

# NASA Technical Memorandum 89098

## The Fault-Tree Compiler

(NASA-TM-89098) THE FAULT-TREE COMPILER  
(NASA) 40 p CSCI 09B

N87-20765

G3/61 Unclas  
45405

Anna L. Martensen  
Ricky W. Butler

**JANUARY 1987**

**NASA**

National Aeronautics and  
Space Administration

Langley Research Center  
Hampton, Virginia 23665-5225

## CONTENTS

INTRODUCTION .....	1
Fault Tree Construction .....	2
THE FTC USER INTERFACE .....	6
Basic Program Concept .....	6
Fault Tree Definition Syntax .....	6
Hierarchical Fault Trees .....	12
FTC Commands .....	15
FTC Graphics .....	18
EXAMPLE FTC SESSIONS .....	19
Outline of a Typical Session .....	19
Examples .....	19
CONCLUDING REMARKS .....	27
REFERENCES .....	28
APPENDIX A .....	A-1
Theory .....	A-1
Solution Technique .....	A-4
APPENDIX B - ERROR MESSAGES .....	B-1

## 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 wide-spread support and is often used as a failure analysis tool by reliability experts. Though conceptually simple, especially for those with a knowledge of basic circuit logic, the fault tree can be a powerful tool. In its basic form, the fault tree has limitations; however, recent advances in the tree notation have often overcome these shortcomings. Even the basic fault tree, though, can be useful in preliminary design analysis. The goal of the Fault Tree Compiler (FTC) program is to provide the user with a tool which can readily describe even the largest fault tree and, once given the tree description and event probabilities, can precisely calculate the probability of the top event in the tree. Automatic sensitivity analysis can also be performed, providing the user with a powerful design analysis capability.

The motivation for the development of the Fault Tree Compiler began with the observation that the Computer Aided Reliability Estimation (CARE III) program (ref. 3) 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 newer, faster solution was developed and implemented in FTC. To that was added a new front-end with a high-level language description of the fault tree. The improved solver and the Fault Tree Compiler's input language and sensitivity analysis capabilities provide a powerful fault tree solver. In short:

- 1) The FTC program has a simple yet powerful input language, which is easily mastered.
- 2) Automatic sensitivity analysis is provided.
- 3) The mathematical solution technique is exact to the five digits in the solution(s).

- 4) A hierarchical capability is provided which can reduce effort and run 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 technique has been implemented in FORTRAN, and the remaining code is Pascal. The program is designed to run on the Digital Corporation VAX computer operating under the VMS operating system.

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

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 small 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 FTC fault tree and is referred to as a common mode event. A useful feature of the FTC program is its ability to handle these common mode events.

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 appears in the example (to describe the hydraulic failure and the top event), boxes could have appeared above any event, basic or non-basic. Logic gates delineate the causal relations which ultimately result in the top event. In the FTC program the following gates are allowed:

