REAL-TIME PROCESS PLANT FAULT DIAGNOSIS

ZAHEDI BIN FISAL
Doctor of Philosophy

THE UNIVERSITY OF ASTON IN BIRMINGHAM

April 1989

The University of Aston in Birmingham

Real-time Process Plant Fault Diagnosis

Zahedi Bin Fisal **PhD 1989**

SUMMARY

Operators can become confused while diagnosing faults in process plant while in operation. This may prevent remedial actions being taken before hazardous consequences can occur. The work in this thesis proposes a method to aid plant operators in systematically finding the causes of any fault in the process plant. A computer aided fault diagnosis package has been developed for use on the widely available IBM PC compatible microcomputer. The program displays a coloured diagram of a fault tree on the VDU of the microcomputer, so that the operator can see the link between the fault and its causes. The consequences of the fault and the causes of the fault are also shown to provide a warning of what may happen if the fault is not remedied. The cause and effect data needed by the package are obtained from a hazard and operability (HAZOP) study on the process plant. The result of the HAZOP study is recorded as cause and symptom equations which are translated into a data structure and stored in the computer as a file for the package to access. Probability values are assigned to the events that constitute the basic causes of any deviation. From these probability values, the *a priori* probabilities of occurrence of other events are evaluated. A top-down recursive algorithm, called TDRA, for evaluating the probability of every event in a fault tree has been developed. From the *a priori* probabilities, the conditional probabilities of the causes of the fault are then evaluated using Bayes' conditional probability theorem. The *posteriori* probability values could then be used by the operators to check in an orderly manner the cause of the fault. The package has been tested using the results of a HAZOP study on a pilot distillation plant. The results from the test shows how easy it is to trace the chain of events that leads to the primary cause of a fault. These method could be applied in a real process environment.

Key words : Fault Diagnosis, HAZOP, Cause and Symptom Equations, Fault Tree, Bayes' Theorem, Probability

ACKNOWLEDGEMENTS

The author wishes to express his profound gratitude and appreciation to the following persons and institutions without whom, this thesis would not be a success.

- Dr. M.C. Jones for supervising this research work from December 1984 to August 1987.

- Dr. J.P. Fletcher who took over the supervision from September 1987.

- Mr. T.O. Folami for his advice on matters concerning the pilot distillation plant.

- Mr. Jason Court of Tektronix UK Ltd. for his kind assistance in obtaining the colour hardcopy from the VDU of the computer.

- The National University of Malaysia, the author's employer, for granting study leave so that the author could pursue his further studies at the University of Aston.

- The Government of Malaysia for sponsoring the author throughout his study at the University of Aston.

To:

My loving parents,

Fisal and Asmah

My loving and patient wife,

Rukiah

And my lovely children,

Farah Dayana

Ihsanulfitri

Asma' Amirah

LIST OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Page

LIST OF PLATES

CHAPTER ONE

1. INTRODUCTION

Human requirements are met using natural resources. The chemical process industries have evolved to efficiently change the natural materials to a more useful form. For many such transformations, energy in the form of heat, pressure or motion is involved, in either a single form or in some combination. The presence of physical energy can be potentially destructive if not handled properly. This can cause loss of containment and result in explosions, serious fire or release of toxic material. This thesis is about a method of analysing a chemical process so that accidents, arising from loss of containment of materials in the process, causing major hazards can be prevented.

Marshall [1] states that the beginning of the era of major chemical hazard is said to have started at Opau, Ludwigshafen in Germany when on the 21st September 1921 a store containing about 4000 tonnes of ammonuim nitrate mixture to be used as a fertilizer exploded, killing over 500 people. Since then there have been many other major incidents causing many injuries and fatalities. Table 1.1 lists some of the incidents that have occurred on chemical process plants. The reported incidents are given in references [2] and [4].

It can be seen from Table 1.1 that many of the reported accidents have occurred since the Second World War. It is during this period and up to the present day, that there have been rapid developments in the

Table 1.1 Occurrence of Major Accidents in the Chemical Process Industry in the Twentieth Century

	Date	Location	Nature of incident	No. of fatalities
1	13/12/26	St. Albans France	Release of Chlorine	19
2	28/5/28	Hamburg Germany	Release of Phosgene	11
3	13/12/29	Syracuse New York, USA	Release of Chlorine	1
4	24/12/39	Zarnesti Rumania	Release of Chlorine	60
5	20/10/44	Cleveland Ohio, USA	Fire involving liquified natural gas	128
6	5/11/47	Rauma, Finland	Release of Chlorine	19
7	28/7/48	Ludwigshafen Germany	Explosion involving dimethyl ether	207
8	4/4/52	Walsum Germany	Release of Chlorine	7
9	4/6/64	Antwerp Belgium	Explosion involving ethylene oxide	4
10	4/1/66	Feyzin, France	Fire involving propane	17
11	8/8/67	Lake Charles Louisiana, USA	Explosion involving isobutane	7
12	21/1/68	Pernis, Holland	Explosion involving mixed hdrocarbons	2
13	10/8/71	Amsterdam Holland	Explosion involving butadiene	8
14	1/6/74	Flixborough UK	Explosion involving cyclohexane	28
15	10/2/75	Antwerp Holland	Explosion involving ethylene	6
16	25/2/75	Longview Texas, USA	Explosion involving ethylene	4
17	7/11/75	Beek, Holland	Explosion involving propylene	14
18	3/4/77	Umm Said Qatar	Explosion and fire involving natural gas liquids	6
19	19/11/84	Mexico City Mexico	Explosion and fire involving liquified petroluem gas	144
20	3/12/85	Bhopal, India	Release of methyl isocyanate	3000

chemical, petrochemical and petroleum industries. Plants have grown in size, typically by a factor of 10 [6], since it was thought that there was no need to limit the throughput, particularly if the process was continuous rather than batchwise. Plants built in this period are often single stream. They use long sequential process trains with considerable recycle. Processes have become complex, using large items of high performance equipment. Operating conditions have become more severe, such that the operation of such plants is relatively difficult. Complex instrumentations and highly centralised control systems are used to reduce manning levels.

The above developments have resulted in an increased potential for major accidents to occur. The most frequent accident is loss of containment of materials. This can result in toxic release or production of a vapour cloud sufficiently large as to cause a serious fire or explosion. When such accidents do occur, losses in human, economic and ecological terms are likely.

Concern about the safe operation of plants has developed ever since steam boilers were used to provide power and heat in factories towards the end of the nineteenth century. At that time boiler explosions were frequently occurring, producing many casualties [2]. Most of the industrial safety activities and legislation on safety since then have been based entirely on the humanitarian principle that prevention of personal injury was the main consideration [7]. One such law in Britain to impose health and safety duties on people at work is the Factories Act 1961.

Lessons from major accidents have changed the approach to industrial safety from programmes concerned solely to minimise the risk of personal injury to the concept that safe working is an integral part of the efficient operation of a plant. This implies that preventing accidental damage to plant, machinery, buildings, processes, raw materials and the environment surrounding the plant, known commonly as loss prevention, is as important as preventing accidents to employees in the plant and people elsewhere. The United Kingdom Health and Safety at Work Act 1974 which is superimposed on the Factories Act 1961 not only provides protection for people at work but also includes provisions for prevention of risk to the health and safety of the general public which may arise from work activities [8]. The Act therefore provides for means of controlling major hazards to be introduced as statutory regulations.

Even before the year in which the Health and Safety at Work Act was passed, there was already concern in the United Kingdom about major hazards that could arise in chemical process plants [9]. The Robens Committee set up in 1970 to look into the matter of health and safety at work recognised the potential for major hazards to occur in the industry and in their report [5] recommended two steps to be taken in this area. The first was to set up a specialised unit within the Factory Inspectorate to keep under surveillance those hazards which had the potential for injuring the public. The second was to set up an advisory committee to give advice on explosive and flammable substances. It was only after the Flixborough disaster in 1974 [10, 19], that the second recommendation was acted upon, when the Advisory Committee on Major Hazards was set up by the Health and Safety

Commission to consider safety problems associated with large-scale industrial premises conducting potentially hazardous operations. The terms of reference given to the committee as stated in their first report [3] are:

> "To identify types of installation (excluding nuclear installation) which have the potential to present major hazards to employees or to the public or the environment, and to advise on measures of control, appropriate to the nature and degree of hazard, over the establishment, siting, layout, design, operation, maintenance and development of such installations, as well as over all development, both industrial and non-industrial, in the vicinity of such installations."

The Advisory Committee on Major Hazards sat for nine years and in their deliberations produced three reports [3, 4, 5] where a comprehensive series of recommendations were made. One of the recommendations was that special measures of a legislative character were required, over and above the general provisions of the Health and Safety at Work Act 1974, to take account of the need to safeguard the public. At around the same time, the European Community Directive on the Major Accident Hazards of Certain Industrial Activities [11] was published. This is said by Marshall [12] to have drawn heavily on studies incorporated in the First and Second Reports of the Advisory Committee on Major Hazards. As a result the Control of Industrial Major Accident Hazards Regulations 1984 (CIMAH Regulations) were passed by the British Parliament. A requirement of the CIMAH

Regulations is that all manufacturers must prove to the Health and Safety Executive that they have identified existing major accident hazards, adopted the appropriate safety measures, and provided the persons working on the site with information, training and equipment in order to ensure their safety.

One technique for hazard identification and assessment recommended by the Advisory Committee on Major Hazards in their Second Report [4] is hazard and operability study, known more commonly as HAZOP [13,14,15]. The origins and development of HAZOP will be discussed in Chapter 3. A HAZOP study is normally carried out by a team of experts who provide knowledge and experience appropriate to the objectives of the examination and to the stage of development of the project. An attempt is made to foresee every conceivable fault that can occur on every plant item. This is done by applying guide words such as NO, MORE, LESS, AS WELL AS, PART OF, REVERSE and OTHER THAN in turn to process intentions such as flow, temperature, pressure, liquid level or heat transfer. Causes of these faults are noted and the possible consequences are predicted. Data obtained from HAZOP study can be voluminous. Techniques have been developed to structure the data. One such structure is the Fault Tree [13, 16, 18].

Qualitatively a fault tree describes in pictorial form the paths which can lead from a primary or basic event through any intermediate sub-events, to a top event which usually is the hazardous failure. Quantitatively a reliability study may be carried out using the fault tree to calculate the probability of occurrence of a particular event from basic events which can cause the failure. The basic events are at the bottom of the tree.

Thus the HAZOP study with the aid of fault trees enables the design engineer to see how potential fault conditions propagate in the plant. In this way, a systematic modification of the plant design can be made so as to reduce the probability of any hazardous event occurring during plant operation [17]. Fault tree analysis can be used to investigate the cost benefit and effectiveness of implementing safety devices in process plants [17].

However, no matter how well a process plant is designed incorporating all aspects to prevent losses and making it as safe as possible, fault conditions may occur during plant operation. These faults may be caused by unexpected equipment failures. Examples of such failures are pumps not delivering the correct flowrate, control valves not responding to required positions, measuring instrument malfunction giving wrong readings and leakages due to worn seals or gaskets. Hence to detect abnormal conditions during plant operation, monitoring of the plant is an important function. In most modern plants, the monitoring function is done by computer control systems which raise an alarm if a process variable goes outside specified limits or if some equipment is not in a specified state. However not all plant faults are detectable by the process computer. Human operators are needed to monitor the state of process variables which are not on-line to the process computer and also to check for unusual events such as leakages or measuring instrument malfunction.

Once an alarm has been raised, showing that an abnormal condition has occurred, actions have to be taken by process operators to prevent the fault propagating to other events which may be hazardous. The plant may have to be shut down safely, by the operator or by an automatic safety protection system, so that the fault can be diagnosed and repairs can be made. This will incur financial losses due to lost production and time while diagnosis and repairs are being done.

However some fault conditions may not warrant such drastic action. The fault can be diagnosed and corrective action taken whilst the plant is still running. For this procedure to be successful, the diagnostic and corrective tasks have to be completed within a period of time dependent on the type of fault and the type of process equipment involved. This time period is the time before the fault develops to the point where a more serious event may occur. The corrective task may fail to address the fault, and then the plant has to be shut down safely before serious consequences develop.

Diagnosing a fault condition and identifying its likely causes can be a difficult and confusing task. Plant operators usually take short-cut methods to check plant items which they think are the most probable causes, without considering all the alternatives [33]. This course of action will not always be successful in finding the cause of the fault. Also, so many alarms may occur simultaneously that plant operators become confused. A systematic procedure for fault diagnosis has the potential to simplify the information presented to the operators. This simplifies the operator's task of finding the cause of fault conditions whenever they occur. A computer system can be used to implement such an effective fault finding method so that appropriate corrective actions can be taken soon enough.

Studies have shown that operators respond much better to diagrammatic representation than to a table or list of figures on a sheet or on the visual display unit (VDU) of a computer [20]. Thus a diagrammatic fault tree display on a VDU showing the alarmed event can be used as the base for a computer-aided diagnosis system. Such a display could show likely causes as branches of the fault tree and the consequences if the fault is not remedied within a time limit. Further, displaying of conditional probabilities of the likely causes will help the operator to rank the likely causes to be checked and validated. The time by which action should be taken can also be shown.

It has been the objective of this research to investigate and develop a fault diagnosis system using a microcomputer. This is a prototype for a system to aid plant operators to find causes of an alarm during plant operation. An interactive package has been produced to display on the VDU the possible causes of a fault condition that is to be investigated in the form of a fault tree. The package also displays the consequences of the fault condition and the probability estimates of the possible causes. This objective has been achieved to the point where the program can be demonstrated analysing fault trees derived from a real plant.

CHAPTER TWO

2. FAULT DIAGNOSIS

2.1 : Introduction

A prime concern in the chemical industry is to prevent losses either due to occurrence of major hazards or unscheduled stoppages or poor product quality. This is achieved by monitoring process variables and controlling their fluctuations within a desired range. In modern process plants, the task of monitoring and controlling process variables is almost entirely handled by an automatic control system, which tends increasingly to be based on a process computer.

When operating conditions vary outside their designed limits, the control system will generate alarm signals to notify operators the existence of abnormal conditions in the plant. If the abnormal conditions are left uncorrected, the variations of the process variables could result in catastrophic events such as an explosion, fire or the release of toxic chemicals. Some form of control or corrective actions have to take place to prevent any accident that might propagate from the fault. The form of control or corrective actions depends on the specific occassion prevailing and the immediate goal that is to be achieved at the time the fault has occured.

Rasmussen [21] outlines the appropriate actions to be taken following the detection of an accidental maloperation. First, a preselected set of critical variables is scanned to test whether there exist a data pattern related to preplanned safety action such as shutting down the plant. The goal for this action is to prevent damage to the plant equipment involved which can lead to a more serious accident. This sequence of actions can be performed by the operator, but also is in fact the function behind an automatic safety system.

If no immediate danger is present, the subsequent intention may be to compensate the effect of the fault. The objective is to avoid drastic automatic safety actions by finding possible means for counter-actions from the causal or functional flow of the system. The effect of this action is to move the plant to a new state, not necessarily similar to the original operating condition, so as to protect it from adverse consequences without shutting down the plant.

The two forms of action stated above can be done without consideration of the primary cause of the situation.

The ultimate goal is to restore the normal condition prior to occurence of the fault, whether or not the plant has been shut down or compensatory action has been executed. This is achieved by locating the components that cause the fault in order to be able to adjust or repair them.

Diagnosis of process malfunctions can be a difficult task for process operators because of the interactions of process components [24]. Various methods of fault diagnosis have been proposed in order to aid the process operator in finding the cause of a fault. Most of the methods require the use of computers to help the operator in the decision making process. This chapter reviews some of the computer aided fault diagnosis methods that have been published.

2.2 : Aspects of Fault Detection

A fault in the process plant is easily detected by directly checking measurable variables for upward or downward transgression of fixed limits or trends. This supervisory task can be easily automated using the process computer acting as a limit-value monitor, sending an alarm signal whenever a variable exceed its intended limits. However, this form of fault detection produces large number of alarms. Lees [22] has discussed the defects of process alarm systems.

There are several factors which cause alarms to be numerous. One is the confusion between alarms and statuses. A status indicates that an item is in a particular state, for example, a pump not running. If however the pump should be running but it is not, then this state constitutes an alarm. The confusion arises when alarm displays are also used as status displays. Thus if a section of a plant is not in use, there may be a whole block of alarm signals permanently up on the alarm display, even though these are strictly not alarm conditions.

Another cause of alarm proliferation is when the process has a number of different states. For example, an alarm during normal operation may not be a genuine alarm during startup. Alarms may also occur due to maintenance work on instruments. A further cause of alarms is when trip systems are activated. When a trip operates, it tends to give rise to a number of consequential alarms.

The use of process computers has increased the scope for improvements to alarm systems [23]. Such work on improved alarm systems falls into two stages [22]. The first is the development of alarm handling facilities such as types of alarm display and alarm suppression. The types of display are closely related to the human factors aspects of fault diagnosis. The second stage is the development of facilities for analysing alarms and disturbances. The purpose of such facilities is to separate real alarm signals from other types of signals. The other types of signals include signals indicating status, signals associated with mode of operation of the plant other than the normal operation mode, signals expected as a result of operator or trip action and spurious signals due to instrument maintenance and instrument malfunction [22].

A general account of alarm analysis has been given by Lees et al [22, 23, 34]. Most of the developments on computer based alarm analysis have been done on nuclear power stations as described by Kay [35], Welbourne [36] and Patterson [37]. Proposals for applications in the chemical industries have been made by Baarth and Maarleveld [38], Andow et al [39, 40], Himmelblau [27, 29] and Berenblut and Whitehouse [30]. The basis of the alarm analysis schemes reported by the above authors is to create a data base containing the cause and effect relationship of a particular plant. A general analysis program then interogates the data base in real time and builds up an analysis of the alarms as they occur. Thus such alarm analysis systems not only detects faults in the plant, but can also be extended to diagnose the faults that cause the alarm.

Using either conventional alarms or alarm analysis methods, various types of fault can be detected but only after the measured variables have been considerably affected. There may not be enough time for taking corrective actions after acknowledging the alarm. The use of process computers enables the use of modeling and estimation methods for earlier detection of faults. The discussion of modeling and estimation methods is given in the next section.

2.2.1 : Modeling and Estimation Methods

Mathematical models of the plant are used to estimate values of the process state variables, process parameters and quantities characteristic to the process under normal operating conditions. Process state variables can be measurable or unmeasurable. Process parameters are constants or time dependent coefficients in the process which appear in the process model that relate the output signal to the input signals. Examples of process coefficients are viscosity, heat capacities, transfer coefficients (heat and mass), and resistances. Characteristic quantities are those that describe the performance of the equipment such as efficiency (e.g. of heat exchanger or steam generator), or fuel consumption per production unit or time. These values provide the reference case whereby the boundaries which define a fault or faults are set.

From measurements of observed responses in the steady state or the unsteady state and for known (or unknown) process inputs, process state variables, parameters or characteristic quantities are estimated during plant operation using the same models. By comparing these estimates with those of the reference case via statistical tests, the existence of faults can be ascertained.

A benefit of modeling and estimation techniques is that faults can be inferred based on unmeasureable process state variables or parameters or characteristic quantities. Issermann [26] gives a summary of how such detection techniques can be applied. A comprehensive description of various tools in detecting faults in chemical and petrochemical plants using modeling and estimation techniques can be found in the book by Himmelblau [27].

Modeling and estimation techniques can only detect and diagnose faults in the equipment under investigation. For example, the cause of poor performance of heat exchangers can easily be pinpointed to [28] :

1. fouling;
2. leakage between the tube side and shell side;
3. vibration and cracking;
4. corrosion;
5. improper original design for usage.

If the cause of malfunction of an equipment may be due to interactions of other equipments, extensive modeling and computation is required at the plant level [24]. A considerable amount of computer time is needed to solve the model equations, particularly when dynamic models are used. Thus, the modeling and estimation method may not be practicable when applied to real-time fault detection and diagnosis.

2.2.2 : Pattern Recognition Methods

In the pattern recognition approach, inference as to whether the process plant is in a faulty condition is derived from patterns of the measured variables. The patterns are actually combinations of the features of the measured variables. The features are descriptions of the measured variables, for example, whether high, normal or low, computed according to some criterion. Each combination determines whether the plant is in either the normal state or in a certain class of fault or an anomaly. An anomaly is when the feature combination exhibits a physically impossible situation indicating that there may be malfunctions in the measuring instrument.

Techniques in the pattern recognition method include cluster analysis, acoustic analysis, vibration analysis and fault dictionaries. Himmelblau [29] gives a comprehensive description of these methods. Of the techniques mentioned, the fault dictionary is more of a method for diagnosing faults and is described later in the chapter.

2.3 : Techniques of Fault Diagnosis

2.3.1 : Introduction

Some of the techniques of fault detection described previously are in actual fact aids for fault diagnosis. One method of fault diagnosis is the pattern recognition approach such as the use of fault dictionaries or decision tables as proposed by Berenblut and Whitehouse [30]. There are other methods of fault diagnosis using the pattern recognition approach. These methods differ in the way the cause and effect relationships of process variables are obtained and also the way the diagnosis algorithm operates. The following sections describe briefly some of the techniques of fault diagnosis.

2.3.2 : Fault Diagnosis Using Fault Dictionaries

A fault dictionary is a set of decision rules where each rule is represented by a combination of the features of the measured variables. To each rule is ascribed the fault or faults that cause the outcome of the feature combination. The application of this form of fault detection and diagnosis has been discussed by Berenblut and Whitehouse [30]. In fact their work was a result from a project called "Anticipator" as described by Munday [31]. Basically, "Anticipator" is a surveillance system designed to provide real-time analysis of plant variables during operation and to recognise existence of hazard conditions or events that may lead to hazard conditions.

To illustrate the application of the fault dictionary, consider the buffer tank system taken from the paper by Berenblut and Whitehouse [30]. as shown in figure 2.1. This system includes a feed-back control loop which regulates the outflow rate to maintain the tank level, H1, more or less constant. In addition, the inflow and outflow rates are themselves measured. During normal operation, the valves V1 and V4 are closed.

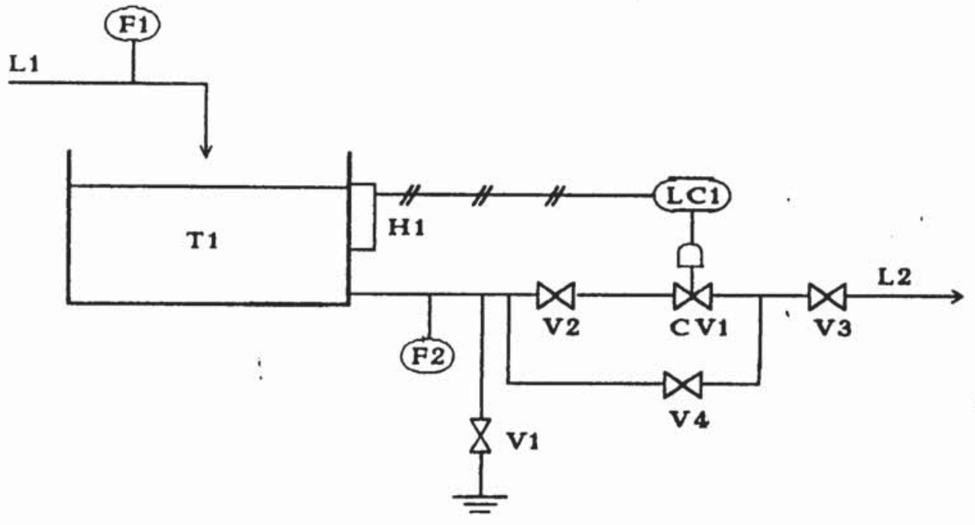


Figure 2.1 : Schematic diagram of a buffer tank [30].

If at some instant in time F1 and CV1 are in their normal range, and F2 is in the low range, the combination of these features indicates that there is a blockage in the exit line. If no corrective action is taken, the level in the tank will increase and eventually overflow.

A matrix can be built up where each row represents the features of a measured variable and each column, known as the feature vector, represents the combination or pattern of the features of the measured variables. Each feature vector describes the existence of an event or some events. A complete set of feature vectors makes up the matrix that comprises the fault dictionary.

Each event that causes the outcome of a particular feature pattern is labelled so that it can be ascribed to its corresponding feature vector in the last row of the matrix. A complete list of labelled events for the case of the buffer tank in figure 2.1 is shown in table 2.1. The features that describe the state of measured variables are also given labels, e.g. N for normal, H for high, L for low, C for closed and O for open. The set of feature vectors corresponding to the event that a blockage in the exit line exists is shown in table 2.2.

Table 2.1 : Possible fault conditions of figure 2.1 [30]

Label	Event
1	Normal operation
2	Pipe leakeage between F2 and CV1
3	V4 open in error
4	Blockage in exit line
5	Leak in tank
6	Abnormal throughput
A	Anomaly or physically impossible combination

Table 2.2 : Part of fault dictionary indicating existence of blockage in exit line of figure 2.1 [30]

Outcome (rule)		3	4	6	12	13	15	21	22	24
Variables:	F1	N	N	N	H	H	H	L	L	L
	CV1	N	O	O	N	O	O	N	O	O
	F2	L	N	L	L	N	L	L	N	L
Event		4	4	4	4	4	4	4	4	4

However, implementing fault dictionaries for fault detection and diagnosis on a computer has certain limitations. This is because the complexity of the fault dictionary increases drastically with the number of measurable variables and the degree of quantization of the measurements. For the case of the buffer tank in figure 2.1, the number of possible patterns or rules that can be derived is $3^3 = 27$. If there are 10 variables that are measured in a plant, the number of patterns would be $3^{10} = 59049$. Thus it may not be practicable to prepare a fault dictionary and store it in the computer if the number of measurements become large.

Berenblut and Whitehouse [30] presented a method of reducing the number of rules without affecting the outcome of the diagnosis. This is done by combining rules which have the same event and all but one of the variables have the same features. For example, rule numbers 3, 12 and 21 can be combined into one pattern as shown in figure 2.2.

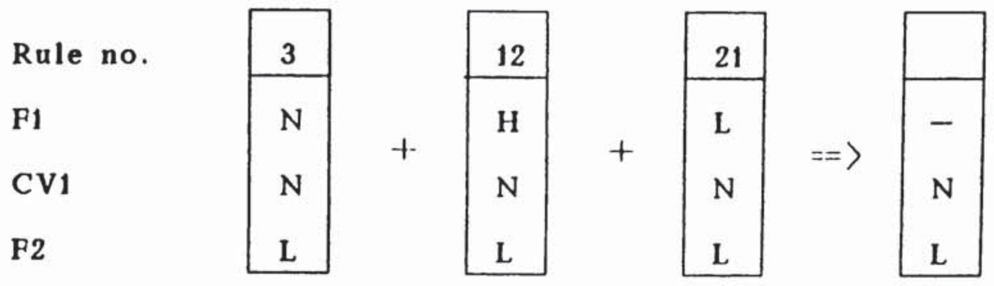


Figure 2.2 : Method of combining rules to shorten fault dictionary

The feature for F1 is replaced by a dash to indicate a "don't care" feature. This implies that whatever condition F1 is in, the combination of the features of the other variables will indicate that the event common to the three rules will have occurred. The shortened fault dictionary for the buffer tank is shown in table 2.3.

Table 2.3 : Shortened fault dictionary for buffer tank [30]

Rule	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
F1	N	—	—	—	H	—	—	—	N	H	L	H	L	L	N
CV1	N	N	N	O	O	O	C	C	C	C	C	N	N	O	O
F2	N	H	L	N	H	L	N	H	L	L	L	N	N	H	H
Event	1	2,3	4	4	6	4	2,3	2,3	5	5	6	5	A	A	A

A requirement for using fault dictionaries or decision tables is that plant measurements are correct and reliable. It is possible that a measurement transmitted to the computer does not represent the actual value of the variable. Consequently, the decision table may either indicate a normal operation whereas in actual fact an event has occurred, or vice versa resulting in false alarms, or diagnosing the wrong event that has occurred.

As an example, referring to figure 2.1 and table 2.3, suppose an alarm is raised due to rule 12 indicating that there is a leak in the tank. If there is a malfunction in F1 whereby the actual state of the input stream is in the normal range such that normal operation still prevails, then the alarm raised is false. On the other hand, if the actual state of the input stream is in the low range, then rule 13 should have been applied indicating an anomaly. This anomaly may be caused by malfunction in the level transmitter to the level controller or the control valve not responding to commands from the controller.

To reduce the probability of false alarms, additional rules involving duplicated measurements or indirect measurements may have to be incorporated in the decision table [32]. Even if additional rules are incorporated, the operator has to diagnose the cause of anomalies or abnormal throughput.

An alternative method to overcome the shortcomings of the fault dictionary is to relate a single measured variable which has deviated from its normal range to all its possible causes, including instrument malfunctions. For example, referring to table 2.3, the cause of F2 being in the high state may be:

1) pipe leakage betwen F2 and CV1

2) V4 open in error

3) high state of the input stream, F1

4) F2 indicating higher than what is actual.

The first two causes are the events due to rules 2 and 8. The third cause comes from rule 5, the abnormal throughput. Whether this cause is true or false is established by checking the state of F1 either indirectly by the operator or directly on-line to the computer. If cause number 3 proved to be the cause of F2 being high, then events that leads to the high state of the input stream can be determined by checking its cause and effect relationship upstream of the buffer tank. The fourth cause, the measuring instrument malfunction, takes care of the anamolies due to the pattern of rules 14 and 15. Thus by one cause and effect relationship, several rules are incorporated. By combining with other cause and effect relations, the cause of a fault in the process plant could be easily be diagnosed. The cause and effect relationship is easily visualised if it is put in a graphical representation such as a fault tree.

Berenblut and Whitehouse proposed that data to build the fault dictionary of a plant come from operability studies. An operability study depends on the expertise of the designers of the plant as well as the managers who operate the plant. Thus the fault diagnosis method by Berenblut and Whitehouse is experience orientated. A detailed description of operability studies is given in chapter 3.

2.3.3 : Fault Diagnosis Using Functional Model Equations

Other than operability studies, the cause and effect relationship between process state variables can be obtained logically from simple models of the plant. The alarm analysis method for fault detection and diagnosis described by Andow [39]; Andow and Lees [40]; and Martin-Solis et al [41] uses functional model equations to represent the relationship between variables of plant items.

In a functional equation, each of the process variables on the left-hand side is a function of all the process variables on the right-hand side. As an illustration, referring to figure 2.1 for the open tank only, the relevant functional equations are [40]:

$$\frac{dH1}{dt} = f(F1, -F2) \qquad \text{..... 2.1}$$

$$F2 = f(H1, -P2) \qquad \text{..... 2.2}$$

P2 is the pressure of the outlet stream. Equation 2.1 states that the level in the tank will increase with either an increase in the inlet flowrate or a decrease in the outlet flowrate. For equation 2.2, the outlet flowrate will increase with either an increase of the tank level or a decrease of the outlet pressure.

Other equipment including pipelines, valves, controllers and many others can be modelled accordingly. From the topography of the plant, and the functional equations of each plant item, a network of process variables that interact with each other can then be constructed. However, not all process variables are measured and therefore do not have alarms on them. Thus a process alarm network can be produced from the process variable network by eliminating all those process variables on which there are no alarms.

Andow and Lees [39, 40] used a list processing technique to store the process alarm network in the computer. Apart from the alarms due to the measured variables, deduced alarms are also stored together with their associated process alarms. Deduced alarms are those associated with mechanical faults of the plant items, such as leakages, sensor malfunction, blockages and valves wrongfully opened or closed.

When a set of alarms occurs in real-time, the relations between alarms are derived by interogating the process alarm network. In fact the process alarm network is a form of an alarm tree. When a fresh alarm occurs, a check is made to determine whether the alarm is part of an existing alarm tree which is developing as the effect of an existing prime cause alarm. If the fresh alarm is not part of the tree, it is classed as another prime cause alarm. If it is part of the tree, it is classed merely as a new alarm.

However, the alarm analysis system as described above is not adequate for fault diagnosis. It was developed primarily to give the relations between a set of alarms rather than to diagnose a mechanical fault [50]. The information given from the alarm analysis is essentially restricted to an indication that a potential fault path exists. Also, the alarm analysis using the process alarm network could not indicate that there may be several alarms that must occur simultaneously to initiate a fresh alarm.

An alternative method using the same functional model equations was developed by Martin—Solis et al [41]. Instead of producing the process variable network, Martin-Solis use mini-fault trees to represent diagramatically the cause and effect relationship of variables that have deviated from their intended values. The way a variable can be caused to deviate is obtained from the relationship found in the functional model equations. Thus, the deviated variable on the left-hand side of the functional equation becomes the top event of a mini-fault tree and the deviations of the right-hand side variables become the base events of the tree connected by an **OR** gate. Information additional to that given in the functional equations which relates to mechanical failures are also added to the mini-fault tree.

A full fault tree is then synthesised by selecting a top event and developing each branch of the tree using the appropriate mini-fault trees. The characteristics and properties of a fault tree are described in chapter 3.

An important advantage of the fault tree over the process alarm network is that events that must simultaneously occur to cause the top event of a mini-fault tree can be included by putting them below an **AND** gate. Also, the state of unmeasured variables can be inferred using the fault tree, whereas the process alarm network has all the variables which could not raise alarm signals removed.

However, using the functional equations either to automatically build up the process alarm network or the fault tree has some limitations. The functional equations can only relate how an increase or decrease of a variable effects other variables. The effect of deviations such as no flow or reverse flow in the pipeline cannot be obtained from the model equations. Functional models do not contain logic about the combination of values of variables needed for some outcomes. Knowledge of the process and engineering judgement are essential to include such simultaneously occurring events as well as mechanical malfunctions in the fault tree. The inclusion of simultaneous occurring events and events due to mechanical failures has to be done manually.

2.3.4 : Fault Diagnosis Using Signed Digraph

Another approach to modelling process plants is the use of directed graphs or digraphs as it is well known. A signed digraph consists of nodes representing the variables and directed branches that connect the nodes that have a cause and effect relationships. A positive or negative sign is assigned to each branch to indicate how the initial node of the branch affects its terminal node. As an illustration of what a signed digraph graph looks, consider the buffer tank system as shown in figure 2.1. Using L1 and L2 to represent the inlet and outlet flow rate respectively and H1 to represent the level in the tank, the signed digraph is shown in figure 2.3.

