PERTT, an interactive microcomputer program to perform fault tree analysis

by

Karen Deborah Daniels Ford

A Thesis Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major: Nuclear Engineering

Iowa State University
Ames, Iowa

1982

TABLE OF CONTENTS

TABLE OF FIGURES

CHAPTER I. INTRODUCTION

Statement of Purpose

The reliability analysis of systems characteristic of many modern technologically complex facilities such as nuclear power plants, high performance aircraft and satellites requires special procedures coupled with the reliability and memory of high-speed computers. One of the better-known procedures is called "fault tree analysis", so named because the graphic representation of the system in terms of operating components and subservient inputs such as fluids, powers and signals, resembles a tree, with branches and twigs. The analogy continues as small branches join to form larger branches just as the successful functioning of small components permits the successful operation of larger or dependent components.

As the numbers of components and inputs increase, and as the description of how the components function and as probabilities associated with various functional modes are assigned, computer capabilities become essential.

Many programs have been developed to perform qualitative and quantitative fault tree analysis, but problems exist in using them. Data entry for these programs is clumsy, for the programs are not interactive. If one wishes to change a part of a fault tree, one must fumble through a stack of cards to find the corresponding card, retype the card or cards, and insert them, being careful to keep them in the proper place in the deck. The programs are slow, both in terms of real time and computer time. Since existing fault tree analysis programs use an array format, the number of inputs to a gate is limited, often inconveniently, by prior programming decisions. Since the programs use a good deal of computer time and require considerable memory space, they are expensive to run and, as a result, are usually run at night, or during other low-demand times. There are occasions when such a delay is not acceptable, such as when the user must iterate to determine optimal values.

For these reasons, existing codes are inconvenient for design studies, although they are useful for evaluating the reliability and performance of already existing and fixed systems. The PERTT (PASCAL Editor and Recursive Tree Traversal) code has been developed for a design environment. It is written for an APPLE II microcomputer, but can easily be transferred to any other interactive system in which PASCAL is implemented. It consists of two programs: one to construct a fault tree and save it on disk (henceforth called the editor) and a second program to use this fault tree to find the cut sets and their failure probabilities as well as the total system failure probabilities. The editor is a "menu-driven" program which makes adding, deleting, or changing an event a simple, understandable procedure. Thus, the impact of a change in a subsystem upon the performance of the system may be easily computed and evaluated during the design process.

The versatility of this code is demonstrated by the fact that it was originally intended to apply this program to compute the failure probability of the cryogenic system of a conceptual tokamak fusion reactor. A search of the literature revealed that essential component reliability data were unavailable, since many of the components have not been built or tested. Thus, the analysis could not be completed. Instead, the equivalent or related probabilities were taken from the fission reactor industry and the PERTT programs used to establish component reliability goals for designers in order that a fusion reactor might meet safety standards comparable to those of the fission industry.

Equipment Needed[1,2]

The following equipment is required to use PERTT:

APPLE II^{TM} computer with UCSD PASCAL^{TM} language system

One or more 5 1/4" flexible disk drives

APPLE 3: disk containing the file SYSTEM.APPLE

FAULTREE: disk containing the APPLE system files:

```
SYSTEM. MISCINFO
SYSTEM.PASCAL
SYSTEM.LIBRARY
```

and the PERTT code files:

```
FAULTTREE.CODE
EDITOR.CODE
```

---

[1]APPLE II
c Apple Computer, Inc., 1978

[2]UCSD PASCAL
c Regents of the University of California, 1979

CHAPTER II. EXPLANATION OF TERMS

Fault Tree Analysis

A fault tree is a model of a system, which facilitates the qualitative exploration of the origin of a specified undesirable event and the quantitative determination of the probability that the event will occur. The fault tree consists of "gates" which show the relationships of initiating events which cause or contribute to a subsequent event. The faults, or events, can be associated with mechanical failure, human error, or natural phenomena. A fault tree is a convenient tool for qualitative and quantitative analysis of safety systems and the probabilities for their failure.

A bottom, or primary event is an event whose antecedent events have not been further explored, either because more information is not necessary or because more information is not available. If the fault tree is to be analyzed quantitatively, probabilities must be furnished for all of the bottom events. An intermediate event occurs because of the action of other events acting through logic gates. Therefore, probabilities for these events are determined by the bottom event probabilities and the logic which connects them with the intermediate event.

A fault tree contains a combination of one or more types of gates. An AND gate shows that the output event occurs if and only if all of the input events occur. An OR gate shows that the output event occurs if and only if one or more of the input events occur. A k-out-of-n gate is similar to an n-input AND gate, but only k out of the n input faults must occur in order to trigger the output fault. A k-out-of-n gate can be represented by a system of AND and OR gates. A NOT gate, or complemented event has only one input. The output event of a NOT gate occurs if the input event has not occurred.

A cut set of a fault tree is a group of events which lead to system failure. A minimal cut set is a subset of a cut set which consists of a "smallest group" of these events. System failure (the top event) occurs if and only if each event in at least one minimal cut set occurs.

Components share a common location if no barrier exists to insulate one of them from an event that can affect both of them. A common link is a dependency, such as a common energy source or a common water supply, between components. A common mode, or common cause failure, is a failure that occurs due to the failure of a common link.

Programming Terms

PASCAL is a computer language that was developed in 1971 by Professor Nicklaus Wirth for scientific and commercial programming [6]. Many implementations of standard PASCAL exist, each with minor differences from the others. The implementation used for the PERTT code developed in this thesis is UCSD PASCAL, developed at the University of California, San Diego (UCSD), for use with micro-computers.

A file, or disk file, is a collection of information stored on disk. It may be accessed by the use of a file name. A file may contain a program, data for a program, or any other information. An editor is a computer program that allows the user to create or change a disk file. One can create an editor which can create a general file or a specific type of file. The editor referred to in this thesis is the fault tree editor which comprises one of two programs in the PERTT package. An editor often contains "menus", which are simply lists of options from which the user may choose. The computer is said to "prompt" the user when it prints a question on the screen for which the user must supply an answer via the keyboard. An interactive program is a program for which the user must supply input data during the execution.

Reading a variable consists of transferring its value from the disk file or the console to the memory of the computer. A Boolean variable is a variable that may assume one of only two values: TRUE or FALSE. A string variable, or string, is a variable identified by a sequence of alphabetic characters and/or numbers. The characters of a string have no individual significance.

Recursive programming

A recursive program is a program that calls itself, and thus nests an operation within another iteration of the same operation. Recursive programs can provide elegant solutions to certain classes of programming problems. For example, the factorial function can be implemented by the following PASCAL program:

```
FUNCTION FACT (N:INTEGER):INTEGER;
     BEGIN
        If N=1 then FACT :=1;
        ELSE FACT :=N*FACT(N-1);
     END;
```

This program loops around, multiplying by (N-1) during each loop until 1 is reached. Of course, this particular function can also be implemented using looping, but recursion in applications such as tree traversal becomes very convenient.

Pointers and linked lists

In many computer languages such as FORTRAN, arrays are used to handle ordered lists of data. However, arrays are not always the most efficient means of handling this type of problem. For example, in order to insert a number between the first two numbers of the array:

```
3.687
7.292
4.189
```

one must first check to see that the dimension of the array is at least 4. Then, the third number must be moved to the fourth position, the second number must be moved to the third position, and the new number must be inserted into the second position. When long lists are involved, this can be a very time-consuming process. If the data set is occasionally small, an array is an inefficient use of space, because the programmer must initially establish and reserve the space required for the largest data set to be handled. This is accomplished by means of a dimension statement and insures that in many cases, much of the reserved space will not be used and will not be available for any other program.

PASCAL has implemented a device which helps to alleviate these problems. It represents each entry in the list as a record of one or more values, plus a pointer referring explicitly to the item which follows it in the list, as shown in Figure 1.

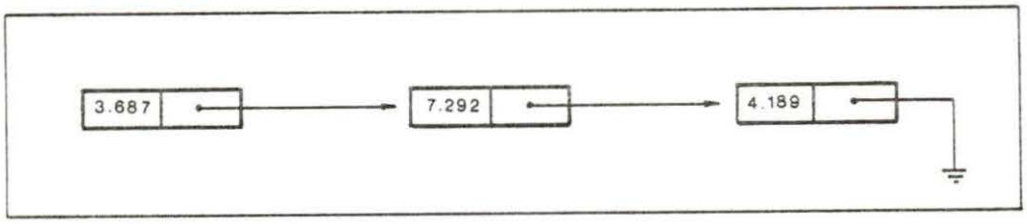

Figure 1. Representation of a simple linked list

Thus, considering the previous example, to insert a number, for example 2.395, into the second position of this list, a new record is created, with a pointer to the record 7.292. Then, the pointer from 3.687 is redirected to the new entry, as shown in Figure 2.

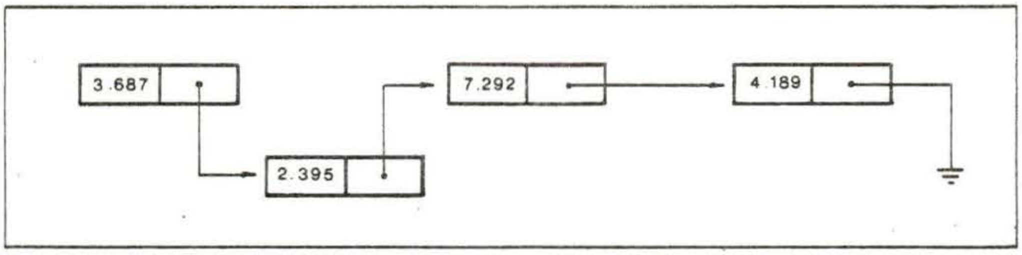

Figure 2. Adding a record to a linked list

This type of structure is known as a linked list. The linked list is exactly as long as required, but the size of an array must be fixed in advance. When PASCAL pointers are used to implement the linked list, the records are stored at any location in memory not taken up by program. The records are accessed by means of a pointer, called the top pointer, which points to the linked list. Thus, no program changes are required to take advantage of the larger amount of memory in a larger, faster system.

CHAPTER III. REVIEW OF PREVIOUS WORK

Fault Tree Analysis Codes

Fault tree analysis requires the preparation or existence of a fault tree. A fault tree is a symbolic representation of actual hardware which is involved in originating, transmitting, processing and using fluid flow, information flow, heat flow or the flow of instructions to accomplish some purpose. Fault tree analysis is the study of the chain of events which may prevent the accomplishment of the desired purpose and of the relative probabilities that individual faults and chains of faults may occur.

The PREP and KITT codes [25] were released in 1970, and were the first computer codes developed for the evaluation of fault trees. The codes are written in FORTRAN for the IBM 360 computer. The minimal cut sets of the fault tree are found by the PREP code and the event probabilities are determined by the KITT code. The cut sets are determined by one of two means. A deterministic method of fault tree analysis in which all possible combinations of failure events are successively tried to determine which combinations cause the system to fail, is provided by the COMBO option. For large fault trees, this requires a great deal of computer time. FATE, the second option, uses Monte Carlo simulation to find the most probable minimal cut sets. Unfortunately, FATE is not guaranteed to find all of the minimal sets. The cut sets required by the KITT code to find time-dependent reliability information are provided by the PREP code.

As might be expected, modifications to the original codes have been developed in recent years. The MOCUS code [24] was designed in 1972 to provide input to the KITT code, because deterministic testing (COMBO) was found to be too slow and Monte Carlo simulation (FATE) was not guaranteed to find all of the cut sets. It uses a "top down" logic to successively replace each gate in the tree by its inputs until each gate has been replaced by bottom events. It is written in FORTRAN for the IBM 360 computer. TREEL and MICSUP, [14] developed in 1975, serve the same purpose as MOCUS. They use essentially the same logic, but work from the bottom up.

The ALLCUTS code [23] was written in 1975, and uses a top-down algorithm similar to MOCUS. It was written in FORTRAN for a CDC 6600 computer, and uses 34700 words of memory. It is more memory-efficient than many other codes. In addition to providing the minimal cut sets, ALLCUTS provides probability calculation as an option.

The WAM code is used to compute minimal cut sets, and the BAM code calculates the associated system unavailability. It performs these calculations by setting up a truth table with all possible combinations of events. The WAM-BAM codes [23] were written in FORTRAM for the CDC 7600 computer, and use 61440 words of memory.

The PL-MOD code [13] performs quantitative and qualitative analysis of a fault tree by "modularizing" it. The modularization process divides the tree into independent subtrees and works not from the cut sets but from a description of the fault tree. It uses the list processing features of PL-1 to perform the modularization. PL-MOD is unique in that it can handle k-out-of-n gates and complemented events.

The COMCAN code [16] performs common cause failure analysis on the fault tree cut sets. It identifies various possible common cause failures, and finds the common locations in the fault tree. It sets up a susceptibility fault tree for each common location and finds the cut sets for each of these trees. It is written in FORTRAN for the IBM-360 computer.

Superconducting Magnets for Fusion Reactors

These codes or the PERTT code developed in this thesis can be used to analyze fault trees corresponding to any system. The system chosen as a relevant example is the one involved with the reliability of superconducting magnets for fusion reactors. Fusion reactors are considered to be strong contenders for supplying energy beginning in the 21st century. The interacting particles must be given enough energy for the reaction to occur and must be kept in close proximity long enough for the reaction to occur. One method for keeping the particles together is called magnetic confinement and requires the production and maintenance of a number of different but related magnetic fields.

A tokamak, [20] or magnetic confinement fusion reactor involves two systems of magnets: the steady-state toroidal magnet system and the pulsed poloidal magnet system. The magnet system is shown in Figure 3. The toroidal coils are D-shaped for mechanical reasons. The poloidal coil system consists of the ohmic heating coils, the equilibrium field coils, and the divertor coils. The ohmic heating coils and the plasma loop act as the primary and secondary sides of a transformer. During the pulsed changes in magnetic flux, currents are induced in the plasma to produce the poloidal magnetic field and to heat the plasma. Because of the toroidal geometry, the poloidal field is stronger near the center of the reactor and the plasma loop tends to expand. Therefore, "equilibrium coils" are needed to counteract this effect. They subtract from the field near the center of the reactor and add to it on the outside. The divertor coils are used to prevent particles that have escaped the plasma from reaching the first wall, the innermost physical boundary surrounding the plasma. If escaping particles were to strike the first wall, impurities would be liberated by spallation and become an unwanted part of the plasma. In addition, the loss of energetic particles would result in plasma cooling.