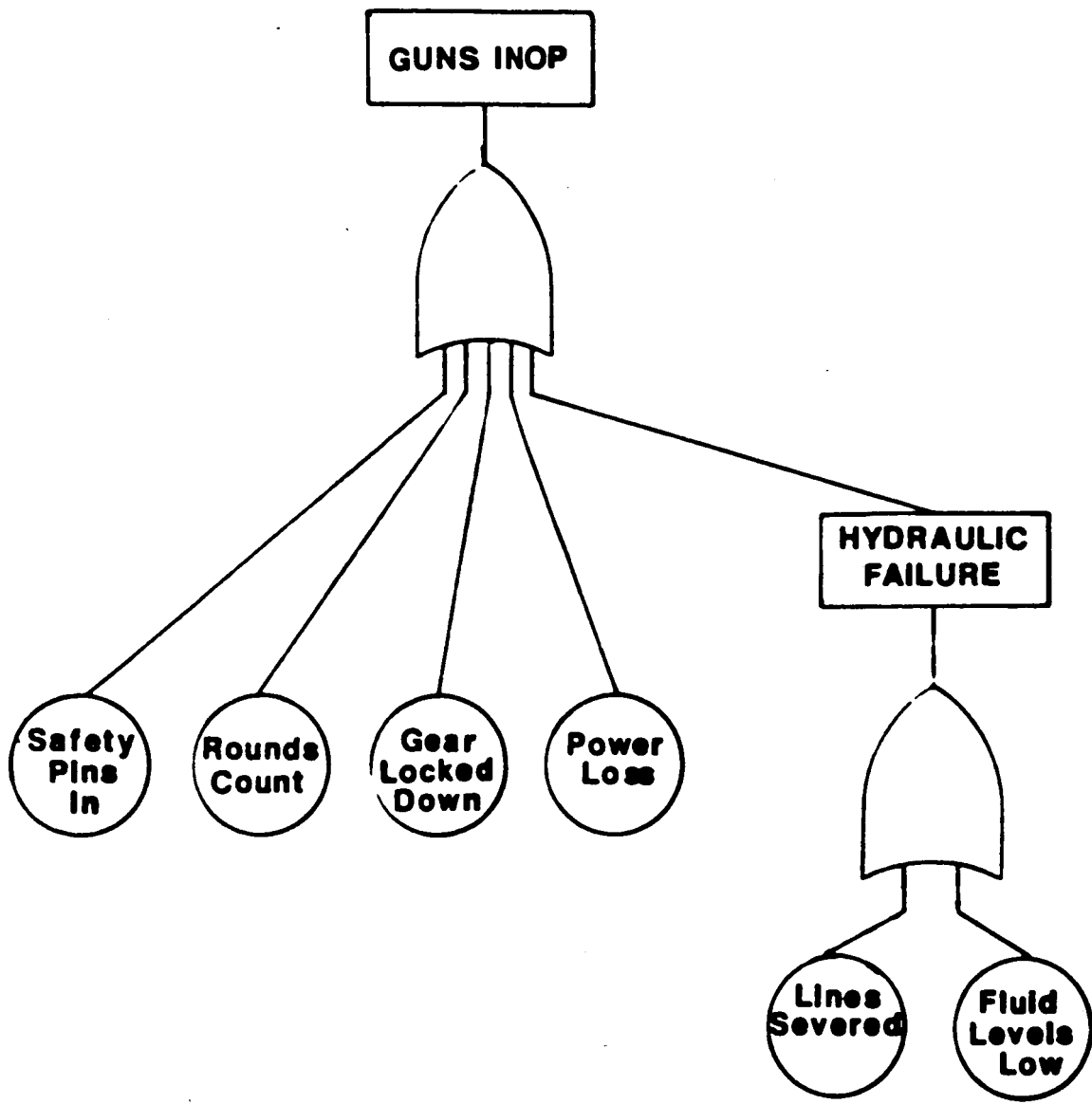
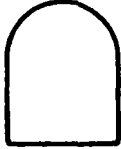
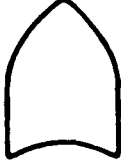
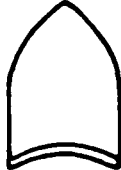
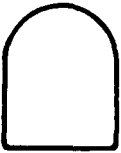



Figure 1

Gate Symbol	Gate Name	Result of Gate
	AND gate	The output occurs when all input events occur simultaneously. An arbitrary number of input events are allowed.
	OR gate	The output occurs when one or more of the input events occur. An arbitrary number of input events are allowed.
	EXCLUSIVE OR gate	The output occurs when one but not both of the input events occur. Only two input events are allowed.
	M of N gate	The output occurs when m of the n inputs occur. The number of inputs must be greater than or equal to m.
	INVERT gate	This gate performs a complement (applying DeMorgan's Law) on the input. The number of inputs must equal one.

Fault trees are typically constructed by starting with the top event (usually an 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. The following short example illustrates the top down process by which a fault tree is constructed:

The McDonnell Douglas F-15C fighter has three primary weapons 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 pre-flight 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.

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 pre-flight 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 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 can not 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 then 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 "FTC Fault Tree Definition Syntax" section of this paper.