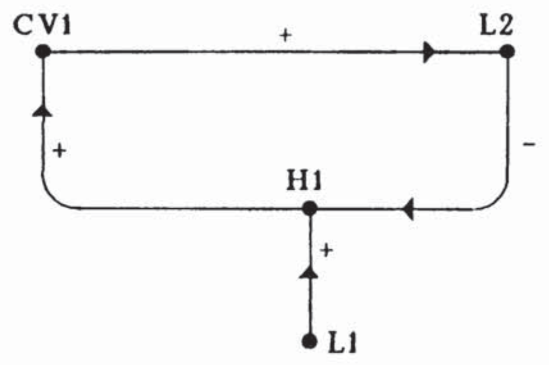


Figure 2.3 : Signed digraph for buffer tank system of figure 2.1

A branch with a positive sign indicates that if the value of the variable represented by the initial node increases or decreases, then the value of the variable represented by the terminal node also increases or decreases respectively. On the other hand, if the branch has a negative sign, an increase or decrease in value of the initial node will decrease or increase respectively the value of the terminal node.

From patterns of the measured variables, the signed digraph is transformed to a cause-effect or CE graph [44]. The analysis of the CE graph will determine the cause of a fault in the plant. Work on fault diagnosis algorithms using the signed digraphs as models of every equipment in the plant were reported by Tsuge et al [42, 43, 45]; Iri and Aoki [44]; Shiozaki et al [46]; and Umeda et al [47].

As an illustration, consider the buffer tank of figure 2.1 and its corresponding signed digraph as shown in figure 2.3 Suppose that F1 and F2 indicate that the respective inlet, L1, and outlet, L2, flow rates are above the normal range, CV1 also indicates that its position is above its normal range, and H1 is normal. The pattern of the measured variables using the notation of Iri and Aoki [44] is:

$$(F1,\ H1,\ CV1,\ F2) = (+,\ 0,\ +,\ +) \qquad \ldots\ldots\ 2.3$$

The positive signs assigned to the state variables indicate that the deviations are due to the increase in value of the variables above the intended limits. If the values of the state variables decreased below the intended limits, then negative signs are assigned to the variables. The zero assigned to a variable indicates that there is no change in its state, as it is in the normal mode of operation. The CE graph to represent the pattern of equation 2.3 is shown in figure 2.4.

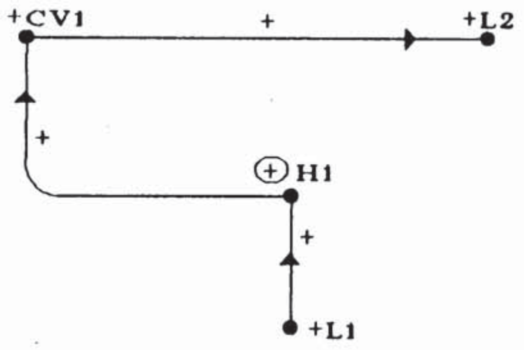


Figure 2.4 : CE graph for pattern in equation 2.3 on the signed digraph of figure 2.2

Normally, when a node representing a state variable is assigned a zero, it is not included in the CE graph [44]. However, for the case of the buffer tank, H1 is being controlled which suppresses the tendency of the level in the tank becoming greater than its normal range, such that H1 has a normal value. In order to take into account the special roles of controllers and control elements, Iri and Aoki

[44] introduced special signs to represent the outcome of controlled variables as if they are not controlled. Iri and Aoki use the sign ⊕ or ⊖ to represent the state of a variable which would have the sign + or - respectively without control but which is not seen abnormal due to the operation of the controller. The sign ⊕ is assigned to H1 in figure 2.3 because an increase in value of L1 will definitely increase the level in the tank if H1 is not controlled.

In building up the CE graph from the signed digraph, consistency must be maintained so that the cause and effect relationship is logically sound [44]. For example, referring to the signed digraph of figure 2.3, the branch between the nodes L2 and H1 indicates that an increase in the value of L2 will decrease the value of H1 if the level is not controlled. If the sign ⊖ is assigned to H1, it will not be logical since L1 is assigned the sign + and the branch connecting the nodes L1 and H1 has the positive sign. So, to be consistent, the branch connecting nodes L2 and H1 is removed in the CE graph as shown in figure 2.4.

The nodes in the CE graph are termed as the strongly connected components of the graph [44, 47]. If a strongly connected component is not a terminal node connected with other initial nodes, then that node is termed the maximum strongly connected component. The method of fault diagnosis using the signed directed graph approach is to search for the maximum strongly connected components in the CE graph. If there is more than one maximum strongly connected component found in the CE graph, then these nodes are termed the candidates for the cause of the fault in the plant.

Not all the maximum strongly connected components found in a CE graph can be the cause of the fault in the plant. The operator has to manually check all the candidates to determine the actual cause of the fault. For the case of the CE graph of figure 2.4, the maximum strongly connected component is L1 and thus the cause of the pattern in equation 2.3 is the high flow rate of the inlet stream.

Although fault diagnosis using the signed digraph is feasible, it can only handle cases when state variables exceed their upper or lower bounds. There is no facility for handling deviations such as reverse flow or no flow. Also, the fault diagnosis algorithm requires that the pattern of the measured variables be correct. Thus the reliability of the measuring instruments is important. With this in mind, the algorithm cannot diagnose that the cause of a fault in the plant is due to instrument malfunction. Another drawback of the signed digraph method for finding causes of faults in chemical plant is the poor diagnostic resolution. Tsuge et al [42] reported that the signed digraph algorithm using on-line sensor data yielded an average of 23 fault candidates out of 53 possibilities.

2.3.5 : Fault Diagnosis Using Fault Trees

From the previous discussions, it can be seen that fault diagnosis with the aid of fault trees overcomes the drawbacks that have been pointed out for other methods. An advantage of the fault tree method over the methods that have been discussed above is that operators can trace the cause and effect relationship of the various deviations that can lead to the abnormal condition in the plant.

Previous work on fault diagnosis using fault trees has been described by Lihou [62] and Martin-Solis et al [41]. A detailed description of fault trees is given in chapter 3.

2.3.6 : Fault Diagnosis Using Expert Systems

Another technique of fault diagnosis is the use of expert systems to aid the operator in finding the cause of a fault. An expert system is defined by Bramer (see reference 65) as "a computer system which embodies organised knowledge concerning some specific area of human expertise, sufficient to perform as a skilful and cost effective consultant".

An expert system may be viewed as comprising of four major components [52] :

1. The knowledge base
2. The case specific data
3. The inference mechanism or inference engine
4. The man-machine interface

The knowledge base is the knowledge specific to the domain of problems that the system is intended to solve. For the case of process plant fault diagnosis, the knowledge consists of the cause and effect relationships of the deviations in the state variables, instrument malfunctions and mechanical failures. Integrated into the knowledge base could be knowledge obtained from proven rules and generally acknowledged hypotheses such as the plant models; and subjective knowledge obtained on the basis of long-term experience of experts. One way of organising the logical structure of knowledge for the fault diagnosis system is in the form of production rules [52, 53, 54]. Production rules are conditional expression of the form [53] :

$$\text{IF} \quad \mathbf{A} \quad \text{THEN} \quad \mathbf{B} \qquad \ldots\ldots 2.4$$

where **A** is the detectable consequence or fault, and **B** is the set of all possible causes of **A**. If a failure is conditional on several causes, then the rule can be expressed as:

$$\text{IF} \quad \mathbf{A} \quad \text{THEN} \quad \mathbf{C_1} \text{ AND } \mathbf{C_2} \ . \ . \text{ AND } \mathbf{C_i} \ . \ . \text{ AND } \mathbf{C_n} \ \ldots\ldots 2.5$$

where C_i, $i = 1$ to n, are the causes that simultaneously must occur to cause **A**. Another form of the production used by Kumamoto et al [54] is:

$$\text{IF} \quad \mathbf{A} \quad \text{AND} \quad \mathbf{F} \quad \text{THEN} \quad \mathbf{B} \qquad \ldots\ldots 2.6$$

where **A** is the faulty state of the system, **F** is the observable fact

that qualifies **A**, and **B** is the cause that brought about the existence of **A** and **F**.

The case-specific data are the knowledge about the particular problem to be solved. It is used for keeping track of the problem status, the input data for the particular problem, and the relevant history of what has thus far been done.

The "inference engine" is the heart of the expert system. It is just a computer program that uses the knowledge base and the case-specific data in either a forward or backward inference chain to obtain a solution to the problem.

The man-machine interface is fundamentally concerned with the way the expert system presents its findings to the user. The usual form of interface is in the form of an interactive dialogue between the computer and the user. This is to enable specific data required by the expert system to be supplied by the user. Reasons to justify every step of the expert system's findings can be displayed whenever required by the user. This enables the user to evaluate the assertions during the cause isolation process [54].

There have been several publications on application of expert system techniques for process plant fault diagnosis. A review of some of the expert systems that have been published can be found in reference [52]. Almost all the reported expert systems are prototypes.

Niida et al [55] described a prototype expert system for diagnosing faults in a pressure relief system. The knowledge structure is in the form of fault trees obtained from cause and effect analysis of the system.

Kumamoto et al [84] give a simple example of an expert system applied to fault diagnosis of a ship's engine cooling system. The knowledge to represent the behaviour of the system is based on an AND/OR tree structure compiled into IF-THEN rules.

Kramer and Palowitch [58] proposed that the diagnostic rules that form the knowledge base of an expert system are to be obtained from the signed digraph of the chemical process system. However, problems associated with fault diagnosis using signed digraphs described earlier will be inherent in the expert system.

Chester et al [56] described a prototype expert system called FALCON for real-time fault diagnosis of chemical process plants. In their system, the knowledge representation comes from interconnected models of plant components that make up the process. The diagnosis is based on causal model approach. In this approach, the models that describe cause and effect relationships and observed data describing the state of the process are used to reason out all the possible faults that might be hypothesized. Yamada and Motoda [57] also use this approach for fault diagnosis of a nuclear reactor power plant.

Nelson [59] also produced an expert system for diagnosing nuclear reactor accidents called REACTOR. In his system, the knowledge base contains two types of knowledge: function-orientated knowledge and event-oriented knowledge. Function-orientated knowledge describes the configuration of the reactor system and how its components work together to perform a given function. Event-orientated knowledge describes the expected behaviour of the reactor under known abnormal conditions. It is contained in a series of production rules.

Other developments of expert systems specifically for diagnosing faults have been reported by Andow [49], Sgurev et al [60], Fox et al [61], and Klemes and Krus [53].

An advantage of the expert system technique for fault diagnosis is the capability, on demand, of explaining the advice or action to be taken. This is done by incorporating accumulated experience, judgment and the heuristics of process engineers and operators into the automated reasoning system. In theory, the use of expert systems for fault diagnosis in real time, by direct access to plant measurements, allows for plant operators to be able to diagnose any fault that occurs during plant operation. Together with a fault tree display as an interface to show how the fault propagates from its basic causes, a better understanding of how the fault occurred can be obtained.

However, implementing expert system techniques in real time fault diagnosis can have some problems. Plant status may change very fast which implies that the response time for the diagnosis is critical. Therefore a high inference speed is needed. Kramer [111] states that for a large chemical plant, the number of rules may be as high as 10,000 to 100,000. Taking into account that the plant status may change whilst the diagnosis is in process, an exhaustive search for the cause of a fault may not be possible in real time.

2.4 : Human Factors Aspects in Fault Diagnosis

The study of human behaviour in areas such as vigilance, information processing and decision making is part of the discipline of ergonomics or human factors [23]. Human factors also includes the study on allocation of function between man and machine and the design of equipments such as displays and controls that enhance the perfomance of the human being in carrying out specific tasks. Such human factors studies are needed to identify and solve difficulties which may arise in developing man-machine systems [23]. The following account discusses some of the human factors aspects relevant to fault detection and diagnosis in the chemical plant.

Whenever an alarm sounds that an abnormal condition is present in the chemical plant, it is expected that operators respond to administer the fault that has occurred. Fault administration consists of three stages [34]:

1. fault detection
2. fault diagnosis
3. fault correction

The task of administering faults requires that the operator makes decisions in a complex or unforeseen situation. To be effective in decision making, the operator must know the current state of the plant. A study by Bainbridge [63] indicates that typically an operator has a mental model of the state of the plant, that he extrapolates this state into the future and that he then uses an overview display to sample information at a frequency sufficient to confirm his extrapolation. Thus a properly designed information display is important to almost all the operator's tasks including fault administration. A good display is when the operator can obtain the information required at a glance [22]. Conventional control panels and mimic diagrams often give relatively good overview displays.

Edwards and Lees [34, 64] dealt in detail the human factors aspects in process control. Although much of the knowledge given by Edwards and Lees is relevant to process control, it can be applied to process alarm systems. The prime function of an alarm system is to assist the operator in fault detection which is the first stage of fault administration.

The second stage of the operator's task is fault diagnosis. Rasmussen and Jensen [33] found that in the task of repairing electronic equipment, man tends to explore first those paths which on the basis of his experience he regards as the high probability paths. Low probability paths are not considered until he is satisfied that the high probability paths can be ruled out. This search strategy according to Rasmussen and Jensen is quite efficient. By contrast, the search strategy used by computer aided fault diagnosis methods described earlier follow a fixed sequence regardless of probability [22]. The computer program would become more efficient if it follows the human's search strategy. This can be done by including relevant probability information into the fault diagnosis package.

When using a computer aided fault diagnosis system, the VDU is an important interface for the operator to obtain information as well as to give the relevant information required by the system. The design of the display is crucial with respect to operator assimilation of the information. Operators respond much better when the information is presented diagramatically rather than a list of figures [20]. One form of diagramatic representation is a fault tree. A fault tree aids the operator in visualising the relationship between the abnormal condition of the plant and all its possible cause. A detail description of fault trees is given in chapter 3.

CHAPTER THREE

3. HAZARD AND OPERABILITY STUDY AND FAULT TREE ANALYSIS

3.1 : Hazard And Operability Study

3.1.1 : Introduction

Hazard and operability studies, or HAZOP as it is well known in its abbreviated form, is a formal study of process plants aimed at discovering potential causes of hazardous consequences or operating difficulties [13,14]. The concept of HAZOP is that a multidisciplinary team reviews the plant methodically in a series of meetings with the aim of detecting all possible ways in which the plant departs from the intentions of its designers. It is based upon the supposition that most problems are missed because the system is complex rather than because of a lack of knowledge on the part of the design team [13].

Although the general objective of HAZOP is to identify hazards and operability problems, the underlying reason for the study may be varied. Examples of reasons for a study might be to:

1. Check the safety of a design.
2. Decide whether and where to build the plant.
3. Check operating and safety procedures.
4. Improve the safety of an existing piece of equipment.
5. Verify that safety instrumentation is reacting to the best parameters.

6. Prepare data for fault diagnosis.

The key for a successful HAZOP is the use of word models to stimulate thought and creativity during the study. The word models consist of **PROPERTY WORDS** that focus attention on the designed intention of plant items, and **GUIDE WORDS** that focus attention on possible deviations. Typical property words are flow, pressure, temperature and level. Guide words are simple words which are used to qualify or quantify property words in order to guide and stimulate the creative thinking process and so discover deviations. Examples of guide words are **NO**, **MORE**, **LESS** and **REVERSE**. Table 3.1 shows the different types of guide words and their meanings as published by the Chemical Industries Safety and Health Council [66].

Once the purpose, objectives and scope of a HAZOP study have been defined, the steps for carrying out the study are:

1. Selection of the HAZOP team members.
2. Making preparations for the study.
3. Actual study carried out by the team.
4. Recording the results.

3.1.2 : The HAZOP study team

HAZOP study of a process plant is usually carried out by a team of experts, each with their own specialisation. It is based on the principle that several experts with different backgrounds can interact

Table 3.1 : HAZOP guide words and meanings [66]

Guide words	Meanings	Comments
NO or NOT	The complete negation of the design intent.	No part of the intention is achieved.
MORE	Quantitative increase.	These refer to quantities and properties such as flowrates, temperatures, concentrations and pressures as well as activities such as 'HEAT' and 'REACT'.
LESS	Quantitative decrease.	
AS WELL AS	Qualitative increase.	All the design and operating intentions are achieved together with some additional activity.
PART OF	Qualitative decrease.	Only some of the intentions are achieved.
REVERSE	Logical opposite of the intention.	This is mostly applicable to activities, for example reverse flow or chemical reaction.
OTHER THAN	Complete substitution.	Something quite different from the design intention happens.

and identify more problems when working together, rather than working separately and combining their results. Members of the team are chosen to provide knowledge and experience appropriate to the objectives of the study and the stage of development of the project.

The leader of the team who chairs meetings of the team need not be an expert on the plant under study, but should be a person experienced in the HAZOP technique whose job is to ensure that the HAZOP guidelines are followed. Kletz [68] proposed that for a newly designed plant and an existing plant, the composition of the study team should be as shown in table 3.2 and table 3.3 respectively.

3.1.3 : Preparation for the HAZOP study

The preparative work for a HAZOP study involves obtaining the necessary data and arranging the timetable for holding the meetings. Typically, the data consists of various drawings in the form of line diagrams, flowsheets, plant layouts, isometrics, and fabrication drawings, which should be verified as up-to-date.

For a continuous plant, the existing flowsheets or pipe and instrument drawings usually contain enough information for the study. The sequence for the study is straightforward, starting at the beginning of the process and progressively working downstream, applying the guide words at specific study nodes which are usually pipe sections and plant components where the process parameters such as flow, temperature, pressure, level, heat exchange and others

Table 3.2 : HAZOP study team for a new plant [68]

Team member	Comment
Design engineer	Usually a mechanical engineer and, at this stage of the project, responsible for minimising the cost but not for hazards and operating problems.
Process engineer	Usually the chemical engineer who drew up the flow sheet.
Commissioning manager	Usually a chemical engineer who will start up and operate the plant.
Instrument design engineer	Requirement for plant with sophisticated control, alarm and trip systems.
Research chemist	If new chemistry is involved.
Independent chairman	An expert in the HAZOP technique, not the plant. Should ensure that the team follow the procedure. Leader of the team.

Table 3.3 : HAZOP study team for an existing plant [68].

Team member	Comment
Plant manager	Responsible for operation.
Process foreman	Someone who knows what actually happens rather than what is supposed to happen.
Plant engineer	Responsible for mechanical maintenance and therefore knowledgeable about many of the faults that occur.
Instrument manager	Responsible for instrument maintenance including testing of alarms and trips.
Process investigation manager	Responsible for investigating technical problems.
Independent chairman	An expert in the HAZOP technique and leader of the team

have identified design intent. These study nodes are identified first before the meetings are held.

For batch plants, the preparative work is more extensive, primarily because of the sequential nature of the process. Thus, operation sequences form a larger part of the HAZOP data. The operations information can be obtained from operating instructions, logic diagrams, or instrument sequence diagrams. If operators are physically involved in the process rather than simply controlling the process, their activities are represented by means of process flow charts.

3.1.4 : The Team Review

The HAZOP study provides opportunities for people in the team to let their imagination go free and think of all possible ways in which hazards or operating problems might arise, but at the same time cover all malfunctions by applying a systematic procedure. The approach to HAZOP study is to critically examine the full description of the process laid down on the process and instrument diagrams and seeks to answer definite questions in a systematic manner. This method of critical examination approach was discussed by Elliott and Owen [67].

Each part of the design will be subjected to a number of questions to explore every conceivable way in which that design could deviate from the design intention. This usually produces a number of theoretical deviations. Each deviation is then considered in order to determine whether it is meaningful or unrealistic. If the deviation is unrealistic, then the derived consequences will be rejected [67]. For those that remain, some of the consequences may be trivial and would not be considered further.

Figure 3.1 shows the procedure to critically examine the plant during the HAZOP study. The success or failure of the examination will be built on the following factors [67]:

a) The completeness and accuracy of drawings and other data used as the basis for the study.

b) The technical skills and insights of the study team.

c) The ability of the study team to use the critical examination approach as an aid to their imagination in visualising deviations, causes and consequences.

d) The ability of the study team to maintain a sense of proportion, particularly when assessing the seriousness of the hazards which are identified.

Although the critical examination approach provides a comprehensive analysis during the HAZOP study, the review team may have to spend many hours on a single process and instrument diagram. This will incur a significant expense to complete the study. Lihou [17] has given a classified set of rules to speed up the HAZOP study as well as to avoid producing ambigous or illogical cause and consequence relationships for the deviations. These rules are given in Appendix A.

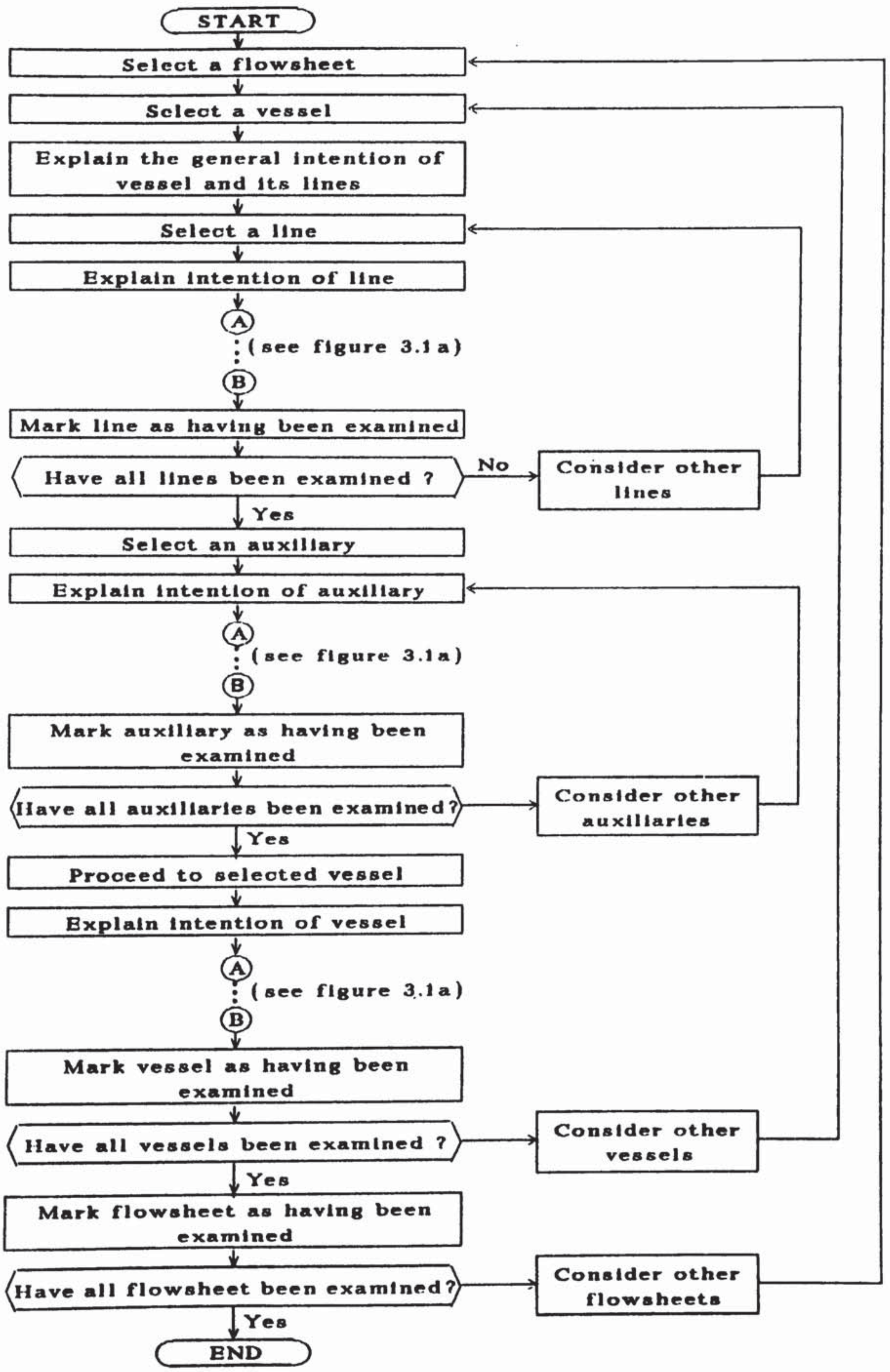

START
Select a flowsheet
Select a vessel
Explain the general intention of vessel and its lines
Select a line
Explain intention of line
A
(see figure 3.1 a)
B
Mark line as having been examined
Have all lines been examined ?
No
Consider other lines
Yes
Select an auxiliary
Explain intention of auxiliary
A
(see figure 3.1a)
B
Mark auxiliary as having been examined
Have all auxiliaries been examined?
Consider other auxiliaries
Yes
Proceed to selected vessel
Explain intention of vessel
A
(see figure 3.1a)
B
Mark vessel as having been examined
Have all vessels been examined ?
Consider other vessels
Yes
Mark flowsheet as having been examined
Have all flowsheet been examined?
Consider other flowsheets
Yes
END

Figure 3.1 : Flowsheet of HAZOP procedure

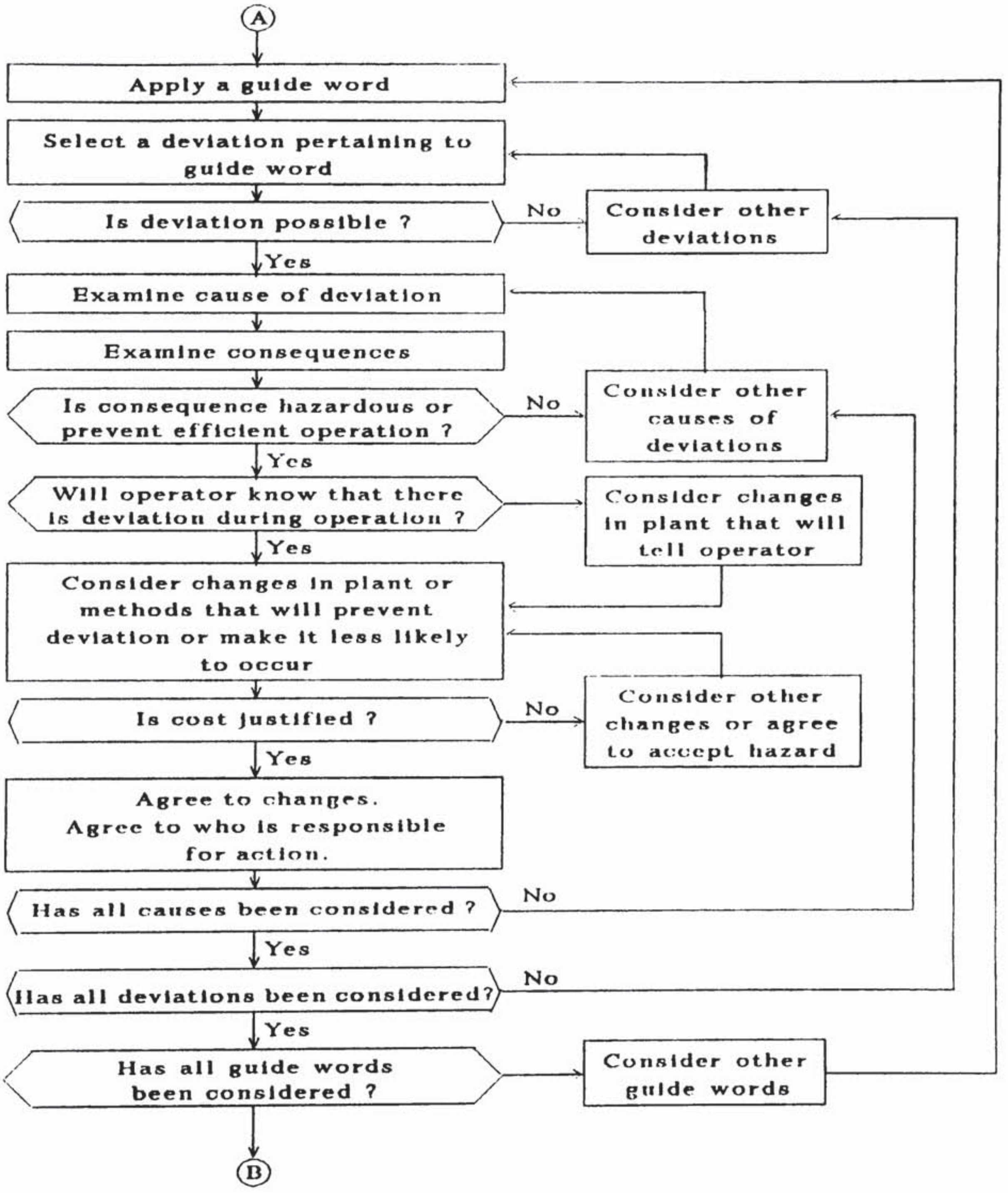

A
Apply a guide word
Select a deviation pertaining to guide word
Is deviation possible ?
No
Consider other deviations
Yes
Examine cause of deviation
Examine consequences
Is consequence hazardous or prevent efficient operation ?
No
Consider other causes of deviations
Yes
Will operator know that there is deviation during operation ?
Consider changes in plant that will tell operator
Yes
Consider changes in plant or methods that will prevent deviation or make it less likely to occur
Is cost justified ?
No
Consider other changes or agree to accept hazard
Yes
Agree to changes. Agree to who is responsible for action.
Has all causes been considered ?
No
Yes
Has all deviations been considered?
No
Yes
Has all guide words been considered ?
Consider other guide words
B

Figure 3.1 a (continuation of figure 3.1)

3.1.5 : Recording of results

During the team review of the plant, all the significant remarks may not have been recorded. Thus it is essential that the team members review the report produced from the HAZOP meetings and then come together for a report review session. The process of reviewing the recorded findings will often fine-tune these findings and uncover others that were missed. The success of this process demands a good recording scheme. One form of recording information obtained during HAZOP study is by using a table of entries of guide word, deviation, possible causes, consequences and action required [13]. Another form is by use of checklist [69,70].

Depending on the depth of the study, the team may not have to follow strictly the direction of flow as laid down on the process and instrument diagram [68]. If there is some unresolved doubt about an equipment or a line, other parts of the plant have to be worked out first so that sufficient information is gathered to provide data for its solution. To keep track on the remote interactions and to keep the operability records up to date may be quite difficult when using the HAZOP table or check list. The team have to sieve through piles of recorded results to get the relevant information. Retrieving the required operability data becomes more difficult when modifications are proposed to the process design during the study.

3.1.6 : Problems with HAZOP

Some of the shortcomings of HAZOP have been discussed earlier. A problem is the time required to complete a HAZOP study. A HAZOP study requires many hours of meetings to consider every plant item of the chemical process. When modifications to the plant design are proposed during the HAZOP study, more meetings are required to consider whether there are any effects on the operation and safety of the modified plant. Spending so much time on HAZOP can be very costly and also affect the motivation of the experienced engineers of the HAZOP team.

Another problem is the presentation of the accumulated data from the HAZOP study so that it is easy to comprehend. The use of a HAZOP table or a check list may not be the best way to relate a deviation on one plant item to its basic causes. The way the HAZOP data is presented is also important for evaluation of the reliability of a plant item. Lawley [13] proposed the use of a logic tree, which is a form of fault tree, drawn manually from data compiled from the HAZOP table, so that the team can visualise the relationship of each deviation. The probability of occurrence of each deviation also can be calculated from the relationship shown in the logic tree. However, it can be difficult and time consuming to obtain the logic tree from the HAZOP table, making sure that any deviation that is relevant is not missed.

3.1.7 : Computerisation of HAZOP recordings

A way to overcome the problems with HAZOP discussed previously is to use a computer to aid the HAZOP study. The use of a computer to store the findings from the HAZOP study and automatically draw a fault tree on a suitable output device will enable the study team to better comprehend the relationship between the deviations considered. The study team can easily obtain and study the necessary information if there are unexpected operating difficulties or changes in the design intentions. Another benefit of computerisation of the HAZOP data is that the effect of remote interactions can be studied by using a fault tree display.

To obtain the desirable effects stated above, a suitable means of computer storage of the data must be developed. A way to store the HAZOP findings in the computer is to use Cause and Symptom Equations, as devised by Lihou [71], to express the data recorded during the HAZOP study. Cause and Symptom Equations are discussed in detail in the next section.

3.2 : Cause and Symptom Equations from HAZOP

Lihou [71] showed that outcomes from HAZOP can be recorded in terms of equations that can be stored in a computer. Deviations in a pipeline or equipment when any of the guide words is applied to a property word can be connected to its causes by **Cause Equations**. The way the equipment responds to a deviation can be described by a

Symptom Equation. The responses are at certain nodes in the equipment which can be used to show connections between output deviations of one plant item and input deviations of another. Thus the cause and symptom equations represent the cause and effect relationship of a plant item and also between one plant item and another. Any modifications to the plant design can be accounted for by adding more equations or by just changing some of the equations.

3.2.1 : Coding of Deviations

To make recording of cause and symptom equations compact, Lihou [71] introduced a coding system whereby a fault on a plant item is represented by the item identification code followed by a set of numbers or letters within parenthesis. The numbers usually represent, in order, the property word, guide word and, if appropriate, a chemical component that is, or could be, present in the plant item. Letters and single numbers are used in Lihou's notation to represent the failure mode of process items. Tables 3.4 and 3.5 give examples of the meanings of the numbers and letters when used to represent a fault in a plant item.

As an illustration, consider a pipeline having the reference number LINE101 on the process and instrument diagram. This line can be simply be referred to as L101. The deviant state of no flow in this line due to a malfunction can be recorded as L101 NO FLOW. The guide word is NO and the property word is FLOW. Using Lihou's coding system, and keeping in mind the order in which the indices are to

Table 3.4 : Meaning of index numbers in parenthesis following a line, vessel or node number [71].

Index number	Property word	Guide word
1	Flow	NO
2	Temperature	LESS
3	Pressure	MORE
4	Level	AS WELL AS
5	Concentration	PART OF / FLUCTUATION
6	Absorb	REVERSE
7	Heat transfer	OTHER THAN

Table 3.5 : Equipment failure modes indicated by a single index number or letters in parenthesis [17].

Equipment type	Index number			Letters and their meanings
	0	-1	1	
Alarm	Failed			(FD) Failed to danger (IG) Ignored
Controller	No signal	Set too low	Set too high	
Control Loop	One or more valves closed	Giving less flow	Giving more flow	
Level switch		Set low or stuck high	Set high or stuck low	(FD) Failed to danger (FS) Failed spuriously
Line	Fully blocked	Partly blocked		(L) Leaking (BV) Blocked valves (RV) Restricted valves
Pneumatic trip valve (3 way)	Vent branch isolated	Leaking to vent	Open to vent	
Switch (not level)		Set low	Set high	(FD) Failed to danger (FS) Failed safe
Transmitter	No signal	Indicating too low	Indicating too high	
Valve	Closed or blocked	Insufficiently open or passing	Open or open too much	

appear, the coded deviation is L101(11). The first index number 1 codes the property word FLOW, The second index number, which is by chance equal to 1, codes the guide word NO. It will read as FLOW NO. The reason for the property word to precede the guide word is that it is easier to record deviations on the operability sheet under groups of process parameters such as flow, temperature and pressure.

If the deviant state of a chemical component needs to be recorded, for example MORE CONCENTRATION of component A in line LINE101, then the coded information will be L101(531). The third number, 1, in the parenthesis refers to component A. The codes to represent chemical components will depend on the particular system in use.