The required magnetic fields are produced by electric currents passing through coils. For conventional conductors like copper, the I^2R heating loss is very great and provides a limit to the magnetic field that can be achieved. Certain materials, called superconductors, have the property that when they are cooled below a certain critical temperature (around 2-20 K) the resistance becomes zero. Thus, the

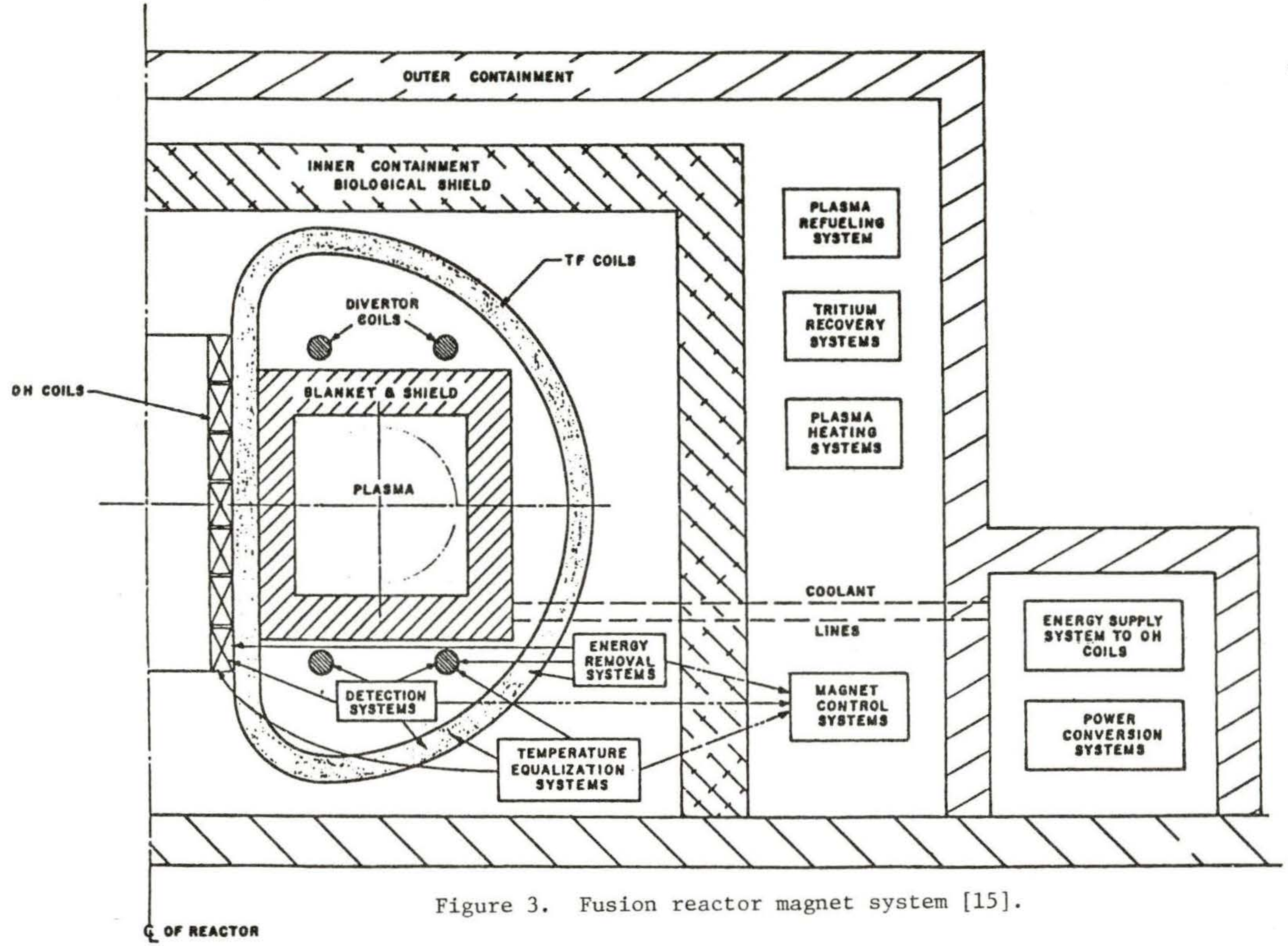

Figure 3. Fusion reactor magnet system [15].

magnetic field that can be obtained using superconducting magnets is significantly higher than that of conventional magnets.

Three superconducting materials are presently used in magnets: NbTi, Nb_3Sn, and V_3Ga. NbTi is cheaper and more ductile than either Nb_3Sn or V_3Ga. Ductility is an important property for wire which is to be bent into a coil. NbTi has the disadvantage of a lower critical temperature and critical magnetic field. It is, however, currently favored for magnets producing fields up to 9 Tesla.

Reliability, discussed more fully in Chapter VI, will be a major problem with fusion reactors. The magnets will be exposed to fast neutron and gamma irradiation originating in the plasma, mechanical stress, asymmetric forces, torques, and the consequences of pulsed operation. The problems associated with radiation damage, cooling, large mechanical forces, and the discharge of stored energy must be solved by the designers of the magnet systems if the successful operation of tokamak reactors is to be assured.

CHAPTER IV. ALGORITHMS

Fault Tree Editor (EDITOR code)

The fault tree editor reads data provided by the user by means of a console and arranges them in an order such that they can be processed in the FAULTTREE program. The editor consists of a number of procedures, each of which performs a specified operation on the data.

The MENU procedure

The MENU procedure clears the screen and displays a "menu" to the user. The menu consists of a list of seven options from which the user may choose: (1) add a node; (2) remove a node; (3) append files; (4) change a node, (5) display a node, (6) write the tree to a file name, or (7) exit the program. For each option, a subroutine is called.

The READTREE procedure

The READTREE procedure recursively reads the elements of a fault tree from disk into memory and stores it as a linked list. The linked list shown in Figure 5 corresponds to the fault tree in Figure 4. Each gate in the tree corresponds to two kinds of records in the linked list. One type of record is the NODE, which contains the actual information describing the gates: name, function, probability, and number of inputs. A bottom event is represented

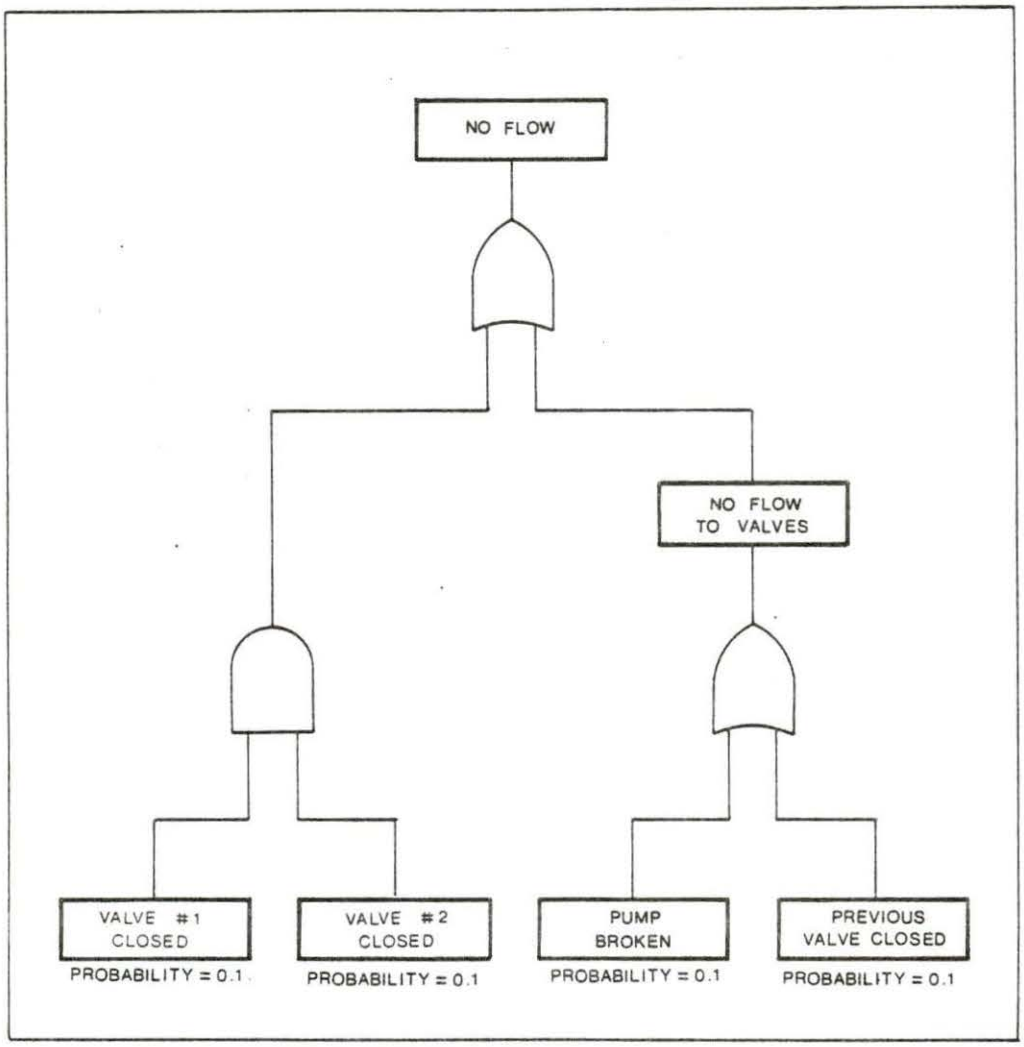

Figure 4. Sample fault tree

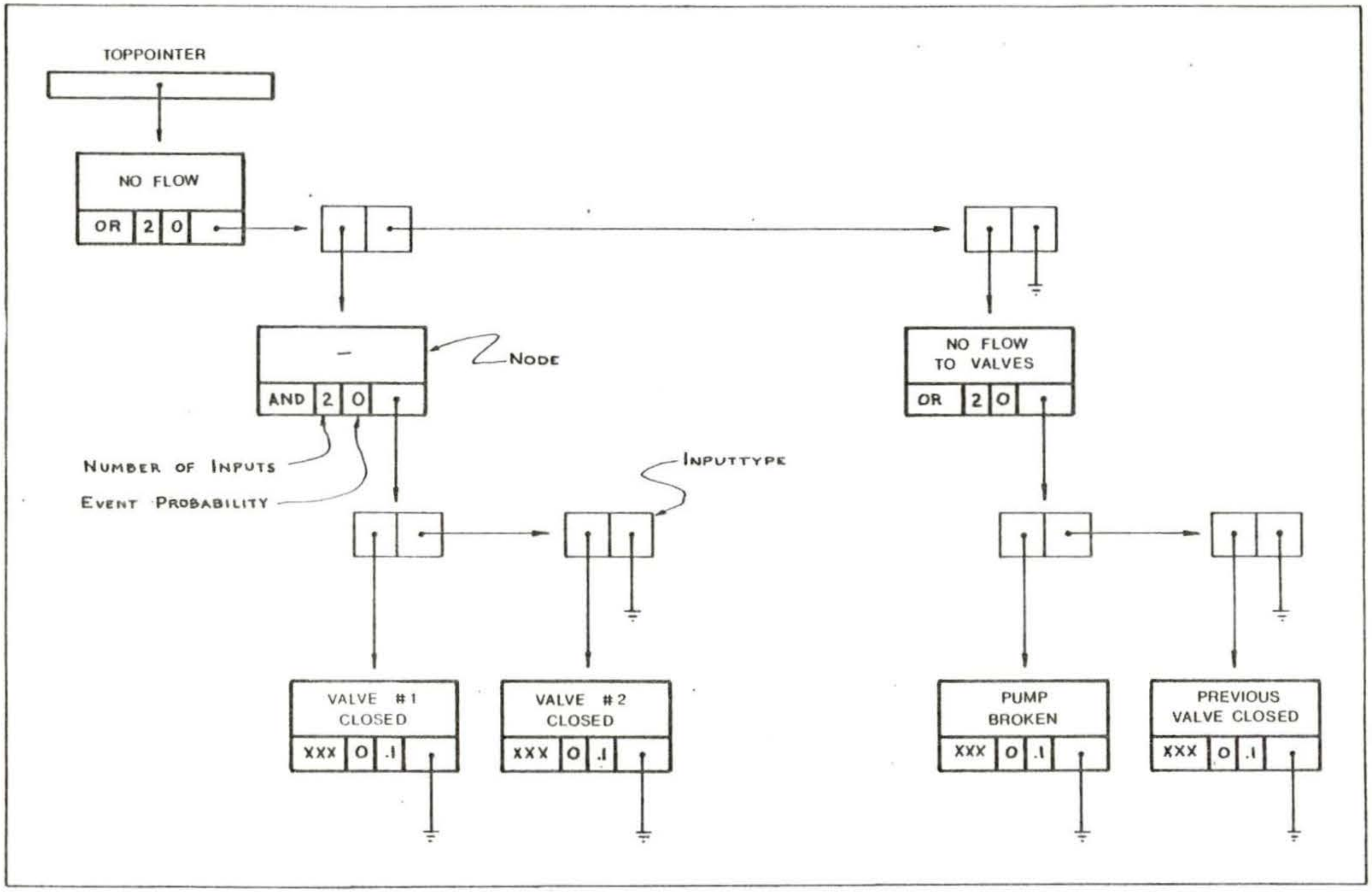

Figure 5. Representation of sample fault tree after passing through READTREE

in Figure 5 with a gate function of XXX. The second type of record, the INPUTTYPE, contains the pointers which connect each NODE to its inputs and outputs.

The SRCHTREE procedure

The SRCHTREE (SeaRCH TREE) procedure recursively searches the fault tree for a gate specified by the user. This procedure is called by many of the other procedures to find a gate which is to be changed or displayed. It inputs a gate name and the pointer to the top event of the tree (henceforth called the top pointer), and outputs the proper gate and a Boolean variable that is set to TRUE if the gate is found and FALSE if it is not found.

The FINDPARENT procedure

The FINDPARENT procedure is used to find the gate preceding a given gate in the tree (its parent, or output gate). FINDPARENT is called by the DELGATE procedure (described below) to reduce by one the number of inputs to the deleted gate's parent. It is also used to check for errors in the DELGATE procedure to insure that the user does not leave an AND or an OR gate with a single input. Inputs to this procedure are the top pointer of the tree and the name of the gate that is to be deleted. Output variables are the gate to be deleted and its output gate, and a Boolean variable set to TRUE if the output gate is found and to FALSE if it is not found.

The ADDGATE procedure

The ADDGATE procedure is called when the user wishes to add a gate to the tree. It prompts the user for the name of the gate to which an input is to be added. It checks to see if this gate is a bottom event, and if so, it will force the user to add at least two gates. This has been implemented to prevent the use of single-input AND or OR gates. It also checks to make sure that the tentative name of the new gate has not previously been used.

When error-checking is complete, the ADDGATE procedure will set up a NODE and an INPUTTYPE for the new gate. The results of the ADDGATE procedure are shown in Figure 6. The procedure increments the number of inputs to the output gate by 1.

The DELGATE procedure

The DELGATE (DELete GATE) procedure is called when the user wishes to remove a gate from the fault tree. The operation of the procedure is shown in Figure 7. Note that the gate remains in memory after its deletion but cannot be retrieved. This is because the standard DISPOSE command is not implemented in UCSD PASCAL. Error checking is done to insure that one of two inputs is not deleted and that the gate to be deleted is a bottom event. After error checking is complete, if the gate to be deleted is the first input to its output gate, then the pointer from the output gate is directed to the second input. If the gate is the ith input (i < number of inputs), then the pointer from the (i-1)th gate is