For more information on fault trees, Reference [2] is recommended. The next section discusses the user interface for the FTC program and example FTC sessions follow. Concluding remarks, Appendix A—The Solution Technique, and Appendix B—Error Messages complete the paper.

## THE FTC USER INTERFACE

### Basic Program Concept

The user of the FTC program must define his fault tree using a simple language. This language will be discussed in detail in this section. There are two basic statements in the 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 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.

### FTC 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 are floating point numbers. The Pascal 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 by the READ command described below), 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 is given for this variable. The FTC system will compute the system reliability as a function of this variable. If the system is run in graphics mode (to be described later), then a plot of this function will 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 example. Thus, suppose POINTS = 4, then

Geometric:

```
XV = 1 TO* 1000;
```

where the values of XV used would be 1, 10, 100, and 1000.

Arithmetic:

```
XV = 1 TO+ 1000;
```

where the values of XV used would be 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. By following the above syntax with BY "inc", the value of POINTS is automatically set such that the value is varied by adding or multiplying the specified amount. For example,

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

sets POINTS equal to 5 and the values of V used would be 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 used would be 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 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 evaluates to a number between 0 and 1.

Alternately, the user can specify the rate of an event. This can be accomplished 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 using the following formula:

$$\text{Prob[ event ]} = 1.0 - \text{EXP}( - \text{<rate-expression>*TIME } )$$

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,

$$F(t) = 1 - e^{-\lambda t}.$$

Gate Definition - Once all of 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> : { OR  
                INV  
                XOR  
                AND } ( <input>, <input>, ... );
```

or

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

The <output-id> is the name of the (non-basic) event which is the output of the gate. The type of gate is indicated by the reserved words OR, AND, INV, XOR or OF as follows

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

Any number of input events may be included within the parentheses. The following gate definition statements are valid:

```
G1: AND(X, Y, Z);
G2: OR(A1, A2, A3);
PLANE_CRASH: 2 OF (ENGINE1_FAILS, ENGINE2_FAILS, HYDRAULIC_FAILURE);
DESPAIR: OR(GET_THE_FLU, MOTHER_IN_LAW_STAYS_2_WEEKS,
            MEET_DAUGHTERS_NEW_BOYFRIEND);
```

### 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);
```

Unfortunately, in order to integrate these sections into one fault tree, the subsystem has to be replicated four times (each replicate with different names):

```
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 the above model, four events in the main tree are given the probability of the subsystem, i.e. 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. The effect of the change in the failure probability of a component in the previous model could be investigated 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(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);



## FTC Commands

Two types of commands have been included in the user interface. The first type of command is initiated by a reserved word:

EXIT        INPUT        PLOT        READ        RUN        SHOW

The second type of command is invoked by setting one of the special constants

CARE3        ECHO        LIST        POINTS        TIME

equal to one of its pre-defined values.

**EXIT** - The EXIT command causes termination of the FTC program.

**INPUT** - This 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?
```

and a new constant LVAL is created that is equal to the value input by the user. Several constants can be interactively defined using 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."

**READ** - A sequence of FTC statements may be read from a disk file. The following interactive command reads FTC statements from a disk file named SIFT.MOD:

```
READ SIFT.MOD;
```

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

**RUN** - After a fault tree has been fully described 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 specified. 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:

RUN(10);           — change POINTS to 10 then run.  
RUN(,0);           — change LIST to 0 and run.  
RUN(20,1);         — change POINTS to 20 and LIST to 1 then run.

**SHOW** - The value of an identifier may be displayed by the following command:

SHOW ALPHA;

---

**CARE3** - If set equal to 1 the program will generate a file containing the fault tree in the CARE III syntax. The default value of 0 specifies that no CARE III file be written. The name of the generated file is CARE3.TRE. 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 insert 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 entitled "Example FTC Sessions.")