Although seven guide words are sufficient [71] , the list of property words can be quite extensive. Examples of other property words are React, Purge, Calorific Value, Ph and Viscosity. To avoid ambiguity if there are more than 9 property words and chemical components, slashes are used to separate the codes for the property word, guide word and chemical components for all deviations [29]. For example, the code to represent MORE CONCENTRATION of component A in line L101 will be L101(5/3/1) instead.

The codes in table 3.5 are used to represent malfunctions of equipments or control instruments. For example, the cause of no flow in line **L101** may be due to total blockage at some point in the line. Therefore the code to represent this malfunction is **L101(0)**.

3.2.2 Cause Equations

For a cause equation, deviation in a pipeline or equipment is written at the left hand side of the equation followed by an equals sign (=) which means "is caused by". On the right hand side of the equation are the causes of the deviation separated by either plus signs (+) or asterisks (*). The plus sign, representing the Boolean **OR**, is used to indicate that the deviation is caused by either of the causes, before and after the sign. The asterisk, representing the Boolean **AND**, is used to indicate that the causes before and after the sign should happen simultaneously to cause the deviation on the left hand side of the equation.

To illustrate how cause equations are formulated, consider figure 3.2 which is part of an ammonia let-down system [30]. Input to the vessel C1 via the line L1 is a two-phase mixture of cooled liquid ammonia and synthesis gas containing some ammonia gas and traces of methane. The vessel separates the two-phase flow where the synthesis gas, plus some ammonia gas and methane, is discharged via line L5, and the liquid ammonia containing dissolved synthesis gas leaves via line L2. Liquid level in C1 is controlled at about 40% by LIC1. During normal operation, valve V3 is always closed and V1 and V2 open. The causes of no flow in line L2 could be written as:

(no flow, line L2) **is caused by** (no flow into line L2)
or (line L2 blocked)
or (line L2 valves blocked)

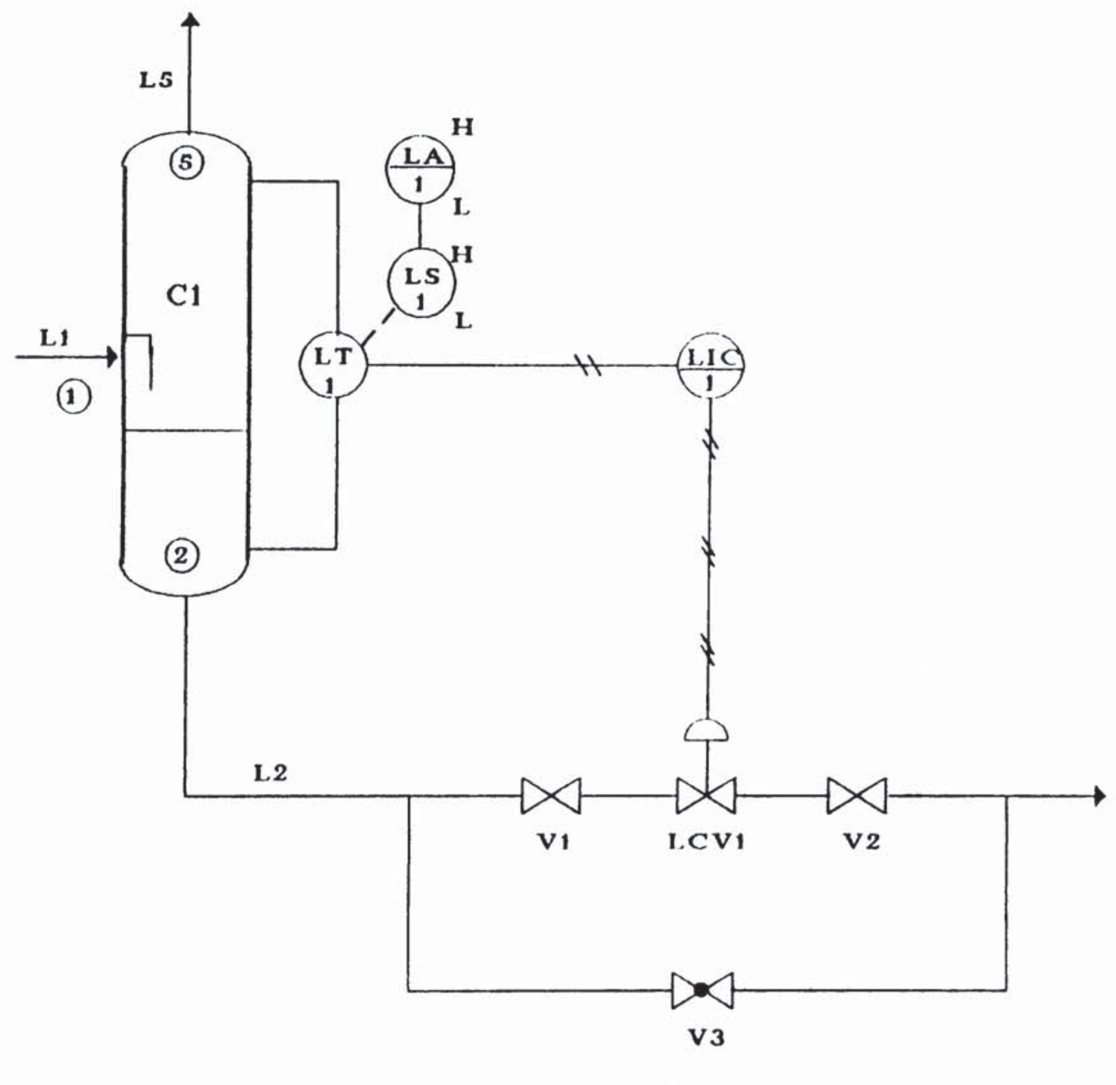


Figure 3.2 : Part of an ammonia let down system [17].

The cause equation in its coded form is

L2(11) = N2(11) + L2(O) + L2(BV) 3.1

The cause equation for L2(BV) is

L2(BV) = V1(O) + LCV1(O) + V2(O) 3.2

which means that (line L2, valves blocked) is caused by one or more of the valves V1, LCV1 or V2 being closed or blocked.

To show an example of the use of the **AND** operator, consider the causes of loss of liquid level in vessel C1 which can be written as:

(no level, vessel C1) **is caused by** (level transmitter LT1 indicating higher than actual)
or (level control loop giving too much flow in line 2 AND low level alarm failed to danger)

The coded cause equation is

C1(41) = LT1(1) + LCL1(1) * LAL1(FD) 3.3

LT1(1) indicates a malfunction in the level transmitter. LCL1(1) and LAL1(FD) can be further developed to give the following cause equations:

$$LCL1(1) = LIC1(-1) + LCV1(1) + V3(1) \qquad \text{..... 3.4}$$

$$LAL1(FD) = LSL1(1) + LAL1(0) + LAL1(IG) \qquad \text{..... 3.5}$$

Equation 3.4 means that the level control loop, LCL1, giving too much flow is caused by the set point of the level controller, LIC1, set too low or the control valve, LCV1, open too much or the bypass valve, V3, open. Equation 3.5 means that the low level alarm, LAL1, failed to danger is caused by the low level switch, LSL1, set high or stuck low or the low level alarm, LAL1, failed or ignored when activated.

3.2.3 : Symptom Equations

For symptom equations, the deviation in input streams to an equipment is written at the left hand side of the equation followed by a dash sign (-) which means "causes the following". On the right hand side of the equation are responses to the deviation at certain nodes in the equipment. If there are several nodal responses, these are separated by asterisks to show that all the nodal responses in the equation will occur. The nodes are points where streams enter and leave major equipment and each is given a unique number. In some circumstances, output streams can also cause symptoms in the equipment from which the stream is leaving. For example, changes in flow rate of an output liquid stream may cause responses in the liquid level in the equipment.

As an illustration, consider again figure 3.2. The consequences of no flow into the vessel C1 via line 1 can be written as:

(no flow, line L1) **causes the following** (no flow at node 2)
AND (no flow at node 5)

The coded Symptom Equation has the form :

L1(11) - N2(11) * N5(11) 3.6

Symptom equations are primarily used to connect the deviations of input streams to the responses of the output streams of major plant equipment. For example, from equation 3.1, one of the causes of no flow in line L2 is the nodal response N2(11). Looking at the symptom equation 3.6, N2(11) is a consequence of no flow in line L1. Hence, it can be inferred that a cause of no flow in line L2 is the deviant state of no flow in line L1. The causes of L1(11) are obtained from developments of cause and symptom equations of lines and equipments upstream of the vessel C1. In this way, the cause and effect relationship of deviant states between one plant item and another, not necessarily adjacent to one another, can be obtained.

While the cause and symptom equations are concise, it is difficult to comprehend. The relationships can more easily be visualised by converting the cause and symptom equations into fault trees. This can be done with a suitable computer program.

3.3 : Fault Tree Analysis

3.3.1 : Introduction

Fault tree analysis or FTA in its abbreviated form, is a deductive technique that focuses on one particular fault and provides a method for determining the causes of that fault. A fault tree as defined by Himmelblau [72] is a diagrammatic description in the form of a logic tree to show how an event may occur from sequences of faults and failure events. It makes use of logic symbols and text blocks to build up the tree. The description of the logic symbols used in FTA are given in Appendix B.

The method of FTA starts with the statement of an undesired event, usually a system failure or an accident, and works backwards through the plant configuration to trace equipment failures and operational or procedural errors that may lead to the undesired event. Thus, in a fault tree, the undesired event called the top event, is at the top of the diagram and the sequences of events that may cause the undesired event form the branches of the fault tree.

The advantage of the fault tree representation is that the structure of the sequences and combinations of causes and effects are clearly laid out for the human observer to trace and diagnose the causes of an abnormal condition in a chemical plant. Fussell [73] stated that the major values of FTA are in:

i) Directing the analyst to ferret out failures deductively.

2) Pointing out the aspects of the system important in respect of the failure being considered.

3) Providing a graphical aid giving an appreciation of design to those in management who are not directly involved.

4) Providing options for qualitative or quantitative system reliability analysis.

5) Allowing the analyst to concentrate on one particular failure at a time.

6) Providing the analyst with genuine insight into system behaviour.

In addition, studies have shown that FTA is helpful in the field of operator training, decision making, start up and shut-down procedures, alarm analysis and in the writing of operating manuals [74,75].

3.3.2 : Terms Used in Fault Tree Analysis

Different names and terms are used to describe the building blocks and properties of the fault tree. Some of these are given below.

An **accident** or **system failure** is the term used to describe the unplanned and unwanted occurrence of an event or a group or series of events to produce a loss or a near loss. An example of an accident is when a pressure vessel ruptures due to the internal pressure exceeding the design limits of the vessel. An example of a system failure is when a distillation column does not give the desired separation as required.

A **basic** or **primary event** is the term used to represent a basic equipment fault or failure, or human error that requires no further development into more basic faults or failures. If the fault under investigation is a system failure, repair of the equipment fault or failure will bring the system back to normal. An example of a primary event is when a valve becomes blocked cutting off flow.

An **intermediate** or **secondary event** is the term used to represent a fault event that results from the interactions of other fault events. Usually, secondary events are deviations in process parameters. An example is when the expected flowrate in a pipe line is not acheived. To remedy a secondary event fault requires that the primary events that cause it be repaired.

A **common cause event** is an event which causes multiple intermediate events to occur. On the fault tree diagram, a common cause event is repeated at several places on the fault tree. Another name given to a common cause event is a **repeated event**. An example of a repeated event is when cooling water supply is cut off affecting several units of the process.

The way the intermediate or secondary event develops depends on the **logic gate** that defines the input conditions which must be met in order for the event to occur. Logic gates that are most frequently used to develop fault trees are the basic **AND** and **OR** Boolean operators. The **AND** gate indicates that the output event occurs only when all the input events occur. The **OR** gate indicates that the output event occurs if any of the input events occur. Other logic gates frequently in use are the **Exclusive OR** and the **NOT** gates [82, 94]. An **Exclusive OR** gate, sometimes given the abbreviation **EOR** or **XOR**, has two input events and the output event occurs if either, but not both, of the input event occur.

3.3.3 : Fault Tree Synthesis

For a large complex system such as a chemical plant, the process of generating and analysing the fault trees can be time consuming and expensive. Rasmussen stated that the WASH-1400 study on the assessment of accidental risk of a nuclear power plant took about 25 man-years of effort to complete [77] . Several workers have developed algorithmic methods and computer programs to overcome this shortcoming.

Fussell [78] presented a method for fault tree synthesis of electrical systems called the Synthetic Tree Model (STM). His method constructs fault trees from input data consisting of failure transfer functions of components in the system. The component failure transfer functions are in actual fact mini-fault trees, in which the top event is the component output failure and the basic events are the component input failures. The fault tree of the whole system is constructed by combining the relevant component mini fault trees. In electrical systems, components have only two states and system dynamics are not significant. These simple elements do not represent fully the elements required for a chemical plant. Thus, the Synthetic Tree Model is not well suited to a chemical process system where system dynamics are important and process parameters have variable states. However, work done by Fussell formed the basis of fault tree applications in chemical processes taken up by other workers.

Powers and Tomkins [79,80] devised a systematic approach for automated fault tree construction for chemical systems. In their method, each unit in the process system is modelled by information flow relationship as described by Rudd and Watson [81] instead of the general unsteady state mass, energy and momentum transport differential equations. Only the steady state mass energy and momentum balances of the unit in question need to be solved to show how a change in magnitude and direction of input variables affects the output variables. Hence, by knowing the cause and effect relationship that describe the behaviour of a unit, nearly all possible failure modes can be predicted leading to the development of failure models for the unit. The mini fault trees generated from the failure models formed the basis for synthesizing a larger fault tree of the system for investigating a hazard event.

Salem et al [82] use decision tables to model failure modes of chemical process components in their computer program called CAT

(Computer Automated Tree) for constructing fault trees. The advantage of using decision tables is that modelling is not restricted to the hardware of the system. Complex operator actions and interactions that may produce a hazardous event also can be modelled. Thus, a fault tree with the top event that may be caused by wrong sequences of actions by operators can be constructed.

Lapp and Powers [51] presented another method of modelling system failure modes which is the digraph (directed graph) model. A computer package called Fault Tree Synthesis (FTS) was produced to synthesise fault trees using the digraphs as data.

Andow and Lees [40] proposed the use of functional model equations to model components in a process system. Their work was related to process alarm analysis by a computer, whereby the data structure as input to the program was synthesised from the functional model equations and the plant topology. Martin-Solis et al [41] later use the same type of modelling for their work in fault tree synthesis for design and real-time fault diagnosis.

Fault trees can also be synthesised from HAZOP study of a plant, as described by Lawley [13] and Kletz [14]. The events in the fault trees constructed are stated in sentences or short phrases which require a lot of space and can be quite cumbersome. Also, the extraction of relevant information from results of the HAZOP study, usually in the form of HAZOP tables, can be quite difficult. However, by using a coding system and representing the HAZOP information as cause and symptom equations, as proposed by Lihou [17, 62], the construction of fault trees is made easier.

3.3.4 : Fault Trees From Cause and Symptom Equations

Each cause equation is in actual fact a mini fault tree. As an illustration, referring to figure 3.2, the fault tree of the undesired event C1(41) is obtained by combining the cause equations 3.3, 3.4 and 3.5. as shown in figure 3.3. The symbols and denote the **OR** and **AND** gates respectively.

For symptom equations, each nodal response is treated as a top event and the deviation that cause the response is treated as the cause of the top event. If there are several symptom equations having the same nodal response, the deviations that cause the response are treated as inputs via an **OR** gate. Nodal responses in symptom equations are used as a bridge to relate the deviant states of an output stream to the deviant states of the input stream of an equipment. Referring to figure 3.2, the relationship of no flow in line L2 and no flow in line L1 is expressed as the combination of cause equation 3.1 and symptom equation 3.6. The fault tree of this combination is shown in figure 3.4. Thus, from the list of cause and symptom equations from a HAZOP study, a large fault tree can be built up and the propagation of abnormal states or faults from one plant item to another can be easily traced.

3.3.5 : Fault Tree Evaluation

Once the fault tree has been constructed, it can be studied both qualitatively and quantitatively. Qualitative evaluation refers to

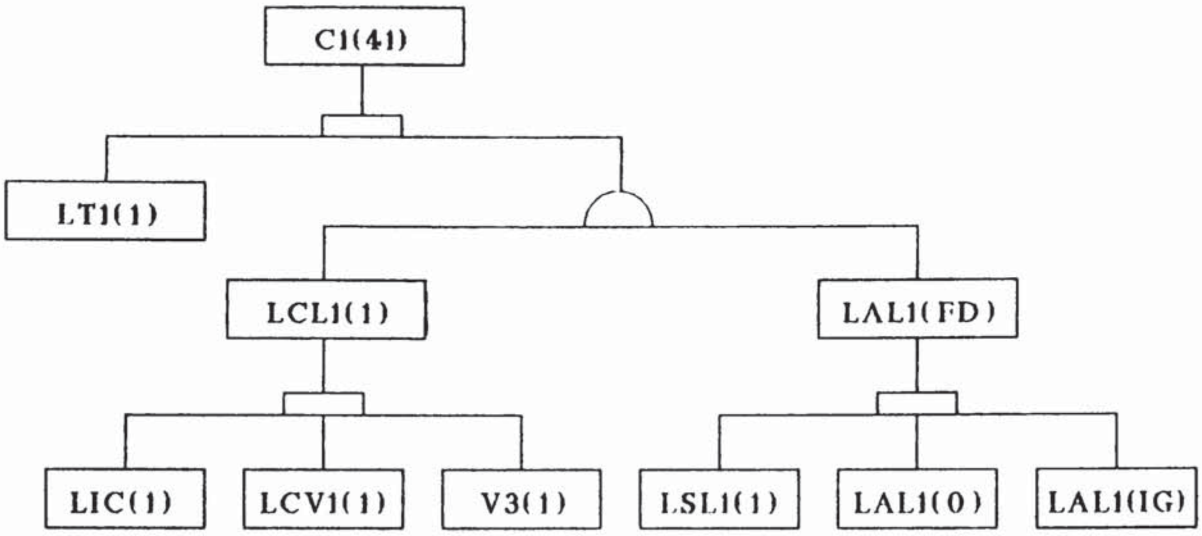


Figure 3.3 : Fault tree after combining equations 3.3, 3.4 and 3.5

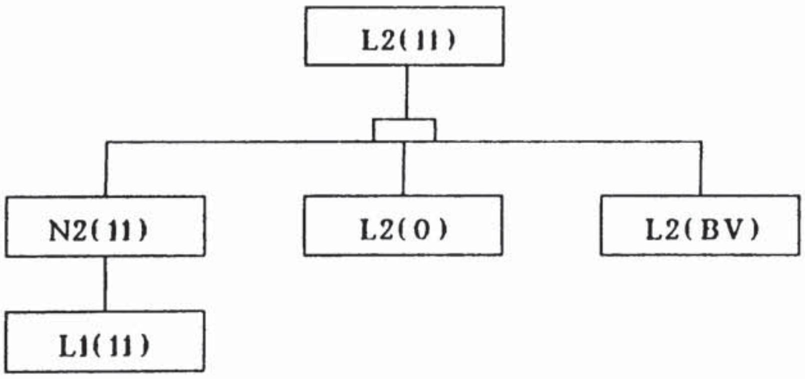


Figure 3.4 : Fault tree for the combination of equations 3.1 and 3.6

identifying the unique modes by which the top event can occur. The unique modes of occurrence are termed **minimal cut sets** or **mode failures**. A **cut set** is a collection of basic events, such that when all occur, the top event is guaranteed to occur. A **minimal cut set** is the smallest group of basic events that must all simultaneously occur in order for the top event to occur. The finite collection of all unique minimal cut sets of a fault tree represents all the unique, non redundant ways by which the top event can occur. Thus, the minimal cut sets represent those set of events which are critical with regard to the occurrence of the top event. For example, in the fault tree of figure 3.3, if events V3(1) and LSL1(1) occur simultaneously, the top event will occur. V3(1) and LSL1(1) form a minimal cut set of the fault tree.

The minimal cut sets represent a source of information about the state of the process and makes possible the calculation of process performance. By assigning probability values to the basic events, and using the minimal cut sets obtained, the probability of occurrence of the top event can be calculated. The calculation of event probabilities constitutes the quantitative evaluation of the fault tree.

Knowledge of the probability of the top event will determine whether or not the risks involved are acceptable. If needed, changes in the system can be made to reduce the risks involved. Decisions as to what changes are to be made can be guided by looking into the events comprising any minimal cut set for the system failure with significant probability. With the proposed changes, the fault tree is restructured to verify whether there is any significant reduction of the risks involved. This loop of evaluation, decision and verification is repeated until an acceptable fault tree is obtained. This shows the importance of minimal cut sets and probability evaluation of fault trees. Methods and algorithms for obtaining minimal cut sets and fault tree probabilities are discussed in detail in chapter four.

CHAPTER FOUR

4. FAULT TREE PROBABILITY CALCULATIONS

4.1 : Introduction

Fault trees are logical statements of various ways in which a system can fail to perform a defined function. Although fault trees are qualitative in nature, they provide a framework for quantitative probabilistic analysis. The analysis consists of assigning probabilities to each bottom event and combining them, as prescribed by the fault tree, to obtain the probability of the top event.

The bottom events of a fault tree are assumed to be primary events, in the sense that occurrence of a primary event is not effected by other primary events and also will not affect the outcome of other primary events. Therefore, the occurrence of each primary event is independent of all other primary events. Thus for the purpose of calculating the probability of the top event of a fault tree, the bottom events are assumed to be statistically independent of each other.

The calculated probability values of top events can be used to evaluate process design modifications suggested to reduce the risk of any hazardous event occurring. The probability of occurrence of the top event is reduced if the probability of any of the bottom events is reduced. The significance of the reduction depends on the occurrence of **AND** and **OR** gates in the fault tree. For bottom events connected by an **AND** gate, significant reduction can be obtained by reducing the probability of any of the bottom events. For the case of **OR** gates, significant reduction can only be achieved by reducing those with the larger probability.

In real-time fault diagnosis, probability estimates of events in the fault tree can be used as a guide for searching out the cause of an occurring alarm. Lihou [62] presented a strategy for checking the possible causes of an alarm event with the aim of minimising the probable time for fault finding, fault diagnosis and corrective action. In his method, the time required to check each possible fault is divided by the probability of that fault causing the alarm. The optimal order for checking faults is in ascending order of this quotient.

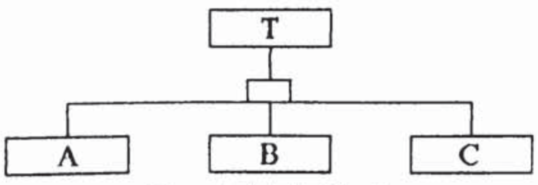
However, the probability values that Lihou used are those calculated *a priori*, i.e., before any event had actually happened. Once there is an alarm, the probability of the alarm event has changed to the value 1, indicating the existence of the alarm event. The probabilities of causal events in the fault tree have to be recalculated since there now exists the condition that a particular event has occurred. The recalculated probability values known as conditional probabilities or *posteriori* probabilities can then be used in the strategy to search optimally for the cause of the alarm. The posteriori probability provides a truer picture of which causal event is the most likely cause of the alarm, for example is it a true of false alarm.

To facilitate quantitative analysis, it is convenient to represent the fault tree in a mathematical form, and Boolean algebra is an appropriate tool for this purpose. Once a fault tree has been represented by its equivalent Boolean equations, probabilities can be evaluated by applying probability laws and axioms. There is an analogy between the Boolean equations and the cause and symptom equations of Lihou [71].

This chapter discusses how Boolean equations to mathematically represent a fault tree are obtained, how probability data is obtained and methods of evaluating *a priori* and *posteriori* probabilities of a fault tree.

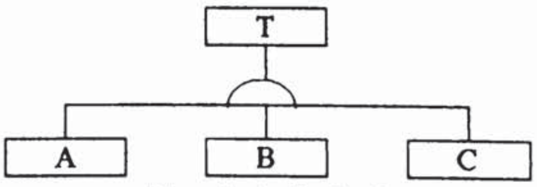
4.2 : Laws of Boolean Algebra

In the Boolean algebra of fault trees, the basic quantities are the events. Each unique event is assigned a unique symbol for convenience of notation and manipulation. Logical operators in the fault tree, mainly **AND** and **OR** gates are simply graphical symbols which represent Boolean operations on the various events. For example, the **OR** gate is equivalent to the Boolean operator $\vee$ and represents the union of the events attached to the gate. The **AND** gate is equivalent to the Boolean symbol $\wedge$ representing the intersection of events. Figures 4.1 and 4.2 shows the fault trees and the equivalent Boolean representation of **OR** and **AND** gates respectively.



$T = A \vee B \vee C$

Figure 4.1 : **OR** Gate



$T = A \wedge B \wedge C$

Figure 4.2 : **AND** Gate

In general, fault trees involve several levels of secondary events between the top undesired event and the bottom events. The fault tree is represented mathematically by writing an equivalent Boolean equation for each gate. By eliminating the secondary events from these equations, an expression is obtained which defines the top event in terms of only the primary events. The primary events are those which must be quantifiable to proceed with the analysis, i.e., for which if data exist, the top event can be quantified by use of this equation.

To carry out the elimination of secondary events, basic laws of Boolean algebra as shown in Table 4.1 [83] are used. To illustrate the elimination and transformation of a fault tree, figure 4.3 is used as an example.

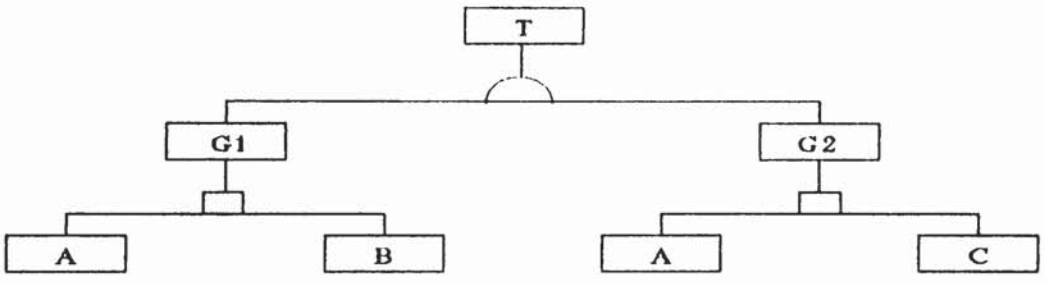


Figure 4.3 : A fault tree with secondary events

The Boolean expressions for the fault tree in figure 4.3 are:

$T = G1 \wedge G2$ 4.1

$G1 = A \vee B$ 4.2

$G2 = A \vee C$ 4.3

Table 4.1: Rules for Boolean Manipulation [83]

Property	Boolean Expression
Commutative Law	a) $A \vee B = B \vee A$ b) $A \wedge B = B \wedge A$
Associative Law	a) $A \wedge (B \wedge C) = (A \wedge B) \wedge C$ b) $A \vee (B \vee C) = (A \vee B) \vee C$
Distributive Law	a) $A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$ b) $A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$
Idempotent Law	a) $A \wedge A = A$ b) $A \vee A = A$
Absorption Law	a) $A \vee (A \wedge B) = A$ b) $A \wedge (A \wedge B) = A \wedge B$ c) $A \wedge (A \vee B) = A$
Complemented Law	a) $A \wedge \bar{A} = 0$ b) $A \vee \bar{A} = 1$
Reduction Law	a) $A \vee (\bar{A} \wedge B) = A \vee B$ b) $(A \wedge B) \vee (A \wedge \bar{B}) = A$ c) $(A \wedge B) \vee (A \wedge B) = A \wedge B$

Substituting the expressions for G1 and G2 into the expression for T:

$$T = (A \vee B) \wedge (A \vee C) \qquad \text{..... 4.4}$$

The top event T is now expressed in terms of the primary events A, B and C. Using the distributive and absorption laws of Boolean algebra, equation 4.4 can be transformed into

$$T = A \vee (B \wedge C) \qquad \text{..... 4.5}$$

The transformed Boolean expression for T in equation 4.5 can now be recast into the associated transformed fault tree as shown in figure 4.4.

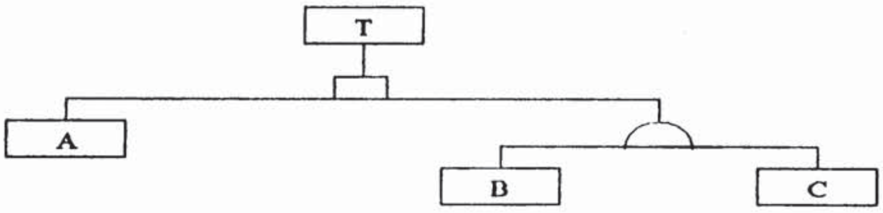


Figure 4.4: Transformed fault tree of figure 4.3

By transforming fault trees to simplified, equivalent forms, the complexity and tediousness of quantification can be reduced. In general, the transformed Boolean expression can be expressed as :

$$T = M_1 \vee M_2 \vee \ldots\ldots \vee M_n \qquad \text{..... 4.6}$$

The terms M_i, i=1 to n, consist of intersections of primary events. Another term commonly used for the expression in equation 4.6 is the sum-of-products (SOP). The expression for M_i is :

$$M_i = C_{i1} \wedge C_{i2} \wedge \ldots \wedge C_{ik} \wedge \ldots \wedge C_{im} \qquad \ldots\ldots 4.7$$

where C_{ik}, k = 1 to m, are primary events.

A property of the sum-of-products is that no M_i is a subset of another M_j for all i and j and $i \neq j$, i.e., the primary events of M_i are not all contained in M_j. M_i is termed the critical path or minimal cut set of the fault tree. A minimal cut set can be intepreted as a unique "failure mode" by which the top event can occur. In the example for the fault tree in figure 4.3, the minimal cut sets are A and $B \wedge C$, which means that T occurs if A occurs or if both B and C occurs.

4.3 : Failure Probability Data

4.3.1 : Introduction

A fault in a system may sometimes result from the breakdown of a single component or from a combination of several component failures. Failure probability data of equipment are obtained from statistical data from past observations. One form of failure probability data of equipment is what is termed the "mean time between failures". Other ways of representing failure probability data include the use of statistical distributions known as probability density functions, derived from past observations of failures. These include the binomial, multinomial, Poisson, negative exponential, normal, lognormal, Weibull, rectangular, gamma, Pareto and extreme value distributions. Hasting and Peacock [84] give a comprehensive summary of the properties of the above mentioned distributions. Only the negative exponential and Weibull distribution are discussed in this section.

4.3.2 : Mean Time Between Failures

Mean time between failures, or MTBF as it is commonly known, is a measure of the expected time between two consecutive failures. Suppose that an equipment has failed N times, each time being repaired and put back into service. The time interval between successive failures i and i+1 is t_i. Assuming that the equipment is as good as new after repair and the failures occurred randomly, then the mean time between failures, MTBF, is given as

$$MTBF = \left(\sum_{i=1}^{N} t_i \right) / N \qquad ..... \ 4.8$$

The unbiased estimate $\hat{\sigma}^2$ of the variance, σ^2 of MTBF from the observed data is

$$\hat{\sigma}^2 = \frac{1}{N-1} \sum_{i=1}^{N} \left(t_i - MTBF \right)^2 \qquad ..... \ 4.9$$

From the value of $\hat{\sigma}$, the next occurrence of failure can be estimated with a certain limit of confidence. Loosely speaking, there is a 68% probability that the next time of failure after repair, t_{N+1}, will be in the range $MTBF - \hat{\sigma} \le t_{N+1} \le MTBF + \hat{\sigma}$. Similarly, there is a 95% probability that $MTBF - 2\hat{\sigma} \le t_{N+1} \le MTBF + 2\hat{\sigma}$.

4.3.3 : Probability Density Functions

If f(t) is the failure probability density function of an equipment and F(t) is the probability that it experiences the first failure by time t, given that it is good as new at time zero, then

$$F(t) = \int_0^t f(t)\,dt \qquad \text{..... 4.10}$$

The probability that an equipment experiences a failure per unit time, given that it was repaired and good as new at time zero and has survived to time t is known as the failure rate of the equipment denoted by r(t). It is obtained using the relation

$$r(t) = \frac{f(t)}{1 - F(t)} \qquad \text{..... 4.11}$$

If the failure rate of a component is known, the failure probability and the failure probability density function can be evaluated using the following identities:

$$F(t) = 1 - \exp\left[- \int_0^t r(t)\,dt \right] \qquad \text{..... 4.12}$$

$$f(t) = r(t)\exp\left[- \int_0^t r(t)\,dt \right] \qquad \text{..... 4.13}$$

4.3.3.1 : Negative Exponential Distribution

The negative exponential distribution is defined as

$$f(t) = \lambda \exp(-\lambda t) \qquad \text{..... 4.14}$$

Substituting equation 4.14 into equation equation 4.10 and integrating between the limits, the failure probability is given as

$$F(t) = 1 - \exp(-\lambda t) \qquad \text{..... 4.15}$$

Substituting equations 4.14 and 4.15 into equation 4.11,

$$r(t) = \lambda \qquad \text{..... 4.16}$$

Hence, for negative exponential distributions, λ is the failure rate and is assumed to be independent of time or age or environmental influence. Since the failure rate is constant, it is equivalent to the reciprocal of the MTBF of the equipment.

$$MTBF = 1/\lambda \qquad \text{..... 4.17}$$

Thus for random occurence of failure whence the MTBF can be evaluated, the failure probability and the failure probability density function can be obtained using equations 4.12 and 4.13 respectively.

Suppose that the unit for the failure rate is per year. Equation 4.15 with $t = 1$ can be expanded as follows:

$$F(1) = \lambda - \lambda / 2! + \lambda / 3! - \ldots\ldots\ldots \qquad \ldots\ldots 4.18$$

If λ is 0.1 yr^{-1} or less, the second and subsequent terms in equation 4.18 may be neglected making the failure probability and failure rate have similar values. But this is not the case all the time because failure probability and failure rate are two different statistics. Typical values of failure rates for process plant equipments are shown in the Appendix C.

4.3.3.2 : Weibull Distribution

Some equipment may not fail randomly. Breakdown may be due to its design, quality of construction and the environment in which it is placed. A general probability density function to take into account these factors is the Weibull distribution and expressed as follows:

$$f(t) = (\beta/\eta)(t/\eta)^{\beta-1} \exp\{-(t/\eta)^{\beta}\} \qquad \ldots\ldots 4.19$$

where

η = scale parameter or the characteristic age of the equipment. A high value indicates well designed equipment, good quality control, large factors of safety and operating below capacity.

β = shape parameter which defines the type of distribution.

Comparing equations 4.14 and 4.19 with $\beta = 1$ and η = MTBF will show that the negative exponential distribution is a special case of the Weibull distribution.