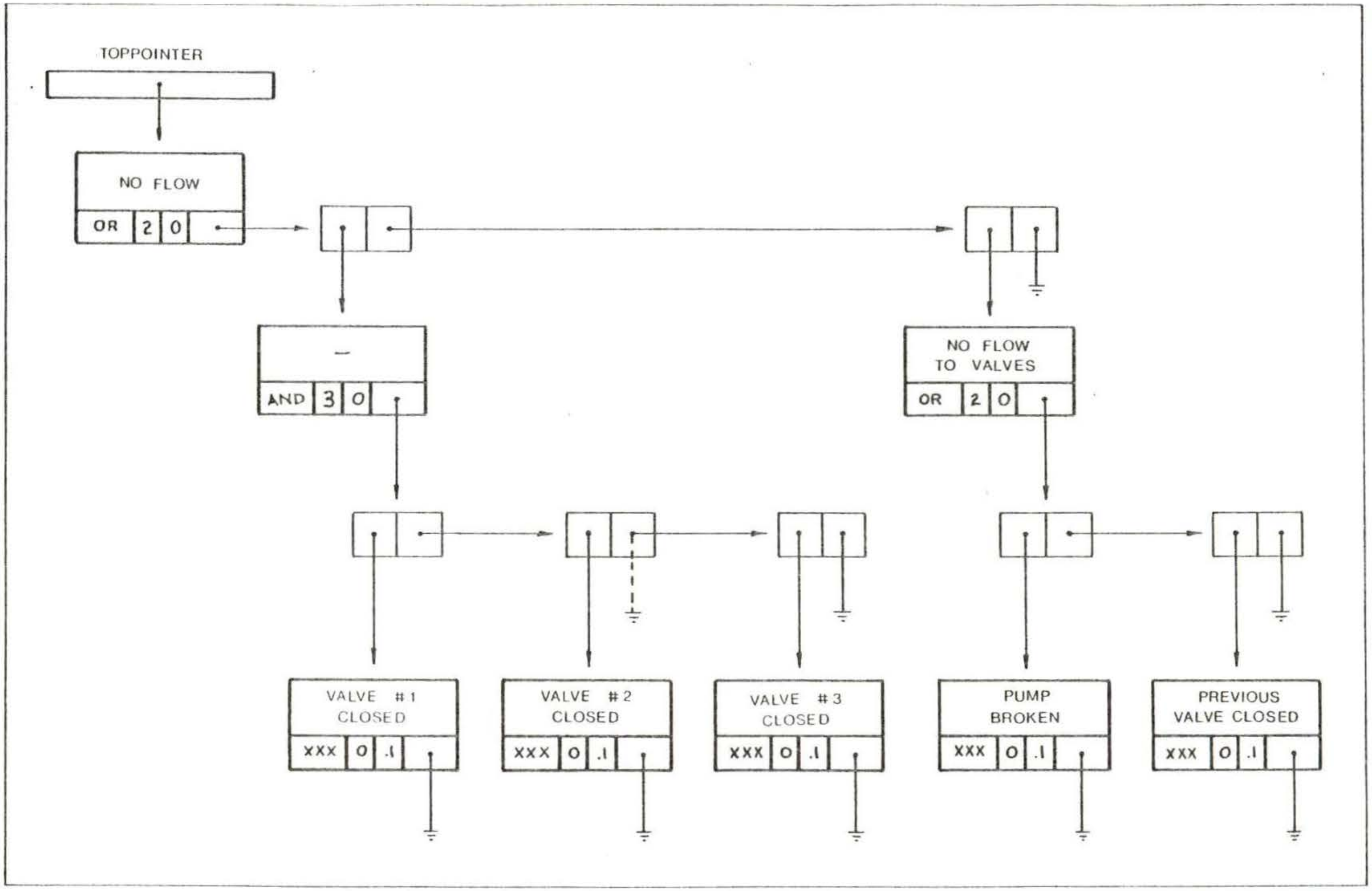

Figure 6. Representation of adding a gate to the sample fault tree

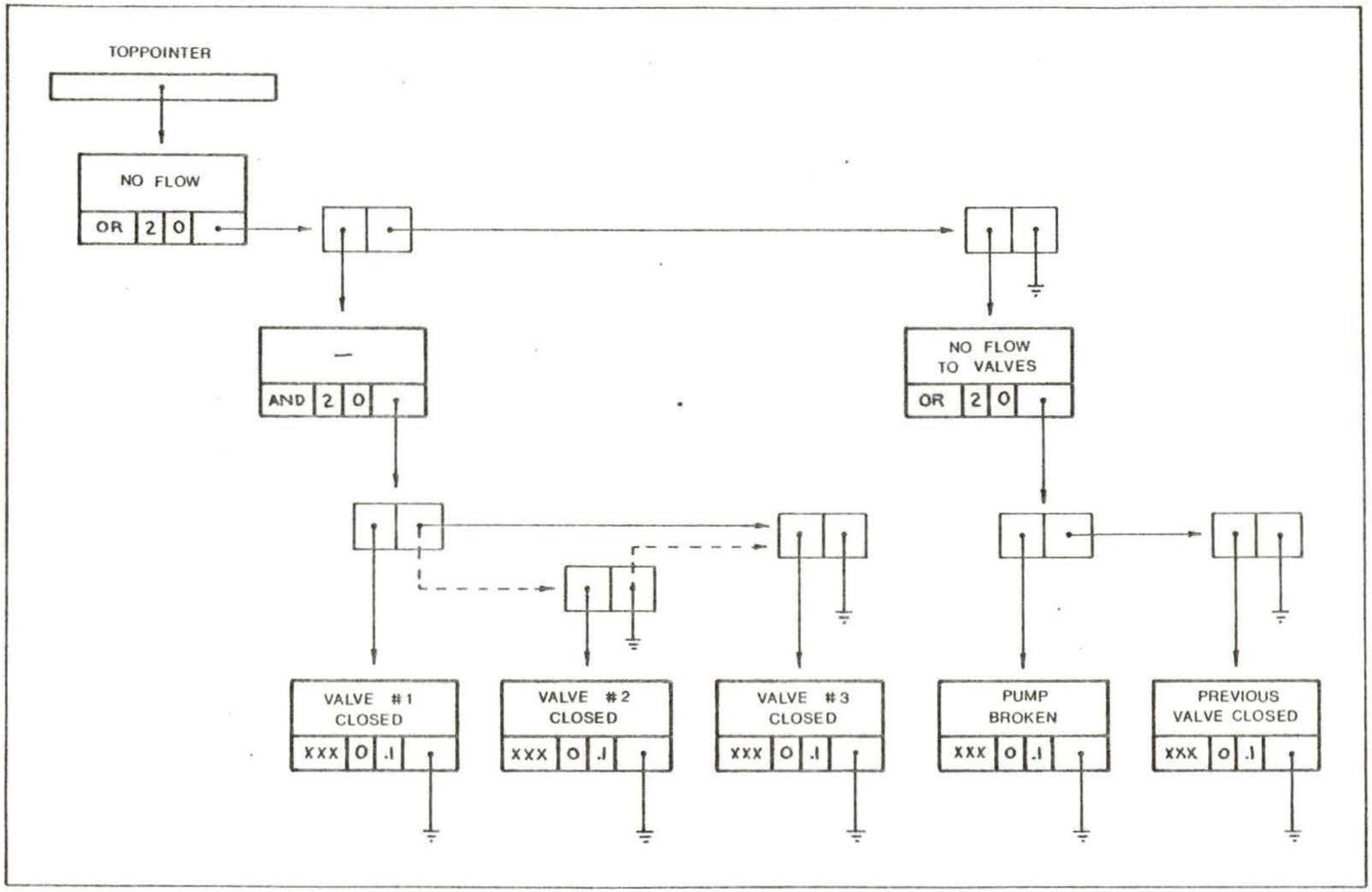

Figure 7. Representation of deleting a gate from the sample fault tree

redirected to the (i+1)th gate. If the gate is the last input, then the pointer leading from the next-to-last input is redirected from the deleted event to NIL.

The APPEND procedure

This procedure reads the components of an already existing fault tree from a file, making its top event an input to a gate specified by the user in the tree being edited. The number of inputs to the output gate is incremented by 1, and the READTREE procedure is called, using the output gate specified by the user as the top pointer.

The CHANGE procedure

When the user wishes to change the name, function, or probability associated with a gate, the CHANGE procedure simply sets the value of the variable equal to the new value. When the user wishes to change the output of a gate, the OUTPUTCHANGE procedure is called. First, the gate is added to the end of the input list of the new output gate. Then, the gate is deleted from the old output gate. Note that, in contrast to the DELGATE procedure, no memory is lost during this delete operation.

The WRITETREE procedure

The WRITETREE procedure recursively writes the gate names, functions, numbers of inputs, and failure probabilities contained in the fault tree to the console, to the printer, or to a disk file. It is not automatically called when the user wishes to exit the program. Thus, all newly entered data are lost if the user does not use the "write" option before exiting the program.

Fault Tree Traversal (FAULTTREE Code)

The FAULTTREE code uses a fault tree produced by the editor to find all of the minimal cut sets and their probabilities. The READTREE procedure found in the editor is used to read the contents of the fault tree from disk and store it into memory.

The FINDCUTSETS procedure

The procedure FINDCUTSETS takes the output from READTREE and processes it into a list of cut sets as shown in Figure 8. To add a new record to the current cut set it first calls the procedure ADD. The ADD procedure is used to check for a bottom event, and to set a Boolean variable, ISBOTTOM, equal to TRUE or FALSE. A new record is added to the list and the appropriate values are put into that record. This new record is then returned to the procedure FINDCUTSETS.

If an AND gate is being processed, then each of the inputs of the AND gate is added to the current cut set. If it is processing an OR gate, the process is somewhat more complex. For an N-input OR gate, N-1 copies of the current cut set are made, using the procedure COPY. A different input of the OR gate is added to the end of each copy. Then, the MERGECUTSETLISTS procedure is used to link all of the copies together into a new cut set list.

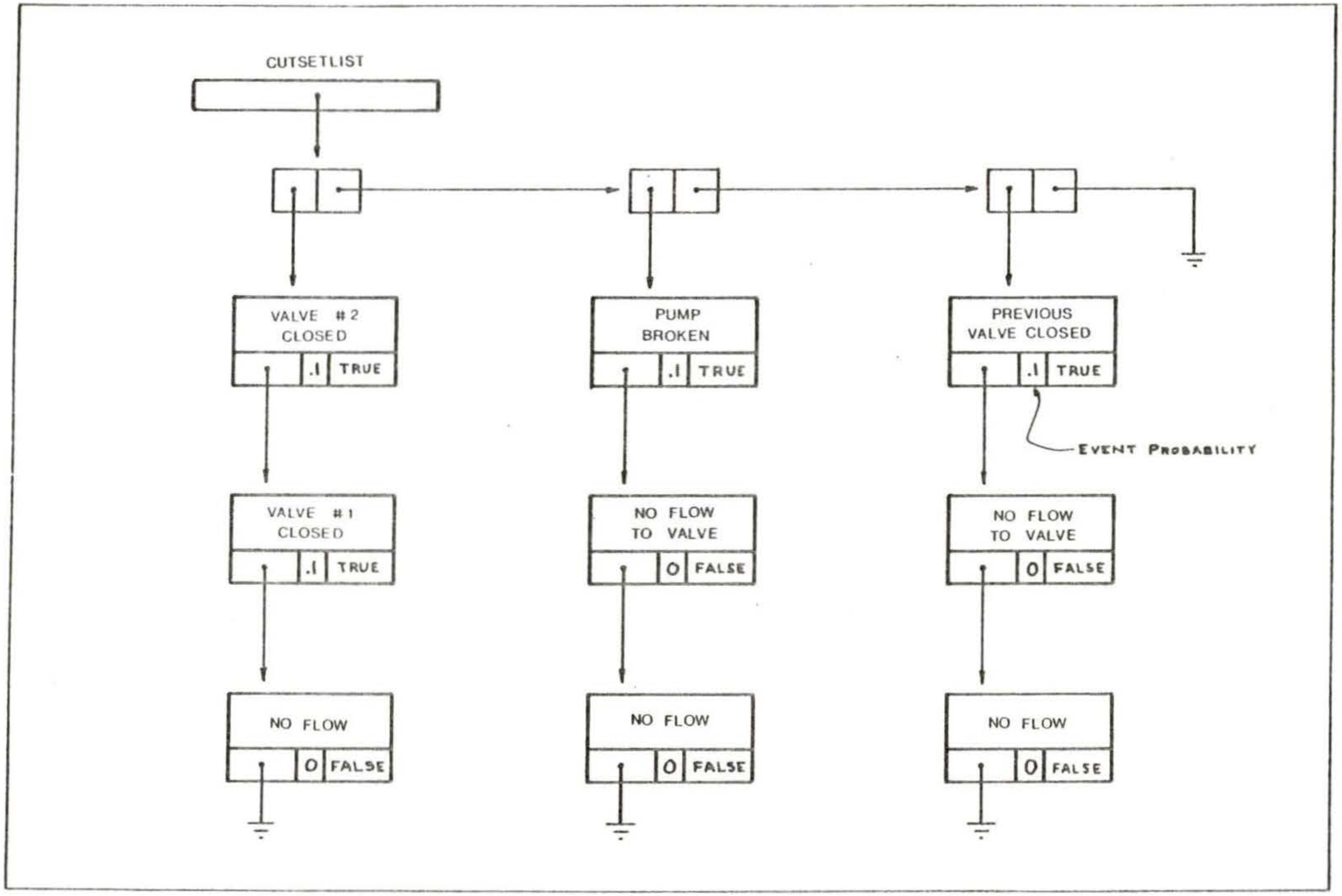

Figure 8. Representation of sample fault tree after passing through FINDCUTSETS

The FINDPROB procedure

The FINDPROB procedure computes the occurrence probability associated with each cut set and the occurrence probability associated with the top event. It uses the cut sets produced by the FINDCUTSETS procedure and event probabilities supplied by the user. The probability associated with a given cut set is defined to be the product of the probabilities of the bottom events belonging to the specified cut set.

$$P_j = \prod_{i=1}^{n_j} P_{ij}$$

where P_j is the probability of the jth cut set;

P_{ij} is the probability of the ith bottom event of the jth cut set;

n_j is the number of bottom events in the jth cut set.

The probability P of occurrence of the top event is determined by the relation:

$$P = 1-\prod_{j=1}^{n} (1-P_j)$$

where n is the number of cut sets.

The WRITECUTSETS procedure

This procedure produces a list of cut sets for user consideration. It traverses the linked list produced by FINDCUTSETS, stopping to write out the name of each gate (1) to a data file,

(2) to the console, or (3) to the printer. An asterisk is printed beside each bottom event. At the end of each cut set, the associated probability of occurrence is printed, and at the end of the cut set list, the probability of the top event is printed.

Deviations from Standard Pascal

Strings are used instead of packed arrays to facilitate user entry. This is because UCSD PASCAL requires that user-provided input fill a packed array and that the user must include enough spaces to accomodate the dimension of the array. Thus, for each event name, the user would need to type 40 characters if packed arrays (i.e. standard PASCAL) were used. The MEMAVAIL command, which determines the size of the available memory is nonstandard, but most implementations of PASCAL have an equivalent function.

File manipulation is different for each PASCAL machine, but in the PERTT programs standard PASCAL I/O has been used wherever possible.

CHAPTER V. USER'S MANUAL

Editor

<u>Beginning to edit a fault tree</u>

Put the APPLE 3: in disk drive #1 and turn on the computer and monitor. Wait until the light on the disk drive turns off, then remove the APPLE 3: disk and insert the FAULTTREE: disk. Press <RESET> and then press the <X> key. The computer will prompt, "Execute what file?". Type EDITOR and press the <RETURN> key. This executes the editor program.

The computer will then give the prompt:

```
DO YOU WANT TO:
     EDIT A CURRENTLY EXISTING FAULT TREE (E)
     START A NEW FAULT TREE (N)
PLEASE TYPE E OR N
```

If a fault tree already stored on disk is to be altered, type <E> and then <RETURN>. The computer will prompt, "TYPE FILE TO BE EDITED". Type the name of the data file containing the desired fault tree and press <RETURN>. The file will be read into the memory of the computer.

If a new data file (a new fault tree) is to be created, type <N> and then<RETURN>. The computer will prompt "TYPE THE NAME OF THE TOP EVENT". Type a name of up to 40 characters and press <RETURN>.

Building the fault tree

In order to build a fault tree, the user may select options from the following menu on the screen:

```
MEMORY AVAILABLE = xxxxx.  DO YOU WANT TO:
     ADD A NODE (A)
     REMOVE A NODE (R)
     APPEND A FILE TO THIS FILE (P)
     CHANGE A GATE (C)
     DISPLAY A GATE (D)
     WRITE TO A FILE NAME (W)
     EXIT THE PROGRAM (E)
PLEASE TYPE A,R,P,C,D,W, OR E.
```

Type desired operation and press <RETURN>. The memory available statement indicates the number of bytes of memory available for fault tree data. Adding a gate to a fault tree requires approximately 30-40 bytes. When the available memory becomes small, the user should try to write the file to a name, exit the file, and read the file back in. This is because the deleting procedure causes memory to be temporarily lost, as explained in Chapter IV.

Adding a node After the user types <A>, the computer will prompt, "TO WHICH GATE WOULD YOU LIKE TO ADD AN OUTPUT". Type the gate name and press <RETURN>. The computer will prompt, "TYPE PROBABILITY OF OCCURRENCE FOR NEW NODE". Type a probability between 0 and 1 and press <RETURN>. If this gate is, or will be a bottom event, this probability is significant. For all other cases, enter 0. If the output gate was a bottom event, the following prompt will appear on the screen:

```
SINCE (event name) IS A BOTTOM EVENT, IT MUST HAVE AT LEAST 2 INPUTS.
WOULD YOU LIKE IT TO BE AN AND (A) OR AN OR (O) GATE? (TYPE L TO
LEAVE THE ADD PROCEDURE AND DELETE THE GATE JUST ADDED)

PLEASE TYPE A, L, OR O
```

If the user types L, the gate that has just been added is deleted. If the user types A or O, the output gate is changed to an AND gate or an OR gate, and another gate may be added by the user. This is to assure that the final fault tree contains no single-input AND or OR gates. The computer will prompt, "DO YOU WISH TO ADD ANOTHER GATE TO (event name)". Type YES or NO and press <RETURN>.

Removing a gate After the user types A , the computer will prompt, "WHAT GATE DO YOU WISH TO DELETE". Type the gate name and press <RETURN>. If the gate has inputs, the user will be asked if he wishes to delete everything below the gate. If the answer is YES, then all of the inputs will be deleted. If the answer is NO, then no gates will be deleted. If the gate above the gate to be deleted (the "OUTPUT" gate) has only two inputs, the output gate will be displayed and the computer will prompt, "DO YOU WISH TO DELETE BOTH INPUTS TO (event name)? TYPE Y OR N." If <Y> is typed, both inputs will be deleted and the output gate will be a bottom event. The computer will prompt for a probability for the gate. If <N> is typed, the gate will not be deleted. If <N> is typed and gates below one input have already been deleted, these gates have been deleted permanently. They will not be returned to the fault tree. If the user wishes to delete a gate without deleting its inputs, he must redirect the outputs of the input gates he wishes to retain. This may be accomplished by means of the "change" option, described below.