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

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

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

**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. 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.

### 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 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:

XLOG        plot x-axis using logarithmic scale  
YLOG        plot y-axis using logarithmic scale  
XYLOG       plot both x- and y-axes using logarithmic scales  
NOLO        plot x- and y-axes with normal scaling

XLEN=5.0    set x-axis length to 5.0 in.  
YLEN=8.0    set y-axis length to 8.0 in.  
XMIN=2.0    set x-origin 2 in. from left side of screen  
YMIN=2.0    set y-origin 2 in. above bottom of screen

The PLOTINIT and PLOT+ commands are used 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

3-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 using the PLOT+ command. The parameters of this command are identical to the PLOT command. The only difference is that the data is saved, so it can be displayed in conjunction with subsequent run data. 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 ten 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, etc., as desired. Altering the value of a constant identifier does not affect any transitions entered previously even though they were defined using 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.

##### Examples

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

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

\$ FTC

FTC V1.0     NASA Langley Research Center

```
1? lambda = 1e-4;
2? X: 1.0 - exp( -lambda*time);
3? Y: 1.0 - exp( -lamda*time);
      ^ IDENTIFIER NOT DEFINED
3? Y: 1.0 - exp( -lambda*time);
4? top: or(x,y);
5? run
```

Pr[ TOP EVENT ]

-----  
1.99800E-03

\*\*\* WARNING: SYNTAX ERRORS PRESENT BEFORE RUN  
0.420 SECS. CPU TIME UTILIZED

6? 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 APPENDIX B.

Example 2 - 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 of the actuator subsystems 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 V1.0 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

20: (ACT\_SYS1\_FAILS,ACT\_SYS2\_FAILS,ACT\_SYS3\_FAILS,ACT\_SYS4\_FAILS);

21: TOP: OR(GATE1, HYDRAULIC\_FAILURE, ELECTRONICS\_FAILURE);

22? run

	Pr[ACT_SYS_]
-----	4.93777E-03
	Pr[TOP EVENT]
-----	6.46555E-04

1.260 SECS. CPU TIME UTILIZED

23? exit

Example 3 - 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. 1.)