Lihou [85] summarised how the value of β found for equipment influences the cause of failure. For equipment found with $\beta < 1$, indicates incompetent or incomplete maintenance. When gradual deterioration or wearout is the sole cause of failure, β values of 3 to 3.5 will be found. The probability density function will be nearly symmetrically distributed about MTBF, corresponding to the well known normal distribution. When failures are due to a combination of deterioration and superimposed random overloads, values of between 1 and 3 are found. When $\beta = 1$ the system is totally out of control and failures occur randomly. Methods for finding the values of β and η are discussed briefly in reference [84].

Substituting equation 4.19 into equation 4.10 and integrating between the limits,

$$F(t) = 1 - \exp\{ -(t/\eta)^{\beta} \} \qquad \text{..... 4.20}$$

The failure rate is obtained by substituting equations 4.19 and 4.20 into equation 4.11.

$$r(t) = (\beta/\eta)(t/\eta)^{\beta-1} \qquad \text{..... 4.21}$$

4.3.4 : Failure Probability For Standby Safety Equipment

Standby safety equipment such as trips, pressure relief valves and non-return valves may fail and the failure will remain undetected until there is a need. In order to detect failure of safety systems, they are inspected and tested periodically. If the inter-inspection interval is T (years), the proportion of time for which the system will remain in the failed state during any test interval is called the Fractional Dead Time (FDT). For the purpose of evaluating probabilities of fault trees involving safety equipment, FDT is used to represent the failure probability of the equipment. The value of FDT is obtained by the relation:

$$FDT = \frac{1}{T} \int_0^T F(t)\,dt \qquad \text{..... 4.22}$$

For the negative exponential distribution, which represents random failures, substituting equation 4.15 into equation 4.22 and integrating between the limits will give :

$$FDT = 1 - \{1 - \exp(-\lambda T)\} / \lambda T \qquad \text{..... 4.23}$$

Expanding equation 4.23 gives

$$FDT = \lambda T/2! - (\lambda T)^2/3! + (\lambda T)^3/4! - \ldots\ldots \qquad \text{.....4.24}$$

Typically $\lambda T << 1$, making the second and subsequent terms of equation 4.24 insignificant. Hence equation 4.24 is reduced to

$$FDT = \lambda T/2 \qquad \text{.....4.25}$$

A period of time is required during inspection and testing of safety equipments known as the proof testing period. If t' is the proof testing period, the fraction t'/T is considered as part of the fractional dead time. During the testing period, human operators are needed to monitor and perform certain operations to effectively simulate the function of the safety devices. Suppose that the number of operations needed is n and for each operation i there is an associated error probability e_i. The total fractional dead time is thus

$$FDT = (\lambda T/2) + (t'/T) + \sum_{i=1}^{n} e_i \qquad \text{..... 4.26}$$

4.4 : Principles of Fault Tree *A Priori* Probability Evaluation

4.4.1 : Introduction

Once the probability failure data are assigned to primary events, the probabilities of the top and secondary events of a fault tree can then be evaluated. Basically there are two approaches for evaluating the probability of non-primary events in a fault tree. One approach is to evaluate the probability directly from its structure. This can be done by traversing the fault tree from the bottom to the top, evaluating first the probability of every secondary event which have only primary events as its inputs and then, using the results obtained, calculating the probabilities of other secondary events and eventually the top event of the fault tree. Another approach is to express every non-primary event in the fault tree as a sum-of-products. The probabilities are then evaluated from the Boolean expressions obtained. This involves determination of the minimal cut sets of the event under consideration.

The following account in this section describes the rules and axioms to carry out probability calculations.

4.4.2 : Probability Rules

Calculation of the probability of any non-primary event in a fault tree basically involves the intersection rule and union rule which are described below.

1. Intersection Rule

If an event T which occurs only if all its causal events C_i occur, $i = 1$ to n, then the logical expression for T is

$$T = C_1 \wedge C_2 \wedge \ldots\ldots \wedge C_n \qquad \ldots\ldots 4.27$$

For all C_i independent of each other, the probability of T is

$$P(T) = P(C_1) * P(C_2) * \ldots\ldots\ldots * P(C_n) \qquad \ldots\ldots 4.28$$

Generally,

$$P(T) = \prod_{i=1}^{n} P(C_i) \qquad \ldots\ldots 4.29$$

2. Union Rule

If T can be caused by either of two of its causal events C_1 and C_2, the logical expression for T is

$$T = C_1 \vee C_2 \qquad \ldots\ldots 4.30$$

For C_1 and C_2 independent of each other, the probability of occurrence of T is

$$P(T) = P(C_1) + P(C_2) - P(C_1) * P(C_2) \qquad \ldots\ldots 4.31$$

Generally, if T occurs due to any of its causal events C_i, i=1 to n, the logical expression for T is

$$T = C_1 \vee C_2 \vee \ldots\ldots\ldots\ldots \vee C_n \qquad \ldots\ldots 4.32$$

The probability of occurrence of T for all independent C_i is

$$P(T) = \sum_{i=1}^{n} P(C_i) - \sum_{i=2}^{n} \sum_{j=1}^{i-1} P(C_i) * P(C_j)$$

$$+ \sum_{i=3}^{n} \sum_{j=2}^{i-2} \sum_{k=1}^{j-1} P(C_i) * P(C_j) * P(C_k)$$

$$- \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$+ (-1)^{n-1} \prod_{i-1}^{n} P(C_i) \qquad \ldots\ldots 4.33$$

A simpler expression for obtaining the probability of T involves using the complement of events. For the event T not to occur, all the C_i must not occur. Hence the probability of event T not to occur, i.e. probability of $\overline{T}$ is given as

$$P(\overline{T}) = \prod_{i=1}^{n} [\, 1 - P(C_i)\,] \qquad \text{..... 4.34}$$

Since T and $\overline{T}$ are mutually exclusive, the probability of occurrence of T is simply

$$P(T) = 1 - \prod_{i=1}^{n} [\, 1 - P(C_i)\,] \qquad \text{..... 4.35}$$

Another possible case is if all C_i of equation 4.32 are mutually exclusive. Then equation 4.33 is reduced to

$$P(T) = \sum_{i=1}^{n} P(C_i) \qquad \text{..... 4.36}$$

4.4.3 : Probability Evaluation of Fault Trees With Repeated Events.

Sometimes there exist some basic events appearing more than once in a fault tree. Applying directly the probability laws in a bottom up calculation of the top event probability will not give the correct result. As an illustration consider the fault tree in figure 4.3 which has a repeated event A.

$$P(G1) = P(A) + P(B) - P(A)*P(B) \qquad \text{..... 4.37}$$

$$P(G2) = P(A) + P(C) - P(A)*P(C) \qquad \text{..... 4.38}$$

Since G1 and G2 are inputs to an **AND** gate to cause T, the probability of T would be:

$$P(T) = P(G1)*P(G2) \qquad \text{..... 4.39}$$

However, G1 and G2 are not independent because both can be caused by the repeated event A. Thus multiplying the values of the probabilities of G1 and G2 as expressed in equation 4.39 will not give the exact probability of T.

To overcome problems arising for probability calculations from repeated events in a fault tree, special techniques have to be used. Lihou and Jones [16] proposed the use of a Bayesian conditional probability method. The method essentially states that when there are several probable outcomes for an event of which there is no certain knowledge that one or more of other events can cause its occurrence, the probability of occurrence of the event is evaluated by considering all the possibilities and simply adding their probabilities.

For example, suppose that T is the top event of concern and A is an event appearing in several places in the fault tree. Two possible outcomes need to be considered, i.e. the outcomes of T when A occurs and when A does not occur. Expressed in probability terms, the probability of occurrence of T is:

$$P(T) = P(T/A)*P(A) + P(T/\bar{A})*P(\bar{A}) \qquad \text{..... 4.40}$$

where

$P(T/A)$ is the probability of T given A has occurred,

$P(T/\bar{A})$ is the probability of T given A does not occur,

$P(A)$ is the probability of occurrence of A,

$P(\bar{A})$ is the probability of non-occurrence of $A = 1 - P(A)$

The evaluation of $P(T/A)$ and $P(T/\bar{A})$ can be done using Baye's conditional probability rule which states that the probability of two events X and Y occurring at the same time is given as:

$$P(X \wedge Y) = P(X/Y)*P(Y) = P(Y/X)*P(X) \qquad \text{..... 4.41}$$

Rearranging the expression in equation 4.41 :

$$P(X/Y) = \frac{P(X \wedge Y)}{P(Y)} \qquad \text{..... 4.42}$$

As an illustration, consider again the fault tree of figure 4.3. Looking at the fault tree structure, when A occurs, T is bound to occur and when A does not occur, the occurrence of T is dependent on B and C occurring together. Using Baye's Theorem:

$$P(T/A)*P(A) = P(T \wedge A) \qquad \text{..... 4.43}$$

$$P(T/\bar{A})*P(\bar{A}) = P(T \wedge \bar{A}) \qquad \text{..... 4.44}$$

The Boolean expression for T is given by equation 4.5. The expressions for $(T \wedge A)$ and $(T \wedge \bar{A})$ are:

$$T \wedge A = \left[A \vee (B \wedge C) \right] \wedge A = A \qquad \text{..... 4.45}$$

$$T \wedge \bar{A} = \left[A \vee (B \wedge C) \right] \wedge \bar{A} = \bar{A} \wedge B \wedge C \qquad \text{..... 4.46}$$

Thus:

$$P(T \wedge A) = P(A) \qquad \text{..... 4.47}$$

$$P(T \wedge \bar{A}) = P(\bar{A}) * P(B) * P(C) \qquad \text{..... 4.48}$$

Substituting equations 4.47 and 4.48 into equations 4.43 and 4.44 respectively and eventually substituting the results into equation 4.40:

$$P(T) = P(A) + P(\bar{A}) * P(B) * P(C) \qquad \text{..... 4.49}$$

But $P(\bar{A}) = 1 - P(A)$ 4.50

Hence, after sustituting equation 4.50 into equation 4.49:

$$P(T) = P(A) + P(B) * P(C) - P(A) * P(B) * P(C) \qquad \text{..... 4.51}$$

Equation 4.51 gives the correct probability value of the top event of figure 4.3.

In principle, the conditional probability method can be applied for any number of repeated events. For a fault tree with two repeated events A and B:

$$P\ (T) = P(T/A,B)*P(A)*P(B) + P(T/\overline{A},B)*P(\overline{A})*P(B)$$
$$+ P(T/A,\overline{B})*P(A)*P(\overline{B}) + P(T/\overline{A},\overline{B})*P(\overline{A})*P(\overline{B}) \quad ..... \ 4.52$$

The disadvantage with this method is that the number of evaluations increases exponentially with the number of repeated events. From equation 4.40, with one repeated event, the number of terms to be evaluated on the right hand side of the equation is two. With two repeated events, the number of terms to be evaluated is four, as shown in equation 4.52. Hence for n repeated events, the number of terms to be evaluated will be 2^n.

One way to overcome the problems of using the Bayesian conditional probability in dealing with repeated events is to express the top event of a fault tree as a Boolean sum-of-products which may produce independent minimal cut sets. The minimal cut sets approach to fault tree probability evaluation is discussed in the next section.

4.4.4 : Probability Evaluation Using Minimal Cut Sets

The transformation of a fault tree by Boolean algebra produces an expression termed the sum-of-products. Each term in the sum-of-products is in actual fact a minimal cut set of the fault tree. Since minimal cut sets contain only primary events, the probability of the top event is easily evaluated, if probability data are available for the primary events.

A minimal cut set of a fault tree represents a unique failure mode of the top event and every element of the set must happen simultaneously for the top event to occur. Thus the probability of a minimal cut set is calculated by using the probability intersection rule. If M_i represent a minimal cut set, its probability is obtained by evaluating the equation :

$$P(M_i) = \prod_{j=1}^{m} P(C_{ij}) \qquad \text{..... 4.53}$$

where C_{ij} is the jth primary event in the ith minimal cut set which has m unique primary events, assumed to be independent.

If a fault tree has n minimal cut sets represented by M_i, $i = 1$ to n, and no primary event in M_i appears in M_k for every other $k \neq i$ and for every $i = 1$ to n, then the minimal cut sets of the fault tree are said to be s-independent of each other. For such a case, each minimal cut set can be treated as a primary event. Thus the probability of the top event is:

$$\begin{aligned} P(T) \;=\; & \sum_{i=1}^{n} P(M_i) - \sum_{i=2}^{n} \sum_{j=1}^{i-1} P(M_i) * P(M_j) \\ & + \sum_{i=3}^{n} \sum_{j=2}^{i-2} \sum_{k=1}^{j-1} P(M_i) * P(M_j) * P(M_k) \\ & - \ldots\ldots\ldots\ldots\ldots\ldots \\ & + (-1)^{n-1} \prod_{i=1}^{n} P(M_i) \qquad \text{..... 4.54} \end{aligned}$$

Whether or not there are repeated events in a fault tree, there is no guarantee that the minimal cut sets obtained will be s-independent of each other. For example, the fault tree in figure 4.3 has a repeated event A, and the minimal cut sets obtained as the sum-of-products in equation 4.5 are s-independent. Thus by applying equations 4.53 and 4.54 to determine the probability of the top event, the same result as in equation 4.51 would be obtained.

For comparison, consider the fault tree in figure 4.5 originally given by Fussell [86] which has a repeated event B. By applying Boolean algebra, the sum-of-products expression for the fault tree in figure 4.5 is :

$$T = (A \wedge B) \vee (A \wedge C \wedge D) \qquad \ldots\ldots 4.55$$

Clearly the minimal cut sets obtained as shown in equation 4.55 are not s-independent and applying directly equation 4.54 to obtain the probability of the top event will give an incorrect result.

An example of a fault tree with no repeated events but generating non s-independent cut sets is shown in figure 4.6. The sum-of-products expression for the fault tree in figure 4.6 is :

$$T = (A \wedge B) \vee (A \wedge C) \qquad \ldots\ldots 4.56$$

The minimal cut sets in equation 4.56 are not s-independent. So applying equation 4.54 to obtain the probability of T is not valid.

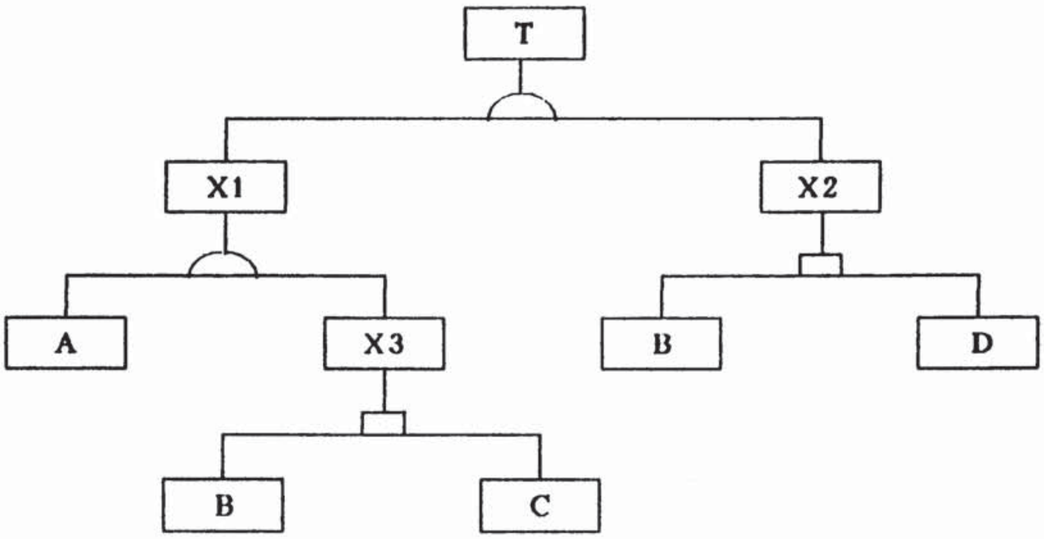


Figure 4.5 : A Fault Tree by Fussell [86]

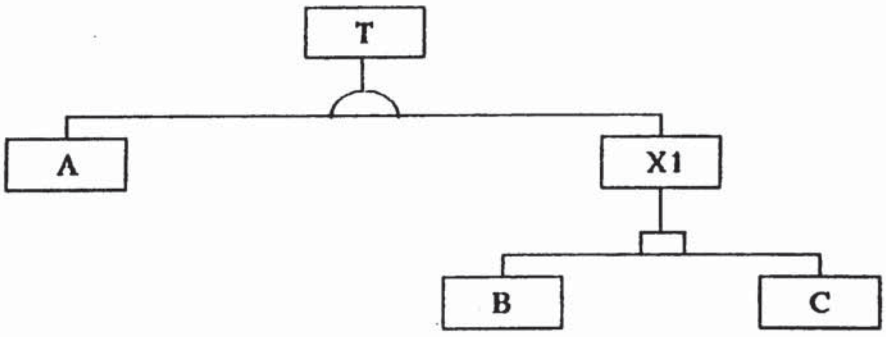


Figure 4.6 : A fault tree with no repeated events

However, using the bottom-up probability calculation approach, first obtaining the probability of X1 by applying the union rule and then applying the intersection rule, the correct probability would be obtained.

4.4.5 : Conclusion

The main problem in evaluating the probability of a fault tree is the effect of repeated events such that the probability rules cannot be applied because of existence of non-independent terms in the calculation. The conditional probability method to take care of the effect of repeated events creates the independence required for applying the probability rules. However, the use of the conditional probability method can be very tedious and time consuming if the number of repeated events in the fault tree is large. The alternative is to use the minimal cut sets method. However, there is no guarantee that independent minimal cut sets will be obtained, even if there is no repeated events in the fault tree.

Development of methods to overcome the problems of non-independence in fault tree probability evaluation that can be implemented on a computer has been going on for the past decade. Publications of computer programs and algorithms to evaluate probability of fault trees using the minimal cut sets approach and directly from the fault tree structure are discussed in the next two sections.

4.5 : Probability Evaluation Algorithms Using Minimal Cut Sets

4.5.1 : Introduction

In the minimal cut sets approach to fault tree probability evaluation, the minimal cut sets of every non-primary event in the fault tree has to be determined first. A fault tree of a complex chemical plant can be very large involving many levels of secondary events between the top event and the primary events. Generating the minimal cut sets of such a fault tree manually can be time consuming and tedious. Development of ways to use a computer to perform such a task has been going on for the past decade. Publications of algorithms and computer programs to speed up minimal cut sets generation and also the calculation of fault tree probabilities are numerous. Some of the work done by researchers in this field is discussed below.

4.5.2 : Algorithms For Generating Minimal Cut Sets

4.5.2.1 : Introduction

The technique for generating minimal cut sets of a fault tree can be categorised into two main approaches, either a top down or bottom up procedure. A top down algorithm begins with the top event and works downward to the primary events while a bottom up algorithm begins with with primary events and works upward to the top event.

4.5.2.2 : Top Down Algorithms for Determining Minimal Cut Sets

In 1972, Fussell and Vesely [92] developed a top down approach for determining the minimal cut sets of a fault tree using Boolean substitution method. A key point of this methodology is that an **AND** gate alone always increases the size of cut sets while an **OR** gate alone always increases the number of cut sets. To obtain the minimal cut sets, this method requires that the Boolean Indicated Cut Sets be obtained first. A Boolean Indicated Cut Set is defined such that the primary events in the set are different and appear only once. If it happens that there are more than one of the same primary event in the set, application of the Boolean absorption rule will make the set conform to the definition stated above. When the Boolean Indicated Cut Sets have been determined, a simple search procedure is used to determine the minimal cut sets. This is done by removing supersets. A superset is a cut set which has other cut sets as sub-sets. Supersets are also known as redundant sets.

In 1974, Fussell et al [86] published a computer package called MOCUS based on the method by Fussell and Vesely. The MOCUS algorithm can be stated as follows:

1. Locate the top event in the first row and column of a matrix.

2. Iterate either of the fundamental permutations (a) or (b) in a top down fashion:
   (a) if inputs to the event are via an OR gate, replace the event by its inputs in a vertical arrangement so as to increase the number of cut sets.

(b) if inputs to the event are via an AND gate, replace the event by its inputs in a horizontal arrangement so as to enlarge the size of the cut sets.

3. When the elements of the matrix are all primary events, the rows of the matrix represent the cut sets of the top event. Each cut set is checked for repetition of primary events. When there is more than one occurrence of the same event in the cut set, the repetition is removed so that the event appears only once in the cut set. This produces the Boolean Indicated Cut Sets. Then the minimal cut sets are obtained by removing supersets.

The main virtue of MOCUS is its simplicity and its ability to generate all the cut sets of the top event. However, a large amount of the computer resources is needed since all the cut sets have to be generated before being reduced to obtain the minimal cut sets. Also, two stages of comparisons have to be done in MOCUS which increases the execution time as the size of a cut set and the number of cut sets generated become larger. The first stage is when events in a cut set are compared with each other to remove repetitions of primary events. The second stage is when all the cut sets generated are compared with each other to remove supersets. Modifications to MOCUS and algorithms based on MOCUS have since been developed to overcome some of the problems stated above.

In 1976, Bengiamin et al [91] developed a technique using similar Boolean substitution method as developed by Fussell and Vesely [92].

The difference in the two methods is that the algorithm by Bengiamin et al sifted out all repeated events which are inputs to **OR** gates from the fault tree to produce a reduced fault tree. From the reduced fault tree, cut sets are obtained using the method by Fussell and Vesely. The cut sets obtained are referred to as Group 1 cut sets and are all minimal.

The Group 1 cut sets are further processed by substituting the partners of the repeated events with various combinations of their corresponding repeated events. Partners of a repeated event are those events that appear as inputs together with the repeated event to the same **OR** gate. The cut sets resulting from this process are known as Group 2 cut sets. The minimal cut sets are obtained by weeding out supersets in Group 2 cut sets. Bengiamin et al claimed that the computing time in determining the minimal cut sets when using their algorithm is much shorter than the Fussell and Vesely method. This is achieved due to the early elimination of non-minimal cut sets when determining the Group 1 cut sets.

In 1977, Caldarola et al [90] produced the Karlsruhe Computer Program which is similar to the top down algorithm MOCUS [86]. Certain extra features are incorporated in the Karlsruhe program to make it more efficient than MOCUS. Before determining the minimal cut sets, the fault tree is ordered in the form of a list. Primary events are first entered into the list followed by secondary events. A secondary event is accepted into the list only if its inputs have already been accepted into the list. The top event is at the bottom of the list. The advantage of this ordered list is that it ensures that each element of the fault tree is completely defined so that logical errors can be detected. Determination of the cut sets proceeds from the bottom of the list.

Another feature inherent in the Karlsruhe program that makes it more efficient than MOCUS is the indentification of "super events" or pseudo primary events. If all the primary events that are the inputs of a secondary event are not found as primary events of other secondary events, then the first secondary event is known as a "super event". Once a super event has been identified, it is treated as a primary event and need not be expanded to its basic events during the determination of cut sets.

Finally, the Karlsruhe program follows the criterion to expand a row, i.e. a cut set, down to its primary events before proceeding to the next row. Caldarola claimed that the extra features added reduce the execution time and storage area of the computer as compared to MOCUS.

In 1978, Rasmusson and Marshall [93] developed a top down algorithm called FATRAM which is similar to MOCUS. FATRAM's main aim is to take care of repeated events more efficiently than MOCUS so that the computer time required for determining the minimal cut sets is much less. This is done by reducing the amount of comparisons for removing supersets. FATRAM also requires less computer storage than MOCUS.

The FATRAM algorithm follows the same substitution method of MOCUS except that in stage 2, **OR** gates which have primary events as inputs are not resolved yet. When all **OR** gates with gate inputs and all **AND** gates are resolved, supersets which may exist are removed. This is due to repeated secondary events or repeated primary events as inputs to **AND** gates appearing in the fault tree. After this stage, repeated primary events in the unresolved **OR** gates are processed. For each repeated event, it replaces the unresolved gate of which it is an input to form a new set. When all the repeated events have been processed, any supersets that exist are removed. After this stage, the remaining **OR** gates are resolved with the remaining non-repeated primary events. The cut sets obtained after this stage are all minimal.

In 1986, Limnois and Ziani [96] developed a method to reduce the number of comparisons for eliminating redundant cut sets. In their algorithm, prior knowledge of the repeated events is required. The cut sets of a top event are first obtained using MOCUS [86]. Then cut sets with the repeated events are grouped into one set and comparisons made on this to eliminate non-minimal cut sets. Cut sets which do not contain the repeated events are already minimal.

The algorithms described above are mostly modifications on MOCUS designed to improve execution time and reduce the storage area. However, a drawback common to all the algorithms described above is that only the minimal cut sets of the top event under consideration is determined for one traversal of the fault tree. To obtain the minimal cut sets of secondary events in the fault tree, each event has to be treated as a top event and apply the chosen algorithm again. The execution time on the computer will be greatly reduced if during one fault tree traversal, the minimal cut sets of every non-primary event can be determined.

4.5.2.3 : Bottom Up Algorithms for Determining Minimal Cut Sets

In 1971, Semanderes [87] put forward a computer program coded in FORTRAN called ERLAFT which formulates the simplest logic expression for each secondary event in a fault tree in terms of the basic events that combine to cause it. The algorithm uses a bottom up approach and the Boolean reduction techniques discussed earlier, to produce the sum-of-products of the top event being evaluated. The problem with the Boolean reduction technique as reported by Semanderes is that it requires a great deal of computer memory for most fault trees found in practice. To reduce usage of computer memory, Semanderes used prime numbers and the property of prime numbers as expressed in the unique factorization theorem [88].

Each basic event is assigned a prime number. A particular combination of basic events can be expressed uniquely as a single number which is equal to the product of the prime numbers. From this single number, the basic events that make up the combination, i.e., a cut set can be determined by factoring the number into its prime factors. If a unique number is a factor of another, it indicates that the later has a combination of basic events which is a superset of the former. In this way supersets can be easily removed, and the final result obtained will be all the minimal cut sets of the top event.

In 1977, Wheeler et al [89] introduced a fault tree analysis program called FAULTRAN which was written in FORTRAN. The program is divided into two parts. The first part determines the minimal cut sets using the bottom up approach. The second part determines the probability of the top event.

The Wheeler method uses a bit vector representation where each basic event is assigned a single 1 in a unique position in a sequence of binary digits. The output of an AND gate is the bit by bit logical sum of the inputs which can be obtained with the intrinsic **OR** function supplied with FORTRAN. The output of an **OR** gate is the list of non-duplicated inputs. The benefit of using bit manipulation is that redundant events in a cut set are automatically eliminated.

Once all the cut sets have been obtained, still in their bit vector representation, simple logical test are used to identify redundant cut sets which can be eliminated. Suppose V1 and V2 are the vectors representing two cut sets. The logic for redundancy test in the actual FORTRAN implementation is as follows:

1) If "OR(V1,V2).EQ.V1" is true, V1 is redundant. If however the test is not true, then do the next test.
2) If "OR(V1,V2).EQ.V2" is true, V2 is redundant. If neither test is true, no redundancy is detected.

The advantage of using logic operations on bit vectors in the FAULTRAN algorithm is that comparing individual basic events between two cut sets for determining redundant cut sets is avoided, producing faster and more efficient code. Another benefit in ERLAFT as well as in FAULTRAN is that they are capable of generating the minimal cut sets of all secondary events and the top event in a fault tree, thereby making it possible to calculate the probability of events with one traversal of the fault tree.

In 1975, Bennetts [94] developed a bottom up algorithm using reverse Polish notation. The basic strategy of the algorithm is to derive an algebraic description of the top event as a function of the primary events and operators (**AND** and **OR** gates) as a reverse Polish expression and then to "unpack" this expression into its equivalent sum-of-products form. Later Nakashima and Hattori [95] improved on Bennetts's algorithm so as to reduce the computation time during weeding out redundant cut sets. The computer program developed by Nakashima and Hattori is called BUB-CUTS. The main reason that BUP-CUTS is faster than Bennetts's algorithm is that prior knowledge of the repeated events of the fault tree is required by BUB-CUTS whereas there is no such requirement in Bennetts's algorithm. However, for both BUP-CUTS and Bennett's algorithm, it is not possible to obtain the minimal cut sets of every non-primary event in the fault tree with one traversal of the fault tree. Thus, the determination of the minimal cut sets and eventually the probability of every non-primary event in a fault tree has to be done separately.

There are numerous other computer programs for determining minimal cut sets of a fault tree that have been developed. Most of them involve more complicated processing than the ones mentioned above and concentrate on determining the minimal cut sets of the top event of a fault tree. For example, the Kumamoto and Henley [98] algorithm operates on the dual fault tree. A dual fault tree is defined as a fault tree whose original **AND** and **OR** gates have been converted to **OR** and **AND** gates respectively. A computer program called DICOMICS, developed by Garibba et al [97], first segments the fault tree to several fault subtrees. Minimal cut sets of the fault subtrees are determined and then combined to construct the minimal cut sets of the top event. Other minimal cut sets determination programs, just to mention a few, are RESIN [99], AFTP [100] and the Jasmon and Kai algorithm [101].

4.5.3 : Algorithms For Probability Calculation Using Minimal Cut Sets

4.5.3.1 : Introduction

The probability of a minimal cut set is calculated using equation 4.53. Generally, if a non-primary event T has n minimal cut sets, i.e. M_i, i=1 to n, then the probability of occurrence of T is:

$$P(T) = \sum_{i=1}^{n} P(M_i) - \sum_{i=2}^{n} \sum_{j=1}^{i-1} P(M_i \wedge M_j)$$
$$+ \sum_{i=3}^{n} \sum_{j=2}^{i-2} \sum_{k=1}^{j-1} P(M_i \wedge M_j \wedge M_k)$$
$$- \cdots\cdots\cdots$$
$$+ (-1)^{n-1} \prod_{i=1}^{n} P(M_1 \wedge M_2 \wedge \ldots \wedge M_n) \qquad \ldots\ldots 4.57$$

If all the minimal cut sets are s-independent, then equation 4.57 will become equation 4.54.

It has been shown that the minimal cut sets of a non-primary event in a fault tree may not be s-independent. Basically, there are two methods to overcome the problem of non s-independence of minimal cut sets in calculating the top event probability. The first method is to manipulate the intersection of minimal cut sets when applying the probability union rule so that primary events appear only once in each product. The second method is by manipulating the minimal cut sets to make them mutually exclusive or disjoint. These methods are discussed below.

4.5.3.2 : Techniques For Manipulating Intersections Of Minimal Cut Sets

Vesely [102, 103] developed a computer package called KITT for evaluating reliability parameters of large and complicated fault trees. The package requires as input the minimal cut sets of the event whose probability is to be evaluated. The basic assumption is that the basic events that constitute the minimal cut sets are independent. The probability of each minimal cut set is first determined using equation 4.53.

Consider a general minimal cut set intersection with m terms, i.e. $M_i \wedge M_j \wedge \ldots\ldots \wedge M_m$, where $i \neq j \neq \ldots\ldots \neq m$. Vesley stated that this intersection can be considered as a cut set or failure mode. Since the primary failures are assumed independent, and a mode failure exists if and only if all its primary failures exist, then

$$P(M_i \wedge M_j \wedge \ldots\ldots \wedge M_m) = \prod^{+i,..,m} q \qquad \ldots.. \; 4.58$$

where q is the probability of the basic event found in $M_i \wedge M_j \wedge \ldots. \wedge M_m$. The product symbol is defined such that

$\prod^{+i,..,m}$ = the product of unique basic event quantities where the basic event occurs in at least one of the mode failures i, j, ..., m.

A particular basic event quantity, i.e., its probability, thus occurs at most once in the product and occurs only if the basic event is a member of at least one of the minimal cut sets denoted above the product symbol. Computation of $P(M_i \wedge M_j \wedge \ldots. \wedge M_m)$ using equation 4.57 therefore simply consists of collecting the unique basic events which are members of one or more of the m minimal cut sets and then multiplying the probabilities of these basic events. If Q_0 is the probability of the non-primary event that is to be evaluated with N minimal cut sets and Q_i is the probablity of the ith minimal cut set, then rewriting equation 4.57:

$$Q_0 = \sum_{i=1}^{N} Q_i - \sum_{i=2}^{N}\sum_{j=1}^{i-1} \prod^{+i,..,j} q + \sum_{i=3}^{N}\sum_{j=2}^{i-2}\sum_{k=1}^{j-1} \prod^{+i,..,k} q$$

$$- \ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$+ (-1)^{N-1} \prod^{+1,..,N} q \qquad \ldots.. \; 4.59$$

The exact value Q_0, can straightforwardly be determined using equation 4.59. KITT ensures that basic events in the product terms occur only once before multiplication is carried out.

Wheeler et al [89] use the same equation 4.59 in their FAULTRAN algorithm for evaluating probabilities. The bit manipulation technique is used to ensure that primary events appear only once in each product combination. Caldarola et al [90] in their Karlsruhe Computer Program also use equation 4.59 but expressed in a different form.

The computer time required to evaluate exact probabilities using equation 4.59 depends on the number of minimal cut sets found for the top event. The number of product terms to be evaluated on the right hand side of equation 4.59 increases exponentially with the number of minimal cut sets. If there are N minimal cut sets, the number of product terms will be $2^N - 1$.

4.5.3.3 : The Disjoint Cut Set Technique

The products of minimal cut sets when using the probability union rule can be eliminated if the minimal cut sets are made in such a way that they are disjoint or mutually exclusive. If there are N disjoint cut sets, then the number of terms to be evaluated is also N. The probability of the top event is evaluated by applying the probability rule for mutually exclusive events where the probabilities of each disjoint cut set are just added up.

Bennetts [94, 104] summarised the theory of the disjoint technique as follows:

(a) Pairs of terms in the sum of product expression are compared to determine their disjointness. The pair are disjoint if and only if one term contains at least one Boolean variable and the other term contains the same variable but in its complemented form. If the pair of terms are not disjoint, then a modification must be made to render it disjoint relative to the other.

(b) If a pair of terms M_i and M_j are found to be non-disjoint, define their relative complement $RC = M_i \backslash M_j$ as the non empty set $\{y_1, y_2, \ldots\ldots\ldots, y_n\}$, where all y's are written in the uncomplemented form. The elements of RC are those variables found in M_i but not found in M_j. The procedure by which M_i and M_j are converted into a disjoint collections of terms is described by the following expression:

$$\begin{aligned} M_i \vee M_j = M_i &\vee (\bar{y}_1 \wedge M_j) \vee (y_1 \wedge \bar{y}_2 \wedge M_j) \vee \ldots\ldots\ldots \\ &\vee (y_1 \wedge y_2 \wedge \ldots\ldots\ldots \wedge y_{r-1} \wedge \bar{y}_r \wedge M_j) \\ &\vee \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ &\vee (y_1 \wedge y_2 \wedge \ldots\ldots\ldots\ldots\ldots\ldots\ldots \wedge \bar{y}_n \wedge M_j) \end{aligned} \quad \ldots\ldots 4.60$$

Effectively, equation 4.60 is based on a controlled reintroduction of missing variables to individual products on a variable by variable basis.