Appending a file This command will read a fault tree from a file, making its top event an input to a specific gate of the fault tree being edited. After the user types <P> the computer will prompt, "INPUT TO WHICH GATE". Type the name of the output gate and press RETURN . The computer will ask for the name of the file to be appended to the tree being constructed. Type it and press RETURN .

Changing a gate When the user types <C>, the computer will prompt, "WHAT GATE WOULD YOU LIKE TO CHANGE". Type the gate name and press RETURN . A menu will appear on the screen as follows:

```
DO YOU WANT TO:

     CHANGE THE NAME OF THE GATE (N)

     CHANGE THE FUNCTION OF THE GATE (F)

     CHANGE THE OUTPUT OF THE GATE (O)

     CHANGE THE PROBABILITY OF THE GATE (P)

     TO EXIT THE CHANGE PROCEDURE TYPE E

PLEASE TYPE N,F,O,P, OR E
```

When the user types <N> or <P>, the computer prompts for the new name or probability. When the user types <F>, the gate function is changed from AND to OR or vice versa. The <O> command is used to change the position of a gate within the tree; in other words, the gate is deleted from one position and added to another. When a gate is moved, all of its inputs are also moved.

After the change is made in the tree, the computer prompts, "DO YOU WANT TO MAKE ANY OTHER CHANGES TO (event name)? TYPE Y OR N". Type YES or NO.

Displaying a gate   When <D> is typed, the computer will prompt for a gate name. Type it and press RETURN . The computer will print the name of the gate at the top of the screen. If the gate is a bottom event, its probability will be displayed. Otherwise, the function (AND or OR) will be displayed, along with the name and probability of each of the inputs.

Writing to a file name   When <W> is pressed, the computer will prompt for the name of an output file. Type a file name to have the tree stored on disk, or type CONSOLE: or PRINTER: to display the tree on the screen or the printer. Be sure to type a ":" after the words "console" and "printer"; otherwise, the tree will be stored on disk as a file named "console" or "printer". If output is sent to the screen or to the printer, it will not be sent as a tree, but as a long string of data.

Exiting the program Before you type <E>, be sure to write the tree to a disk file. Otherwise, all newly entered data will be lost.

Fault Tree Traversal

After a data file has been created using the editor, the minimal cut sets and failure probabilities may be found using the FAULTTREE program. Type <X>. The computer will prompt, "Execute what file?". Type FAULTTREE and press <RETURN>. The computer will ask for an input file name. Type the name of the data file and press RETURN . The computer will then prompt for an output file name. Type a file name to write the cut sets and probabilities to a disk file, or type CONSOLE: or PRINTER:. For each cut set, the events will be listed and a cut set probability will be given. A total failure probability will be given at the end of the listing.

CHAPTER VI. APPLICATION TO THE MAGNET SYSTEMS OF A FUSION REACTOR

Safety Problems in Superconducting Magnet Systems

Reliability is a major problem of existing superconducting magnets. In one study [21] out of twenty experimental magnets observed, fourteen had failed due to inadequate cooling, electrical insulation breakdown, inadequate mechanical support, and failures in powering and safety systems.

Radiation effects on superconducting magnets are significant. Over a 30 year lifetime, neutron fluences (the total number of neutrons falling on a unit area) are estimated to be approximately $10^{18}/\text{cm}^2$, which would lead to a displacement rate of $1.8\text{x}10^{-3}$ dpa. This would give approximately a 10% reduction in J_c, the critical current, at 4.2 K, for NbTi [20]. It appears that this degradation could be fairly easily tolerated in UWMAK-I, the University of Wisconsin Tokamak Study Design. [15].

Cooling of superconducting magnets can also be a problem. Two possibilities for cooling are being explored: bath cooling in liquid helium and forced cooling with two-phase helium. The main disadvantage of bath cooling is that the heat transfer rate depends upon the surface orientation. If the conductor orientation deviates from vertical, the heat transfer rate will be decreased by approximately one order or magnitude [21]. Much less experimental evidence exists for forced cooling with two-phase helium. Cooling instabilities seem to appear because of helium phase transition behavior.

Stored energy will also be a problem, since the magnets of a fusion reactor will be expected to carry more than 100 times the current carried in present-day superconducting magnets. If a superconducting magnet were to "go normal" (lose its superconducting properties because it exceeds its critical temperature or critical current), the superconducting material will suddenly be subject to I^2R heat losses, which would be enormous in the case of a fusion reactor. Such sudden stored energy deposition into a small area of the magnet could lead to temperature excursions above the melting point of the magnet. The most serious case of released energy is the case of a coil break, studied in detail below. Other mechanisms include dewar leakage, loss of conductor cooling, loss of superconductivity, and power supply failure.

Application to a Conductor Break

The case selected for detailed study consists of a conductor break because this seems to be the most significant accident for a superconducting magnet system. Large amounts of energy would be released in a short time. This would produce large amounts of heat and large releases of helium gas, and would lead to significant structural failure. This structural failure could lead to internal and external missiles, and release of radioactive materials in the form of debris from the destroyed structure.

A fair amount of qualitative information is available concerning accident pathways to a conductor break. A fault tree for a conductor break is shown in Figure 9 [15]. Quantitative information on reliability is lacking however. The failure rate reported for existing superconducting magnets [15] is clearly unacceptable and one can only speculate on the failure rates for those to be used in the future. It is difficult to quantify failure probabilities without direct experience, and thus, one must rely on qualitative judgments as to which accident pathways are most likely.

For these reasons, the PERTT code and fission industry data were used to set goals for operation of the fusion reactor magnet.

Powell [15] in an analysis of fusion reactor design, sets an upper limit, for economic reasons, of 10^{-2} per reactor-year for events which release no radiation, but cause enough damage to the magnet that replacement is necessary. The number 10^{-2} was used for calculation, and it is assumed that the probabilities for more severe accidents would be much lower. If one assumes the appropriateness of the fault tree in Figure 9, and if all of the 29 event probabilities are equal, then to generate an overall failure probability of 10^{-2} per reactor year, the probability for each event would be 3×10^{-4} per reactor year. Of course, it is unlikely that all of the probabilities will be equal. Qualitatively, failure caused by electrical

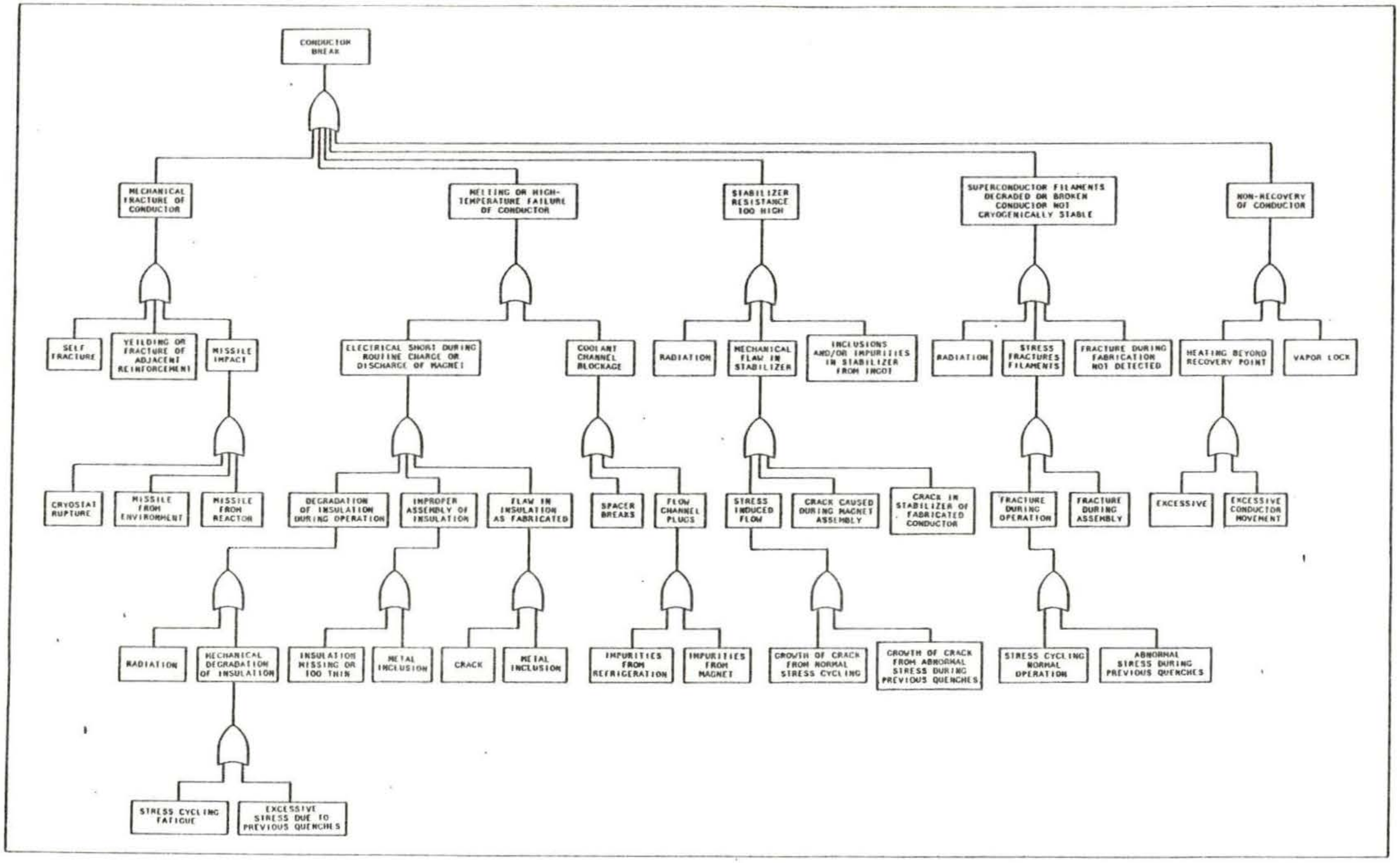
CONDUCTOR BREAK
MECHANICAL FRACTURE OF CONDUCTOR
MELTING OR HIGH-TEMPERATURE FAILURE OF CONDUCTOR
STABILIZER RESISTANCE TOO HIGH
SUPERCONDUCTOR FILAMENTS DEGRADED OR BROKEN CONDUCTOR NOT CRYOGENICALLY STABLE
NON-RECOVERY OF CONDUCTOR
SELF FRACTURE
YEILDING OR FRACTURE OF ADJACENT REINFORCEMENT
MISSILE IMPACT
ELECTRICAL SHORT DURING ROUTINE CHARGE OR DISCHARGE OF MAGNET
COOLANT CHANNEL BLOCKAGE
RADIATION
MECHANICAL FLAW IN STABILIZER
INCLUSIONS AND/OR IMPURITIES IN STABILIZER FROM INGOT
RADIATION
STRESS FRACTURES FILAMENTS
FRACTURE DURING FABRICATION NOT DETECTED
HEATING BEYOND RECOVERY POINT
VAPOR LOCK
CRYOSTAT RUPTURE
MISSILE FROM ENVIRONMENT
MISSILE FROM REACTOR
DEGRADATION OF INSULATION DURING OPERATION
IMPROPER ASSEMBLY OF INSULATION
FLAW IN INSULATION AS FABRICATED
SPACER BREAKS
FLOW CHANNEL PLUGS
STRESS INDUCED FLOW
CRACK CAUSED DURING MAGNET ASSEMBLY
CRACK IN STABILIZER OF FABRICATED CONDUCTOR
FRACTURE DURING OPERATION
FRACTURE DURING ASSEMBLY
EXCESSIVE
EXCESSIVE CONDUCTOR MOVEMENT
RADIATION
MECHANICAL DEGRADATION OF INSULATION
INSULATION MISSING OR TOO THIN
METAL INCLUSION
CRACK
METAL INCLUSION
IMPURITIES FROM REFRIGERATION
IMPURITIES FROM MAGNET
GROWTH OF CRACK FROM NORMAL STRESS CYCLING
GROWTH OF CRACK FROM ABNORMAL STRESS DURING PREVIOUS QUENCHES
STRESS CYCLING NORMAL OPERATION
ABNORMAL STRESS DURING PREVIOUS QUENCHES
STRESS CYCLING FATIGUE
EXCESSIVE STRESS DUE TO PREVIOUS QUENCHES

Figure 9. Conductor break fault tree [15]

shorts, by previous damage from temperature inhomogeneities, by coolant channel blockage, by inadequate quality control during fabrication, and by mechanical fracture seem to be more significant than others [15].

Since the probability of accidental death due to falling meteors, planes, etc. is reported to be 6 x 10^{-6} per person per year, a probability of 1 x 10^{-5} per year was utilized for the probability of external missiles. The probability of internal missiles was set at 10^{-5} per year, also. Powell gives a probability of 10^{-5} per year for an accident that involves containment breach and radioactivity release to the public, and this type of accident is implied if internal missiles are present [15].

The probability of coolant channel blockage was set at 10^{-6} per year. WASH-1400 [22] sets the probability of pipe blockage at 10^{-10} per hour or 9 x 10^{-7} per year, for a pipe diameter greater than 3 inches. As seen in Figure 9, each potential contributor mechanism to coolant channel blockage, spacer breaks, impurities from refrigeration coolant channels, and impurities from the magnet coolant channels will have a probability of 3 x 10^{-7} per year. The probability for a crack in the electrical insulation is also taken to be approximately 10^{-6} per year, because WASH-1400 gives this number as a mean value for cracks in pipes and other components. Insulation is subject to different environmental stresses than pipes, but this number should be correct within an order of magnitude.

Probability for the loss in electric power is higher. WASH-1400 gives a probability of 2×10^{-7} per hour or 2×10^{-3} per year for the fission industry. The figure 2×10^{-3} per year was used in this thesis for calculations. This implies a probability of 3×10^{-4} per year for each of the bottom events to the "electrical short" subtree. Of course, these numbers could be improved by the use of parallel power sources.

The probability of yielding or fracture of adjacent support structure is a qualitative judgment. Its probability was set at 10^{-6} per year. The probability of an undetected flaw during fabrication was taken to be 10^{-5} per year.

CHAPTER VII. RESULTS AND CONCLUSIONS

A package, PERTT, has been written which permits the analysis of fault trees involving up to approximately 100 events using a desk top computer. The package consists of two programs. A listing of the EDITOR program, used to construct and change the program, is found in Appendix A. A listing of the FAULTTREE program, used to find the minimal cut sets and failure probability for the system, is found in Appendix B.

The application of the program to a typical problem revealed certain advantages and shortcomings to the use of a desk top computer.

- The program is easy to use, because the user is prompted for all necessary information.
- A study of the relative advantages of different reactor system arrangements is facilitated.
- The running time is fairly long - 1 1/2 minutes for a 49-event fault tree. However, the turnover time is equal to the running time, and is thus very short.

The use of the PERTT package to analyze the reliability of a superconducting magnet used in a fusion reactor demonstrated an inverse application in which the reliabilities of various components were established based on the stated overall system reliability.

Cut sets found by the PERTT program for the conductor break fault tree in Figure 9 are given in Appendix C. No probability calculations were performed in this run. For the fault tree involved, an execution time of 1 1/2 minutes was required. This could be significantly decreased if a faster computer, such as a VAX, were used.

Each cut set consists of only one bottom event, in addition to intermediate events, because the fault tree consists of only OR gates. This means that the reliability importances of all of the components are equal.

If all of the failure probabilities not enumerated in Chapter VI are taken to be equal, then they would be equal to 6×10^{-4} per reactor-year. Since the probability for electrical shorts and for accidents due to previous damage seems to be higher than the average [15], this probability was set equal to 1×10^{-3} per reactor-year. This results in probabilities for the other events of 5×10^{-4} per reactor year. This cannot be achieved with existing magnets, which are still in the experimental stage.

The PERTT code helped a good deal in performing these calculations, even though for this case it was used to "work backwards" from a system failure probability to component failure probabilities. Seeing the cut sets helped in visualizing the failure paths more clearly, and iteration could be performed to find estimates for the probabilities of the events for which probabilities could not otherwise be quantified. The code was easy to use in this calculation, and was far more convenient than using cards on a mainframe computer.

A desk top computer can aid in fault tree analysis by:

(a) providing immediate and direct access to the computer,

(b) providing results almost immediately, thus furnishing feedback which permits the identification of poor design or events which require more study,

(c) providing inexpensive computer time,

(d) providing a convenient, fast, simple means of changing an already constructed fault tree.

