < 0.5 \times 10^{-d}$$

Thus, there are d digits accuracy.

Domain of Efficiency

Since the program processes vectors in order of increasing number of basic-event failures, it is efficient for systems which have a failure mode involving a small number of basic events. Even very large fault trees can be solved, if they possess dominant failure vectors containing only a few failed basic events. This type of fault tree is found in models of fault-tolerant computer system architectures. For example, if the system is constructed using threefold redundancy, the dominant failure vectors will contain only two basic-event failures.

Concluding Remarks

A new algorithm for solving fault trees has been developed along with an error bound on its accuracy. This algorithm is the mathematical basis for a new reliability analysis program called the Fault-Tree Compiler (FTC). The solution algorithm is especially efficient for the types of fault trees used to model fault-tolerant system architectures. The FTC program has four major strengths: (1) the input language is easy to understand, (2) automatic sensitivity analysis is allowed by varying a parameter over a range of values, (3) the answer provided by the program is precise to within a user-specified level of accuracy, and (4) the program uses a pruning technique which significantly reduces the execution time of the program. Additionally, the use of the hierarchical fault-tree capability can reduce model complexity.

NASA Langley Research Center
Hampton, VA 23665-5225
May 2, 1989

Appendix

Error Messages

Error and warning messages are listed in alphabetical order, with messages beginning with a symbol (i.e., =,], ;) listed at the end.

ALREADY DEFINED AS A GLOBAL CONSTANT—The value defined has been defined previously as a global constant.

ALREADY DEFINED AS A GATE OUTPUT OR EVENT—The value has been defined previously as a gate output or event.

ALREADY DEFINED AS A LOCAL CONSTANT—The value has been previously defined as a local constant.

ALREADY DEFINED AS A RESERVED WORD—The value defined is an FTC reserved word.

ALREADY DEFINED AS A SUBTREE—The value defined has previously been defined as a subtree title.

ARGUMENT TO EXP FUNCTION MUST BE $< 8.80289E+01$ —The argument to the EXP function is too large.

ARGUMENT TO LN OR SQRT FUNCTION MUST BE > 0 —The LN and SQRT functions require positive arguments.

ARGUMENT TO STANDARD FUNCTION MISSING—No argument was supplied for a standard function.

COMMA EXPECTED—Syntax error; a comma is needed.

CONSTANT EXPECTED—Syntax error; a constant is expected.

DIVISION BY ZERO NOT ALLOWED—A division by 0 was encountered when evaluating the expression.

EVENT PROBABILITY > 1 —The event probability was evaluated to a value greater than 1.

EVENT PROBABILITY < 0 —The event probability was evaluated to a value greater than 1.

EXP FUNCTION OVERFLOW—The argument to the EXP function is too large. The value of the argument must be less than $8.80289E+01$.

EXPRESSION CANNOT CONTAIN THE VARIABLE—The variable cannot be defined in terms of itself.

EXPRESSION OVERFLOW—The value of the expression caused arithmetic overflow.

FILE NAME EXPECTED—Syntax error; the file name is missing.

FILE NAME TOO LONG—File names must be 80 or less characters.

IDENTIFIER EXPECTED—Syntax error; the file name is missing.

IDENTIFIER NOT DEFINED—The identifier entered has not yet been defined.

ILLEGAL CHARACTER—The character used is not recognized by the FTC program.

ILLEGAL LN OR SQRT ARGUMENT—The LN and SQRT functions require positive arguments.

ILLEGAL STATEMENT—The command word is unknown to the program.

ILLEGAL NUMBER OF INPUTS TO GATE—The AND and OR gates may have an arbitrary number of inputs; however, the INVERT gate must have only one input, the EXCLUSIVE OR gate must have two inputs, and the M OF N gate must have the number of inputs such that $N - M \geq 0$.

INPUT ALREADY DEFINED AS A VARIABLE—The gate or variable defined in the statement has already been defined globally as a variable.

INPUT LINE TOO LONG—The command line exceeds the 100-character limit.

INTEGER EXPECTED—Syntax error; an integer is expected.

INV GATE MUST HAVE ONLY 1 INPUT—Only one input is allowed for the INVERT gate.

MUST BE IN "READ" MODE—The INPUT command can be used only in a file processed by a READ command.

NOT A VALID EVENT—Events used as gate inputs must be previously defined as a basic event or the output from a previous gate.

NO GATES IN FAULT TREE—The fault tree contains no gates.

NUMBER TOO LONG—Only 15 digits/characters allowed per number.

ONLY 1 VARIABLE ALLOWED—Only one variable can be defined per complete fault tree.

REAL EXPECTED—A floating point number is expected here.

SEMICOLON EXPECTED—Syntax error; a semicolon is needed.