To illustrate the procedure, consider the sum-of-product expression given in equation 4.55. Let $M_1 = A \wedge B$ and $M_2 = A \wedge C \wedge D$. Thus $RC = M_1 \setminus M_2 = \{ B \}$. With reference to M_1, the disjoint sum-of-product is

$$T = (A \wedge B) \vee (\bar{B} \wedge A \wedge C \wedge D) \qquad \text{..... } 4.61$$

Applying the probability law of union of mutually exclusive events,

$$P(T) = P(A)*P(B) + P(\bar{B})*P(A)*P(C)*P(D) \qquad \text{..... } 4.62$$

But $P(\bar{B}) = 1 - P(B)$. Therefore

$$P(T) = P(A)*P(B) + P(A)*P(C)*P(D) - P(A)*P(B)*P(C)*P(D) \qquad \text{..... } 4.63$$

The result obtained in equation 4.63 gives the correct probability of the top event of the fault tree in figure 4.5. Making $M_1 = A \wedge C \wedge D$ and $M_2 = A \wedge B$ will produce a different disjoint sum-of-product expression but the same probability expression as equation 4.63.

Generating minimal cut sets and then applying the disjoint algorithm to evaluate fault trees can take up quite a considerable computer time to process. Recent works by Jionsheng [105] and Ramadaan [106] has shown that the usage of computer time can be reduced by using the disjointness principle to produce disjoint sum-of-products directly from the fault tree without determining the minimal cut sets.

4.5.4 : Discussion and Conclusion

The minimal cut sets approach to fault tree probability evaluation involves two stages of processing. The first stage is the minimal cut set generation from the fault tree structure. Problems arise if the fault tree contains several repeated secondary and primary events which can produce repeated terms in a cut set and also redundant cut sets. As large fault trees produce large cut sets as well as large numbers of cut sets, much time and effort is spent on comparing pairs of elements in every cut set produced to ensure that there is no repetition of terms within the cut set and also on comparing pairs of cut sets to remove redundant cut sets. Thus most of the developments in the generation of minimal cut sets of fault trees as described previously, have been done either to devise methods to handle repeated events more efficiently so that there will be few comparisons to be made, or to improve on methods for removing repetitions within cut sets and also for removing redundant cut sets.

The second stage is the actual probability calculation. Here, problems arise when the minimal cut sets obtained are not all s-independent of each other. If the evaluation of fault tree probability based on equation 4.57 is to be used, further processing has to be done so that every intersection of minimal cut sets contains no duplication of the primary events. If instead the disjoint cut set technique is to be used, complicated processing is involved to make all the minimal cut sets disjoint or mutually exclusive.

Considering all the drawbacks and virtues of the algorithms described in this section as well as the discussion above, it is not viable to implement the minimal cut set approach to fault tree probability evaluation on a microcomputer. The main reason is the limited amount of memory available on the microcomputer to contain the codes for the various phases of minimal cut set generation and probability evaluation. The only alternative is to apply a direct probability calculation from the fault tree structure, discussed in the next section.

4.6 : Probability Calculations Directly From Fault Trees

4.6.1 : Introduction

The fault tree contains secondary events through which the basic events propagate to the top event. In the minimal cut set approach to fault tree probability calculations, the minimal cut sets of every non-primary event have to be determined first before the probability can be evaluated. Thus the computer time required to calculate the probability of every event in the fault tree can be large. An alternative approach to fault tree probability calculations is by direct computation from the tree structure. With this approach, the probabilities of all the events in the fault tree may be evaluated with one traversal of the tree. Several algorithms for evaluating probabilities directly from the fault tree are reviewed below.

4.6.2 : Algorithms For Direct Evaluation Of Fault Tree Probabilities

An algorithm for direct computation of fault tree probabilities was put forward by Lee [76] where the Boolean expression of the fault tree is converted to reverse Polish notation. Indirectly the method restructures the fault tree to form a binary tree where there are only two inputs to every gate. The probability of the top event is evaluated using the bottom up approach. Two nodes are taken at a time and the probability law of union or intersection is applied till the top event is reached. Lee claimed that his algorithm calculates the exact probability accurately and very quickly. The only problem in Lee's algorithm is that it can only be applied for fault trees with no repeated events.

Feo [107] produced a computer package called PAFT 77 in which the basic laws of probability as expressed by equations 4.29 and 4.33 are used. To handle repeated events, Feo used the conditional probability as expressed in equation 4.40 or 4.52. Apart from calculating probabilities of top events as well as secondary events PAFT 77 also calculates the marginal importance of basic events. The marginal importance of a basic event is a measure of the influence of that event on the probability of occurrence of the top event.

Koen and Carnino [108] used a list processing technique in compiling the events of a fault tree. The algorithm for calculating probabilities is based on recognition of patterns, i.e., standard fault tree configuration. The fault tree is scanned for known patterns stored in a library. Whenever a recognisable pattern is found in the tree, it is replaced as a pseudo primary event and its probability is calculated. This process is repeated until the original tree is reduced to a single node when the probability of the top event is determined. However Koen and Carnino's algorithm suffers from lack of generality because it cannot handle repeated events where there may be interdependency between one pattern and others.

Page and Perry [109, 110] proposed a top down recursive method in calculating probabilities of fault trees. The algorithm is known as TDPP (Top-Down Page-Perry). The probability calculation algorithm is coded as a subroutine in a main program. Within that subroutine, it can call itself to obtain data which have not been calculated yet. This recursion makes it possible for the computer code to be small enough to be implemented on microcomputers where memory size is limited. The language used by Page and Perry is PASCAL, which supports recursion. PASCAL also supports sets and set operations. The structure ensures that elements in a set appear only once. Indirectly the Boolean idempotent and absorption laws are applied for the collection of inputs to an event.

One advantage of TDPP is its ability to handle repeated events efficiently. A brief description of the theory behind TDPP is given below.

4.6.3 : The Top Down Recursive Algorithm - TDPP

The development of a top down recursive algorithm for direct computation of fault tree probabilities was first presented by Page and Perry in reference [109]. However, the published algorithm can only be used on fault trees restructured into a binary form.

In reference [110], Page and Perry presented an improved version of their original algorithm. Instead of working on binary trees, TDPP can work on any fault tree having **AND** and **OR** gates. The function PROB in TDPP acts on two sets S1 and S2 which are passed as parameters. PROB(S1,S2) computes the probability that every event in S2 occurs and that at least one event in S1 occurs. This implies that if a non-basic event corresponds to an **OR** gate, all its inputs are placed in S1. However if the non-basic event corresponds to an AND gate, all its inputs are placed in S2. An array LEAVES is used to check for s-independence in S1 and S2. The array LEAVES is set up such that LEAVES[n], for any node n, will be the set of all nodes representing basic events below node n in the tree.

The probability of a top event T, using TDPP is computed via the call PROB({ }, {T}). Inside PROB, there are two distinct cases which will be acted upon depending on whether or not S1 is empty. The logic of the algorithm is described below.

Case I : Evaluation of PROB({ }, S2), i.e. S1 empty. A preliminary simplification is made where all AND gates in S2 are replaced by their inputs. This process is repeated until no AND gates remain in S2. After this operation, there are three cases numbered 1 - 3. The case that applies determines the recursion that is to be used.

1. If S2 contains only basic events, then PROB returns the product of their probabilities. If S2 is empty, PROB returns the value 1.

2. If S2 consists of a single node representing an **OR** gate, then PROB calls PROB(S1, { }) where S1 is the set of inputs to the single node in S2.

3. If S2 contains at least one **OR** gate as well as other elements, determine if there is a node n in S2 representing an event that is s-independent from the other elements in S2. This checking uses the array LEAVES to obtain the property such that LEAVES[n] is disjoint from LEAVES[m] for all other elements m in S2.

   3a. If such a node n is found, then PROB is evaluated by the recursion:

   PROB({ }, {n}) * PROB({ }, S2-{n}).

   3b. Otherwise, pick an **OR** gate node m in S2 and evaluate PROB via the call PROB(S1, S2-{m}) where S1 is the set of inputs of node m.

Case II : Evaluation of PROB(S1, S2) when S1 is not empty. A preliminary simplification is made where all **OR** gates in S1 are replaced by their inputs until no **OR** gates remain. Following this operation there are four cases, numbered 1 - 4:

1. If it is found that $S1 \cap S2$ is non-empty, PROB calls PROB({ },S2). This implies that there is at least one event in S1 also an element of S2. The call to PROB({ }, S2) represent a simple Boolean reduction.

2. If S1 contains a single node, PROB calls PROB({ }, S1+S2).

3. If S1 contains more than one element, determine if there is a node n in S1 which is s-independent from the other nodes in S1 and from all nodes in S2. This determination uses the array LEAVES such that LEAVES[n] is disjoint from LEAVES[m] for all other elements m in S1 and that LEAVES[n] is disjoint from LEAVES[k] for all elements k in S2. If such a node n exist, then PROB returns the value

   PROB({ },{n}) * [PROB({ }, S2) - p] + p

   where p = PROB(S1-{n}, S2).

4. If no node n is found in the above case, pick any node m in S1. PROB returns a value by using the following recursion

   PROB({ }, S2+{m}) + PROB(S1-{m}, S2)-PROB(S1-{m}, S2+{m})

The recursion in case II.4 above is a special use of

$$P((A \vee B) \wedge C) = P(A \wedge C) + P(B \wedge C) - P(A \wedge B \wedge C) \qquad ..... 4.64$$

If A and C are s-independent and A and B∧C are also s-independent, equation 4.64 can be expressed as

$$P\{(A \vee B) \wedge C\} = P(A) * \left[P(C) - P(B \wedge C) \right] + P(B \wedge C) \qquad \ldots.. 4.65$$

Equation 4.65 forms the reasoning behind the use of the recursion in case II.3.

As an illustration, using the fault tree of figure 4.5, the recursion process for determining the probability of T is shown in figure 4.7. From the trace of the recursion, the probability of T is:

$$P(T) = P(A)*[P(C)*[P(B)*[1-P(D)]+P(D)-P(B)]+P(B)] \qquad \ldots..4.66$$

Expanding equation 4.66, the same result as equation as 4.63 will be obtained. TDPP is more efficient than Page and Perry's original algorithm. This is due to less calls to PROB in TDPP than in their original algorithm and also because the fault trees do not have to be converted into a binary form.

To further reduce the number of calls to PROB, Page and Perry use a modularization technique to make the size of the fault tree smaller [110]. This is done by merging two primary inputs into a single one if the two always appear together as siblings and always with the same type parent, i.e. either **AND** gate or **OR** gate. When two such primary inputs are merged, the correct probability for the newly created pseudo primary input is determined by the parent type so as to represent the logical union or intersection. This merging process is

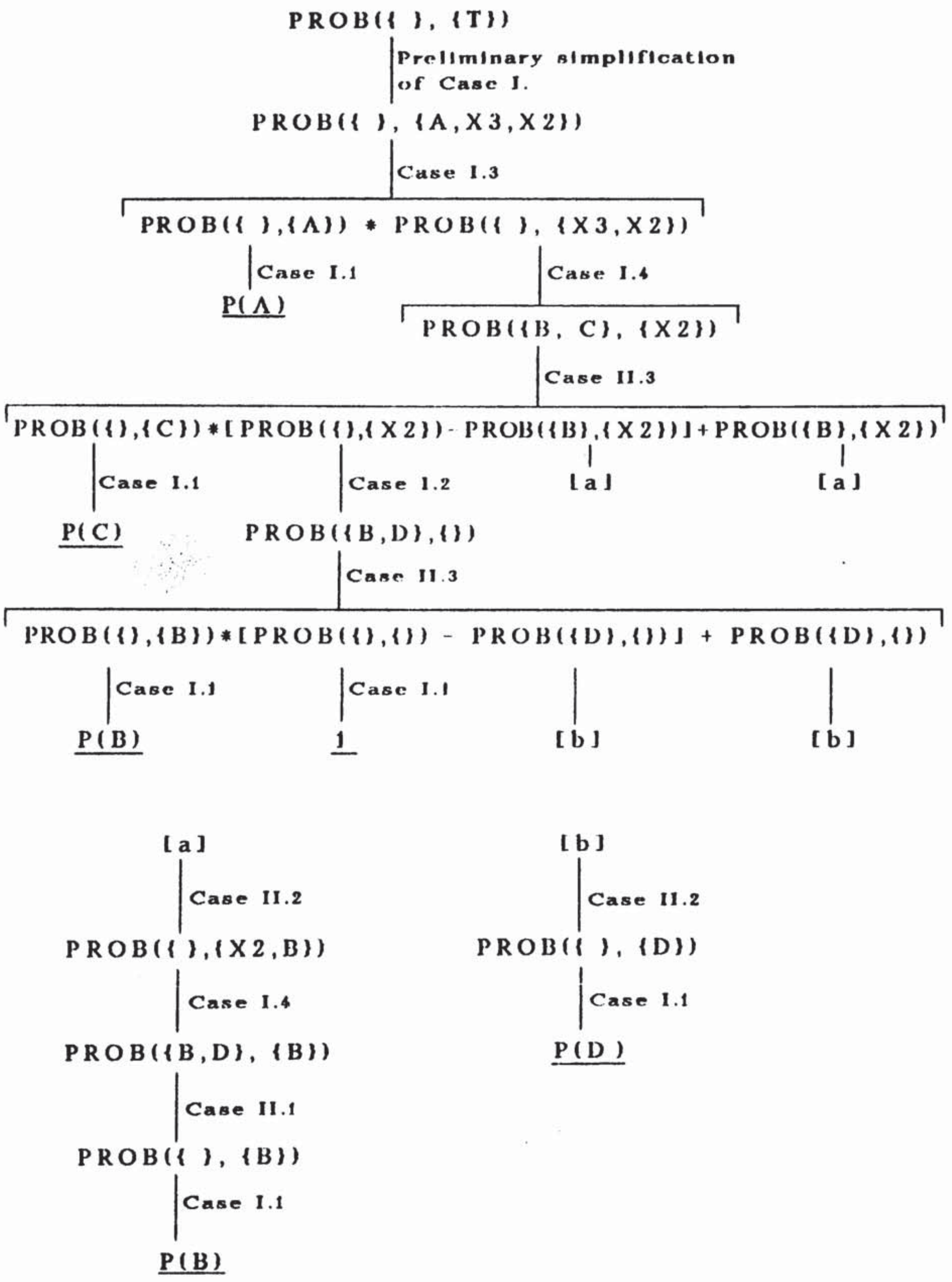


Figure 4.7 : Recursion trace of algorithm TDPP on fault tree of figure 4.5

repeated, operating on the newly created pseudo primary inputs until no further mergers are possible. Then any **AND** gate or **OR** gate that is left with one input as a result of the modularization is reclassified as a primary input and assigned the correct probability.

If the fault tree has no repeated events, the bottom up modularization as suggested by Page and Perry [110] would evaluate the probabilities of all the events in the fault tree without using the top down recursive algorithm. Thus modularization is advantageous if there are many repeated events in the fault tree.

4.6.4 : Discussion and Conclusion

Among the algorithms to evaluate probabilities directly from the fault tree stucture described in this section, TDPP is the most attractive. This is due to its simplicity and compactness of the computer code which is easily implemented on a microcomputer. Also, TDPP is capable in handling repeated events present in a fault tree, without the user knowing that this has occurred. It is for these reasons that the author of this present work has chosen the basic principles of TDPP for evaluating the *a priori* fault tree probability as part of the research.

One drawback in TDPP is in the way it was set up such that the probabilities of non-basic events are calculated one at a time. This is due to the preliminary simplifications in cases I and II until the sets S1 and S2 contain no OR gates and AND gates respectively. Thus during the recursion process, the probabilities of secondary events having the same gate type as the top event are not evaluated. To determine the probability of a secondary event, the whole recursion process is repeated with that secondary event treated as a top event.

TDPP can easily be modified so that during the recursion process, the probabilities of independent secondary events or sub-trees can be determined and stored without using the bottom-up modularization process. If the calculated probability of the secondary event is required again by other recursions, the data can be used instead of going through the recursion process. This lessens the execution time on the computer. The implementation of the modified TDPP is described in detail in chapter 5.

4.7 : Fault Tree *Posteriori* Probability Evaluation

4.7.1 : Introduction

When an alarm has occurred, the probability of the alarm event has changed to the value 1, indicating the existence of the alarm event. Since there is the condition that an event has occurred, the probability values of its causal events and its consequences are no more those calculated *a priori*. *Posteriori* or conditional probabilities of the causes and consequences of the occurred event have to be calculated to provide a truer picture of the situation. The object of calculating *posteriori* probabilities is to rank the causal events that contribute to the occurrence of the alarm. These values can be used as a guide in searching for the most likely cause of the alarm. The method of evaluating the *posteriori* probability is based on Bayes' Theorem which is discussed briefly below.

4.7.2 : Bayes' Theorem

The definition of Bayes' Theorem has already been defined in section 4.4.3. The calculation of the conditional probability of an event given another event has occurred is given by equation 4.43 which is reprinted below:

$$P(X/Y) = \frac{P(X \wedge Y)}{P(Y)} \qquad \ldots\ldots \ 4.66$$

where

P(X/Y) is the probability of X given that Y has occurred.

P(X ∧ Y) is the *a priori* probability that both X and Y has occurred together.

P(Y) is the *a priori* probability of occurrence of Y.

4.7.3 : Conditional Probability of Causal Events

Suppose all C_i, i=1 to n, are connected to T by an **OR** gate. The expression for T is given by equation 4.32. Let's suppose that T has occurred and C_j is suspected as the cause of T.

$$T \wedge C_j = (C_1 \vee C_2 \vee \ldots\ldots \vee C_j \vee \ldots\ldots \vee C_n) \wedge C_j \qquad \ldots\ldots \ 4.67$$

Using the laws of Boolean algebra on equation 4.67 :

$$T \wedge C_j = C_j \qquad \text{..... 4.68}$$

Applying equation 4.42, the conditional probability of occurrence of C_j given that T has occurred is:

$$P(C_j / T) = \frac{P(C_j)}{P(T)} \qquad \text{..... 4.69}$$

Suppose all C_i, i=1 to n, are connected to T by an **AND** gate instead. The expression for T is given by equation 4.27. Let's suppose that T has occurred and C_j is suspected as the cause of T.

$$T \wedge C_j = (C_1 \wedge C_2 \wedge \dots \wedge C_j \wedge \dots \wedge C_n) \wedge C_j \qquad \text{..... 4.70}$$

Using the laws of Boolean algebra on equation 4.70 :

$$T \wedge C_j = T \qquad \text{..... 4.71}$$

Applying equation 4.42, the conditional probability of occurrence of C_j given that T has occurred is:

$$P(C_j / T) = \frac{P(T)}{P(T)} = 1 \qquad \text{..... 4.72}$$

Equation 4.72 can be explained by the fact that all C_i must have occurred to cause T. Therefore the probability of any of the causal event that is connected to T by an AND gate must be unity.

There are cases when the alarm event has its causes connected by combinations of **OR** and **AND** gates. As an illustration, consider the fault tree in figure 4.3. The logical Boolean representation is given by equation 4.5, i.e.:

$$T = A \vee (B \wedge C) \qquad \text{..... 4.5}$$

Suppose that T has occurred. The conditional probability of B having occurred is given by:

$$P(B/T) = \frac{P(T \wedge B)}{P(T)} \qquad \text{.....4.73}$$

Using the laws of Boolean algebra:

$$\begin{aligned} T \wedge B &= \{A \vee (B \wedge C)\} \wedge B \\ &= (A \wedge B) \vee (B \wedge C) \qquad \text{.....4.74} \end{aligned}$$

Substituting equation 4.74 into equation 4.73:

$$P(B/T) = \frac{P\{(A \wedge B) \vee (B \wedge C)\}}{P(T)} \qquad \text{.....4.75}$$

Thus for the above case, the probability of the intersection of T and B has to be determined first before the conditional probability of B given T has occurred can be evaluated. This can be done by applying the **TDPP** algorithm discussed earlier, operating on an empty **S1**, and **S2** containing T and B.

4.7.4 : Conditional Probability of Consequent Events

Generally, there are two classes of consequent events that have to be considered. The first class of consequent events is the consequences of the event that has occurred i.e. the alarm event. The second class of consequent events is events which are neither causes nor consequences of the occurred event but are consequences of a cause of the alarm event.

4.7.4.1 : Conditional Probability of the Consequences of an Event Which Has Occurred.

Suppose that an event T_j is a secondary event in a fault tree and is in some combination with other T_i, $i \neq j$, through an **OR** gate to cause Q, i.e., a consequence of T_j. The conditional probability of Q occurring given T_j has occurred is

$$P(Q/T_j) = \frac{P(Q \wedge T_j)}{P(T_j)} = \frac{P(T_j)}{P(T_j)} = 1 \qquad \ldots.. 4.76$$

This implies that any consequences connected to the occurring fault by an **OR** gate will happen. Only a time delay for the consequence to respond to the occurring fault may give enough time for operators to take remedial actions.

However if Q is connected to the event T_j and other T_i, $i \neq j$, by an **AND** gate, the conditional probability of Q given T_j has occurred is

$$P(Q/T_j) = \frac{P(Q \wedge T_j)}{P(T_j)} \qquad \text{..... 4.77}$$

For such a case, $(Q \wedge Tj)$ is equal to Q. Therefore,

$$P(Q/T_j) = \frac{P(Q)}{P(T_j)} \qquad \text{..... 4.78}$$

The conditional probability of Q given T_j has occurred will not be unity but will significantly increase in value from its *a priori* value. For this case, there may be no restriction of time to find the cause of the alarm and to take remedial actions.

4.7.4.2 : Conditional Probability of the Consequences of a Causal Event

Suppose that T can be caused by A and B occurring together. The Boolean expression for T is:

$$T = A \wedge B \qquad \text{....4.79}$$

The conditional probability of occurrence of A given T has occurred is 1.

Suppose that there is another event S that can be caused by A or C. The Boolean expression for S is:

$$S = A \vee C \qquad \text{....4.80}$$

The intersection of T and S is:

$$T \wedge S = (A \wedge B) \wedge (A \vee C) = A \wedge B \qquad .....4.81$$

The conditional probability of occurrence of S given T has occurred is:

$$P(S/T) = \frac{P(T \wedge S)}{P(T)} = 1 \qquad .....4.82$$

Equation 4.82 shows that S must have occurred, although it is not a cause or a consequence of T. This is due to the occurrence of A, common to both S and T.

Suppose that there is an event R that has inputs A and D via an **AND** gate. The Boolean expression for R is:

$$R = A \wedge D \qquad .....4.83$$

The intersection of T and R is given as:

$$T \wedge R = (A \wedge B) \wedge (A \wedge D) = A \wedge B \wedge D \qquad .....4.84$$

The conditional probability of R given T has occurred is:

$$P(R/T) = \frac{P(A \wedge B \wedge D)}{P(A \wedge B)} \qquad .....4.85$$

For this case, the occurrence of R depends on A and D occurring together. Since A has occurred to cause T to occur, it will influence the outcome of R. This is reflected in equation 4.85 which shows that the *posteriori* probability of R given that T has occurred significantly increases in value from its *a priori* probability. In fact, if $(A \wedge B)$ is independent from D, then the *posteriori* probability of R given that T has occurred is equal to the *a priori* probability of D.

Consider another case where the occurred event is caused by the occurrence of one of its causes. Suppose an event Q can be caused by the occurrence of A or E. The Boolean expression for Q is:

$$Q = A \vee E \qquad \text{..... 4.86}$$

The intersections of Q and S, and Q and R are:

$$Q \wedge S = (A \vee E) \wedge (A \vee C) = A \vee (C \vee E) \qquad \text{.....4.87}$$

$$Q \wedge R = (A \vee E) \wedge (A \wedge D) = A \wedge D \qquad \text{....4.88}$$

The conditional probability of S given Q has occurred is:

$$P(S/Q) = \frac{P\{A \vee (C \wedge E)\}}{P(A \vee E)} \qquad \text{.....4.89}$$

The conditional probability of R given Q has occurred is:

$$P(R/Q) = \frac{P(A \wedge D)}{P(A \vee E)} = \frac{P(R)}{P(Q)} \qquad \text{.....4.90}$$

Equations 4.89 and 4.90 show that for both cases where S and R which are consequences of a cause of the occurred event Q, their posteriori probabilities significantly increase from their *a priori* values.

4.7.5 : Discussion and Conclusion

In some cases, the *posteriori* probability of an event in a fault tree can be determined very rapidly without involving any Boolean manipulation. Such cases are:

1) when a causal event is connected to the alarm event via an **OR** gate or through a chain of other events via a series of **OR** and **DIR** gates, its *posteriori* probability is determined by dividing its *a priori* probability with the *a priori* probability of the alarm event.

2) when a causal event is connected to the alarm event via an **AND** or a **DIR** gate, or through a chain of other events via a series of **AND** and **DIR** gates, its *posteriori* probability is 1.

3) when a consequent event has the alarm event as an input via a **DIR** gate or an **OR** gate, or through a chain of other events via a series of **DIR** and **OR** gates, its *posteriori* probability is 1.

4) when a consequent event has the alarm event as an input with several other events via an **AND** gate; or through a chain of other events via a series of **AND** and **DIR** gates, the *posteriori* probability for that event is obtained by dividing its *a priori* probability by the *a priori* probability of the alarm event.

5) when an event which is neither a consequence nor a cause of the alarm event; but is one of several symptoms of a causal

event whose *posteriori* probability value is 1; and is connected to the causal event either by a **DIR** or an **OR** gate, or through a chain of several events via a combination of **OR** and **DIR** gates, then the *posteriori* probability for that event is 1.

In other cases, the probability of the intersection between the alarm event and the event whose *posteriori* probability that is to be evaluated, has to be determined first. The result obtained when divided by the *a priori* probability of the alarm event gives the *posteriori* probability for that event. The above method has to be applied when determining the posteriori probability of:

1) a causal event which is connected to the alarm event through a chain of other events via a combination of **AND** and **OR** gates.

2) a consequent event which is not a direct symptom of the alarm event, but having the alarm event in combination with other events as inputs via mixtures of **AND**, **OR** and **DIR** gates.

3) an event which is neither a consequence nor a cause of the alarm event; but is one of several symptoms of a causal event of the alarm event; and is connected to the causal event by an **AND** gate, or through a chain of several events via a combination of **AND**, **OR** and **DIR** gates.

The determination of the probability of the intersection of the alarm event and the event whose *posteriori* probability is to be evaluated can be done using the **TDPP** algorithm described earlier. For this case, **TDPP** will operate with an empty **S1** and **S2** containing the alarm event, and the causal or consequent event. However, **TDPP** has its drawbacks discussed earlier. Also, since the *a priori* probability of every event in the fault tree data file has been calculated, there is no need for the preliminary requirement of **TDPP** that only primary events must be in the sets **S1** and **S2** before actually calculating the probability. **TDPP** can be modified for the purpose of evaluating *posteriori* probabilities such that when an s-independent event is encountered in the recursion, its *a priori* probability is directly obtained from the fault tree data file.

The computer program for evaluating the *posteriori* probabilities of events in a fault tree given that an event has occurred for cases discussed above is described in chapter 5. The algorithm for evaluating the probability of intersection of events based on **TDPP** is also described in chapter 5.

CHAPTER FIVE

5. THE DIAGNOSIS PACKAGE

5.1 : Introduction

The analysis discussed in chapters 3 and 4 has been built into a diagnosis package. The package consists of a coloured fault tree display on the screen showing the alarm event, its likely causes and consequences. The *a priori* and *posteriori* probabilities also can be displayed. This helps the operator in tracing the most probable line of events that cause the alarm. The package has been written in Microsoft Quickbasic and runs on widely available IBM compatible computers using PC-DOS or MS-DOS operating system. A requirement of the algorithms is recursion and Quickbasic supports this. Some of the features of Microsoft Quickbasic are described in Appendix D.

Data for the display are obtained from hazard and operability study records in the form of the cause and symptom equations discussed in chapter 3. The cause and symptom equations contain information which must be stored in a form which can be easily and quickly accessed by the package. This is done by translating the cause and symptom equations into a data structure and storing this in a random access file. The file contains all the relevant information about each event, such as the type of event; its causes; its consequences; its *a priori* probability and its *posteriori* probability due to the alarm event. Each event can either be a primary event or an event with one or more inputs via an **OR** or an **AND** gate. The cause and consequence information contains pointers to the data about other events involved. The diagnosis package has only to search through the file once for the name of the alarm event and then by using the pointers, all the information required can easily be obtained without doing any more searches.

The PC-DOS/MS-DOS filing system uses filenames of up to 8 characters with a 3 character extension separated by a full stop. The extension is normally used to indicate the type of information in a file. This convention has been extended in this work to provide a consistent filename system. For example, text files are often given the extension **.TXT**, e.g. **EXAMPLE.TXT**. This is used for cause and symptom equations files which are prepared by hand.

5.2 : The Data Preparation Program - TRANSLAT

The data preparation program is a separate part of the diagnosis package. The program is called **TRANSLAT**. It consists of three stages which are:

1. Translation of a text file containing the cause and symptom equations into a fault tree data structure to be stored in a random access file. This is the fault tree data file and is given a name with the extension **.COD**, e.g. **EXAMPLE.COD**.

2. Creation of a primary events file which stores for each event, the set of all primary events that could cause that event. The

primary events file is also a random access file. This is done for the purpose of checking whether events are s-independent when calculating probabilities. The file will have a name with the extension **.PRI**, e.g. **EXAMPLE.PRI**.

3. Input of *a priori* probability data for all primary events which are stored in the **.COD** file.

4. Calculation of the *a priori* probability of every event in the data file. These are stored in the **.COD** file.

5.2.1 : Fault Tree Data File Structure

The structure of the fault tree data file developed by the author is a fixed record length database management system. The file consists of several records. Each event is assigned a unique record number or address. In each record, there are several fields containing all the relevant information about the event. Table 5.1 indicates how each record is divided into its various fields.

As an illustration, consider an arbitrary cause equation and its fault tree in figure 4.1. Part of the data file containing the information about the cause equation will be as shown in figure 5.1.

Although the number of fields reserved for storing the addresses of the branches of an event is 4, it does not mean that the number of branches connected by the same gate on the right hand

Table 5.1 Fault Tree Data Structure

Field Name	Field Type	No. of Bytes	Function
e$	Character	20	To store name of event.
g$	Character	3	To store type of gate or event according to the following criterion: 1. If **e$** is a primary event, **g$** will contain **PRI**. 2. If **e$** has only one branch, **g$** will contain **DIR.** 3. If **e$** has several branches connected by an AND gate, **g$** will contain **AND.** 4. If **e$** has several branches connected by an OR gate obtained from a cause equation, **g$** will contain **ORC** . 5. If **e$** has several branches connected by an OR gate obtained from several symptom equations, **g$** will contain **ORS**.
bc$	Numeric, integer	2	To store the number of branches connected to **e$**.
cau$(1)	Numeric integer	2	To store the address, i.e. record numbers of the branches connected to the event in **e$**. These fields are called the branch address fields.
cau$(2)		2	
cau$(3)		2	
cau$(4)		2	
bq$	Numeric, integer	2	To store the number of consequences of event in **e$**.
con$(1)	Numeric, integer	2	To store the address of the consequences of the event in **e$**. These fields are called the consequence address fields.
con$(2)		2	
con$(3)		2	
con$(4)		2	
pr$	Numeric real	4	To store the a priori probability of event in **e$**.
cpr$	Numeric, real	4	To store the conditional probability of event in **e$**.

Rec.	e$	g$	bc$	cau$				bq$	con$				. . . .
no.				(1)	(2)	(3)	(4)		(1)	(2)	(3)	(4)	
⋮													
10	T	ORC	3	11	12	13	0	0	0	0	0	0	. . . .
11	A	PRI	0	0	0	0	0	1	10	0	0	0	. . . .
12	B	PRI	0	0	0	0	0	1	10	0	0	0	. . . .
13	C	PRI	0	0	0	0	0	1	10	0	0	0	. . . .
⋮													

Figure 5.1 : Part of data file for fault tree of figure 4.1.

Rec	e$	g$	bc$	cau$				. . . .
no.				(1)	(2)	(3)	(4)	
⋮								
30	A	ORC	8	31	32	33	35	. . . .
31	B	PRI	0	0	0	0	0	. . . .
32	C	PRI	0	0	0	0	0	. . . .
33	D	PRI	0	0	0	0	0	. . . .
34	E	PRI	0	0	0	0	0	. . . .
35	&1	CON	4	34	36	37	39	. . . .
36	F	PRI	0	0	0	0	0	. . . .
37	G	PRI	0	0	0	0	0	. . . .
38	H	PRI	0	0	0	0	0	. . . .
39	&2	CON	2	38	40	0	0	. . . .
40	J	PRI	0	0	0	0	0	. . . .
⋮								

Figure 5.2 : Part of data file containing information about equation 5.1

side of the cause equation is limited to 4. The data file is structured in such a way that any number of branches for each event can be handled. As an illustration, consider the following arbitrary cause equation:

$$A = B + C + D + E + F + G + H + J \qquad \text{.....} \; 5.1$$

Figure 5.2 shows how the cause equation is translated into the data file.

When a record has the number of branches field, **bc$**, containing a value greater than 4, the first three branch address fields contain the addresses of three of the branches of that record, and **cau$(4)** will contain the address of a unique dummy event known as the continuation event. In the record of this continuation event, the branch address fields will contain the addresses of the rest of the branches of the above record. The name of a continuation event usually start with the character **&** followed by a number. The gate type field, **g$**, will contain the character string **CON**. If in the record of the continuation event, **bc$** contains the value 4, **cau$(4)** will either contain an address of another continuation event or a named event. This method of assigning dummy continuation events is also applied if an event has more than **4** consequences.

5.2.2 : Translation of Cause and Symptom Equations

A cause equation may have mixtures of **OR** and **AND** gates and events within parenthesis in order to state the logical combination of causes of the top event. For example, consider the system of tanks to supply fluid at a constant rate to another unit operation as shown in figure 5.3. The valve V3 is open during normal operation to relieve pressure on the control control CV1.