The desk top computer has certain limitations, such as a relatively small memory. This means that only small (<100 event) fault trees can be analyzed. Since many complete system fault trees contain more than 500 events, it is clear that a large computer must be involved in the full analysis.

PERTT will be useful in a design situation, where the user must try many different combinations of components to find the optimum safety combination. It will also be useful as a teaching aid in a university classroom situation. If more memory and more speed is needed, it could easily be adapted for a VAX or other system, for standard PASCAL has been used wherever possible. Because of its convenience, PERTT will be extremely valuable in all situations where large memory is not required.

CHAPTER VIII. SUGGESTIONS FOR FURTHER STUDY

It seems likely that the usefulness of the PERTT programs can be increased by adding three options to the procedure. The first option is time dependence for reliability information. The computer could be programmed to find component failure probabilities from failure and test data supplied by the user, and to calculate associated unavailability. These calculations can be performed by hand by the user, but they become tedious when many different events are involved. Reliability importance calculations, as described in Chapter III, are not feasible on a microcomputer because they are too time-consuming.

To provide a more legible and compact fault tree, the program could be written using NOT and k-out-of-n gates. These structures must be represented by groups of AND and OR gates when the PERTT code is being used.

Some form of modularization for the fault tree would also be useful. If the tree could be broken down into subtrees to be stored on disk, larger fault trees could be handled by the system memory.

BIBLIOGRAPHY

1. H. A. Amherd and J. H. Vanston, editors, A Feasibility Study for the Development of Fusion Energy, Technical Report No. ER-778-SR, 1979.

2. R. E. Barlow and F. Proschan, Statistical Theory of Reliability and Life Testing: Probability Models (Holt, Rinehart, and Winston, New York, 1975).

3. Roy J. DeBellis and Zeinab A. Sabri, Fusion Power: Status and Options, Technical Report No. EPRI ER-510-SR, 1977.

4. R. C. Erdmann, Probabilistic Safety Analysis. Final Report, Technical Report No. EPRI-NP-424, 1977.

5. R. C. Erdmann, WAMCUT, a Computer Code for Fault Tree Evaluation. Final Report. Technical Report No. EPRI-NP-803, 1978.

6. W. Findlay and D. A. Watt, PASCAL, an Introduction to Methodical Programming, (Computer Science Press, Potomac, MD, 1978).

7. J. B. Fussell and W. E. Vesely, Transactions of the American Nuclear Society, 15, 262 (1972).

8. R. Karimi, N. Rasmussen, and L. Wolf, Qualitative and Quantitative Reliability Analysis of Safety Systems, Technical Report No. PB81-118325, 1980.

9. William E. Kastenberg and David Okrent, Some Safety Considerations for Conceptual Tokamak Fusion Power Reactors, Technical Report No. EPRI-ER-546, 1978.

10. G. L. Kulcinski, et al., Nuclear Technology, 22, 20 (1974).

11. F. L. Leverenz and H. Kirch, User's Guide for the WAM-BAM Computer Code, Technical Report No. PB-249 824/8SL, 1976.

12. E. E. Lewis, Nuclear Power Reactor Safety (John Wiley and Sons, New York, 1977).

13. Jaime Olmes and Lothar Wolf, A Modular Approach to Fault Tree Analysis, Technical Report No. NUREG/CR-0670, 1979).

14. P. K. Pande, Computerized Fault Tree Analysis: TREEL and MICSUP, Technical Report No. ORC 75-3, 1975.

15. J. Powell, editor, Aspects of Safety and Reliability for Fusion Magnet Systems, Technical Report No. BNL 50542, 1976.

16. D. M. Rasmuson, N. H. Marshall, J. R. Wilson, and G. R. Burdick, COMCAN II-A: A Computer Program for Automated Common Cause Failure Analysis, Technical Report No. TREE-1361, 1979.

17. E. T. Rumble, F. L. Leverenz, Jr., and R. C. Erdmann, Generalized Fault Tree Analysis for Reactor Safety, Technical Report No. EPRI 217-2-2, 1975.

18. P. Shaw and R. F. White, Appraisal of the PREP, KITT, and SAMPLE Computer Codes for the Evaluation of the Reliability Characteristics of Engineered Systems, Technical Report No. WRO-R-57, 1978.

19. Keith Shillington, et al., APPLE PASCAL Reference Manual (Apple Computer, Inc., Cupertino, CA, 1979).

20. M. Soll, Journal of Nuclear Materials, 72, 168 (1978).

21. M. Soll, Kerntechnik, 19, 272 (1977).

22. United States Nuclear Regulatory Commission, Reactor Safety Study: An Assessment of Accident Risks in U.S. Commercial Nuclear Power Plants, Appendix 3, Technical Report No. WASH-1400, 1975.

23. H. J. Van Slyke and D. E.Griffing, ALLCUTS, a Fast, Comprehensive Fault Tree Analysis Code, Technical Report No. ARH-ST-112, 1975.

24. H. E.Vesely, F. F. Goldburg, N. H. Roberts, and D. F. Haasl, Fault Tree Handbook, Technical Report No. NUREG 0492, 1981.

25. H. E. Vesely and R. E. Narum, PREP and KITT: Computer Codes for Automatic Evaluation of a Fault Tree, Technical Report No. IN-1348, 1970.

ACKNOWLEDGMENTS

The author would like to thank Dr. D.M. Roberts for constant moral and technical support, as well as help in correcting the final manuscript. Thanks also go to Steven Ford, who has suffered through the trials and tribulations of the past two years. The excellence of the artwork is due to the insatiable desire of draftsman Lyle Ruppert for oatmeal chocolate-chip cookies. Thanks also go to Jo Sedore for typing the thesis manuscript. Finally, the author expresses appreciation for the technical support provided by Dr. Z. A. Sabri, Dr. Richard Horton, and Dr. Richard Danofsky.