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 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, (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 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 PCS data, IRADC data, or computer failure. The loss of data can be due to data source failure or 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);

```

The above model was available in file IAPSA.MOD prior to the following interactive session.

\$ FTC

FTC V1.0 NASA Langley Research Center

1? echo = 0  
2? read IAPSA

45? run

Pr{TOP EVENT}

-----  
1.76344E-09

12.230 SECS. CPU TIME UTILIZED

**Example 4** - This example illustrates the use of the program to investigate the sensitivity of a fault tree to a parameter.

\$ FTC

FTC V1.0 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? run

V	Pr[TOP EVENT]
0.00000E+00	0.00000E+00
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

4.090 SECS. CPU TIME UTILIZED

27? PLOT

28? DISP COPY

(See Figure 2)

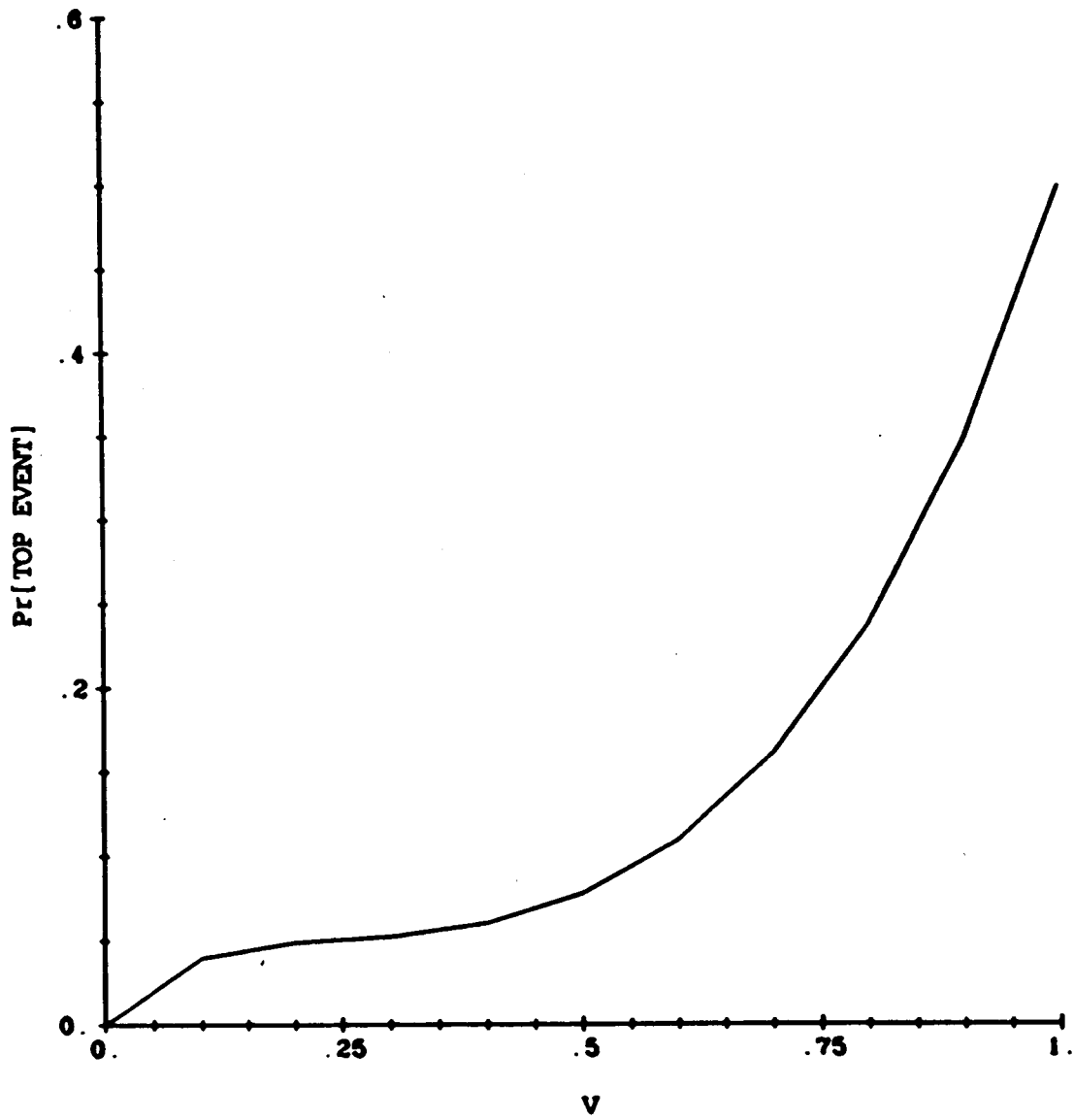


Figure 2

## CONCLUDING REMARKS

The Fault-Tree Compiler is a new reliability analysis program based on combinatorial mathematics. The program has three major strengths: 1) the input language is easy to understand and learn, 2) automatic sensitivity analysis is allowed by varying a parameter over a range of values, and 3) the answer provided by the program is precise. Additionally, the use of the hierarchical fault tree capability can reduce model complexity. The program can be used for an exact analysis, or a simple analysis of any system of interest.

## REFERENCES

1. 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 Contractor Report 178084, June 1986.
2. Henley, E.J.; and Kumamoto, H.: Reliability Engineering and Risk Assessment, Prentice-Hall, Inc., 1981.
3. Bavuso, S.J.; and Petersen, P.L.: CARE III Model Overview and User's Guide, NASA Technical Memorandum 86404, April 1985.

## APPENDIX A

### Theory

The Fault Tree Compiler program 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.

Figure A1 illustrates a representative fault tree. 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.

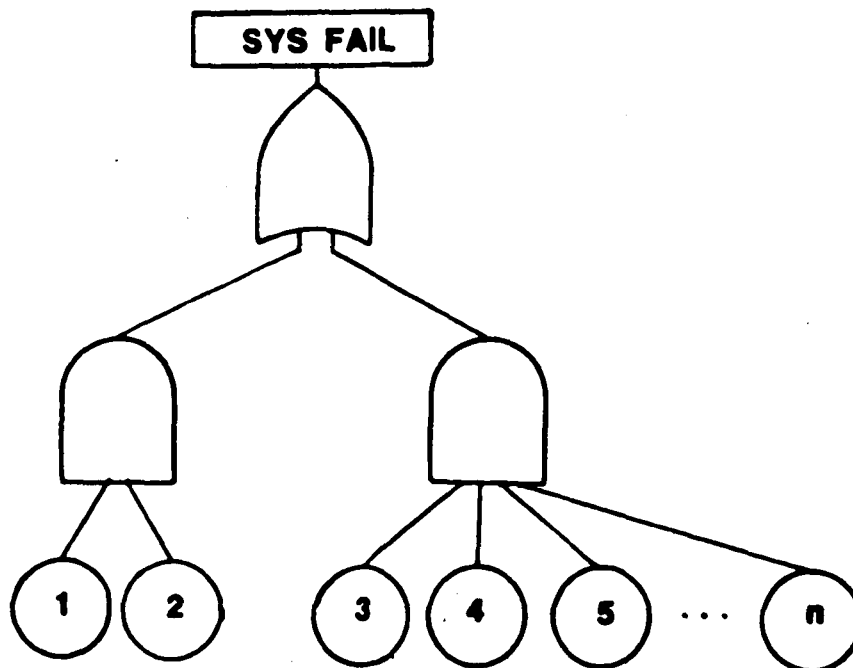


Figure A1.

Let

$A_{i0}$  represent the event component  $A_i$  has not failed

$A_{i1}$  represent the event component  $A_i$  has failed.

By defining the variable  $v_i$  as

$$v_i = \begin{cases} 0 & \text{if system component } i \text{ has not failed} \\ 1 & \text{if system component } i \text{ has failed,} \end{cases} \quad 1 \leq i \leq n$$

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

$$\begin{aligned} P(A_{1v_1} A_{2v_2} \dots A_{nv_n}) &= P(A_{1v_1}) P(A_{2v_2}) \dots P(A_{nv_n}). \\ &= \prod_{i=1}^n P(A_{iv_i}) \\ &= \prod_{i=1}^n \left[ (1-v_i) P(A_{i0}) + P(A_{i1}) v_i \right]. \end{aligned}$$

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 composed of 1's and 0's, where 1 indicates that the component has failed and 0 indicates that the component has not failed. 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)

Clearly, 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  $s_j$ ,



and its associated probability by  $P(s_j)$ . Note that the  $j$ th binary vector can be written in terms of the variables  $v_{1j}, (v_{1j} v_{2j} \dots v_{nj})$ , and that the probability of the  $j$ th system state is