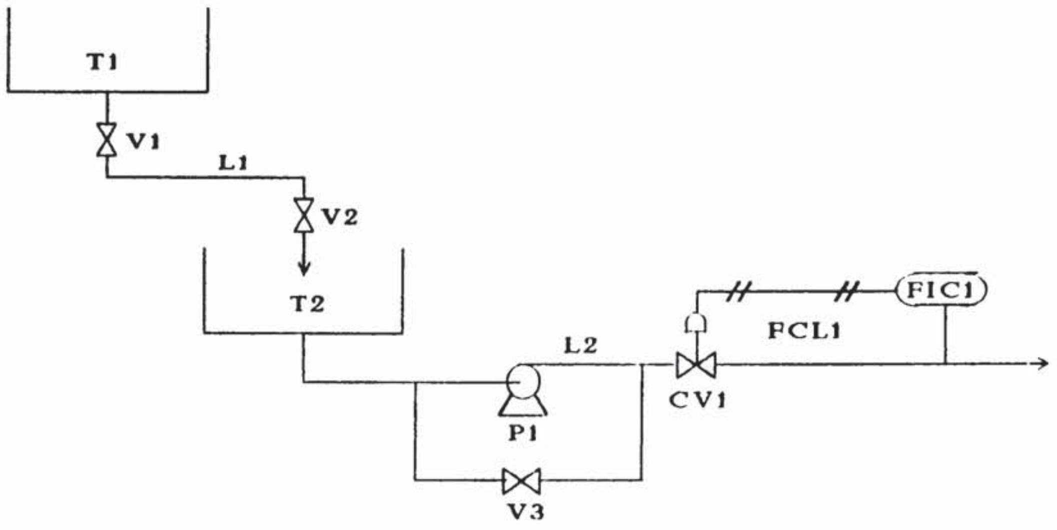


Figure 5.3 : Feed Tank System

No flow in the feed line **L2** can be shown to have the following cause equation:

L2(11)=T2(41)*(T1(41)+V1(0)+L1(0))+P1(0)*V2(0)+FCL1(0) 5.2

Equation 5.2 shows that the top event **L2(11)** has 3 branches as inputs through an **OR** gate which are:

1. T2(41)*(T1(41)+V1(0)+L1(0)

2. P1(0)*V2(0)

3. FCL1(0)

The first two branches are considered as unnamed branches because each has further inputs through an **AND** gate. The third branch which has a named event is considered as a named branch. For branch number 1, it has two inputs through an **AND** gate which are a named branch, **T2(41)**, and an unnamed branch consisting of 3 inputs through an **OR** gate.

Equation 5.2 may be written in any order but having the same logical combination. For example:

L2(11)=(T1(41)+V1(0)+L1(0))*T2(41)+FCL1(0)+P1(0)*V2(0) 5.3

or

L2(11)=P1(0)*V2(0)+(T1(41)+V1(0)+L1(0))*T2(41)+FCL1(0) 5.4

In order to be consistent in translating cause equations with unnamed branches, each unnamed branch is assigned a unique dummy event. The dummy event is given a name starting with the character * followed by a number.

Unnamed branches can appear in a cause equation in two ways. The first way is when a group of events or branches are connected by **OR** gates within parenthesis. The second way is when a group of events or branches are connected by **AND** gates but not within parenthesis and the inputs to the top event is via an **OR** gate. The program for translating cause equations into the data file can recognise unnamed branches in whatever order they are written such as shown by equations 5.2, 5.3 and 5.4.

The logic of the program for translating the cause and symptom equations to be stored into the fault tree data file is as follows:

1. Read in an equation from the text file.

2. Distinguish whether equation is a cause or a symptom equation. This is done by searching for the character "=" or "-" in the equation.

3a. If the the character "=" is found, subroutine **CAUSE** is called to translate the cause equation into the data file.

3b. If the character "-" is found instead, subroutine **SYMPTOM** is called to translate the symptom equation into the data file.

4. Repeat steps 1 to 3 until end of text file is reached.

5.2.2.1 : Description of Subroutine CAUSE

Subroutine **CAUSE** handles the translation process for cause equations. A flowchart is given in figure 5.4. When **CAUSE** is called, a search pointer, **i**, is at the position where the character "=" is located. The string of characters to the left of this pointer constitute the event name forming the top event of the cause equation. An address, i.e. record number in the data file is assigned to this event. This is done by the subroutine **ASSIGN**.

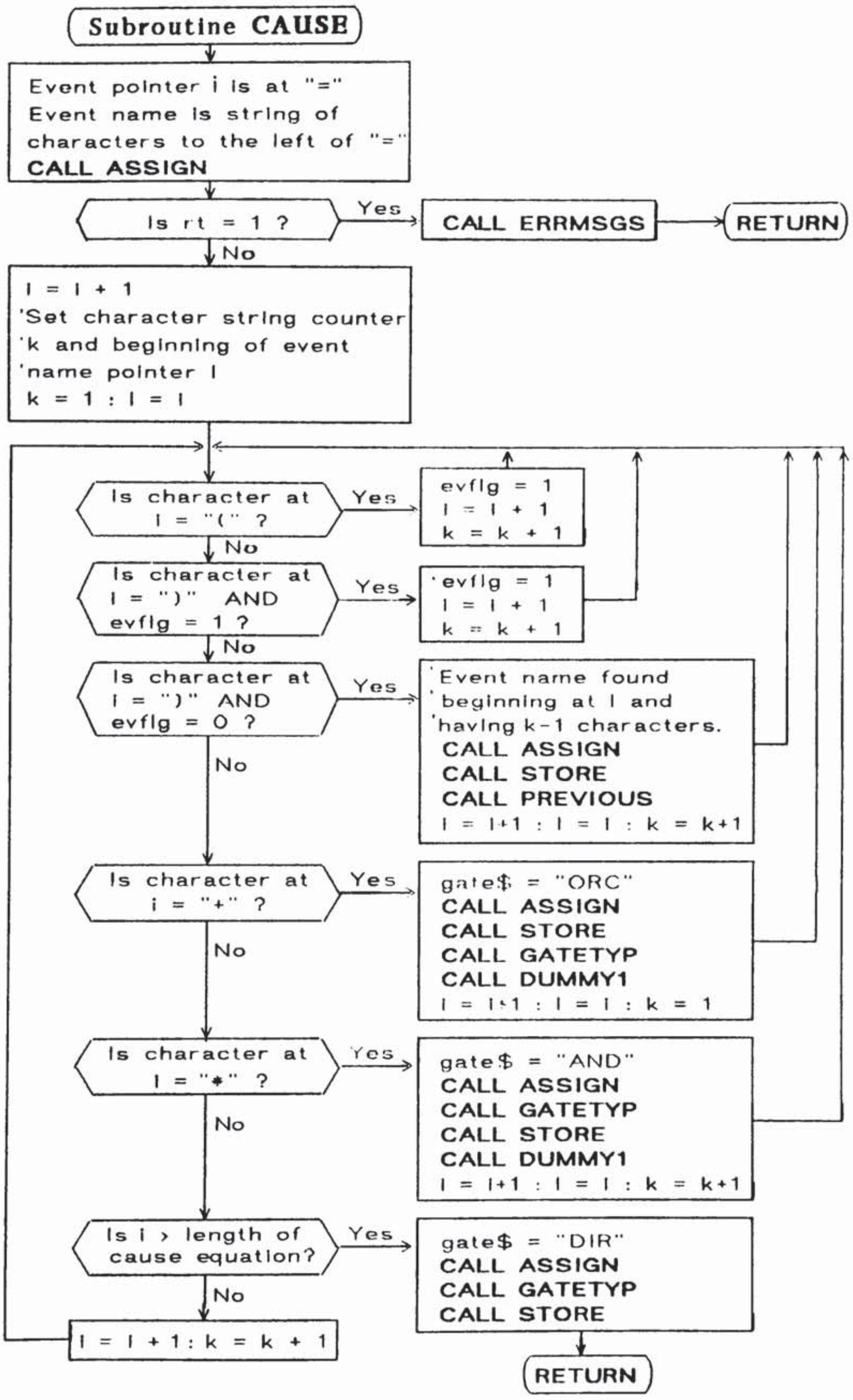


Figure 5.4 : Flowchart of Subroutine CAUSE

Subroutine **ASSIGN** includes an error checking procedure to determine if the top event of the cause equation has already been processed. This is done by determining whether or not the event has already been assigned an address. If the determination proves positive, then the field storing the gate type is checked. If **g$** contains other than **PRI**, it implies that an equation having the same top event had already been processed and that there are errors in the list of cause and symptom equations. An error flag, **rt**, is set and passed back to **CAUSE** so that the cause equation under process is not translated. Subroutine **ERRMSGS** is then called to tell the user about the error and where in the text file the error has occurred. Control of the program is returned to **TRANSLAT** so that the next equation in the list is processed.

If there is no error, **ASSIGN** returns the address of the top event to **CAUSE** stored in the variable **ca**. The value in **ca** is then stored in an integer variable **tp**. **CAUSE** then proceeds to search for event names that appear on the right hand side of the cause equation by scanning every character from left to right using increments of the search pointer **i**.

To check whether or not the cause equation has unnamed branches due to events within parenthesis, subroutine **DUMMY1** is called. **DUMMY1** first checks whether the character at the search pointer **i** is an open parenthesis, "(". If the character at **i** is not the open parenthesis, **DUMMY1** will return to **CAUSE** with no change in any of the variables.

If however "(" is found at **i**, this will indicate that an unnamed branch has been detected. A dummy event is created to represent the unnamed branch. An address in the data file is then assigned to the dummy event. The address of the top event stored in **tp** is saved temporarily in an array **pg(g, 1)** which acts like a last in, first out, stack. The index **g** in the array acts as an indicator to show the depth of nested parenthesis found in the cause equation. Every time a dummy event is created, the value of **g** is incremented by 1 before the contents of **tp** is stored in **pg(g, 1)**. Then the address of the newly created dummy event is stored in **tp**. This procedure is repeated for every subsequent open parenthesis found. When **DUMMY1** returns to **CAUSE**, the search pointer, **i**, will be at the last "(" found and the top event which **CAUSE** will refer to will be the last dummy event created whose address is stored in **tp**.

Event names used in coding cause and symptom equations may have parentheses which contain the codes for the guide words and property words. **CAUSE** is able to distinguish these parentheses which are part of the event name and those that require dummy events to be created. This is done by setting a flag, **evflg**, to 1 when the character "(" is found after **DUMMY1** has been called. When the character ")" is found and **evflg** is 1, **evflg** is set to 0. In this way, parentheses which are part of the event name are sifted out.

However if the character at **i** is a close parenthesis, ")", and **evflg** equal to 0, this will indicate that reference to a dummy event as the top event is about to end. After processing the event name associated with the above discovery, further processing will have to be referred to the previous top event whose address is stored in **pg(g, 1)**. The procedure for getting the previous top event address is done by calling subroutine **PREVIOUS**.

In subroutine **PREVIOUS**, the address in **pg(g, 1)** is stored in **tp** and **g** is decremented by 1. The search pointer, **i**, is then incremented by 1. If the character at **i** is also a ")", the above process is repeated. If the character at **i** is "*" or "+" instead, then the variable **gate$** is assigned the string **AND** or **ORC** respectively. **PREVIOUS** then checks the gate type stored in **g$** at the new top event address **tp**. If the content of **g$** is the same as **gate$**, **PREVIOUS** will return back to **CAUSE**. If however the content of **g$** is **ORC** and **gate$** is **AND**, subroutine **ANDGATE1** is called, otherwise if **g$** contains **AND** and **gate$** contains **ORC**, subroutine **GATETYP** is called instead.

The function of subroutine **ANDGATE1** is to create a dummy event since the gate found at **i** is an **AND** gate whereas the gate type field of the present top event at address **tp** contains **ORC**. The content of the last branch address field that had been filled at record number **tp** is replaced by the address of the dummy event created. The value that has been replaced is then placed in the first branch address field at the record number of the dummy event. The content of **tp** is then saved in the stack array at **pg(g, 2)**. A flag, **pf(g)**, is set to 1 to indicate that a dummy event has been created to represent a branch due to combination of events not within parentheses. When control of the program is returned to **CAUSES**, **tp** will contain the address of the dummy event which will be referred to when subsequent event names are found.

Subroutine **GATETYP** is called to replace the contents of the field **g$** at record number **tp** by the contents of **gate$** only if **g$** contains the string **PRI**. If **g$** contains the same character string as **gate$**, control of the program returns to its caller without anything being done. However, if **g$** is not the same as **gate$**, **GATETYP** will either call subroutine **ORGATE** if **g$** contains the string **AND** or subrourtine **ANDGATE2** if **g$** contains **ORC**.

The action in subroutine **ORGATE** depends on the contents of the flag **pf(g)**. If **pf(g)** is 1, **ORGATE** will return to its caller with **tp** containing the address stored in **pg(g, 2)** and **pf(g)** reset to 0. However if **pf(g)** is 0, a dummy event is created which will replace the top event name in **e$** at record number **tp.** The top event is then assigned a new address which will be stored in **tp**. All the information regarding the consequences of the top event is also transfered to the new record number. When control of the program is returned to **CAUSE**, the same top event name will still be referred but at the new address stored in **tp**.

The purpose of calling subroutine **ANDGATE2** is to create a dummy event whose inputs is via an **AND** gate. First the contents of **tp** is saved in the stack array at **pg(g, 2)** and the flag **pf(g)** is set to 1. A dummy event is then created whose address will be stored in the next unfilled branch address field at record number **tp**. When control of the program is returned to **CAUSE**, contents of **tp** will be the address of the dummy event created.

When a causal event has been found and assigned an address in the data file by the subroutine **ASSIGN**, subroutine **STORE** is called to store the contents of **ca** in a branch address field at the record number **tp**. If all the branch address fields had been filled, **STORE** will call subroutine **EXPCAUS** to expand the number branch of address fields by creating a continuation event. **STORE** also stores the consequence of the event found by storing the content of **tp** in a consequence address field at the record number **ca**. If all the consequence address fields had been filled, subroutine **EXPCONS** will be called to create a continuation event for expanding the number of consequence address fields.

As an illustration, consider equation 5.4. When the search pointer is at the character "=", the top event name L2(11) is recognised. Figure 5.5 shows the state of the data file at various stages of the search pointer.

5.2.2.2 : Description of Subroutine SYMPTOM

Subroutine **SYMPTOM** handles the translation process for symptom equations. When **SYMPTOM** is called, the search pointer **l** is located at the character "-". The event name found at this stage is the causal event of the symptom equation. Subsequent searches for events on the right hand side of the equation that are the symptoms of the causal event is done by looking for the character "*". Subroutines **ASSIGN** and **STORE** described previously are used by **SYMPTOM** to get or assign an address of each event found and to store the address of

i
↓
L2(11)=P1(O)*V2(O)+(T1(41)+V1(O)+L1(O))*T2(41)+FCL1(O)

Rec.	e$	g$	bc$	cau$				bq$	con$				. . .
no.				(1)	(2)	(3)	(4)		(1)	(2)	(3)	(4)	
1	L2(11)	PRI	O	O	O	O	O	O	O	O	O	O	. . . . .

(i)

---

i
↓
L2(11)=P1(O)*V2(O)+(T1(41)+V1(O)+L1(O))*T2(41)+FCL1(O)

Rec.	e$	g$	bc$	cau$				bq$	con$				. . .
no.				(1)	(2)	(3)	(4)		(1)	(2)	(3)	(4)	
1	L2(11)	AND	1	2	O	O	O	O	O	O	O	O	. . . . .
2	P1(O)	PRI	O	O	O	O	O	1	1	O	O	O	

(ii)

---

i
↓
L2(11)=P1(O)*V2(O)+(T1(41)+V1(O)+L1(O))*T2(41)+FCL1(O)

1. Before calling subroutine GATETYP

Rec.	e$	g$	bc$	cau$				bq$	con$				. . .
no.				(1)	(2)	(3)	(4)		(1)	(2)	(3)	(4)	
1	L2(11)	AND	2	2	3	O	O	O	O	O	O	O	. . . . .
2	P1(O)	PRI	O	O	O	O	O	1	1	O	O	O	
3	V2(O	PRI	O	O	O	O	O	1	1	O	O	O	

2. After calling subroutine GATETYP

Rec.	e$	g$	bc$	cau$				bq$	con$				. . .
no.				(1)	(2)	(3)	(4)		(1)	(2)	(3)	(4)	
1	# 1	AND	2	2	3	O	O	1	4	O	O	O	. . . . .
2	P1(O)	PRI	O	O	O	O	O	1	1	O	O	O	
3	V2(O)	PRI	O	O	O	O	O	1	1	O	O	O	
4	L2(11)	ORC	1	1	O	O	O	O	O	O	O	O	

(iii)

---

i
↓
L2(11)=P1(O)*V2(O)+(T1(41)+V1(O)+L1(O))*T2(41)+FCL1(O)

Rec.	e$	g$	bc$	cau$				bq$	con$				. . .
no.				(1)	(2)	(3)	(4)		(1)	(2)	(3)	(4)	
1	# 1	AND	2	2	3	O	O	1	4	O	O	O	. . . . .
2	P1(O)	PRI	O	O	O	O	O	1	1	O	O	O	
3	V2(O)	PRI	O	O	O	O	O	1	1	O	O	O	
4	L2(11)	ORC	2	1	5	O	O	O	O	O	O	O	
5	# 2	PRI	O	O	O	O	O	1	4	O	O	O	

(iv)

Figure 5.5 : Various stages of translating equation 5.4 into data file.

L2(11)=P1(0)*V2(0)+(T1(41)+V1(0)+L1(0)$\overset{\overset{i}{\downarrow}}{)}$*T2(41)+FCL1(0)

Rec. no.	e$	g$	bc$	cau$ (1)	(2)	(3)	(4)	bq$	con$ (1)	(2)	(3)	(4	. . .
1	# 1	AND	2	2	3	0	0	1	4	0	0	0	......
2	P1(0)	PRI	0	0	0	0	0	1	1	0	0	0	
3	V2(0)	PRI	0	0	0	0	0	1	1	0	0	0	
4	L2(11)	OR	2	1	5	0	0	0	0	0	0	0	
5	# 2	OR	3	6	7	8	0	1	4	0	0	0	
6	T1(41)	PRI	0	0	0	0	0	1	5	0	0	0	
7	V1(0)	PRI	0	0	0	0	0	1	5	0	0	0	
8	L1(0)	PRI	0	0	0	0	0	1	5	0	0	0	

(v)

L2(11)=P1(0)*V2(0)+(T1(41)+V1(0)+L1(0))$\overset{\overset{i}{\downarrow}}{*}$T2(41)+FCL1(0)

Rec no.	e$	g$	bc$	cau$ (1)	(2)	(3)	(4)	bq$	con$ (1)	(2)	(3)	(4	......
1	# 1	AND	2	2	3	0	0	1	4	0	0	0	......
2	P1(0)	PRI	0	0	0	0	0	1	1	0	0	0	
3	V2(0)	PRI	0	0	0	0	0	1	1	0	0	0	
4	L2(11)	ORC	2	1	9	0	0	0	0	0	0	0	
5	# 2	ORC	3	6	7	8	0	1	9	0	0	0	
6	T1(41)	PRI	0	0	0	0	0	1	5	0	0	0	
7	V1(0)	PRI	0	0	0	0	0	1	5	0	0	0	
8	L1(0)	PRI	0	0	0	0	0	1	5	0	0	0	
9	# 3	AND	1	5	0	0	0	1	4	0	0	0	

(vi)

L2(11)=P1(0)*V2(0)+(T1(41)+V1(0)+L1(0))*T2(41)$\overset{\overset{i}{\downarrow}}{+}$FCL1(0)

Rec no	e$	g$	bc$	cau$ (1)	(2)	(3)	(4)	bq$	con$ (1)	(2)	(3)	(4)	. . .
1	# 1	AND	2	2	3	0	0	1	4	0	0	0	....
2	P1(0)	PRI	0	0	0	0	0	1	1	0	0	0	
3	V2(0)	PRI	0	0	0	0	0	1	1	0	0	0	
4	L2(11)	ORC	2	1	9	0	0	0	0	0	0	0	
5	# 2	ORC	3	6	7	8	0	1	9	0	0	0	
6	T1(41)	PRI	0	0	0	0	0	1	5	0	0	0	
7	V1(0)	PRI	0	0	0	0	0	1	5	0	0	0	
8	L1(0)	PRI	0	0	0	0	0	1	5	0	0	0	
9	# 3	AND	2	5	10	0	0	1	4	0	0	0	
10	T2(41	PRI	0	0	0	0	0	1	9	0	0	0	

(vii)

Figure 5.5a (continuation of fig. 5.5)

i
↓
L2(11)=P1(0)*V2(0)+(T1(41)+V1(0)+L1(0))*T2(41)+FCL1(0)

Rec no.	e$	g$	bc$	cau$				bq$	con$				. . . .
				(1)	(2)	(3)	(4)		(1)	(2)	(3)	(4)	
1	# 1	AND	2	2	3	0	0	1	4	0	0	0	. . . . .
2	P1(0)	PRI	0	0	0	0	0	1	1	0	0	0	
3	V2(0)	PRI	0	0	0	0	0	1	1	0	0	0	
4	L2(11)	ORC	3	1	9	11	0	0	0	0	0	0	
5	# 2	ORC	3	6	7	8	0	1	9	0	0	0	
6	T1(41)	PRI	0	0	0	0	0	1	5	0	0	0	
7	V1(0)	PRI	0	0	0	0	0	1	5	0	0	0	
8	L1(0)	PRI	0	0	0	0	0	1	5	0	0	0	
9	# 3	AND	2	5	10	0	0	1	4	0	0	0	
10	T2(41)	PRI	0	0	0	0	0	1	9	0	0	0	
11	FCL1(0)	PRI	0	0	0	0	0	1	4	0	0	0	

(viii)

Figure 5.5 b(continuation of fig. 5.5)

the causal event at the appropriate branch address field at the record number of the symptom events found. Subroutine **GATETYP2** is called to store in the gate type field either the character string **DIR** if the symptom event has only one cause or **ORS** if the symptom has more than one cause. The string **ORS** is used to indicate that the causes of the event at the particular address comes from symptom equations. Figure 5.6 shows the flowchart of the subroutine **SYMPTOM**.

As as an illustration, consider the following arbitrary symptom equations.

A-N1*N2*N3*N4 ...... 5.5

B-N1*N2*N5 ...... 5.6

C-N4*N5 ..... 5.7

Figure 5.7 shows part of the data file that contains the information of the translated symptom equations.

5.2.3 : Primary Events File

Some of the secondary events in a fault tree may not be s–independent with each other because they have one or more primary events in common among the basic causes. A way to check the s-independence of events is by using sets of primary events. If the elements of the set of primary events of event A do not appear in the set of primary events of event B, then A an B is said to be

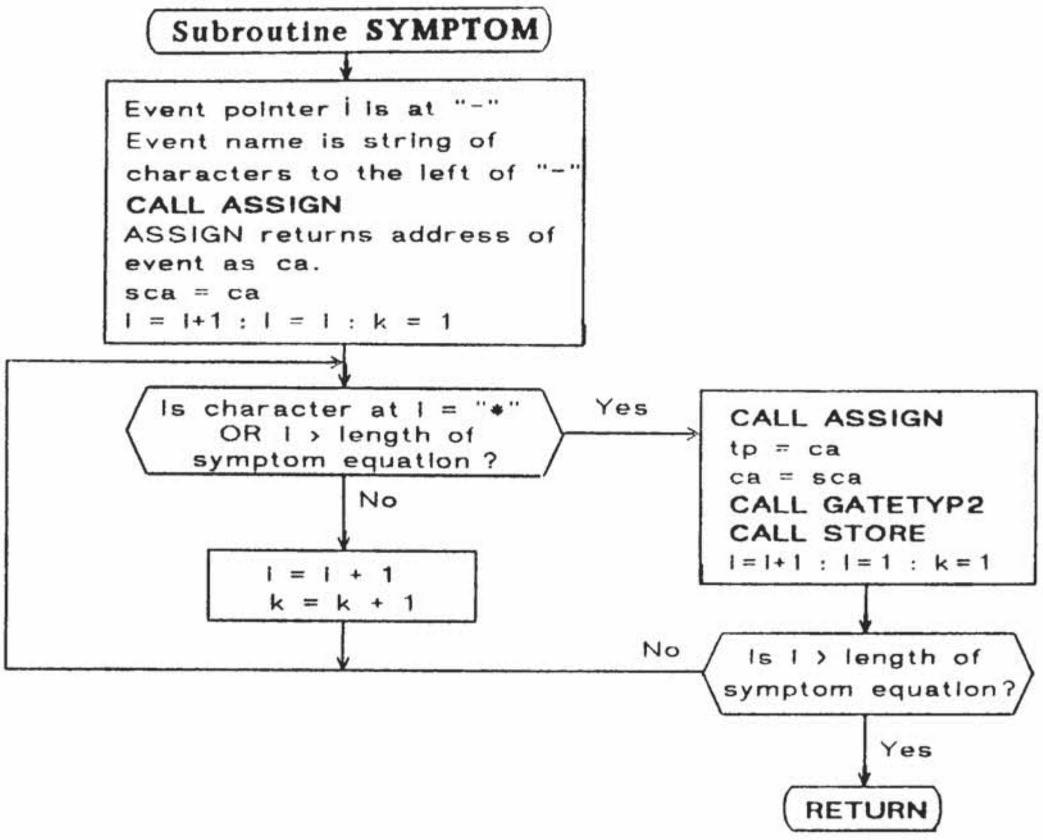

Subroutine SYMPTOM
Event pointer i is at "-"
Event name is string of characters to the left of "-"
CALL ASSIGN
ASSIGN returns address of event as ca.
sca = ca
i = i+1 : l = i : k = 1
Is character at i = "*" OR i > length of symptom equation ?
Yes
No
i = i + 1
k = k + 1
CALL ASSIGN
tp = ca
ca = sca
CALL GATETYP2
CALL STORE
i=i+1 : l=1 : k=1
No
Is i > length of symptom equation?
Yes
RETURN

Figure 5.6 : Flowchart of Subroutine **SYMPTOM**

Rec no.	e$	g$	bc$	cau$ (1)	(2)	(3)	4)	bq$	con$ (1)	(2)	(3)	(4	. . .
:													. . . . .
10	A	PRI	0	0	0	0	0	4	11	12	13	14	
11	N1	ORS	2	10	15	0	0	0	0	0	0	0	
12	N2	ORS	2	10	15	0	0	0	0	0	0	0	
13	N3	DIR	1	10	0	0	0	0	0	0	0	0	
14	N4	ORS	2	10	17	0	0	0	0	0	0	0	
15	B	PRI	0	0	0	0	0	3	11	12	16	0	
16	N5	ORS	2	15	17	0	0	0	0	0	0	0	
17	C	PRI	0	0	0	0	0	2	14	16	0	0	
:													

Figure 5.7 : Part of data file for symptom equations 5.5, 5,6 and 5.7

s-independent. The determination of s-independence of events is one of the requirements for implementing the fault tree probability evaluation algorithm which will be described later.

In this part of **TRANSLAT**, each set of primary events is obtained from the fault tree data file. Every record of the primary event file has 71 fields, each of 2 bytes long. The first field, **nu$** stores the number of primary events in the set and the next 70 fields from **pri$(1)** to **pri$(70)** store the addresses of the primary events in the set. The record number of the event in the primary events file correspond to the same record number in the fault tree data file. At the record number of a primary event, **nu$** will contain the value 1 and **pri$(1)** will contain its own address.

The subroutine to obtain the set of primary events is called **PRIMARY**. It is called after the translation process of the cause and symptom equations is over. The subroutine **PRIMARY** is recursive and can call itself. Suppose k is the address of an event which has n branches whose addresses are x_1, x_2, ..., x_n. Let SET[k], SET[x_1], SET[x_2], ... , SET[x_n] be the respective sets of primary events. The set of primary events of event at address k can be obtained recursively using the identity :

$$\mathrm{SET}[k] = \mathrm{SET}[x_1] \cup \mathrm{SET}[x_2] \cup \; . \; . \; . \; . \; . \; \cup \mathrm{SET}[x_n] \qquad ..... \; 5.8$$

The recursion terminates when k refers to an address of a primary event. The execution of this recursion is very fast and depends linearly on the number of events in the fault tree. For a single fault tree, the primary event file can be set up completely with one traversal.

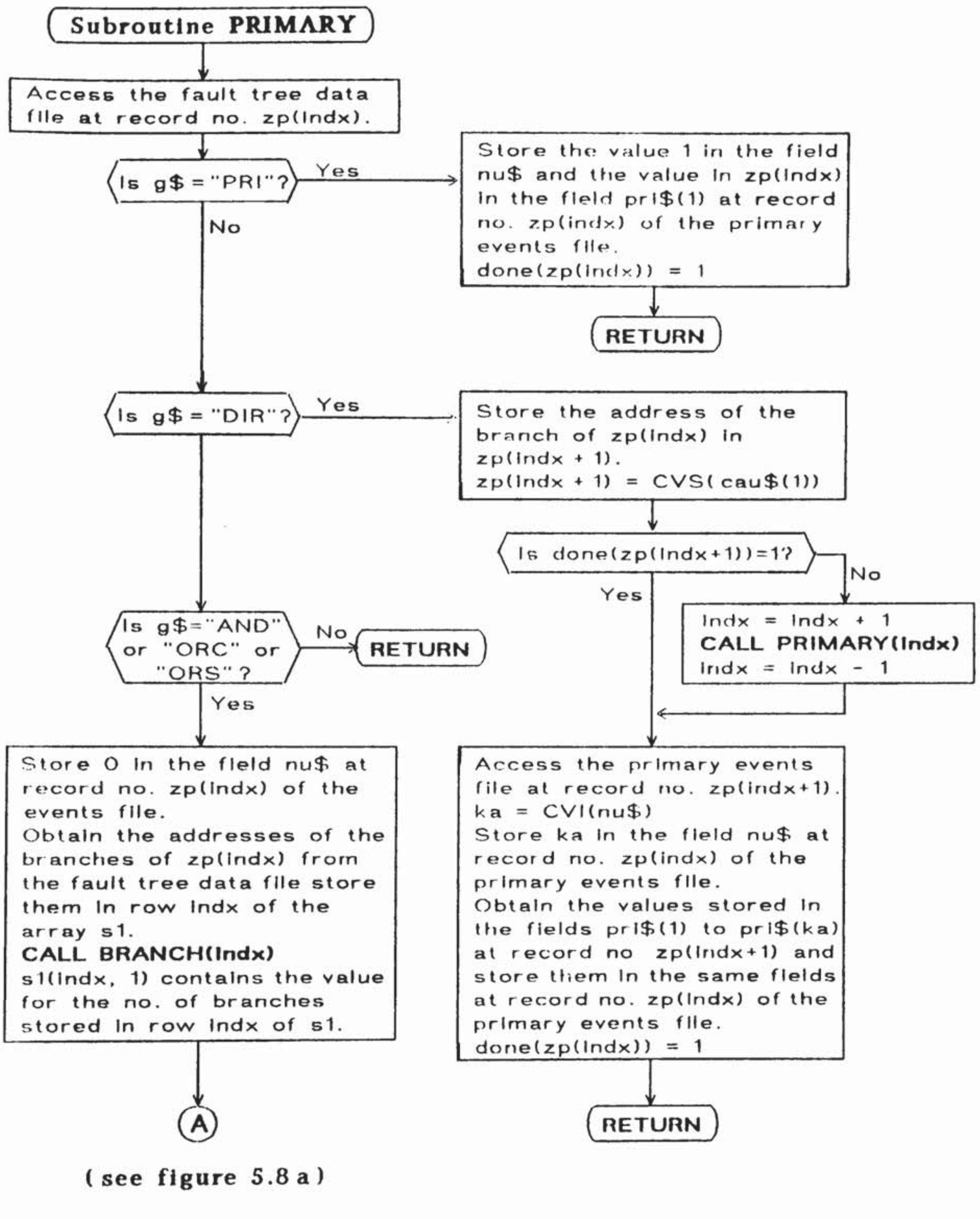
The flowchart of subroutine **PRIMARY** is shown in figure 5.8. When **PRIMARY** is called, the address of the event being processed is passed from the caller as **zp(indx)**. The index **indx** is used to track the depth of recursion being used. The array **done(k)** is used to flag that the event whose address is k has been processed. Whenever any other fault tree which has k as one of its branches, the recursion need not have to be done on that particular event. The set of primary events can be obtained at record number k already stored in the primary event file.

If the event has its input branches via an **OR** or **AND** gate, then **PRIMARY** will call subroutine **BRANCH** to get the address of the branches and store in a temporary two dimensional array **s1(indx, n)**. The first column of **s1(indx, n)** contain the number of branches of the event being processed. Columns 2 to n will hold the addresses of the branches. Then **PRIMARY** will recursively obtain the primary events of the event from each branch stored in **s1(indx, n)**.