APPENDIX A. SOURCE LISTING OF "EDITOR" PROGRAM

The EDITOR program, described in full in Chapters IV and V, is an interactive program used to construct a fault tree. It stores the fault tree in a data file to be processed by the FAULTTREE program, listed in Appendix B.

```
      TYPE STRING40 = STRING[40];
      STRING3   = STRING[3];

      NODE=RECORD        (*THESE ARE THE NODES WITH THE ACTUAL INFORMATION*)
                         (* IN FIGURE 4*)
              NAME:STRING40;   (* NAME OF THE EVENT. *)
              FUNCT:STRING3;   (* "AND", "OR", "XXX" *)
              NMR:INTEGER;     (* NUMBER OF INPUTS   *)
              PROB:REAL;       (* PROBABILITY OF OCCURRENCE IF BOTTOM EVENT *)
              INPUTLIST:^INPUTTYPE
            END;
      NODEP=^NODE;

      INPUTTYPE=RECORD   (*THESE ARE THE "CONNECTOR" NODES IN FIGURE 4*)
                   INPOINTER:^NODE;
                   NEXTIN:^INPUTTYPE
                 END;
      INPUTP=^INPUTTYPE;


VAR
  TOPPOINTER:NODEP;         (* MAIN FAULT TREE AFTER READING *)
  PARENTNODE : NODEP;
  I:INTEGER;
  ANS:STRING40;
  INPUTFILE, OUTFILE:TEXT;

  FILENAME,OUTFILENAME:STRING40;
  SRCHNAME:STRING40;
  SRCHNODE:NODEP;
  ISNAME:BOOLEAN;


PROCEDURE READONELN(VAR STR:STRING40);
  (*GIVES AN ERROR MESSAGE WITH ZERO LENGTH RESPONSES SO THAT THE *)
  (*PROGRAM WILL NOT ABORT*)
  BEGIN
    REPEAT
      READLN(STR);
      IF(LENGTH(STR)=0) THEN
        WRITELN ('INVALID ENTRY -- TRY AGAIN');
    UNTIL (LENGTH(STR))>0;
  END;


PROCEDURE SRCHTREE(CURRENTNODE:NODEP;
                       VAR SRCHNODE    :NODEP;
                       SRCHNAME    :STRING40;
                       VAR ISNAME      :BOOLEAN);

  (*SEARCHES THE TREE FOR THE NODE TO BE CHANGED*)
  VAR CURRENTINPUT:INPUTP;
      I, NMROFINPUTS:INTEGER;

    BEGIN
      ISNAME:= FALSE;
      NMROFINPUTS := CURRENTNODE^.NMR;
      IF (SRCHNAME=CURRENTNODE^.NAME) THEN BEGIN
        (*FOUND THE NODE TO BE CHANGED*)
        ISNAME:= TRUE;
        SRCHNODE := CURRENTNODE;
      END
      ELSE IF (CURRENTNODE^.FUNCT<>'XXX') THEN BEGIN
        CURRENTINPUT := CURRENTNODE^.INPUTLIST;   (*FOR 1ST INPUT OF GATE*)
        SRCHTREE(CURRENTINPUT^.INPOINTER,SRCHNODE,SRCHNAME,ISNAME);
        I:= 2;
        WHILE ((I<=NMROFINPUTS) AND (NOT ISNAME)) DO BEGIN
          CURRENTINPUT := CURRENTINPUT^.NEXTIN;   (*GO TO NEXT INPUT OF GATE*)
          SRCHTREE(CURRENTINPUT^.INPOINTER,SRCHNODE,SRCHNAME,ISNAME);
          I:=I+1;
        END;
      END;
   END;


PROCEDURE READTREE (CURRENTNODE:NODEP);
(* READS TREE UNDER CURRENTNODE FROM INPUT FILE. *)

VAR
  INNAME:STRING40;
  GATE   :STRING3;
  NMROFINPUTS:INTEGER;
  CURRENTINPUT:INPUTP;
  I:INTEGER;
  PROB:REAL;

BEGIN
  IF EOF(INPUTFILE) THEN BEGIN
    WRITELN;
    WRITELN('***ERROR, EXPECTED MORE INPUT.');
    HALT;
  END;
  READLN (INPUTFILE,INNAME);         (* READ DATA FROM INPUT FILE *)
  READLN (INPUTFILE,NMROFINPUTS);
  READLN (INPUTFILE,GATE);
  READLN (INPUTFILE,PROB);
  WRITELN(INNAME);                   (*DEBUG*)
  CURRENTNODE^.NAME  := INNAME;      (* LOAD NODE WITH INPUT DATA *)
  CURRENTNODE^.NMR   := NMROFINPUTS;
  CURRENTNODE^.PROB  := PROB;
  CURRENTNODE^.FUNCT := GATE;
  WRITELN ('MEMORY AVAILABLE = ',MEMAVAIL);

  IF GATE <> 'XXX' THEN BEGIN
    NEW (CURRENTINPUT);              (* GET FIRST INPUT OF GATE *)
    CURRENTNODE^.INPUTLIST := CURRENTINPUT;
    NEW (CURRENTINPUT^.INPOINTER); (* GET GATE FOR FIRST INPUT *)
    READTREE (CURRENTINPUT^.INPOINTER);  (* READ TREE FOR FIRST INPUT *)

    FOR I := 2 TO NMROFINPUTS DO BEGIN   (* FOR EACH OF THE OTHER GATES *)
      NEW (CURRENTINPUT^.NEXTIN);  (* GET NEXT INPUT *)
      CURRENTINPUT := CURRENTINPUT^.NEXTIN;
      NEW (CURRENTINPUT^.INPOINTER);  (* GET GATE FOR THAT INPUT *)
      READTREE (CURRENTINPUT^.INPOINTER); (* READ TREE FOR THAT INPUT *)
    END;
    CURRENTINPUT^.NEXTIN := NIL     (* NO MORE INPUTS *)
  END

  ELSE        (* THE GATE IS OF TYPE XXX *)
    CURRENTNODE^.INPUTLIST := NIL; (* NO INPUTS AT ALL *)
END;

PROCEDURE WRITETREE (CURRENTNODE:NODEP);
(* WRITES THE TREE TO A TEXT FILE *)

VAR CURRENTINPUT : INPUTP;
    I, NMROFINPUTS:INTEGER;
    OUTFILENAME:STRING40;'

    BEGIN
    (* WRITE THE DATA TO THE OUTPUT FILE *)
    NMROFINPUTS := CURRENTNODE^.NMR;
    WRITELN (OUTFILE, CURRENTNODE^.NAME);
    WRITELN (OUTFILE, CURRENTNODE^.NMR);
    WRITELN (OUTFILE, CURRENTNODE^.FUNCT);
    WRITELN (OUTFILE, CURRENTNODE^.PROB);
    IF (CURRENTNODE^.FUNCT <> 'XXX') THEN BEGIN
      CURRENTINPUT := CURRENTNODE^.INPUTLIST;  (*GET FIRST INPUT OF GATE*)
      WRITETREE (CURRENTINPUT^.INPOINTER);   (*WRITE DATA FOR THAT INPUT*)
      FOR I := 2 TO NMROFINPUTS DO BEGIN;   (*FOR EACH OF THE OTHER GATES*)
        CURRENTINPUT := CURRENTINPUT^.NEXTIN; (*GET THE NEXT INPUT*)
        WRITETREE (CURRENTINPUT^.INPOINTER); (*WRITE TREE FOR THAT INPUT*)
      END;
    END;
END;
```

```
(*$IOVERFLOW*)
(*CAUSE OVERFLOW.TEXT TO BE READ*)
PROCEDURE DSP (CURRENTNODE:NODEP);
VAR I:INTEGER;
    CURRENTINPUT:INPUTP;

BEGIN
  PAGE(OUTPUT);
  WRITELN;WRITELN;
  WRITE(CURRENTNODE^.NAME);
  IF CURRENTNODE^.FUNCT='XXX' THEN BEGIN
    WRITELN ('     PROBABILITY = ',CURRENTNODE^.PROB);
  END
  ELSE BEGIN
    WRITELN;
    WRITELN (CURRENTNODE^.FUNCT);
    CURRENTINPUT:= CURRENTNODE^.INPUTLIST;
    WRITELN;WRITELN;
    WRITELN ('INPUTS ARE:');
    WRITE ('     ', CURRENTINPUT^.INPOINTER^.NAME);
    IF (CURRENTINPUT^.INPOINTER^.FUNCT='XXX') THEN
      WRITE('    PROBABILITY = ',CURRENTINPUT^.INPOINTER^.PROB);
    WRITELN;
    FOR I:= 2 TO CURRENTNODE^.NMR DO BEGIN
      CURRENTINPUT:= CURRENTINPUT^.NEXTIN;
      WRITE('     ', CURRENTINPUT^.INPOINTER^.NAME);
      IF CURRENTINPUT^.INPOINTER^.FUNCT='XXX' THEN
        WRITE ('    PROBABILITY = ',CURRENTINPUT^.INPOINTER^.PROB);
      WRITELN;
    END;
  END;
END;


PROCEDURE DISPLAY(CURRENTNODE:NODEP;
                  TOPNODE:NODEP);
(* DISPLAYS A NODE AND ALL OF ITS INPUTS *)

VAR I:INTEGER;
    CURRENTINPUT:INPUTP;
    CURRNAME:STRING40;
    ISNAME:BOOLEAN;
    ANS:STRING40;

  BEGIN
    REPEAT
      WRITELN ('WHAT GATE WOULD YOU LIKE TO DISPLAY');
      READONELN (CURRNAME);
      SRCHTREE(TOPNODE,CURRENTNODE,CURRNAME,ISNAME);
      IF ISNAME THEN BEGIN
        DSP(CURRENTNODE);
      END
      ELSE BEGIN
        WRITELN ('CANT FIND ',CURRNAME);
      END;
      WRITELN ('WOULD YOU LIKE TO DISPLAY ANOTHER GATE?');
      READONELN (ANS);
    UNTIL ANS[1] <> 'Y';
  END;


PROCEDURE CHECKBOTTOM (CURRENTNODE:NODEP);
(* CHECKS TO SEE IF THERE ARE ANY NODES BELOW CURRENTNODE AND DELETES THEM*)
(* IF THE USER WISHES *)
```

```
VAR ANS:STRING40;
    IPROB:REAL;
    PROBOK:BOOLEAN;

 BEGIN
   IF CURRENTNODE^.FUNCT<> 'XXX' THEN BEGIN
     DSP(CURRENTNODE);
     WRITELN ('DO YOU WANT TO DELETE EVERYTHING BELOW ', CURRENTNODE^.NAME);
     WRITELN ('TYPE Y OR N');
     READONELN (ANS);
     IF (ANS[1]<>'Y') THEN BEGIN
       WRITELN ('PLEASE GIVE THESE GATES A DIFFERENT OUTPUT BEFORE');
       WRITELN ('DELETING ', CURRENTNODE^.NAME);
     END
     ELSE BEGIN (*MAKE CURRENTNODE A BOTTOM EVENT*)
       REPEAT
         PROBOK:=TRUE;
         WRITELN ('TYPE PROBABILITY OF OCCURRENCE FOR ',CURRENTNODE^.NAME);
         READLN (IPROB);
         IF (IPROB<0)THEN BEGIN
           WRITELN ('CANT HAVE NEGATIVE PROBABILITY');
           PROBOK:=FALSE;
         END;
         IF (IPROB>1)THEN BEGIN
           WRITELN ('PROBABILITY CANT BE > 1');
           PROBOK:=FALSE;
         END;
       UNTIL (PROBOK=TRUE);
       CURRENTNODE^.PROB:=IPROB;
       CURRENTNODE^.INPUTLIST:=NIL;
       CURRENTNODE^.FUNCT:='XXX';
       CURRENTNODE^.NMR:=0;
     END;
   END;
 END;


PROCEDURE FINDPARENT (CURRENTNODE : NODEP;
                      VAR INNODE : NODEP;
                      INNAME      :STRING40;
                      VAR PARENTNODE: NODEP;
                      VAR FOUND: BOOLEAN);
VAR CURRENTINPUT:INPUTP;
    I,NMROFINPUTS: INTEGER;

BEGIN
  FOUND := FALSE;
  NMROFINPUTS:= CURRENTNODE^.NMR;
  IF INNAME=CURRENTNODE^.NAME THEN BEGIN
    WRITELN ('TOP EVENT - CANT FIND PARENT');
  END
  ELSE IF CURRENTNODE ^.FUNCT <> 'XXX' THEN BEGIN
    CURRENTINPUT := CURRENTNODE^.INPUTLIST;
    IF (CURRENTINPUT^.INPOINTER^.NAME=INNAME) THEN BEGIN
      PARENTNODE := CURRENTNODE;
      INNODE := CURRENTINPUT^.INPOINTER;
      FOUND := TRUE;
    END
    ELSE FINDPARENT (CURRENTINPUT^.INPOINTER,INNODE,INNAME,PARENTNODE,FOUND);
    I:= 2;
    WHILE ((NOT FOUND) AND (I<= NMROFINPUTS)) DO BEGIN
      CURRENTINPUT := CURRENTINPUT^.NEXTIN;
      IF (CURRENTINPUT^.INPOINTER^.NAME = INNAME) THEN BEGIN
        INNODE := CURRENTINPUT^.INPOINTER;
```

```
          PARENTNODE := CURRENTNODE;
          FOUND := TRUE;
        END
        ELSE FINDPARENT(CURRENTINPUT^.INPOINTER,INNODE,INNAME,PARENTNODE,FOUND);
        I := I+1;
      END (*WHILE*)
    END  (*IF*)
END;


PROCEDURE DELGATE (TOPPOINTER:NODEP);
(*DELETES CURRENTNODE FROM THE TREE*)

VAR LASTNODE,OTHERNODE,CURRNODE : NODEP;
    ANS : STRING40;
    LASTINPUT, CURRENTINPUT:INPUTP;
    PROBOK,DELETED,FOUND:BOOLEAN;
    I:INTEGER;
    IPROB:REAL;
    CURRENTNAME:STRING40;

BEGIN
  ANS:=' ';               (*INITIALIZE*)
  REPEAT
    WRITELN ('WHAT GATE WOULD YOU LIKE TO DELETE');
    READONELN (CURRENTNAME);
    SRCHTREE(TOPPOINTER,CURRNODE,CURRENTNAME,FOUND);
    IF NOT FOUND THEN
      WRITELN('CANT FIND ',CURRENTNAME)
    ELSE BEGIN
      FINDPARENT(TOPPOINTER, CURRNODE ,CURRENTNAME, LASTNODE, FOUND);
    END;
    IF NOT FOUND THEN BEGIN
      WRITELN ('TRY ANOTHER NAME?');
      READONELN (ANS);
    END;
  UNTIL ((FOUND) OR (ANS[1]<>'Y'));
  IF  FOUND THEN BEGIN
    CHECKBOTTOM(CURRNODE);     (*MAKE IT A BOTTOM EVENT*)
    IF (CURRNODE^.FUNCT = 'XXX') THEN BEGIN
      IF LASTNODE^.NMR=2 THEN BEGIN
        DSP (LASTNODE);
        WRITELN ('DO YOU WANT TO DELETE BOTH INPUTS TO ',LASTNODE^.NAME);
        WRITELN ('TYPE Y OR N');
        READONELN (ANS);
        IF ANS[1]='Y' THEN BEGIN
        (* MAKE SURE THE OTHER INPUT IS A BOTTOM EVENT*)
          CURRENTINPUT := LASTNODE ^.INPUTLIST;
          IF CURRENTINPUT^.INPOINTER = CURRNODE THEN BEGIN
            OTHERNODE := CURRENTINPUT^.NEXTIN^.INPOINTER;
          END
          ELSE BEGIN
            OTHERNODE := CURRENTINPUT^.INPOINTER;
          END;
          CHECKBOTTOM(OTHERNODE);          (*MAKE IT A BOTTOM EVENT*)
          IF (OTHERNODE^.FUNCT='XXX') THEN BEGIN
            LASTNODE^.INPUTLIST:= NIL;
            LASTNODE^.FUNCT:='XXX';
            LASTNODE^.NMR:= 0;
            REPEAT
              PROBOK:=TRUE;
              WRITELN ('TYPE THE PROBABILITY OF OCCURRENCE FOR ',LASTNODE^.NAME)
              READLN(IPROB);
              IF (IPROB<0)THEN BEGIN
                WRITELN ('PROBABILITY CANT BE LESS THAN 0');
                PROBOK:=FALSE;
              END;
              IF (IPROB>1)THEN BEGIN
                WRITELN ('PROBABILITY CANT BE GREATER THAN 1');
                PROBOK:=FALSE;
              END;
            UNTIL (PROBOK=TRUE);
            LASTNODE^.PROB:=IPROB;
          END;
        END;
      END
      ELSE BEGIN (*IF NUMBER OF INPUTS IS GREATER THAN 2*)
        CURRENTINPUT := LASTNODE^.INPUTLIST;
        IF (CURRENTINPUT^.INPOINTER=CURRNODE) THEN BEGIN
          LASTNODE^.INPUTLIST:= CURRENTINPUT^.NEXTIN;
        END
        ELSE BEGIN
          DELETED:= FALSE;
          I:= 2;
          WHILE ((I<LASTNODE^.NMR) AND (NOT DELETED)) DO BEGIN
            LASTINPUT := CURRENTINPUT;
            CURRENTINPUT := CURRENTINPUT^.NEXTIN;
            IF CURRENTINPUT^.INPOINTER=CURRNODE THEN BEGIN
              LASTINPUT^.NEXTIN := CURRENTINPUT^.NEXTIN;
              DELETED := TRUE;
            END;
            I:= I+1;
          END;
          IF NOT DELETED THEN (*IT MUST BE THE LAST INPUT*)
            CURRENTINPUT^.NEXTIN:= NIL;
          END;
        LASTNODE^.NMR := LASTNODE^.NMR -1;
      END;
    END;
  END;
END;


 PROCEDURE ASSIGN (CURRENTINPUT:INPUTP;
                   CURRENTNODE:NODEP;
                   INAME:STRING40;
                   IPROB:REAL);

 (*ASSIGNS DATA TO THE GATE THAT HAS BEEN ADDED*)
 BEGIN
   NEW (CURRENTINPUT^.INPOINTER);
   (*ASSIGN DATA TO THE GATE*)
   WITH CURRENTINPUT^.INPOINTER^  DO BEGIN
     FUNCT := 'XXX';
     NMR:= 0;
     INPUTLIST := NIL;
     NAME := INAME;
     PROB := IPROB;
   END;
   CURRENTNODE^.NMR := CURRENTNODE ^.NMR+1;
   WRITELN ('MEMORY AVAILABLE = ',MEMAVAIL);
END;


PROCEDURE ADDGATE(CURRENTNODE, TOPPOINTER:NODEP);
(*ADDS A GATE TO THE TREE WITH OUTPUT TO CURRENTNODE*)

VAR I:INTEGER;
   ANS, ANS1, ANS2:STRING40;
   INAME:STRING40;
```

```
    IPROB:REAL;
    CURRENTINPUT:INPUTP;