$$P(s_j) = \prod_{i=1}^n \left[ (1-v_{ij})P(A_{i0}) + P(A_{i1})v_{ij} \right].$$

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,

$$P(s_1 + s_2 + \dots + s_{2^n}) = P(S) = 1.$$

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

$$P(s_i s_j) = 0 \text{ for every } i \text{ and } j \neq i$$

and, for the  $2^n$  mutually exclusive system states,

$$P(s_1 + s_2 + \dots + s_{2^n}) = P(s_1) + P(s_2) + \dots + P(s_{2^n}).$$

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 states representing system failure, and the other subset contains all states that represent system 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, calculating the total probability of system failure by simply summing the probabilities of the system configurations that represent system failure is exact.

## The Solution Technique

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

$$(0\ 0\ 0\ \cdots\ 0_n).$$

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.

Consider a system composed of ten highly reliable components. The probability that any one component fails is low; the probability that all components fail is much, much lower. It is often true that the latter probability, when summed with the former probability, will not affect the answer within five or even ten or more significant digits. A threshold value can be established, and binary vectors with probabilities less than the threshold may be disregarded.

The FTC program establishes a threshold by first calculating a "weight" function. A worst case scenario is solved, where it is assumed that  $2^n - 1$  vectors, all of value  $10^{-k}$ , are to be pruned. The sum of all the values must be less than or equal to  $0.5 \times 10^{-d}$  ( $d$  = the number of significant digits in the FTC answer). The function

$$(2^n - 1) \times 10^{-k} \leq 0.5 \times 10^{-d}$$

is solved for the smallest integer value of  $k$  that satisfies the equation. Once a binary vector representing a system-fail configuration is found, its

probability is multiplied by the weight function value and the threshold is established. The threshold is used to reduce the total number of fault vectors checked through the system tree by eliminating those vectors that would not influence the FTC answer even if they do represent a system-fail configuration. For large systems with many basic events, the time savings can be substantial.

The solution algorithm can be roughly summarized as follows:

- 1) Calculate the weight function (shown above).
- 2) Rank the probabilities of failure. Two fault vectors are maintained, IVEC and JVEC. IVEC represents the basic events in the order in which they were entered in the model. JVEC, on the other hand, ranks the probabilities of failure and orders the basic events from highest  $P(f)$  to lowest. The need for two vector representations will become clear shortly.
- 3) Select the  $(0\ 0\ 0\ \dots\ 0_n)$  fault vector.
- 4) Using the IVEC representation, check to see if the fault vector represents a system fail state.
- 5) If the fault vector represents a system fail state, then

If this is the first system fail fault vector to be found, calculate the probability of the set of events represented by the fault vector. Then multiply this value by the weight function to obtain the threshold value, CUTOFF. Lastly, initialize the total probability of system failure to the fault vector probability.