5.2.4 : Probability Calculation

5.2.4.1 : Introduction

Once the fault tree data file and the primary events file have been created, the *a priori* probabilities of every non primary event then



Subroutine PRIMARY
Access the fault tree data file at record no. zp(Indx).
Is g$ = "PRI"?
Yes
No
Store the value 1 in the field nu$ and the value in zp(Indx) in the field pri$(1) at record no. zp(indx) of the primary events file.
done(zp(indx)) = 1
RETURN
Is g$ = "DIR"?
Yes
Store the address of the branch of zp(Indx) in zp(Indx + 1).
zp(Indx + 1) = CVS(cau$(1))
Is done(zp(Indx+1))=1?
No
Yes
Indx = Indx + 1
CALL PRIMARY(Indx)
Indx = Indx - 1
Is g$="AND" or "ORC" or "ORS"?
No
RETURN
Yes
Store 0 in the field nu$ at record no. zp(Indx) of the events file.
Obtain the addresses of the branches of zp(Indx) from the fault tree data file store them in row Indx of the array s1.
CALL BRANCH(Indx)
s1(Indx, 1) contains the value for the no. of branches stored in row Indx of s1.
Access the primary events file at record no. zp(indx+1).
ka = CVI(nu$)
Store ka in the field nu$ at record no. zp(indx) of the primary events file.
Obtain the values stored in the fields pri$(1) to pri$(ka) at record no zp(Indx+1) and store them in the same fields at record no. zp(Indx) of the primary events file.
done(zp(Indx)) = 1
RETURN
A
(see figure 5.8 a)

Figure 5.8 : Flowchart of Subroutine **PRIMARY**

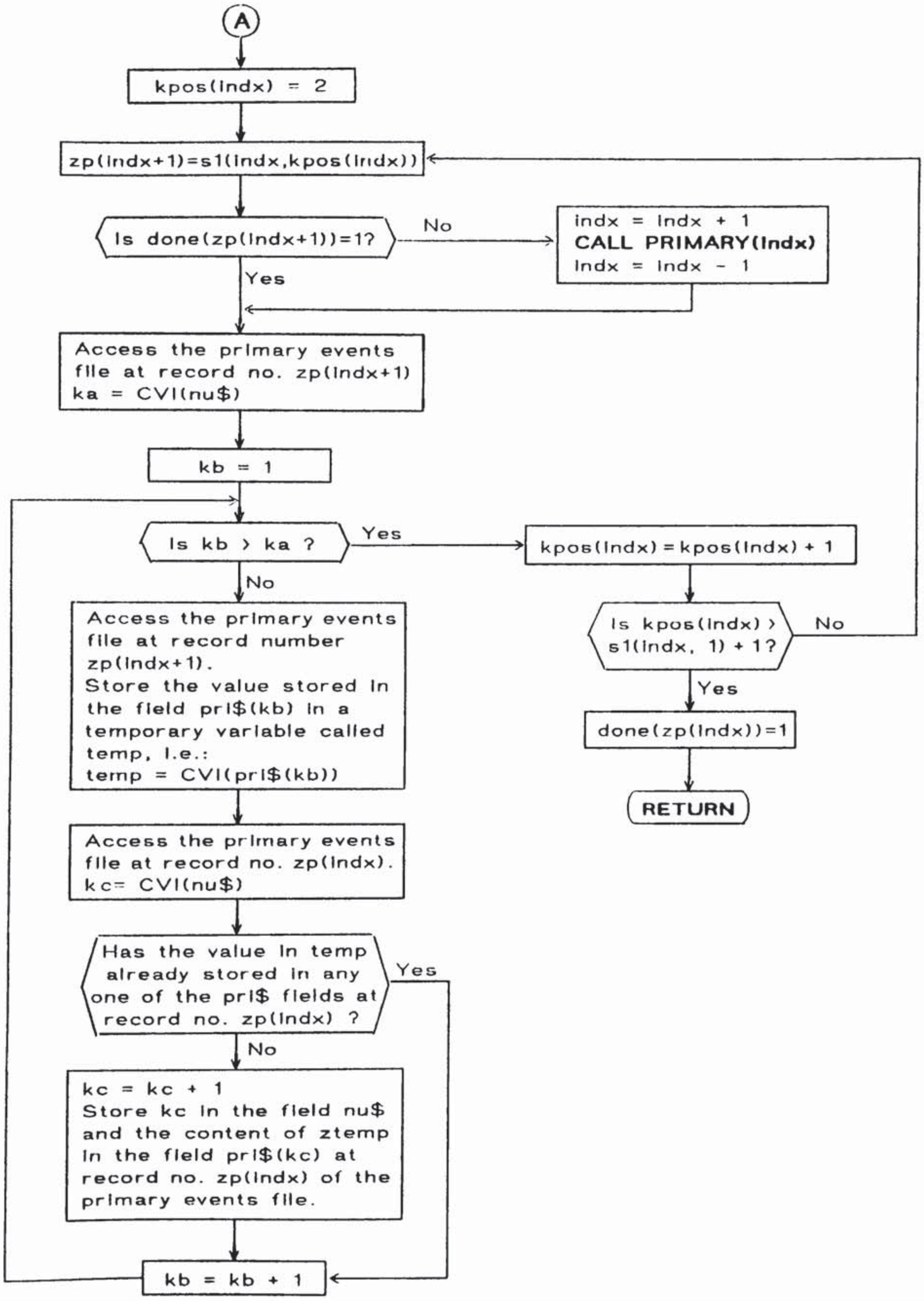

A
kpos(indx) = 2
zp(indx+1)=s1(indx,kpos(indx))
Is done(zp(indx+1))=1?
No
indx = indx + 1
CALL PRIMARY(indx)
indx = indx - 1
Yes
Access the primary events file at record no. zp(indx+1)
ka = CVI(nu$)
kb = 1
Is kb > ka ?
Yes
kpos(indx) = kpos(indx) + 1
No
Access the primary events file at record number zp(indx+1).
Store the value stored in the field pri$(kb) in a temporary variable called temp, i.e.:
temp = CVI(pri$(kb))
Is kpos(indx) > s1(indx, 1) + 1?
No
Yes
done(zp(indx))=1
RETURN
Access the primary events file at record no. zp(indx).
kc= CVI(nu$)
Has the value in temp already stored in any one of the pri$ fields at record no. zp(indx) ?
Yes
No
kc = kc + 1
Store kc in the field nu$ and the content of ztemp in the field pri$(kc) at record no. zp(indx) of the primary events file.
kb = kb + 1

Figure 5.8 a (continuation of fig. 5.8)

can be calculated from *a priori* data for primary events. The *a priori* probabilities are needed when calculating the conditional probabilities of the possible causes of an alarm event in the fault tree display part of the diagnosis package.

The probability calculation algorithm used is based on the top down recursive approach as was proposed by Page and Perry [110] in their development of **TDPP** . It is easily implemented with the use of Microsoft Quickbasic because the programming language supports recursion of subroutines, as required by the algorithm. However, **TDPP** depends only on the probability values of primary events as input data when evaluating the probability of the top event. This implies that if there are s-independent secondary events as branches of the top event, their probabilities are not evaluated during the recursion. Even if their probabilities had already been calculated, they are not being used. Thus every non primary event in the fault tree data file is treated as the top event when determining its probability. If the probability of a top event has been calculated, almost similar recursions would be repeated when evaluating the probabilities of its secondary events. **TDPP** can be modified so as to minimise the number of recursions or calls to the probability calculation subroutine whilst evaluating the probability of the top event of a fault tree and all its secondary events.

5.2.4.2 : The Improved Top Down Recursive Algorithm - TDRA

The improved top down recursive algorithm, **TDRA**, developed by the author of this present work is a modification to Page and Perry's algorithm [110] designed to eliminate redundant recursions when s-independent secondary events are encountered. The algorithm uses the same sets of events, **S1** and **S2** as used in **TDPP**, which contain the addresses of events, as defined in chapter 4.

The main subroutine for calculating the probability of an event is called **PROB**. A one dimensioned array called **done** is used as a flag to indicate whether or not the probability of an event has been calculated and saved in the fault tree data file. If **PROB** requires the probability value of a non primary event, say event **A**, as part of its calculation, the flag **done(A)** is first checked whether it had been set. If **done(A)** is set, **PROB** will obtain the probability value of **A** from the data file. Otherwise, a recursion of **PROB** is used to obtain the probability of **A**. The logical structure of **TDRA** is slightly different from **TDPP** to reduce the number of recursions. The preliminary simplification of getting the branches of the nodes in **S1** or **S2** until there are no more **OR** or **AND** gate nodes in the respective set is not done as the case for **TDPP**. **TDRA** is described below.

0.0 : Preliminary Inspection

The first stage in **PROB** is a preliminary inspection used only in the case when **S1** is not empty. It is to check whether **S1** $\cap$ **S2** is empty. If it is found that **S1** $\cap$ **S2** is not empty, a Boolean reduction is made by making **S1** empty and leaving **S2** intact. Otherwise if **S1** contains a single element, its content is added to **S2** and **S1** itself is made empty. After this preliminary inspection, there will be 2 cases, either **S1** empty or not empty. The case that applies determines the choice of recursion within **PROB** to be used.

1. : Evaluation for the case **S1** is empty.

1.1 : Case when **S2** is empty.

PROB returns to its caller with a probability value of 1.

1.2 : Case when **S2** contains one node.

Check if the node in **S2**, say **m**, is a primary event or the flag **done(m)** has been set equal to 2.

Within case 1.2 :

1.2.1 : Case when the probability data is available.

If either of the checks in 1.2 is positive, **PROB** returns to its caller with the probability of **m** obtained from the fault tree data file.

1.2.2 : Case when the probability data is not available.

Check the node **m** for the type of gate through which it has its input nodes. If **m** is an **AND** or **DIR** node, place its branches in **S2** and make **S1** empty. Otherwise, place the branches of **m** in **S1** and make **S2** empty. The probability evaluation of **m** is made by calling **PROB** which operates on the new sets **S1** and **S2**. After returning from this call, the flag **done(m)** is set equal to 2 to indicate that the probability of the event represented by node **m** has been calculated and saved in the data file. **PROB** then returns to its caller with the probability of node **m**.

1.3 : Case when **S2** contains more than one node.

A variable **ztemp** is first set equal to 1. Then determine if there is an s-independent node in **S2**.

Within case 1.3 :

1.3.1 : Case when an s-independent node is found in **S2** .

Let the s-independent node be **m**. Test to see if the probability of node **m** has already been evaluated.

Within case 1.3.1 :

1.3.1.1 : Case when probability of node **m** has already been evaluated.

Obtain the probability of node **m** from the fault tree data file.

1.3.1.2 : Case when the probability of node **m** has not been evaluated.

First, save the existing content of **S2**. Then examine the type of gate through which node **m** has its input. If the gate is an **AND** or a **DIR**, place the branches of node **m** in **S2** and make **S1** empty. Otherwise, the branches of node **m** are placed in **S1** and make **S2** empty. Evaluate the probability of node **m** by calling **PROB** to operate on the new **S1** and **S2**. When **PROB** returns from this call, save the probability of node **m** in the fault tree data file and set **done(m)** to 2. Then restore the saved values of **S2**.

After evaluation of case 1.3.1.1 or 1.3.1.2, execute the following identity:

ztemp = ztemp * P(m) 5.9

where **P(m)** is the probability of node **m**.

Then remove the node **m** from **S2.** If **S2** is empty after this operation, **PROB** then returns to its caller with the probability value equal to **ztemp**. Otherwise, test to find another s-independent node in **S2**. If there is another s-independent node in the reduced **S2**, repeat the procedure from step 1.3.1. However, if at some stage that no s-independent node can be found, proceed to step 1.3.2.

1.3.2 : Case when there is no s-independent event in **S2**.

Test to see if there are any **AND** and **DIR** nodes in **S2**. If there are such nodes, replace them by their branches.

Within case 1.3.2 :

1.3.2.1 : Case when replacements are made in **S2** .

Remove any repetition of nodes in **S2** and determine if there is an s-independent node in the new **S2**.

Within case 1.3.2.1 :

1.3.2.1.1 : Case when an s-independent node is found in the new **S2**.

Repeat the procedure from step 1.3.1.

1.3.2.1.2 : Case when there is no s-independent node in **S2**.

Repeat the procedure from step 1.3.2.

1.3.2.2 : Case when no replacements are made in **S2** .

Pick an **OR** gate node, say **n**, in **S2**. Place the branches of **n** in **S1**. Create a new **S2** containing all the nodes of the old **S2** except for the node **n**. Evaluate the probability by calling **PROB** to operate on the new **S1** and **S2**. Let the probability evaluated be **P(m)**. Evaluate the identity in equation 5.9 before returning to the caller of the present **PROB**.

2. : Evaluation for the case **S1** is not empty.

First initialise a set **EVSET** so that it is empty. Then determine if there is a node **m** in **S1** that is s-independent from the rest of the nodes in **S1** as well as from all the nodes in **S2.**

2.1 : Case when such a node is found in **S1**.

Add the node, say **m**, to the set **EVSET.** Then remove node **m** from the set **S1**. If **S1** contains 1 node after this operation, proceed to step 2.3. Otherwise, evaluate step 2.1.1.

Within case 2.1 :

2.1.1 : Determine if there is an s-independent node in **S1** which is also s-independent with all the nodes in **S2**. If such a node is found, repeat step 2.1. Otherwise, proceed to step 2.2.

2.2 : Case when no s-independent node is found in **S1**.

Test to see if there are any **OR** and **DIR** nodes in **S1**. If there are such nodes, replace them by their branches.

Within case 2.2 :

2.2.1 : Case when replacements are made in **S1**.

Remove any repetition of nodes in **S1**. Then determine if there is an s-independent node in **S1** which is also s-independent with all the nodes in **S2**. If such a node is found, repeat the procedures from step 2.1. Otherwise, repeat the procedures from step 2.2.

2.2.2 : Case when no replacements are made in **S1**.

Save the existing contents of **S1** and **S2**. Then proceed to step 2.3.

2.3 : Case when the procedures in step 2.1 and 2.2 are executed.

Within case 2.3:

2.3.1 : Case when **EVSET** is not empty.

(a) Call **PROB** operating on a new **S1** containing all the nodes from the saved **S1** and a new **S2** containing all the nodes from the saved **S2**. The probability value returned from this call is saved in the variable **ztemp1**.

(b) Call **PROB** operating on a new **S1** which is empty and a new **S2** containing all the nodes from the saved **S2**. The probability value returned from this call is saved in the variable **ztemp2**.

(c) Get a node, **m** from **EVSET**. The probability of the node **m** is obtained using the procedure as outlined in step 1.3.1.1 or 1.3.1.2. Let the probability of **m** be **P(m)**.

(d) The following identity is then evaluated :

$$\mathbf{ztemp1} = \mathbf{P(m)} * \left[\mathbf{ztemp2} - \mathbf{ztemp1}\right] + \mathbf{ztemp1} \qquad \ldots\ldots 5.10$$

The node **m** is removed from the set **EVSET**. If **EVSET** is not empty after this operation, the procedure from step (c) is repeated. When **EVSET** is empty, **PROB** returns to its caller with the probability value equal to **ztemp1**.

2.3.2 : Case when **EVSET** is empty

(a) Pick a node **m** from the saved **S1**. Call **PROB** operating on a new **S1** which is empty and a new **S2** containing all the nodes from the saved **S2** with additionally the node **m.** The probability value obtained from this call is saved in a variable **ztemp1**.

(b) Call **PROB** operating on a new **S1** containing all the elements of the saved **S1** except the node **m** and a new **S2** which contains all the nodes of the saved **S2**. The probability value obtained from this call is saved in the variable **ztemp2.**

(c) Call **PROB** operating on a new **S1** containing all the elements of the saved **S1** except the node **m** and a new **S2** which contains all the nodes of the saved **S2** with additionally the node **m**. The probability value obtained from this call is saved in the variable **ztemp3**.

When all the above three probability values are obtained, the probability value returned from the present call to **PROB** is given as:

zpv = ztemp1 + ztemp2 - ztemp3 .....5.11

5.2.4.3 : Implementation of TDRA

Microsoft Quickbasic does not support set operations as required by **TDPP** which uses PASCAL as its programming language. Instead, **TDRA** simulates the sets **S1** and **S2** which are passed as parameters in **TDPP** by using 2 two-dimensional arrays **s1** and **s2**, each of size 40 columns by 70 rows, as global variables. Each of **s1** and **s2** is used to store one set of data. In the first column of row **indx** of either **s1** or **s2**, i.e. **s1(indx, 1)** or **s2(indx, 1)**, contains the number of elements in the respective sets. The rest of the columns contain the addresses of events obtained from the fault tree data file. The index **indx**, is used to keep track on the depth of recursion and also to maintain that data in **s1** and **s2** are not lost during the recursive calls to **PROB**. The parameters that are passed whenever **PROB** is called are **indx** and **zp(indx)**. The probability value obtained by **PROB** after completing its function is returned as **zp(indx)**.

Several subroutines are used by **PROB** to help in deciding which case for recursion is to be applied. Subroutine **INTERSEC** is called to check the status of **s1** $\cap$ **s2**. **INTERSEC** will return a flag **true** equal to 1 if it found that **s1** $\cap$ **s2** is not empty. Subroutine **INDEPT** is used to search for an event in **s2** which is s-independent from the other events in **s2**. If the search is successful, **INDEPT** will return a flag **disjt** equal to 1 and the address of the event in the variable **indevt.** Subroutine **INDEPTOR** is used to search if there is an event in **s1** which is s-independent from the other events in **s1** as well s-independent from all the events in **s2**. If **INDEPTOR** finds such an event, it will return the flag **disjt** equal to 1 and the address of the event in the variable **indevt**. Both subroutines **INDEPT** and **INDEPTOR** use the primary events file in their search for an s-independent event in **s2** and **s1** respectively.

The other subroutines called by **PROB** are **REMREP** and **SIBLING**. Subroutine **REMREP** is used to remove repetitions in either set **s1** or **s2** after any replacement of the nodes by their branches. Thus **REMREP** ensures that every element in **s1** or **s2** appears only once in their respective sets. Subroutine **SIBLING** is used to get the addresses of the branches of a node from the fault tree data file which are to replace the node in either **s1** or **s2**.

The way **TDRA** is implemented is shown as a flow chart of the subroutine **PROB** in figure 5.9.

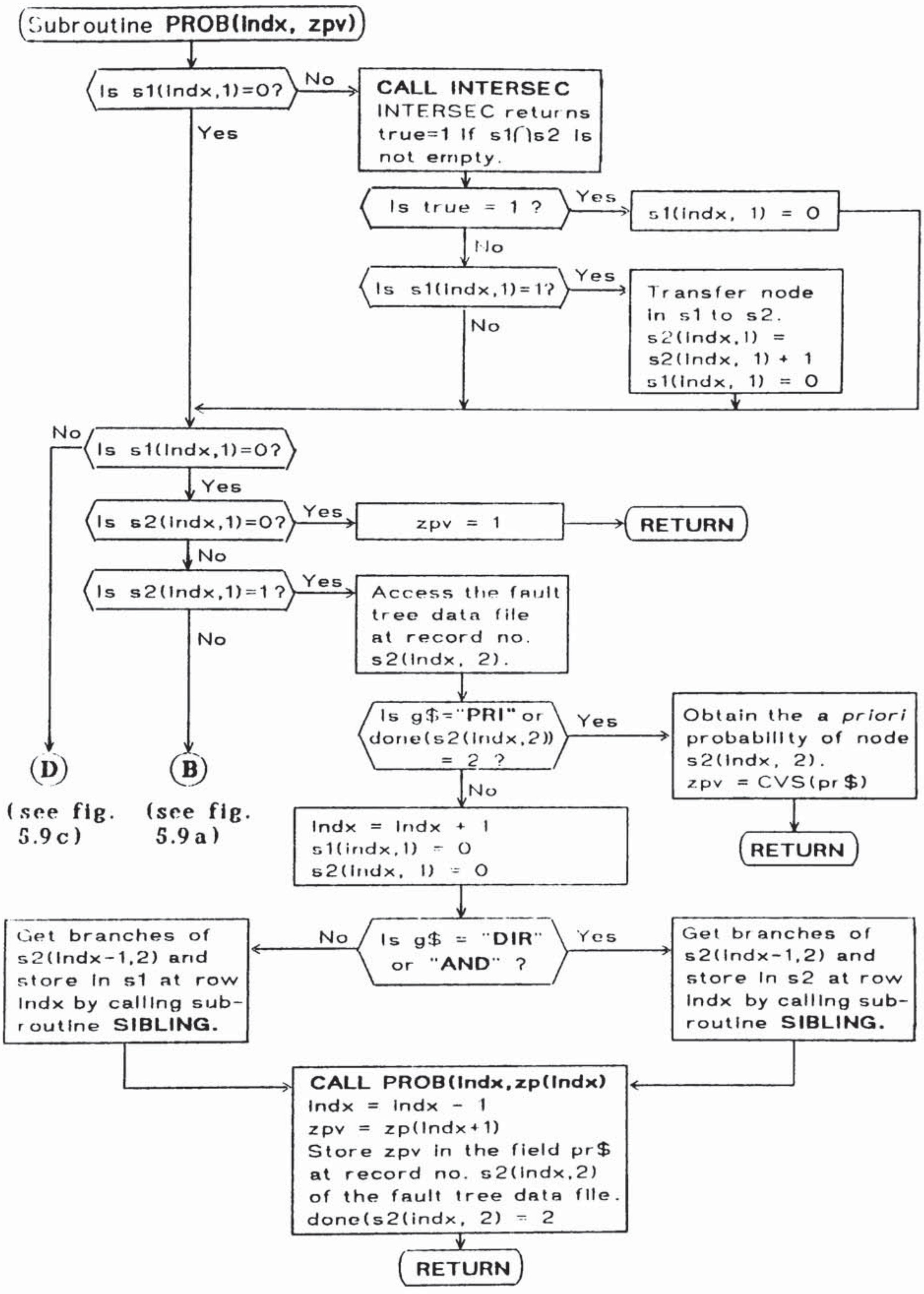

Subroutine PROB(Indx, zpv)
Is s1(Indx,1)=0?
No
Yes
CALL INTERSEC
INTERSEC returns true=1 if s1∩s2 is not empty.
Is true = 1 ?
Yes
s1(Indx, 1) = 0
No
Is s1(Indx,1)=1?
Yes
Transfer node in s1 to s2.
s2(Indx,1) = s2(Indx, 1) + 1
s1(Indx, 1) = 0
No
No
Is s1(Indx,1)=0?
Yes
Is s2(Indx,1)=0?
Yes
zpv = 1
RETURN
No
Is s2(Indx,1)=1 ?
Yes
Access the fault tree data file at record no. s2(Indx, 2).
No
Is g$="PRI" or done(s2(Indx,2)) = 2 ?
Yes
Obtain the a priori probability of node s2(Indx, 2).
zpv = CVS(pr$)
RETURN
No
D
B
(see fig. 5.9c)
(see fig. 5.9a)
Indx = Indx + 1
s1(Indx,1) = 0
s2(Indx, 1) = 0
Get branches of s2(Indx-1,2) and store in s1 at row Indx by calling subroutine SIBLING.
No
Is g$ = "DIR" or "AND" ?
Yes
Get branches of s2(Indx-1,2) and store in s2 at row Indx by calling subroutine SIBLING.
CALL PROB(Indx,zp(Indx))
Indx = Indx - 1
zpv = zp(Indx+1)
Store zpv in the field pr$ at record no. s2(Indx,2) of the fault tree data file.
done(s2(Indx, 2) = 2
RETURN

Figure 5.9 : Flowchart of Subroutine **PROB**

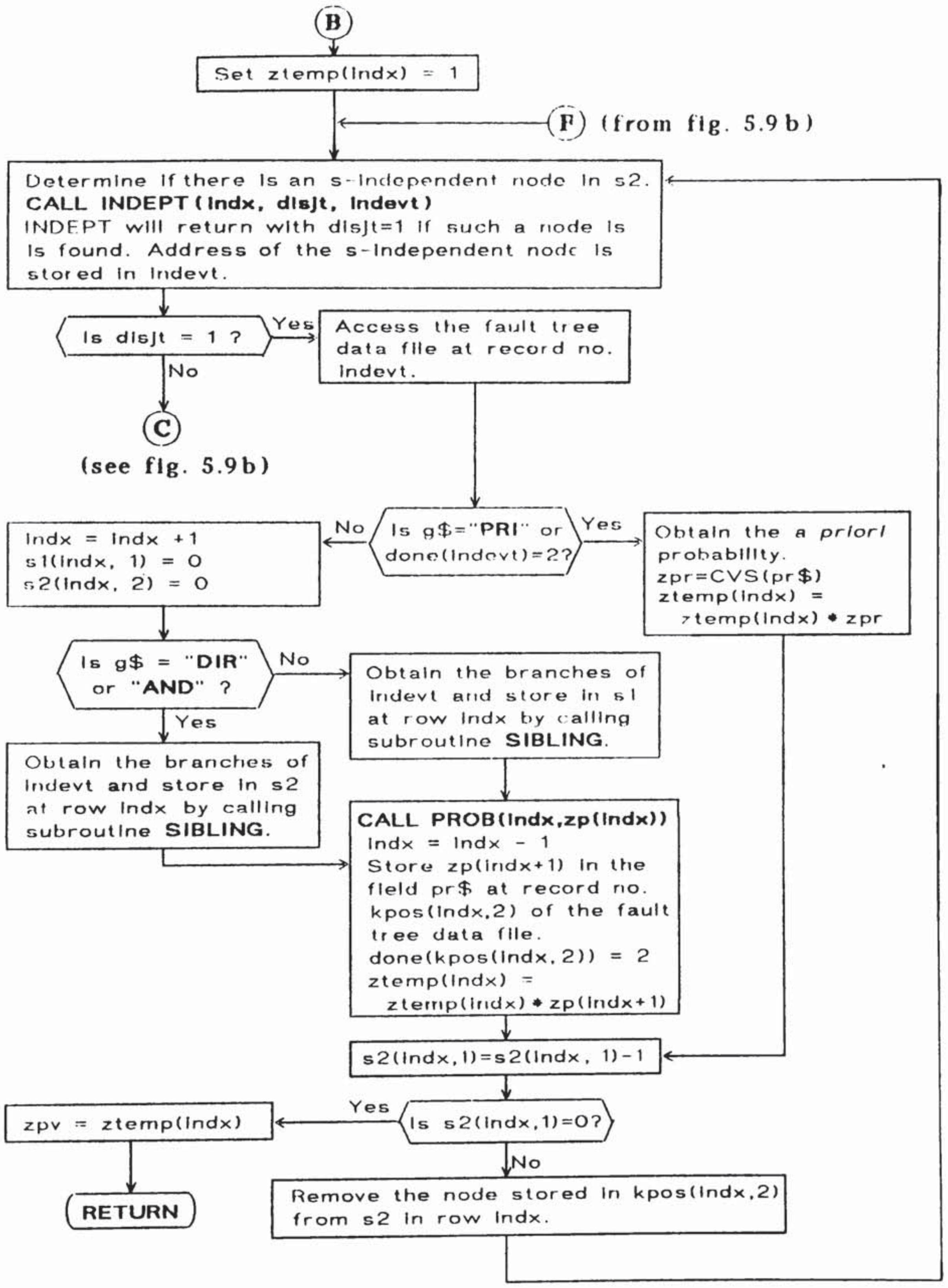

B
Set ztemp(indx) = 1
F (from fig. 5.9 b)
Determine if there is an s-independent node in s2.
CALL INDEPT(indx, disjt, indevt)
INDEPT will return with disjt=1 if such a node is is found. Address of the s-independent node is stored in indevt.
Is disjt = 1 ?
Yes
No
Access the fault tree data file at record no. indevt.
C
(see fig. 5.9b)
Is g$="PRI" or done(indevt)=2?
No
Yes
indx = indx +1
s1(indx, 1) = 0
s2(indx, 2) = 0
Obtain the a priori probability.
zpr=CVS(pr$)
ztemp(indx) = ztemp(indx) * zpr
Is g$ = "DIR" or "AND" ?
No
Yes
Obtain the branches of indevt and store in s1 at row indx by calling subroutine SIBLING.
Obtain the branches of indevt and store in s2 at row indx by calling subroutine SIBLING.
CALL PROB(indx,zp(indx))
indx = indx - 1
Store zp(indx+1) in the field pr$ at record no. kpos(indx,2) of the fault tree data file.
done(kpos(indx, 2)) = 2
ztemp(indx) = ztemp(indx) * zp(indx+1)
s2(indx,1)=s2(indx, 1)-1
Yes
Is s2(indx,1)=0?
No
zpv = ztemp(indx)
RETURN
Remove the node stored in kpos(indx,2) from s2 in row indx.

Figure 5.9a (continuuation of fig. 5.9)

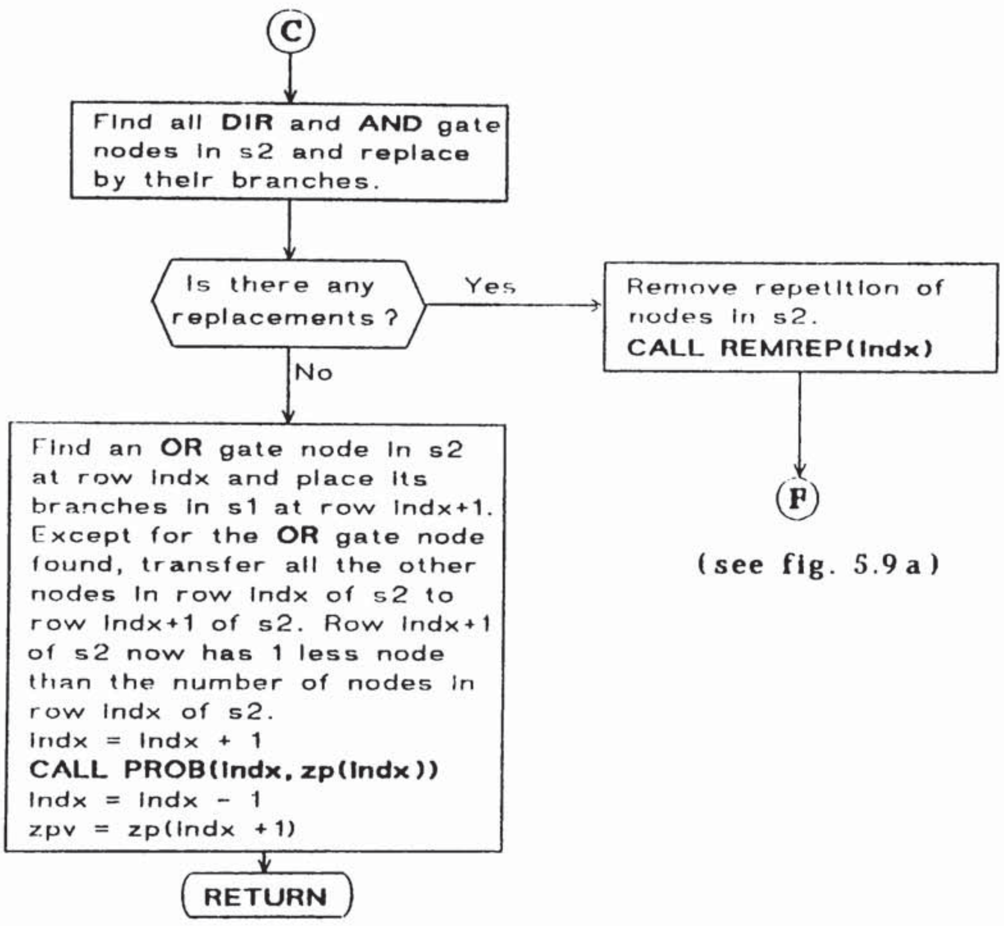

C
Find all DIR and AND gate nodes in s2 and replace by their branches.
Is there any replacements ?
Yes
Remove repetition of nodes in s2.
CALL REMREP(indx)
No
Find an OR gate node in s2 at row indx and place its branches in s1 at row indx+1. Except for the OR gate node found, transfer all the other nodes in row indx of s2 to row indx+1 of s2. Row indx+1 of s2 now has 1 less node than the number of nodes in row indx of s2.
indx = indx + 1
CALL PROB(indx, zp(indx))
indx = indx - 1
zpv = zp(indx +1)
RETURN
F
(see fig. 5.9 a)

Figure 5.9 b (continuation of fig. 5.9 a)

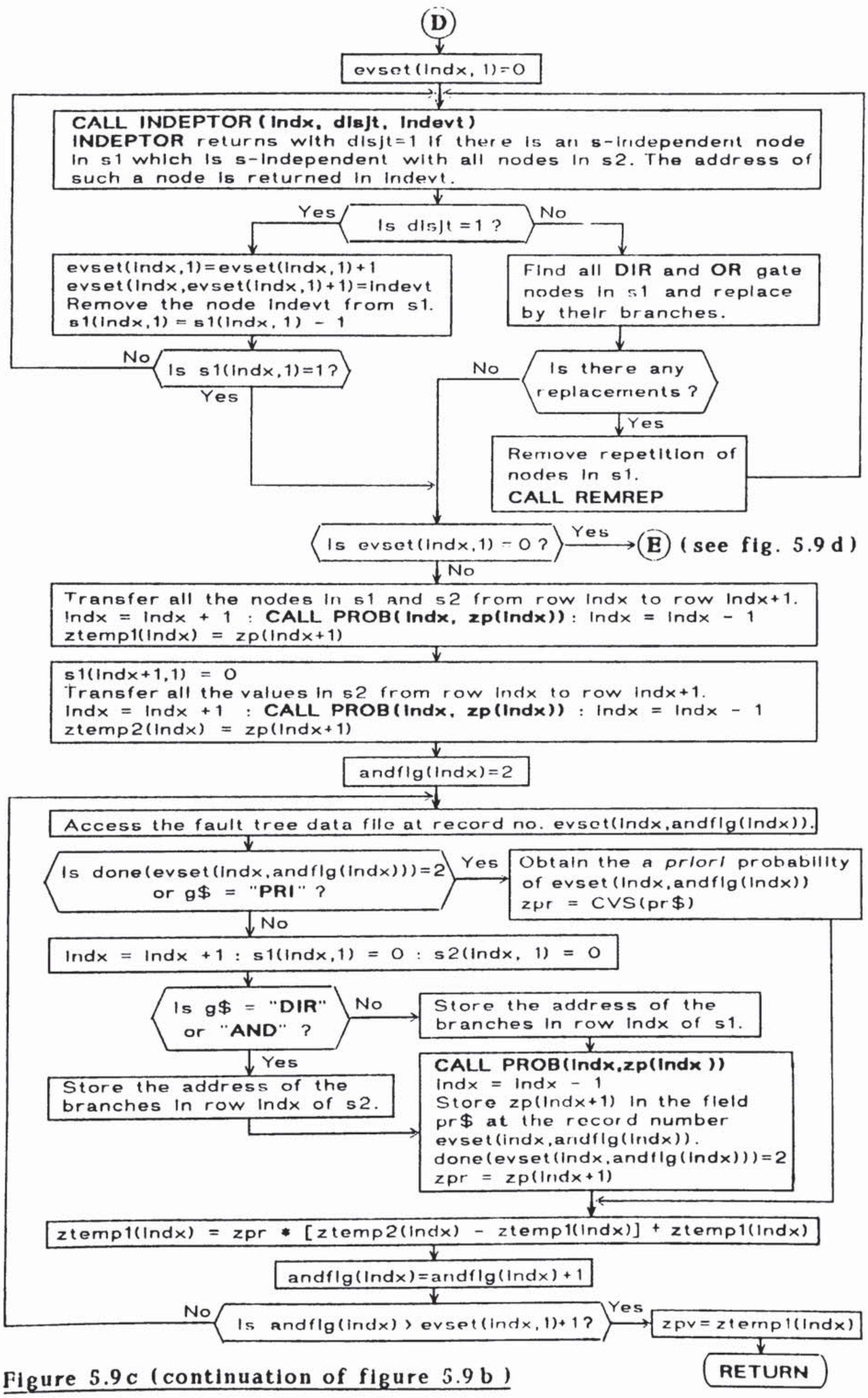
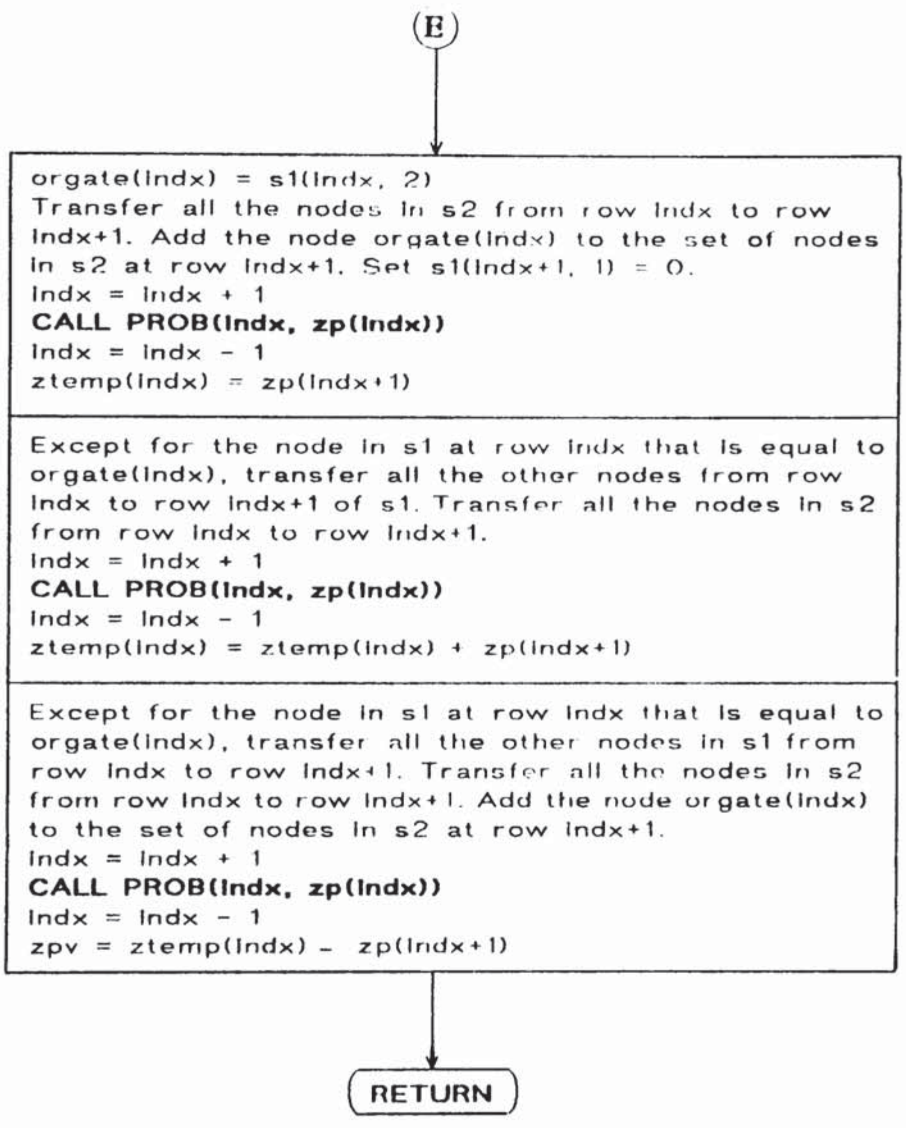