    SRCHNODE:NODEP;
    PROBOK,ISNAME,ADD:BOOLEAN;

PROCEDURE CHANGEBOTTOM ;
  BEGIN
    ADD := TRUE;
    REPEAT
      WRITELN('TYPE NAME OF NODE TO BE ADDED');
      READONELN(INAME);
      SRCHTREE(TOPPOINTER,SRCHNODE, INAME,ISNAME);
      IF ISNAME THEN BEGIN
        WRITELN ('NAME ALREADY EXISTS ON THE TREE. DO YOU WISH TO');
        WRITELN ('TRY ANOTHER NAME?');
        (* DO NOT ALLOW THE SAME GATE NAME TO BE USED TWICE ON THE TREE*)
        READONELN (ANS);
        IF ANS[1] <> 'Y' THEN ADD:=FALSE;
      END;
    UNTIL (ISNAME =FALSE) OR (NOT ADD);
    IF ADD THEN BEGIN
      REPEAT
        PROBOK:=TRUE;
        WRITELN ('TYPE PROBABILITY OF OCCURRENCE FOR NEW NODE');
        READLN(IPROB);
        IF IPROB>1 THEN BEGIN
          WRITELN('PROBABILITY CANT BE GREATER THAN 1');
          PROBOK:=FALSE;
        END;
        IF IPROB<0 THEN BEGIN
          WRITELN ('PROBABILITY CANT BE LESS THAN 0');
          PROBOK:=FALSE;
        END;
      UNTIL (PROBOK=TRUE);
      REPEAT
        WRITELN('SINCE ',CURRENTNODE^.NAME,' IS A BOTTOM EVENT, IT MUST ');
        WRITELN('HAVE AT LEAST 2 INPUTS.  WOULD YOU LIKE IT TO BE AN AND (A)');
        WRITELN('OR AN OR(O) GATE?  (TYPE L TO LEAVE THE ADD PROCEDURE AND');
        WRITELN('DELETE THE GATE JUST ADDED)');
        WRITELN('PLEASE TYPE A, L, OR O');
        READONELN (ANS1);
      UNTIL((ANS1[1]='L') OR (ANS1[1]='A') OR (ANS1[1]='O'));
      IF ANS1[1]='L' THEN ADD:= FALSE;
    END;
    IF ADD THEN BEGIN
      (* GET A SPACE IN MEMORY FOR THE NEW NODE *)
      NEW (CURRENTNODE^.INPUTLIST);
      CURRENTINPUT := CURRENTNODE^.INPUTLIST;
      CURRENTINPUT^.NEXTIN:= NIL;
      ASSIGN (CURRENTINPUT,CURRENTNODE,INAME,IPROB);
      IF ANS1[1] = 'A' THEN CURRENTNODE^.FUNCT := 'AND';
      IF ANS1[1] = 'O' THEN CURRENTNODE^.FUNCT :='OR ';
      WRITELN ('CANT HAVE A SINGLE INPUT AND/OR GATE');
    END;
  END;


BEGIN (*ADDGATE*)
  IF (CURRENTNODE^.FUNCT='XXX')THEN BEGIN
    CHANGEBOTTOM;
  END;
  IF (CURRENTNODE^.FUNCT<>'XXX') THEN BEGIN
    ADD := TRUE;
    REPEAT
      REPEAT
```

```
          WRITELN ('TYPE NAME OF GATE TO BE ADDED');
          READONELN (INAME);
          SRCHTREE (TOPPOINTER, SRCHNODE, INAME, ISNAME);
          IF ISNAME THEN BEGIN
            WRITELN ('NAME ALREADY EXISTS IN TREE. DO YOU WISH TO');
            WRITELN ('TRY ANOTHER NAME?');
            READONELN (ANS);
            IF (ANS[1]<>'Y') THEN ADD:=FALSE;
          END;
        UNTIL ((ISNAME=FALSE) OR (ADD=FALSE));
        IF ADD THEN BEGIN
          REPEAT
            PROBOK:=TRUE;
            WRITELN ('TYPE PROBABILITY OF OCCURRENCE FOR NEW NODE');
            READLN (IPROB);
            IF (IPROB>1)THEN BEGIN
              WRITELN('PROBABILITY CANT BE GREATER THAN 1');
              PROBOK:=FALSE;
            END;
            IF (IPROB<0)THEN BEGIN
              WRITELN ('PROBABILITY CANT BE LESS THAN 0');
              PROBOK:=FALSE;
            END;
          UNTIL (PROBOK=TRUE);
          (*ADD THE NEW GATE IN MEMORY*)
          (*GET TO THE END OF THE INPUT LIST*)
          CURRENTINPUT := CURRENTNODE^.INPUTLIST;
          WHILE CURRENTINPUT^.NEXTIN<>NIL DO BEGIN
            CURRENTINPUT := CURRENTINPUT^.NEXTIN;
          END;
          NEW (CURRENTINPUT^.NEXTIN);
          CURRENTINPUT := CURRENTINPUT^.NEXTIN;
          CURRENTINPUT^.NEXTIN:= NIL;
          ASSIGN (CURRENTINPUT,CURRENTNODE,INAME,IPROB);
          WRITELN ('DO YOU WANT TO ADD ANOTHER INPUT TO ',CURRENTNODE^.NAME);
          WRITELN ('TYPE Y OR N');
          READONELN (ANS2);
          IF ANS2[1]<>'Y' THEN ADD:= FALSE;
        END;
      UNTIL (ADD=FALSE);
    END;
END;


PROCEDURE OUTPUTCHANGE (TOPPOINTER,CURRENTNODE:NODEP);
VAR DELETED,ISAME,FOUND:BOOLEAN;
     INAME:STRING40;
     NEWNODE,INNODE,PARENTNODE:NODEP;
     I:INTEGER;
     LASTINPUT,CURRENTINPUT:INPUTP;
     ANS:STRING40;

BEGIN
  FINDPARENT(TOPPOINTER,INNODE,CURRENTNODE^.NAME,PARENTNODE,FOUND);
  IF FOUND THEN BEGIN
    IF PARENTNODE^.NMR=2 THEN BEGIN
       WRITELN('CANT DELETE ONE OF TWO INPUTS TO ',PARENTNODE^.NAME);
    END
    ELSE BEGIN
      REPEAT
        ISNAME:= FALSE;          (*INITIALIZE ISNAME*)
        WRITELN ('OUTPUT TO WHICH GATE?');
        READONELN (INAME);
        IF (INAME=CURRENTNODE^.NAME) THEN BEGIN
          WRITELN ('CANT OUTPUT A GATE TO ITSELF. ');
    END
    ELSE BEGIN
      SRCHTREE(CURRENTNODE, NEWNODE, INAME, ISNAME);
      IF ISNAME THEN BEGIN
        ISNAME := FALSE;
        WRITELN('CANT CHANGE OUTPUT OF ', CURRENTNODE^.NAME, ' TO AN');
        WRITELN('EVENT THAT CAUSES IT.');
      END
      ELSE BEGIN
        SRCHTREE(TOPPOINTER,NEWNODE,INAME,ISNAME);
        IF NOT ISNAME THEN BEGIN
          WRITELN ('CANT FIND ',INAME,'. ');
        END
        ELSE BEGIN
          IF NEWNODE=PARENTNODE THEN BEGIN
            ISNAME := FALSE;          (*DO NOT CHANGE*)
            WRITELN ('OUTPUT OF ',CURRENTNODE^.NAME,' IS ALREADY AN');
            WRITELN ('INPUT OF ',NEWNODE^.NAME,'.');
          END
          ELSE BEGIN
            IF NEWNODE^.NMR=0 THEN BEGIN
              ISNAME := FALSE;
              WRITELN ('CANT ADD ONLY ONE GATE TO ',NEWNODE^.NAME);
            END;
          END;
        END;
      END;
    END;
    IF NOT ISNAME THEN BEGIN
      WRITELN ('DO YOU WISH TO CHANGE THE OUTPUT OF ');
      WRITELN (CURRENTNODE^.NAME, ' TO ANOTHER GATE?');
      READONELN(ANS);
    END;
  UNTIL ((ISNAME) OR (ANS[1]<>'Y'));
  IF ISNAME THEN BEGIN
    CURRENTINPUT:=NEWNODE^.INPUTLIST;
    (*ADD CURRENTNODE TO END OF NEWNODE*)
    FOR I:= 2 TO NEWNODE^.NMR DO
      CURRENTINPUT:= CURRENTINPUT^.NEXTIN;
    NEW (CURRENTINPUT^.NEXTIN);
    CURRENTINPUT:=CURRENTINPUT^.NEXTIN;
    NEW(CURRENTINPUT^.INPOINTER);
    CURRENTINPUT^.INPOINTER:= CURRENTNODE;
    CURRENTINPUT^.NEXTIN:= NIL;
    NEWNODE^.NMR := NEWNODE^.NMR+1;
    (*DELETE CURRENTNODE FROM PARENTNODE*)
    DELETED := FALSE;
    CURRENTINPUT:= PARENTNODE^.INPUTLIST;
    IF CURRENTINPUT^.INPOINTER=CURRENTNODE THEN BEGIN
      PARENTNODE^.INPUTLIST:=CURRENTINPUT^.NEXTIN;
      DELETED:= TRUE;
    END
    ELSE BEGIN
      I:= 2;
      WHILE ((I<PARENTNODE^.NMR) AND (NOT DELETED)) DO BEGIN
        LASTINPUT :=  CURRENTINPUT;
        CURRENTINPUT := CURRENTINPUT^.NEXTIN;
        IF CURRENTINPUT^.INPOINTER=CURRENTNODE THEN BEGIN
          LASTINPUT^.NEXTIN:= CURRENTINPUT^.NEXTIN;
          DELETED:=TRUE;
        END; (*IF*)
      END;(*WHILE*)
      IF NOT DELETED THEN (*DELETE LAST INPUT*)
        CURRENTINPUT^.NEXTIN := NIL;
    END;
```

```
          PARENTNODE^.NMR  := PARENTNODE^.NMR-1;
        END;
      END;
    END;
  END;


  PROCEDURE CHANGE (TOPPOINTER,CURRENTNODE:NODEP);
  (* CHANGES THE NAME, FUNCTION, PROBABILITY, OR OUTPUT OF A GATE OF THE TREE*)
    VAR ANS, ANS2:STRING40;
        INAME:STRING40;
        DUMMY:STRING3;
        NEWNODE,INNODE,PARENTNODE:NODEP;
        IPROB:REAL;
        PROBOK:BOOLEAN;

    BEGIN
      REPEAT
        REPEAT
          (*DISPLAY A MENU*)
          PAGE(OUTPUT);  (*CLEAR THE SCREEN*)
          WRITELN;WRITELN;WRITELN;WRITELN;
          WRITELN('DO YOU WANT TO:');
          WRITELN;
          WRITELN('    CHANGE THE NAME OF THE GATE (N)');
          WRITELN;
          WRITELN('    CHANGE THE FUNCTION OF THE GATE (F)');
          WRITELN;
          WRITELN('    CHANGE THE OUTPUT OF THE GATE (O)');
          WRITELN;
          WRITELN('    CHANGE THE PROBABILITY OF THE GATE (P)');
          WRITELN;
          WRITELN('    TO EXIT THE CHANGE PROCEDURE TYPE E');
          WRITELN;WRITELN;
          WRITELN('PLEASE TYPE N,F,O,P, OR E');
          READONELN(ANS);
        UNTIL ((ANS[1]='N') OR (ANS[1]='F') OR (ANS[1]='O') OR (ANS[1]='P')
          OR (ANS[1]='E'));
        IF (ANS[1]='N') THEN BEGIN
          WRITELN ('TYPE NEW NAME FOR ',CURRENTNODE^.NAME);
          READONELN (INAME);
          CURRENTNODE^.NAME:= INAME;
        END;
        IF (ANS[1]='F') THEN BEGIN
          IF CURRENTNODE^.FUNCT='AND' THEN BEGIN
            DUMMY:='OR';
            WRITELN ('FUNCTON=OR');      (*DEBUG OUTPUT*)
          END;
          IF CURRENTNODE^.FUNCT='OR' THEN BEGIN
            DUMMY:='AND';
            WRITELN ('FUNCTION=AND');
          END;
          IF CURRENTNODE^.FUNCT='XXX' THEN BEGIN
            WRITELN ('BOTTOM EVENT - CANT CHANGE GATE FUNCTION');
          END
          ELSE BEGIN
            CURRENTNODE^.FUNCT:= DUMMY;
          END;
        END;
        IF (ANS[1]='O') THEN BEGIN
          OUTPUTCHANGE (TOPPOINTER,CURRENTNODE);
        END;
        IF (ANS[1]='P') THEN BEGIN
          REPEAT
            PROBOK:=TRUE;
        WRITELN('TYPE NEW PROBABILITY');
        READLN (IPROB);
        IF (IPROB>1)THEN BEGIN
          WRITELN('PROBABILITY CANT BE GREATER THAN 1');
          PROBOK:=FALSE;
        END;
        IF (IPROB<0)THEN BEGIN
          WRITELN ('PROBABILITY CANT BE LESS THAN 0');
          PROBOK:=FALSE;
        END;
      UNTIL (PROBOK=TRUE);
      CURRENTNODE^.PROB := IPROB;
    END;
    IF (ANS[1]<>'E') THEN BEGIN
      WRITELN ('DO YOU WANT TO MAKE ANY OTHER CHANGES TO ',CURRENTNODE^.NAME);
      WRITELN ('  TYPE Y OR N');
      READONELN (ANS2);
      IF (ANS2[1]<>'Y') THEN ANS:='E';
    END;
  UNTIL (ANS[1]='E');
END;


PROCEDURE APPEND(TOPPOINTER:NODEP);

VAR CURRENTNODE:NODEP;
    APPFILENAME,CURRNAME:STRING40;
    ISNAME:BOOLEAN;
    ANS:STRING40;
    CURRENTINPUT:INPUTP;

 BEGIN
   REPEAT
     ANS:=' ';                    (*INITIALIZE*)
     WRITELN ('INPUT TO WHICH GATE');
     READONELN (CURRNAME);
     SRCHTREE (TOPPOINTER,CURRENTNODE, CURRNAME,ISNAME);
     IF ISNAME THEN BEGIN
       IF CURRENTNODE^.NMR=0 THEN BEGIN
         WRITELN (CURRNAME, ' CANT HAVE A SINGLE INPUT.');
         ISNAME := FALSE;
       END;
     END
     ELSE BEGIN
       WRITELN ('CANT FIND ',CURRNAME,' IN TREE.');
     END;
     IF NOT ISNAME THEN BEGIN
       WRITELN ('DO YOU WISH TO TRY ANOTHER NAME?');
       READONELN (ANS);
     END;
   UNTIL ((ISNAME) OR (ANS[1]<>'Y'));
   IF (ISNAME) THEN BEGIN
     WRITELN ('FILE TO ADD TO ',CURRNAME);
     READONELN (APPFILENAME);
     (* GET TO END OF INPUT LIST*)
     CURRENTINPUT := CURRENTNODE^.INPUTLIST;
     FOR I:= 2 TO CURRENTNODE^.NMR DO
       CURRENTINPUT:= CURRENTINPUT^.NEXTIN;
     NEW (CURRENTINPUT^.NEXTIN);
     CURRENTINPUT := CURRENTINPUT^.NEXTIN;
     CURRENTINPUT^.NEXTIN:= NIL;
     NEW (CURRENTINPUT^.INPOINTER);
     RESET (INPUTFILE,APPFILENAME);
     READTREE(CURRENTINPUT^.INPOINTER);
```

```
        CLOSE (INPUTFILE,LOCK);
        CURRENTNODE^.NMR:=CURRENTNODE^.NMR+1;
      END;
    END;



PROCEDURE MENU(TOPNODE:NODEP);
  VAR ANS,ANS2:STRING40;
      ISNAME:BOOLEAN;
      CURRENTNODE:NODEP;
      APPFILENAME,CURRNAME,OUTFILENAME:STRING40;

BEGIN
  ANS:=' ';  ANS2:=' ';               (*INITIALIZE THE VARIABLES*)
  REPEAT
    REPEAT
    (*DISPLAY THE MENU*)
      PAGE(OUTPUT);    (*CLEAR THE SCREEN*)
      WRITELN;WRITELN;WRITELN;
      WRITELN('MEMORY AVAILABLE=',MEMAVAIL,'.  DO YOU WANT TO:');
      WRITELN;WRITELN;
      WRITELN ('    ADD A NODE (A)');
      WRITELN;
      WRITELN ('    REMOVE A NODE (R)');
      WRITELN;
      WRITELN ('    APPEND A FILE TO THIS FILE (P)');
      WRITELN;
      WRITELN ('    CHANGE A GATE (C)');
      WRITELN;
      WRITELN ('    DISPLAY A GATE (D)');
      WRITELN;
      WRITELN ('    WRITE TO A FILE NAME (W)');
      WRITELN;
      WRITELN ('    EXIT THE PROGRAM (E)');
      WRITELN;WRITELN;
      WRITELN ('PLEASE TYPE A,R,P,C,D,W, OR E');
      READONELN (ANS2);
    UNTIL ((ANS2[1]='A') OR (ANS2[1]='D') OR (ANS2[1]='P') OR (ANS2[1]='C')
     OR (ANS2[1]='R')OR (ANS2[1]='W') OR (ANS2[1]='E'));
    IF ANS2[1]='A' THEN BEGIN           (*ADD A NODE TO THE FILE*)
      REPEAT
        WRITELN ('TO WHICH GATE WOULD YOU LIKE TO ADD AN OUTPUT?');
        READONELN (CURRNAME);
        SRCHTREE (TOPNODE,CURRENTNODE,CURRNAME,ISNAME);
        IF NOT ISNAME THEN BEGIN
          WRITELN ('CANT FIND ',CURRNAME,' IN TREE. DO YOU WISH TO TRY ');
          WRITELN ('ANOTHER NAME?');
          READONELN (ANS);
        END;
      UNTIL ((ANS[1]<>'Y') OR (ISNAME));
      IF ISNAME THEN ADDGATE(CURRENTNODE,TOPNODE);
    END;
    IF ANS2[1]='R' THEN BEGIN         (*DELETE A GATE FROM THE FILE*)
      DELGATE (TOPNODE);
    END;
    IF ANS2[1]='P' THEN BEGIN         (*APPEND A FILE TO THIS FILE*)
      APPEND(TOPNODE);
    END;
    IF ANS2[1]='C' THEN BEGIN           (*CHANGE A GATE OF THE TREE*)
      REPEAT
        WRITELN ('WHAT GATE WOULD YOU LIKE TO CHANGE');
        READONELN (CURRNAME);
        SRCHTREE (TOPNODE,CURRENTNODE,CURRNAME,ISNAME);
        IF NOT ISNAME THEN BEGIN
```