If this is not the first system fail fault vector, simply add its probability of occurrence to the total probability of failure.
- 6) Increment the JVEC fault vector, and generate the equivalent IVEC fault vector. By incrementing the JVEC fault vector and then converting to IVEC, the fault vectors are generated in patterns of decreasing probabilities.
- 7) If at least one fault vector has been found representing a system fail state, calculate the new fault vector's probability of occurrence, PFVOCC.

If a PFVOCC value is calculated, it is compared to the CUTOFF value. If  $PFVOCC < CUTOFF$ , then the next JVEC fault vector with a PFVOCC greater than CUTOFF is generated. If no such vector exists, or the vector  $(1\ 1\ 1\ \dots\ 1_n)$  has been reached, the program jumps to step 9.
- 8) Loop to step 4.
- 9) The total probability of failure is passed to the calling program.

Fault tree solvers exist; it is felt that the FTC program is superior in its tree input language and specification syntax. The use of subtrees to reduce run time and aid in the analysis of large trees is especially powerful. The probability of the top event in the system tree is exact to five significant digits. It is, however, the user's responsibility to assure that basic events in the tree are independent, and that the tree is semantically correct.

The fault tree is often satisfactory for design analysis and system failure studies. Though the fault tree methodology may have limitations for the detailed analysis of complex systems containing dependencies, it can be very useful in preliminary design reviews.

## APPENDIX B

### 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 GREATER THAN 1 - 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 OR 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  $\langle \rangle$ . See the "Gate Definition" section 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: 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.

Standard Bibliographic Page

1. Report No. NASA TM-89098		2. Government Accession No.		3. Recipient's Catalog No.	
4. Title and Subtitle The Fault-Tree Compiler				5. Report Date January 1987	
				6. Performing Organization Code 505-66-21-01	
7. Author(s) Anna L. Martensen and Ricky W. Butler				8. Performing Organization Report No.	
				10. Work Unit No.	
9. Performing Organization Name and Address NASA Langley Research Center Hampton, VA 23665-5225				11. Contract or Grant No.	
				13. Type of Report and Period Covered Technical Memorandum	
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, DC 20546-0001				14. Sponsoring Agency Code	
15. Supplementary Notes Anna L. Martensen, PRC Kentron, Inc., Hampton, Virginia. Ricky W. Butler, Langley Research Center, 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 precise (within the limits of double precision floating point arithmetic) to the five digits in the answer. 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 Corporation VAX 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  Star Category 61		
19. Security Classif.(of this report) Unclassified		20. Security Classif.(of this page) Unclassified		21. No. of Pages 39	22. Price A03

For sale by the National Technical Information Service, Springfield, Virginia 22161