SUB-EXPRESSION TOO LARGE, i.e. > 1.70000E+38—An overflow condition was encountered when evaluating the expression.

SUBTREE RESULT NOT FOUND—The fault tree was unable to calculate subtree top event probabilities. Check for syntax errors in the subtrees.

TOP NOT REACHABLE—No combination of events led to the top event in the system tree.

UNKNOWN GATE TYPE—Verify that the gate type is AND, OR, INV, XOR, or m OF < >. See the section "Gate Definition" of this paper for more information.

VARIABLE MUST BE DEFINED AT GLOBAL LEVEL—Variables may NOT be defined within subtrees; variables must be defined globally.

VMS FILE NOT FOUND—The file indicated on the READ command is not present on the disk. (Note: make sure your default directory is correct.)

WARNING: EVENT PROBABILITY = 1—The event probability was evaluated to a value equal to 1. Although the answer given by the program is still correct the input may not be what was intended.

WARNING: EVENT PROBABILITY = 0—The event probability was evaluated to a value equal to 0. Although the answer given by the program is still correct the input may not be what was intended.

*** WARNING: VARIABLE CHANGED TO A CONSTANT! PREVIOUS EVENTS MAY BE WRONG—If previous basic events have been defined using a variable and the variable name is changed, inconsistencies may appear in the results.

*** WARNING: SYNTAX ERRORS PRESENT BEFORE RUN—Syntax errors were present during the model description process. They may or may not have been corrected prior to the run.

*** WARNING: RUN-TIME PROCESSING ERRORS—Computation overflow occurred during execution.

*** WARNING: REMAINDER ON INPUT LINE IGNORED—The information on the rest of the input line is disregarded.

= EXPECTED—Syntax error; the = operator is needed.

] EXPECTED—A right bracket is missing in the expression.

< EXPECTED—Syntax error; the < symbol is needed.

) EXPECTED—A right parenthesis is missing in the expression.

(EXPECTED—A left parenthesis is missing in the expression.

References

1. Lee, W. S.; Grosh, D. L.; Tillman, F. A.; and Lie, C. H.: Fault Tree Analysis, Methods, and Applications—A Review. *IEEE Trans. Reliab.*, vol. R-34, no. 3, Aug. 1985, pp. 194-203.
2. Bavuso, S. J.; and Petersen, P. L.: *CARE III Model Overview and User's Guide (First Revision)*. NASA TM-86404, 1985.
3. Henley, Ernest J.; and Kumamoto, Hiromitsu: *Reliability Engineering and Risk Assessment*. Prentice-Hall, Inc., c.1981.
4. Cohen, G. C.; Lee, C. W.; Brock, L. D.; and Allen, J. G.: *Design/Validation Concept for an Integrated Airframe/Propulsion Control System Architecture (IAPSA II)*. NASA CR-178084, 1986.



Report Documentation Page

1. Report No. NASA TP-2915	2. Government Accession No.	3. Recipient's Catalog No.	
4. Title and Subtitle The Fault Tree Compiler (FTC) <i>Program and Mathematics</i>		5. Report Date July 1989	6. Performing Organization Code
		8. Performing Organization Report No. L-16529	
7. Author(s) Ricky W. Butler and Anna L. Martensen		10. Work Unit No. 505-66-21-01	11. Contract or Grant No.
9. Performing Organization Name and Address NASA Langley Research Center Hampton, VA 23665-5225		13. Type of Report and Period Covered Technical Paper	
		14. Sponsoring Agency Code	
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, DC 20546-0001		15. Supplementary Notes Ricky W. Butler: Langley Research Center, Hampton, Virginia. Anna L. Martensen: PRC Kentron, Inc., Aerospace Technologies Division, Hampton, Virginia.	
16. Abstract The Fault-Tree Compiler program is a new reliability tool used to predict the top-event probability for a fault tree. Five different gate types are allowed in the fault tree: AND, OR, EXCLUSIVE OR, INVERT, and m OF n gates. The high-level input language is easy to understand and use when describing the system tree. In addition, the use of the hierarchical fault tree capability can simplify the tree description and decrease program execution time. The current solution technique provides an answer precisely (within the limits of double precision floating point arithmetic) within a user-specified number of digits accuracy. The user may vary one failure rate or failure probability over a range of values and plot the results for sensitivity analyses. The solution technique is implemented in FORTRAN; the remaining program code is implemented in Pascal. The program is written to run on a Digital Equipment Corporation (DEC) VAX computer with the VMS operation system.			
17. Key Words (Suggested by Authors(s)) Fault tree Reliability analysis Reliability modeling Fault tolerance		18. Distribution Statement Unclassified—Unlimited Subject Category 62	
19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of Pages 38	22. Price A03