APPENDIX B. SOURCE LISTING OF "FAULTTREE" PROGRAM

The FAULTTREE program, described fully in Chapters IV and V, uses the output of the EDITOR program to find the minimal cut sets and failure probabilities of the fault tree. Sample output is given in Appendix C.

```
(**************    FAULTTREE.TEXT    ***************
  THIS PROGRAM READS AND PROCESSES A GENERAL FAULT TREE HAVING "AND" AND
  "OR" GATES.  THE TREE IS ENTERED BY THE USER BY EDITING A FILE WITH THE
  PROPER FORMAT AND GIVING THE FILE NAME WHEN ASKED.  THE PROCESSING
  CONSISTS OF FINDING ALL OF THE MINIMAL CUT SETS OF THE TREE.  A LISTING
  OF THESE CUT SETS CAN BE ROUTED TO THE TERMINAL SCREEN, THE PRINTER, OR
  TO A DISK FILE FOR LATER USE.  *)


TYPE STRING40 = STRING[40];
     STRING3  = STRING[3];

     NODE=RECORD        (*THESE ARE THE NODES WITH THE ACTUAL INFORMATION*)
                        (* IN FIGURE 2*)
            NAME:STRING40;   (* NAME OF THE EVENT. *)
            FUNCT:STRING3;   (* "AND", "OR", "XXX" *)
            PROB:REAL;       (* PROBABILITY OF OCCURRENCE IF BOTTOM EVENT *)
            INPUTLIST:^INPUTTYPE
          END;
     NODEP=^NODE;

     INPUTTYPE=RECORD  (*THESE ARE THE "CONNECTOR" NODES IN FIGURE 2*)
                 INPOINTER:^NODE;
                 NEXTIN:^INPUTTYPE
               END;
     INPUTP=^INPUTTYPE;

     CUTSETNODE=RECORD    (*THESE ARE THE INFORMATION NODES IN FIGURE 3*)
                  ISBOTTOM: BOOLEAN;  (* TRUE IF NOTHING CAUSES EVENT *)
                  NAME:STRING40;      (* EVENT NAME *)
                  PROB:REAL;          (* PROBABILITY OF OCCURRENCE *)
                  NEXTNODE:^CUTSETNODE
                END;
     CUTSETP=^CUTSETNODE;

     CUTLISTNODE=RECORD  (*THESE ARE THE "CONNECTOR" NODES IN FIGURE 3*)
                   SETPOINTER:^CUTSETNODE;
                   NEXTLIST:^CUTLISTNODE;
                   LISTPROB:REAL;         (* PROBABILITY OF CUTSET OCCURRENCE *)
                 END;
     CUTLISTP=^CUTLISTNODE;

VAR
  TOPPOINTER:NODEP;                  (* MAIN FAULT TREE AFTER READING *)
  CUTSETLIST:CUTLISTP;               (* LIST OF CUT SETS *)
  FILENAME, OUTFILENAME:STRING40;    (* INPUT AND OUTPUT FILE NAMES *)
  INPUTFILE, OUTFILE:TEXT;           (* INPUT AND OUTPUT FILE IDENTIFIERS *)
  PROB,TOTALPROB:REAL;               (* PROBABILITIES OF FAILURE *)


PROCEDURE READTREE (CURRENTNODE:NODEP);
(* READS TREE UNDER CURRENTNODE FROM INPUT FILE. *)

VAR
  INNAME:STRING40;
  GATE   :STRING3;
  NMROFINPUTS:INTEGER;
  CURRENTINPUT:^INPUTTYPE;
  I:INTEGER;

BEGIN
  IF EOF(INPUTFILE) THEN BEGIN
    WRITELN;
    WRITELN('***ERROR, EXPECTED MORE INPUT.');
    HALT;
  END;
  READLN (INPUTFILE,INNAME);          (* READ DATA FROM INPUT FILE *)
  READLN (INPUTFILE,NMROFINPUTS);
  READLN (INPUTFILE,GATE);
  READLN (INPUTFILE,PROB);
  CURRENTNODE^.NAME  := INNAME;       (* LOAD NODE WITH INPUT DATA *)
  CURRENTNODE^.PROB  := PROB;
  CURRENTNODE^.FUNCT  := GATE;

  IF GATE <> 'XXX' THEN BEGIN
    NEW (CURRENTINPUT);               (* GET FIRST INPUT OF GATE *)
    CURRENTNODE^.INPUTLIST  := CURRENTINPUT;
    NEW (CURRENTINPUT^.INPOINTER); (* GET GATE FOR FIRST INPUT *)
    READTREE (CURRENTINPUT^.INPOINTER);  (* READ TREE FOR FIRST INPUT *)

    FOR I := 2 TO NMROFINPUTS DO BEGIN    (* FOR EACH OF THE OTHER GATES *)
      NEW (CURRENTINPUT^.NEXTIN);  (* GET NEXT INPUT *)
      CURRENTINPUT  := CURRENTINPUT^.NEXTIN;
      NEW (CURRENTINPUT^.INPOINTER);  (* GET GATE FOR THAT INPUT *)
      READTREE (CURRENTINPUT^.INPOINTER); (* READ TREE FOR THAT INPUT *)
    END;
    CURRENTINPUT^.NEXTIN  := NIL      (* NO MORE INPUTS *)
  END

  ELSE         (* THE GATE IS OF TYPE XXX *)
    CURRENTNODE^.INPUTLIST  := NIL; (* NO INPUTS AT ALL *)
END;



PROCEDURE ADD(NAME:STRING40;
              FUNCT: STRING3;
              PROB:REAL;
              VAR CUTSETLIST:CUTLISTP);
  (* ADDS EVENT NAME TO EACH CUT SET IN CUT SET LIST.  IF FUNCTION IS
     XXX, THEN FLAGS SET ELEMENT AS BOTTOM. *)
  VAR CURRENTSET:^CUTLISTNODE;
      NEWSETNODE:^CUTSETNODE;

  BEGIN
    IF CUTSETLIST = NIL THEN BEGIN  (* NO SETS IN CUTSET *)
      NEW(CUTSETLIST);                (* CREATE ONE CUTSET *)
      CUTSETLIST^.SETPOINTER  := NIL;(* WITH NOTHING IN IT *)
      CUTSETLIST^.NEXTLIST  := NIL
    END;

    CURRENTSET:= CUTSETLIST;
    WHILE CURRENTSET <> NIL DO BEGIN
      NEW (NEWSETNODE);             (* GET NEW SET ELEMENT *)
      NEWSETNODE^.NAME  := NAME;  (* AND INITIALIZE IT *)
      NEWSETNODE^.PROB  := PROB;
      IF FUNCT = 'XXX'
        THEN NEWSETNODE^.ISBOTTOM  := TRUE
        ELSE NEWSETNODE^.ISBOTTOM  := FALSE;
       (* LINK NEW ELEMENT INTO SET *)
      NEWSETNODE^.NEXTNODE  := CURRENTSET^.SETPOINTER;
      CURRENTSET^.SETPOINTER  := NEWSETNODE;
      CURRENTSET:=CURRENTSET^.NEXTLIST;  (* DO NEXT CUT SET *)
    END;
  END;

PROCEDURE COPY (CUTSETLIST:CUTLISTP;
                VAR CUTSETCOPY:CUTLISTP);
    (* MAKES A COPY OF THE CUT SET LIST AND POINTS CUTSETCOPYN AT IT *)
  VAR CURRENTSET,SETCOPY:^CUTLISTNODE;
```

```
    CURRENTELEMENT,ELEMENTCOPY:^CUTSETNODE;

  BEGIN
    CURRENTSET:= CUTSETLIST;
    NEW (CUTSETCOPY);    (* INIT COPY WITH FIRST CUT SET *)
    SETCOPY  := CUTSETCOPY;

    WHILE CURRENTSET <> NIL DO BEGIN      (* COPY EACH SET *)
      CURRENTELEMENT:= CURRENTSET^.SETPOINTER;
      NEW (ELEMENTCOPY);    (* INIT SET WITH FIRST ELEMENT *)
      SETCOPY ^.SETPOINTER:=ELEMENTCOPY;

      WHILE CURRENTELEMENT<>NIL DO BEGIN    (* COPY EACH ELEMENT IN SET *)
        ELEMENTCOPY^.NAME:= CURRENTELEMENT^.NAME;
        ELEMENTCOPY^.ISBOTTOM := CURRENTELEMENT^.ISBOTTOM;
        ELEMENTCOPY^.PROB := CURRENTELEMENT^.PROB;
        CURRENTELEMENT:= CURRENTELEMENT^.NEXTNODE;  (* NEXT ELEMENT *)
        IF CURRENTELEMENT<>NIL THEN BEGIN
          NEW (ELEMENTCOPY^.NEXTNODE);
          ELEMENTCOPY:= ELEMENTCOPY^.NEXTNODE
        END
        ELSE    (* NO MORE ELEMENTS IN SET *)
          ELEMENTCOPY^.NEXTNODE:= NIL;
      END;

      CURRENTSET:= CURRENTSET^.NEXTLIST;      (* NEXT SET IN LIST *)
      IF CURRENTSET<>NIL THEN BEGIN
        NEW(SETCOPY^.NEXTLIST);
        SETCOPY  := SETCOPY^.NEXTLIST;
      END
      ELSE      (* NO MORE SETS IN LIST *)
        SETCOPY^.NEXTLIST:= NIL;
    END;

  END;


PROCEDURE MERGECUTSETLISTS (VAR RESULT:CUTLISTP;
                                ADDITION:CUTLISTP);
  (* TACKS CUT SET POINTED AT BY ADDITTION ONTO CUTSET POINTRESULT *)
  VAR CURRENTSET:^CUTLISTNODE;

  BEGIN
    IF RESULT = NIL THEN
      RESULT := ADDITION  (* NOTHING IN RESULT *)
    ELSE BEGIN
      CURRENTSET:= RESULT;
      WHILE CURRENTSET^.NEXTLIST<>NIL DO (* FIND END OF RESULT SET LIST *)
        CURRENTSET:= CURRENTSET^.NEXTLIST;

      CURRENTSET^.NEXTLIST:= ADDITION;    (* TACK ADDITION ONTO RESULT *)
    END;
  END;


PROCEDURE FINDCUTSETS(CURRENTNODE:NODEP;
                      VAR CUTSETLIST:CUTLISTP);
  (* DETERMINS THE CUT SET LISTS FOR EACH INPUT CAUSING THE CURRENT EVENT *)
  VAR CURRENTINPUT:^INPUTTYPE;
      RESULTCUTSET:^CUTLISTNODE;
      CUTSETCOPY:^CUTLISTNODE;

  BEGIN
       (* FIRST ADD CURRENT EVENT TO CUTSETLIST *)
      ADD (CURRENTNODE^.NAME,CURRENTNODE^.FUNCT,CURRENTNODE^.PROB,CUTSETLIST);
      IF CURRENTNODE^.FUNCT='AND' THEN
        BEGIN
          CURRENTINPUT:= CURRENTNODE^.INPUTLIST;
          WHILE CURRENTINPUT <> NIL DO BEGIN    (* FIND FOR EACH INPUT *)
            FINDCUTSETS (CURRENTINPUT^.INPOINTER,CUTSETLIST);
            CURRENTINPUT:= CURRENTINPUT^.NEXTIN
          END;
        END

      ELSE IF (CURRENTNODE^.FUNCT = 'OR ') OR
              (CURRENTNODE^.FUNCT = 'OR')
        THEN BEGIN
          CURRENTINPUT  := CURRENTNODE^.INPUTLIST;
          RESULTCUTSET  := NIL;
          WHILE CURRENTINPUT <> NIL DO BEGIN  (* FIND FOR EACH INPUT *)
            COPY (CUTSETLIST,CUTSETCOPY);    (* BUT KEEP RESULTS SEPERATE *)
            FINDCUTSETS (CURRENTINPUT^.INPOINTER,CUTSETCOPY);
            MERGECUTSETLISTS(RESULTCUTSET,CUTSETCOPY);
            CURRENTINPUT:= CURRENTINPUT^.NEXTIN;
          END;

          CUTSETLIST  := RESULTCUTSET;
        END;
  END;


  PROCEDURE FINDPROB (CUTSETLLIST:CUTLISTP);
    VAR CURRENTSET:^CUTLISTNODE;
        CURRENTELEMENT:^CUTSETNODE;

  (* FIND PROBALITY OF OCCURRENCE FOR EACH CUTSET AND FOR TOP EVENT.*)

  BEGIN
    CURRENTSET  := CUTSETLIST;
    TOTALPROB   := 1;           (*PROBABILITY OF NO FAILURE PATHS OCCURRING *)
    WHILE CURRENTSET <> NIL DO BEGIN
    (* REPEAT UNTIL ALL SETS ARE TRAVERSED *)
      CURRENTSET^.LISTPROB := 1; (* PROBABILITY OF PATH NOT OCCURRING *)
      CURRENTELEMENT  := CURRENTSET^.SETPOINTER; (* GO TO FIRST NODE *)
      WHILE CURRENTELEMENT <> NIL DO BEGIN
      (* REPEAT UNTIL ALL ELEMENTS ARE TRAVERSED *)
        IF CURRENTELEMENT^.ISBOTTOM THEN
          (* GET PROBABILITY OF THE BOTTOM ELEMENT *)
          CURRENTSET^.LISTPROB := CURRENTSET^.LISTPROB*CURRENTELEMENT^.PROB;
          (*UPDATE CUTSET PROBABILITY*)
        CURRENTELEMENT  := CURRENTELEMENT^.NEXTNODE;  (* GO TO NEXT ELEMENT *)
      END;
      TOTALPROB  := TOTALPROB*(1-CURRENTSET^.LISTPROB);
      (*UPDATE TOTAL PROBABILITY*)
      CURRENTSET  := CURRENTSET^.NEXTLIST;    (*GO TO NEXT CUTSET*)
    END;
    TOTALPROB  := 1-TOTALPROB;  (*PROBABILITY OF ONE OR MORE PATHS OCCURRING*)
  END;


PROCEDURE WRITECUTSETS (CUTSETLIST:CUTLISTP);
  VAR CURRENTSET:^CUTLISTNODE;
      CURRENTELEMENT :^CUTSETNODE;
  (* PRINTS THE CUTSET LIST TO THE OUTPUT FILE. *)

  BEGIN
    CURRENTSET  := CUTSETLIST;
    WHILE CURRENTSET<>NIL  DO BEGIN    (* PRINT EACH SET IN LIST *)
      CURRENTELEMENT:= CURRENTSET^.SETPOINTER;
      (* LEAVE 3 BLANK LINES BETWEEN SETS *)
     WRITELN(OUTFILE);WRITELN(OUTFILE);WRITELN(OUTFILE);

     WHILE CURRENTELEMENT<>NIL DO BEGIN   (* PRINT EACH ELEMENT IN SET *)
       IF CURRENTELEMENT^.NAME[1] <> '-'
         THEN       (* ONLY PRINT IF THERE IS A NAME *)
            IF CURRENTELEMENT^.ISBOTTOM
              THEN WRITELN(OUTFILE,'*',CURRENTELEMENT^.NAME)
              ELSE WRITELN(OUTFILE,' ',CURRENTELEMENT^.NAME);
       CURRENTELEMENT:= CURRENTELEMENT^.NEXTNODE; (*POINT AT NEXT ELEMENT*)
     END;

     WRITELN (OUTFILE,'PROBABILITY OF OCCURRENCE FOR THIS PATH IS',CURRENTSET
     ^.LISTPROB);
     CURRENTSET:= CURRENTSET^.NEXTLIST;  (* POINT AT NEXT SET IN LIST *);
   END;
   WRITELN (OUTFILE,'PROBABILITY OF OCCURRENCE FOR TOP EVENT IS',TOTALPROB );
 END;


BEGIN
  WRITELN;
  WRITELN ('TYPE THE NAME OF THE INPUT FILE');
  READLN (FILENAME);
  RESET (INPUTFILE, FILENAME);

  WRITELN;
  WRITELN ('WHERE DO YOU WANT THE OUTPUT TO GO?');
  WRITELN ('TYPE CONSOLE: OR PRINTER: OR FILE-NAME');
  READLN (OUTFILENAME);
  REWRITE (OUTFILE, OUTFILENAME);

  NEW (TOPPOINTER);
  READTREE (TOPPOINTER);
  CUTSETLIST := NIL;
  FINDCUTSETS (TOPPOINTER,CUTSETLIST);
  FINDPROB (CUTSETLIST);
  WRITECUTSETS (CUTSETLIST);

  CLOSE(INPUTFILE);
  CLOSE(OUTFILE, LOCK);
END.
```

APPENDIX C. SAMPLE OUTPUT

This sample output is the minimal cut sets for the conductor break fault tree given in figure 9. It is the output of the FAULTTREE program, listed in Appendix B.

```
*SELF FRACTURE
 MECHANICAL FRACTURE OF CONDUCTOR
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000


*YIELDING OF ADJACENT REINFORCEMENT
 MECHANICAL FRACTURE OF CONDUCTOR
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000


*CRYOSTAT RUPTURE
 MISSILE IMPACT
 MECHANICAL FRACTURE OF CONDUCTOR
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000


*MISSILE FROM ENVIRONMENT
 MISSILE IMPACT
 MECHANICAL FRACTURE OF CONDUCTOR
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000


*MISSILE FROM REACTOR
 MISSILE IMPACT
 MECHANICAL FRACTURE OF CONDUCTOR
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000


*RADIATION DAMAGE TO INSULATION
 INSULATION DEGRADATION
 ELECTRICAL SHORT
 HIGH TEMP FAILURE OF CONDUCTOR
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000


*STRESS CYCLING FATIGUE
 MECHANICAL INSULATION DEGRADATION
 INSULATION DEGRADATION
 ELECTRICAL SHORT
 HIGH TEMP FAILURE OF CONDUCTOR
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000


*STRESS DUE TO PREVIOUS QUENCHES
 MECHANICAL INSULATION DEGRADATION
 INSULATION DEGRADATION
 ELECTRICAL SHORT
 HIGH TEMP FAILURE OF CONDUCTOR
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000


*INSULATION MISSING OR TOO THIN
 IMPROPER ASSEMBLY OF INSULATION
 ELECTRICAL SHORT
 HIGH TEMP FAILURE OF CONDUCTOR
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000


*METAL INCLUSION IN INSULATION
 IMPROPER ASSEMBLY OF INSULATION
 ELECTRICAL SHORT
 HIGH TEMP FAILURE OF CONDUCTOR
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000


*CRACK
 FLAW IN INSULATION AS FABRICATED
 ELECTRICAL SHORT
 HIGH TEMP FAILURE OF CONDUCTOR
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000


*METAL INCLUSION
 FLAW IN INSULATION AS FABRICATED
 ELECTRICAL SHORT
 HIGH TEMP FAILURE OF CONDUCTOR
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000


*SPACER BREAKS
 COOLANT CHANNEL BLOCKAGE
 HIGH TEMP FAILURE OF CONDUCTOR
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000


*IMPURITIES FROM REFRIGERATION
 FLOW CHANNEL PLUGS
 COOLANT CHANNEL BLOCKAGE
 HIGH TEMP FAILURE OF CONDUCTOR
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000


*IMPURITIES FROM MAGNET
 FLOW CHANNEL PLUGS
 COOLANT CHANNEL BLOCKAGE
 HIGH TEMP FAILURE OF CONDUCTOR
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000
```

```
*RADIATION DAMAGE TO STABILIZER
 STABILIZER RESISTANCE TOO HIGH
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000


*GROWTH OF CRACK FROM NORMAL STRESS CYCLI
 STRESS INDUCED FLOW
 MECHANICAL FLAW IN STABILIZER
 STABILIZER RESISTANCE TOO HIGH
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000


*GROWTH OF CRACK FROM PREVIOUS QUENCHES
 STRESS INDUCED FLOW
 MECHANICAL FLAW IN STABILIZER
 STABILIZER RESISTANCE TOO HIGH
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000


*CRACK CAUSED DURING MAGNET ASSEMBLY
 MECHANICAL FLAW IN STABILIZER
 STABILIZER RESISTANCE TOO HIGH
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000


*CRACK IN STABILIZER OF CONDUCTOR
 MECHANICAL FLAW IN STABILIZER
 STABILIZER RESISTANCE TOO HIGH
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000


*IMPURITIES IN STABILIZER
 STABILIZER RESISTANCE TOO HIGH
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000


*RADIATION DAMAGE TO SUPERCONDUCTOR
 SUPERCONDUCTOR FILAMENTS DEGRADED
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000


*STRESS CYCLING NORMAL OPERATION
 FRACTURE DURING OPERATION
 STRESS FRACTURES FILAMENTS
 SUPERCONDUCTOR FILAMENTS DEGRADED
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000
```

```
*ABNORMAL STRESS FROM PREV QUENCHES
 FRACTURE DURING OPERATION
 STRESS FRACTURES FILAMENTS
 SUPERCONDUCTOR FILAMENTS DEGRADED
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000


*FRACTURE DURING ASSEMBLY
 STRESS FRACTURES FILAMENTS
 SUPERCONDUCTOR FILAMENTS DEGRADED
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000


*FABRICATION FRACTURE UNDETECTED
 SUPERCONDUCTOR FILAMENTS DEGRADED
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000


*EXCESSIVE B
 HEATING BEYOND RECOVERY POINT
 NON RECOVERY OF CONDUCTOR
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000


*EXCESSIVE CONDUCTOR MOVEMENT
 HEATING BEYOND RECOVERY POINT
 NON RECOVERY OF CONDUCTOR
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000


*VAPOR LOCK
 NON RECOVERY OF CONDUCTOR
 CONDUCTOR BREAK
PROBABILITY OF OCCURRENCE FOR THIS PATH IS 0.00000
PROBABILITY OF OCCURRENCE FOR TOP EVENT IS 0.00000
```