D
evset(indx, 1)=0
CALL INDEPTOR (indx, disjt, indevt)
INDEPTOR returns with disjt=1 if there is an s-independent node in s1 which is s-independent with all nodes in s2. The address of such a node is returned in indevt.
Yes
Is disjt =1 ?
No
evset(indx,1)=evset(indx,1)+1
evset(indx,evset(indx,1)+1)=indevt
Remove the node indevt from s1.
s1(indx,1) = s1(indx, 1) - 1
Find all DIR and OR gate nodes in s1 and replace by their branches.
No
Is s1(indx,1)=1?
Yes
No
Is there any replacements ?
Yes
Remove repetition of nodes in s1.
CALL REMREP
Is evset(indx,1) = 0 ?
Yes
E
(see fig. 5.9 d)
No
Transfer all the nodes in s1 and s2 from row indx to row indx+1.
indx = indx + 1 : CALL PROB(indx, zp(indx)) : indx = indx - 1
ztemp1(indx) = zp(indx+1)
s1(indx+1,1) = 0
Transfer all the values in s2 from row indx to row indx+1.
indx = indx +1 : CALL PROB(indx, zp(indx)) : indx = indx - 1
ztemp2(indx) = zp(indx+1)
andflg(indx)=2
Access the fault tree data file at record no. evset(indx,andflg(indx)).
Is done(evset(indx,andflg(indx)))=2 or g$ = "PRI" ?
Yes
Obtain the a priori probability of evset(indx,andflg(indx))
zpr = CVS(pr$)
No
indx = indx +1 : s1(indx,1) = 0 : s2(indx, 1) = 0
Is g$ = "DIR" or "AND" ?
No
Store the address of the branches in row indx of s1.
Yes
Store the address of the branches in row indx of s2.
CALL PROB(indx,zp(indx))
indx = indx - 1
Store zp(indx+1) in the field pr$ at the record number evset(indx,andflg(indx)).
done(evset(indx,andflg(indx)))=2
zpr = zp(indx+1)
ztemp1(indx) = zpr * [ztemp2(indx) - ztemp1(indx)] + ztemp1(indx)
andflg(indx)=andflg(indx)+1
No
Is andflg(indx) > evset(indx,1)+1?
Yes
zpv= ztemp1(indx)
RETURN

Figure 5.9c (continuation of figure 5.9b)



```
orgate(Indx) = s1(Indx, 2)
Transfer all the nodes in s2 from row Indx to row
Indx+1. Add the node orgate(Indx) to the set of nodes
in s2 at row Indx+1. Set s1(Indx+1, 1) = 0.
Indx = Indx + 1
CALL PROB(Indx, zp(Indx))
Indx = Indx - 1
ztemp(Indx) = zp(Indx+1)
```

```
Except for the node in s1 at row Indx that is equal to
orgate(Indx), transfer all the other nodes from row
Indx to row Indx+1 of s1. Transfer all the nodes in s2
from row Indx to row Indx+1.
Indx = Indx + 1
CALL PROB(Indx, zp(Indx))
Indx = Indx - 1
ztemp(Indx) = ztemp(Indx) + zp(Indx+1)
```

```
Except for the node in s1 at row Indx that is equal to
orgate(Indx), transfer all the other nodes in s1 from
row Indx to row Indx+1. Transfer all the nodes in s2
from row Indx to row Indx+1. Add the node orgate(Indx)
to the set of nodes in s2 at row Indx+1.
Indx = Indx + 1
CALL PROB(Indx, zp(Indx))
Indx = Indx - 1
zpv = ztemp(Indx) - zp(Indx+1)
```

Figure 5.9 d (continuation from fig. 5.9 c)

5.3 : The Fault Tree Display

5.3.1 : Introduction

The fault tree display forms the main part of the diagnosis package. The usual form of a fault tree diagram is that the top event is placed at the top with its causal events below it. This form of display was implemented in work by Jones and Lihou [16] in their computer package called CAFOS, by Ramadaan [106] who used the PERQ workstation for graphical output of fault trees and by Martin-Solis et al [41] in their development of an alarm analysis system.

However, the top down fashion of displaying a fault tree on the screen of a VDU has some limitations. Output of the fault tree diagram on the screen is often in the text mode which can display 80 column by 25 rows of characters. Reserving 20 columns for each event and at least 1 column for spaces between each event, the maximum number of events that can be displayed in a row is 3. If the branches of the top event displayed are secondary events, the display of their causes in the next row below may overlap each other, thus making the output information unintelligible.

In view of the above limitations, a new method of displaying the fault tree was developed so that as much information as possible can be placed on the screen without the user becoming confused. This is done by displaying the fault tree sideways, i.e. the alarm event in the centre of the screen, the causes in a column to the left of the alarm event and the consequences in a column to the right of the alarm event.

A guide to select and examine the most likely cause of the alarm is by using the *posteriori* probability of events calculated after the alarm has been initiated. The theory of conditional probability for determining the *posteriori* probability has already been described in chapter 4. An algorithm to quickly calculate the *posteriori* probability of every event below the alarm event was developed using the *a priori* probability data already obtained in the data preparation part of the package. The *posteriori* probabilities are calculated before displaying the fault tree.

5.3.2 : Structure of the Fault Tree Display

The display of a fault tree on the screen of the VDU is structured to show the alarm event, its causes and consequences in different columns. The screen of the VDU is divided into three columns of equal width. The first column, known as the cause column, is for displaying the causal events of the alarm. The middle column known as the fault column is for the display of the alarm event. The third column, known as the consequence column, is where the consequences of the alarm are displayed. To avoid possible overlapping of the displayed events, causes and consequences are displayed one level at a time. Lines and a rectangle or a triangle to represent an **OR** or **AND** gate respectively are drawn to show the logical connections between events.

The area for drawing the fault tree is from row 2 to row 20 of the screen. Rows 21 to 25 are reserved for the output of messages. For each event to be displayed, three rows are assigned to show the event name and the *a priori* and *posteriori* probabilities above and below the event name respectively. Including empty rows to separate each event, the maximum number of events that can be displayed in each of the three columns is five arranged vertically. Thus, if the number of causes that are input to one gate is greater than five, only five events are displayed in the cause column at a time with the others off the screen. By pressing certain programmed function keys on the keyboard, the user can scroll the cause column up or down to show the other events that cannot fit onto the screen. The fault and consequence columns can also be individually scrolled up or down to show other events if there are more than five events to be displayed in each column. Table 5.2 show what function keys are programmed to do such tasks.

On the display, there is a cursor represented by the first blinking character of the event name. If a cause of the alarm is a secondary event, the user can display its branches by directing the cursor to that event and pressing one of the programmed functions keys on the keyboard, i.e. the F9 key. The causes of the secondary event are displayed in the cause column and the secondary event itself is placed below the bottom most event displayed in the fault column. In this way, the user can trace out any primary cause of the alarm by tracing the line of events from the alarm event to the primary causes.

Any further symptoms of a consequence in the consequence column can also be displayed. This is done by directing the cursor to the consequent event under consideration and pressing the F9 key. This time the consequent event is placed above the top most event displayed in the fault column and its symptoms are displayed in the consequence column. By this method, the user can look into the chain of events that would happen if the alarm is not remedied.

The display of probability values of every event in the fault tree can be switched on or off using certain function keys which are programmed as toggle switches. The *a priori* and *posteriori* probabilities are displayed below and above the event names respectively. Table 5.2 shows which function keys are programmed for such tasks. A help key combination is available which displays table 5.2 on the screen.

The name of each event is in a coded form which the user may not comprehend. By pressing the [Alternate] and [E] keys together, programed as a toggle switch, the explanation of the coded event at the cursor can be switched on. The meaning of the code is displayed on row 22 of the screen. By directing the cursor to other events using the cursor directional keys on the numeric key pad of the keyboard, the meaning of the coded event at the cursor is automatically displayed. Pressing the [Alternate] and [E] keys together again will switch off the explanation mode.

If the user wants to look into the causes and consequences of any other event apart from the alarm event, pressing the [Alternate] and [A] keys together will ask the user to input the name of the event via the keyboard. The event name is then searched in the fault tree

Table 5.2 : Keys that are programed to do certain functions

Key	Function
F1	A toggle to switch on or off the display of *a priori* probabilities.
F2	A toggle to switch on or off the display of *posteriori* probabilities.
F3	To scroll cause column upwards by one event.
F4	To scroll cause column downwards by one event.
F5	To scroll alarm column upwards by one event.
F6	To scroll alarm column downwards by one event.
F7	To scroll consequence column upwards by one event.
F8	To scroll consequence column downwards by one event.
F9	To display the causes or consequences of the event at the cursor. 1. If the cursor is at an event in the cause column, its name is placed below the bottom most event in the alarm column and its causes are displayed in the cause column. 2. If the cursor is at an event in the consequence column, its name is placed above the top most event in the alarm column and its symptoms are displayed in the consequence column. 3. If the cursor is at an event below the alarm event in the alarm column, all the events below that event disappears and its causes are displayed in the cause column. 4. If the cursor is at an event above the alarm event in the alarm column, all the events above that event disappears and its symptoms are displayed in the consequence column.
F10	To display the original fault tree initiated by the alarm.
Alt-A	Ask for the input via the keyboard an event name which is to be browsed for its causes and consequences.
Alt-E	A toggle to switch on or off display of the meaning of the coded event under the cursor.

data file. Then the causes and consequences for that event are displayed in their appropriate column and the input event is placed in the fault column. Pressing the F10 function key will display the original fault and consequence tree initiated by the alarm event.

The fault tree diplay makes full use of the colours available on the VDU to describe the causal and consequent events. The name of events are written in coloured boxes. The colour of a box is known as the background colour and the colour of the letterings of the event name is known as the foreground colour. For the alarm event, the background colour is red and the foreground is yellow. This gives a striking contrast for the user to take notice of the prevailing condition. The colour attributes given to other events in the fault tree is described in table 5.3.

5.3.3 : The Fault Tree Display Program

5.3.3.1 : The Main Program - DISFAULT

The main program for the fault tree display is called **DISFAULT**. Input to the program is the name of the alarm event. **DISFAULT** then searches for the event name in the fault tree data file using the subroutine **FIND.EV**. When the name of the alarm event has been found, **FIND.EV** returns to **DISFAULT** with the address of the event, the number of causal branches, the number of consequences, the *a priori* probability and the *posteriori* probability stored in the variables **top**, **tcbr**, **tqbr**, **zd** and **ze** respectively. At this point, the *posteriori*

Table 5.3 : Colour Attributes of Events in the Fault Tree Display

Description of event	Background	Foreground
Primary event in cause column with only one symptom.	Green	White
Primary event in cause column with more than one symptom.	Green	Black
Secondary event in cause column with only one symptom.	Cyan	White
Secondary event in cause column with more than one symptom	Cyan	Black
Event in consequence column with no other symptoms and caused by the only event in the alarm column	Green	White
Event in consequence column with no other symptoms that may be caused by other events apart from the one in the alarm column.	Green	Black
Event in consequence column with other symptoms and caused by the only event in the alarm column	Cyan	White
Event in consequence column with other symptoms that may be caused by other events apart from the one in the alarm column	Cyan	Black

probability has not been calculated yet. Thus, **ze** contains the value 0.

DISFAULT then calls the subroutine **FORM1** to clear the screen of the VDU and then divide the screen into three columns, each of equal width of 26 characters. Between each column, a space is reserved for printing a vertical line to separate the columns. This takes up the remaining two spaces left after dividing the screen. The headings for each column is printed on the first row of the screen.

DISFAULT then calls the subroutine **DISPLAY** which performs the actual task of diplaying the alarm event, the causes and consequences. After calling **DISPLAY**, the position of the first character of the alarm event is calculated in terms of the screen memory location. Each position on the screen where a character can be printed is assigned two screen memory locations. The first location is to store the ASCII code of the character printed and the second location is to store the code for the attribute of the character. The code for the attribute of a character is a unique number that describes in what foreground and background colour the character is to be printed on the screen. Thus, for a single row on the screen where eighty characters can be printed, there are 160 memory locations to store the ASCII codes and attribute codes. The total amount of screen memory locations for a page of the screen that can fill 80x25 characters is 4000. The screen memory starts at relative address 0 to represent the top left hand corner of the screen. Lets suppose that the first character of the alarm event name is at column **vnum** and row **hnum** of the screen. The relative address where the attribute of the character is stored is given by:

apos = 160 * (**hnum** - 1) + 2 * **vnum** - 1 ..... 5.5

The relative address where the ASCII code of the character is stored is given by:

cpos = **apos** - 1 ..... 5.6

Subroutines **PIXATR** and **CURSOR** are then called consecutively. The function of **PIXATR** is to save the code for the attribute at relative memory location **apos** in a variable called **atr**. The function of **CURSOR** is to set the character at relative memory location **cpos** blinking by changing the attribute stored at memory location **apos**. This is done by adding 128 to the code at **apos** and storing the result at **apos.** The blinking character represents the cursor in the display. Whenever the cursor is directed to other events, the attribute of the character where the cursor is originally located is first reset to its original code stored in **atr**. The code of attribute of the first character of the event to which the cursor has been directed is then saved in **atr** and then that character is set blinking by changing its attribute.

After setting the cursor at the alarm event, the function keys and two other key stroke combinations which have been programed to do certain tasks are set as interrupt keys. Pressing any of the interrupt keys will direct the program to execute a routine particular to the function required by the key stroke. These routines will be described later in the chapter.

After this stage, the program is set to go around in a loop, awaiting the [End] on the numeric key pad to be pressed to end the execution of **DISFAULT**.

5.3.3.2 : Description of Subroutine DISPLAY

Subroutine **DISPLAY** calls several subroutines at different stages, each with its own function. There are 10 stages of subroutine calls and each are described below.

1. Subroutine **FORM2**

Subroutine **FORM2** clears the fault tree drawing area from rows 2 to 20 of the screen and then draw two vertical lines to separate between the cause column and the fault column; and between the fault column and the consequence column.

2. Subroutine **IN.FAULT**

The function of subroutine **IN.FAULT** is to determine the row position for printing the top most event to be displayed in the fault column. If only the alarm event is to be displayed, then its printing position is at row 11 of the screen. If there is more than one event to be shown, then the printing position of the top most event to be displayed is calculated so that the display of the events fits nicely in the fault column. The calculated row position is stored in **alin** and **falin** which are used by other subroutines when the actual output to the screen is being done.

3. Subroutine **CONPROB**

The function of **CONPROB** is to calculate the *posteriori* probabilities of all events in the fault tree. Before subroutine **CONPROB** is called, **DISPLAY** checks if a flag called **doneflg** has been set to 1. The flag **doneflg** was first set to 0 when the main program **DISFAULT** was executed. This tells **DISPLAY** that calculation of the *posteriori* probabilities has not been done yet. When **DISPLAY** acknowledges that **doneflg** is not set to 1 , subroutine **CONPROB** is called. After **CONPROB** returns to **DISPLAY**, **doneflg** is set to 1 so that in future calls to **DISPLAY**, subroutine **CONPROB** will not be called. A detailed description of the algorithm in **CONPROB** is given in section 5.4.

4. Subroutine **GET.CAUSE**

The function of subroutine **GET.CAUSE** is to obtain from the fault tree data file all the relevant information that will be displayed in the cause column. First, **GET.CAUSE** determines the type of gate of the event in the fault column whose causes are to be displayed in the cause column. An ASCII code for the symbol that represents the gate is stored in the variable **sym$**. The symbols to represent an **OR**, an **AND** gate and a **DIR** gate are "█", "▶" and "—" respectively. Then the names of the causal events are obtained and stored in a one dimensional string array called **ca$**. Each of the **n** events stored in **ca$**, where **n** is the number of causal events, are inputs to the gate stored in **sym$**. The type of each event is determined as either a primary or secondary event, and whether it has one or several consequences. This is done to set the display attribute for each event. For the ith event, the code for background colour is stored in **bc(i)** and the code for the foreground colour is stored in **fc(i)**. The background and foreground colours that describe the causal event are shown in table 5.3. The *a priori* and *posteriori* probabilities for that causal event are also obtained and stored in **zc(i)** and **zcp(i)** respectively.

5. Subroutine GET.CONSEQ

The function of subroutine **GET.CONSEQ** is to obtain from the fault tree data file all the relevant information that will be displayed in the consequence column. The event names of the consequences of the event in the fault column are obtained and stored in a one dimensional string array called **qa$**. The description of each of **m** consequent events stored in **qa$**, where **m** is the number of consequences of the event in the fault column, is noted, i.e. whether it has other symptoms or not, and whether it is only caused by the event in the fault column or by other events apart from the event in the fault column. This is done to set the display attribute that describes each. For the jth event, the code of the background colour is stored in **bq(j)** and the code for the foreground colour is stored in **fq(j)**. The *a priori* and *posteriori* probabilities for that event are also obtained and stored in **zq(j)** and **zqp(j)** respectively.

6. Subroutine IN.CAUSE

The function of subroutine **IN.CAUSE** is to calculate the top-most row position for printing the causal event stored in **ca$(1)**

in the cause column. If there is only one causal event to be shown, then the position it will be displayed is at row 11 of the screen. If there are more than five events stored in array **ca$**, then the position at which the event stored in **ca$(1)** will be displayed at row 3 of the screen. If the number of causal events to be shown is less than 5 but more than 1, then **IN.CAUSE** calculates the row position to display the event stored in **ca$(1)** so that the display of all the causal events fits centrally in the cause column. The calculated row position is stored in **clin** and **fclin** which are used by other subroutines when the actual output to the screen is being done.

7. Subroutine **IN.CONSEQ**

The function of subroutine **IN.CONSEQ** is to calculate the top-most row position for printing the consequent event stored in **qa$(1)** in the consequence column. If there is only one consequent event to be shown, then the position it will be displayed is at row 11 of the screen. If there are more than five events stored in array **qa$**, then the position at which the event stored in **qa$(1)** will be displayed at row 3 of the screen. If the number of consequences to be shown is less than 5 but more than 1, then **IN.CONSEQ** calculates the row position to display the event stored in **qa$(1)** so that the display of all the consequent events fits centrally in the consequence column. The calculated row position is stored in **qlin** and **fqlin** which are used by other subroutines when the actual output to the screen is being done.

8. Subroutine **ALARM**

The function of subroutine **ALARM** is to display in the fault column the alarm event and also other events that have been pushed from the cause and the consequence columns. The position at which the top most event in the fault column is printed is at row **alin** of the screen. If there are any other events below the top most event to be displayed in the fault column, then for each event to be printed, **alin** is incremented by 3 and that event is printed at row **alin** of the screen.

9. Subroutine **CAUSES**

Subroutine **CAUSES** displays the causal events that have been stored in array **ca$**. The event stored in **ca$(1)** is printed at row **clin**. For each other event stored in **ca$(i)**, i=2 to n, **clin** is incremented by 3 before that event is printed at row **clin** in the cause column. Then the content of **sym$** is printed at row 11 and column 26 of the screen to show the type of gate the causes are input to the event in the fault column.

10. Subroutine **QUENCE**

Subroutine **QUENCE** displays the consequence events that have been stored in the array **qa$**. The event stored in **qa$(1)** is printed at row **qlin**. For each other event stored in **qa$(j)**, j=2 to m, **qlin** is incremated by 3 before that event is printed at row **qlin** in the consequence column.

5.3.3.3 : Interrupt Service Routines

The function keys, cursor direction keys and certain key combinations are programmed as interrupts to do certain tasks on the fault tree display. When any of the keys are pressed, a small subprogram called an interrupt service routine, particular to the task required, is executed. Whilst the interrupt is being serviced, no other interrupts can be received. After execution of an interrupt service routine, control is returned to the program prior to the interrupt. The names of each interrupt service routine is described below.

1. **TOGGLE1**

TOGGLE1 is executed when the **F1** key is pressed. First it sets the flag **apri** to 0 if previously it is 1 or sets it to 1 if previously it is 0. Then it calls subroutine **APRIORI**. If **APRIORI** is called with **apri**=1, then the subroutine prints out on the screen the *a priori* probabilities above each event displayed. If **APRIORI** is called with **apri**=0, then the subroutine erases all the *a priori* probabilities that have been printed on the screen.

2. **TOGGLE2**

TOGGLE2 is executed when the **F2** key is pressed. First it sets the flag **post** to 0 if previously it is 1 or sets it to 1 if previously it is 0. Then it calls subroutine **POSTER**. If **POSTER** is called with **post**=1, then the subroutine prints out on the screen the *posteriori* probabilities below each event displayed. If **POSTER** is called with

post=0, then the subroutine erases all the *posteriori* probabilities that have been printed on the screen.

3. CAUSEUP

CAUSEUP is executed when the **F3** key is pressed. **CAUSEUP** calls the subroutine **CAUCOLUP**. The function of **CAUCOLUP** is to scroll up the cause column by one event so that any event that is off the screen below the bottom-most event of the display in the cause column can be shown on the screen. When this is done, every event displayed in the cause column also would have moved up, resulting in the previous top-most event displayed move off the screen. If there is no event that is off the screen below the bottom-most event displayed or originally there is only five or less causal events displayed in the cause column, **CAUCOLUP** will output an audible signal telling the user that no scrolling is done.

4. CAUSEDN

CAUSEDN is executed when the **F4** key is pressed. **CAUSEDN** calls the subroutine **CAUCOLDN**. The function of **CAUCOLDN** is to scroll down the cause column by one event so that any event that is off the screen above the top-most event of the display in the cause column can be shown on the screen. When this is done, every event displayed in the cause column also would have moved down, resulting in the previous bottom-most event displayed move off the screen. If there is no event that is off the screen above the top-most event or originally there is only five or less causal events displayed in the cause column, **CAUCOLDN** will output an audible signal telling the user that no scrolling is done.

5. FAULTUP

FAULTUP is executed when the **F5** key is pressed. **FAULTUP** calls the subroutine **FAUCOLUP** which is similar in function with subroutine **CAUCOLUP** except that it acts on events in the fault column.

6. FAULTDN

FAULTDN is executed when the **F6** key is pressed. **FAULTDN** calls the subroutine **FAUCOLUP** which is similar in function with subroutine **CAUCOLDN** except that it acts on events in the fault column.

7. CONSEQUP

CONSEQUP is executed when the **F7** key is pressed. **CONSEQUP** calls the subroutine **QONCOLUP** which is similar in function with subroutine **CAUCOLUP** except that it acts on events in the consequence column.

8. CONSEQDN

CONSEQDN is executed when the **F6** key is pressed. **CONSEQDN** calls the subroutine **QONCOLDN** which is similar in function with subroutine **CAUCOLDN** except that it acts on events in the consequence column.

9. OTHERS

OTHERS is executed when the **F9** key is pressed. The purpose of pressing the **F9** key is to display the causes or consequences of the event at the cursor. If the cursor is at the alarm event and there is no other event displayed below it in the fault column, then an audible signal is output telling the user that nothing is being done. For other cursor positions, first, the event name at the cursor is read from the screen and stored in the string variable **evt$**. This is done by calling the subroutine **EVTNAM**. Then subroutine **FIND.EV** is called to obtain from the fault tree data file the address of the event stored in **evt$**. Then the relevant information for display of the causes or consequences in their respective column is obtained. The information obtained are the event names, display attributes, *a priori* probabilities and *posteriori* probabilities.

If the cursor is at an event in the cause column or at an event below the alarm event in the fault column, then the information about the causes of that event is obtained by calling subroutine **GET.CAUSE**. Then subroutine **DISPLAY1** is called which clears the cause column of any display and output the causal information obtained. If the cursor is at an event in the the cause column, that event is displayed below the bottom-most event in the fault column and its causes are displayed in the cause column. If however, the cursor is at an event below the alarm event in the fault column, all the events displayed below it are removed and its causes are displayed in the cause column.

If the cursor is at an event in the consequence column or at an event above the alarm event in the fault column, then the information about the consequences of that event is obtained by calling subroutine **GET.CONSEQ**. Then subroutine **DISPLAY2** is called which clears the consequence column of any display and output the consequence information obtained. If the cursor is at an event in the consequence column, that event is displayed above the top-most event in the fault column and its consequences are displayed in the consequence column. If however, the cursor is at an event above the alarm event in the fault column, all the events displayed above it are removed and its consequences are displayed in the consequence column.

10. ORIGINAL

ORIGINAL is executed when the **F10** key is pressed. **ORIGINAL** will erase the present display and output the initial fault and consequence tree where the only event displayed in the alarm column is the alarm event, and the cause and consequence columns show the immediate causes and consequences of the alarm event respectively.

11. CURLEFT

CURLEFT is executed when the left cursor direction key on the numeric key pad is pressed. The effect when **CURLEFT** is executed is that the cursor represented by the blinking character of an event name is moved to an event to the left of the first event. If the cursor is already on an event in the cause column, pressing of the left cursor direction key will result in an output of an audible signal telling the user that the cursor cannot move any further left.

12. CURRIGHT

CURRIGHT is executed when the right cursor direction key on the numeric key pad is pressed. The effect when **CURRIGHT** is executed is that the cursor represented by the blinking character of an event name is moved to an event to the right of the first event. If the cursor is already on an event in the consequence column, pressing of the right cursor direction key will result in an output of an audible signal telling the user that the cursor cannot move any further right.

13. CURDOWN

CURDOWN is executed when the down cursor direction key on the numeric key pad is pressed. The effect when **CURDOWN** is executed is that the cursor represented by the blinking character of an event name is moved to an event below the first event. If the cursor is already on the bottom-most event in any column, pressing of the down cursor direction key will result in an output of an audible signal telling the user that the cursor cannot move any further down.

14. CURUP

CURUP is executed when the up cursor direction key on the numeric key pad is pressed. The effect when **CURUP** is executed is that the cursor represented by the blinking character of an event name is moved to an event above the first event. If the cursor is already on the top-most event in any column, pressing of the up cursor direction key will result in an output of an audible signal telling the user that the cursor cannot move any further up.

15. ANOTHER

ANOTHER is executed when the keys **Alternate** and **A** are pressed together. When these keys are pressed, a message ouput on rows 22 to 24 will ask the user to input the name of an event where its causes and consequences are to be browsed. The name of the event is searched from the fault tree data file. When the event has been found, subroutine **DISPLAY** is called to output the fault and consequence tree for that event. When control is returned to the program prior to this interrupt, the present display is maintained and the user can use any of the programmed key combinations to look into other parts of the fault and consequence tree. Pressing of the **F10** key will display again the initial fault and consequence tree initiated by the alarm event.

An error message will be printed if the event input by the user is not found in the fault tree data file. **ANOTHER** will ask the user to input again the correct event name that is to be browsed. By inputting the number 0, nothing is being done and control is returned back to the program prior to this interrupt.

16. EXPCODE

EXPCODE is executed when the keys **Alternate** and **E** are pressed together. The effect of this interrupt is to switch on or off the explanation of the event names under the cursor. The meaning of the coded event is displayed in rows 22 to 24 of the screen. First, **EXPCODE** sets the flag **meant** to 1 if previously it is 0 or sets it to 0 if previously it is 1. Then subroutine **EXPLAIN** is called. If **EXPLAIN** is called with **meant** equal to 1, then the meaning of the event under the cursor is displayed and control is returned to the program prior to the interrupt. If the cursor is moved to other event, the meaning of that event is displayed. The explanation mode is switched off by pressing **Alternate** and **E** keys together, where this time the flag **meant** is set to 0. When **EXPLAIN** is called with **meant** equal to 0, the explanation that has been displayed is erased and when control is returned to the program prior to the second interrupt, nothing is displayed in rows 22 to 24.

17. **HELP**

HELP is executed when the keys **Alternate** and **H** are pressed together. The effect of this interrupt is to display a help screen that tells the user which key to press for the required task to be done on the fault tree display. Pressing the **Esc** key will display again the fault tree on the screen prior to the interupt.

5.4 : *Posteriori* Probability Calculation

5.4.1 : Introduction

The method used in evaluating the *posteriori* probabilitiy of each causal and consequent event of an alarm event is based on Bayes' Theorem on conditional probability, which has been discussed in chapter 4. The program for calculating the conditional probabilities is written as a subroutine called **CONPROB**. It is called from **DISFAULT**, the fault tree display program, after input of the alarm event. A brief description of the algorithm in **CONPROB** is given in this section.

During the evaluation of the posteriori probabilities by **CONPROB**, there may be cases where it is required to evaluate the probability of the intersection of two events. This is acheived by using subroutine **PPROB** which is a simplified version of subroutine **PROB**, used in evaluating the *a priori* probabilities of events in a fault tree. Subroutine **PROB** has been described earlier in this chapter. A brief description of **PPROB** is also given in this section.

5.4.2 : Description of Subroutine CONPROB

Subroutine **CONPROB** evaluates the *posteriori* probabilities of an event and all its consequences before moving on to the next event unevaluated event. A one dimensional array called **done** is used as a flag to indicate whether or not an event and all its consequences have been evaluated. For example, suppose that **m** is the address of an event in the fault tree data file and the *posteriori* probabilities of the event and all its consequences have been evaluated. For such a case, **done(m)** is set equal to 1. This takes care of repeated events in the fault tree, which may be encountered more than once during the *posteriori* probability evaluation process.

A two dimensional array called **evset** is used to contain the addresses of either the causes or the consequences of an event, depending on the stage of the *posteriori* probability evaluation process. The address of the alarm event is already stored in the global variable **top** when **CONPROB** is called.

In subroutine **CONPROB**, first, the content of **top** is stored in **gtop** and a variable **cq** is set equal to 0. The variable **cq** will either contain 0 or 1, depending on whether the *posteriori* probabilities of the consequences of the alarm event or the consequences of a causal event that are to be evaluated. Then the *posteriori* probability of every event that has connections with the alarm event in the fault tree is evaluated according to the following steps:

1. Store the value 1 is in the conditional probability field, **cpr$**, at record number **gtop** in the fault tree data file.

2. Test if the number of consequences at record number **gtop** is greater than **cq**. If the result of the test is false, then proceed directly to step 3. Otherwise call subroutine **REPROB3** to evaluate the *posteriori* probabilities of the consequences of **gtop**. On returning from **REPROB3**, the flag **done(gtop)** is set equal to 1.

3. Test if **gtop** has an input via a **DIR** gate. If the result of the test is false, then proceed directly to step 4. Otherwise store the address of the causal event in **gtop** and set **cq** equal to 1. Then repeat the procedures from step 1.

4. Obtain the addresses of the causal events of **gtop** from the fault tree data file and store in row 1 of the array **evset**. The number of causal events stored is contained in **evset(1, 1)**. From the fault tree data file at record number **gtop**, obtain the contents of the gate type field, **g$**, and store in the string variable **gt$**. **gt$** now contains the type of gate through which **gtop** has its inputs. If the content of **evset(1, 1)** is 0, it implies that **gtop** itself is a primary event. For this case, proceed directly to step 6 below. Otherwise, call subroutine **REPROB1** to evaluate the *posteriori* probabilities of the causal events connected to **gtop** via the gate stored in **gt$**. The chain of events that propagate to **gtop** via a series of **DIR** gates and gates similar to that stored in **gt$** are also evaluated by **REPROB1**. Within subroutine **REPROB1**, the *posteriori* probability of the consequences of every causal event are also evaluated by using subroutine **REPROB3**. On returning from **REPROB1**, all the nodes in row 1 of **evset** will have gate types that are different from that contained in **gt$**.

5. For each node in row 1 of **evset** that has input events, replace it by the address of one of the input events and store the rest of the addresses of the input events in other places of **evest** at the same row. This is done by calling the subroutine **SIBLINGS.** If there are no replacements made, it indicates that all the nodes in row 1 of **evset** represent primary events. For this case, proceed directly to step 6. Otherwise, call subroutine **REMREP** to remove any repetition of nodes in **evset**. Then call subroutine **REPROB2** to evaluate the *posteriori* probabilities of those nodes

in row 1 of **evset** which has not been evaluated. The *posteriori* probabilities of the consequences of these nodes are also evaluated by calling subroutine **REPROB3** within **REPROB2.**

6. Evaluate the *posteriori* probabilities of every event in the fault tree data file which do not have any connection with the occured event. This is done by storing the *a priori* probability in the conditional probability field, **cpr$**, at the address of the event in the fault tree data file.

The flowchart of subroutine **CONPROB** is shown in figure 5.10.

5.4.3 : Description of Subroutines called by CONPROB

In subroutine **REPROB1**, the evaluation of the *posteriori* probability of every node in row 1 of **evset** depends on the content of **gt$** passed from its caller. If **gt$** contains the string "**AND**", then the *posteriori* probability of every node in row 1 of evset is 1. Otherwise, the *posteriori* proabability of each node in row 1 of evset is equal to its *a priori* probability divided by the *a priori* probability of the alarm event.

After evaluating all the nodes in row 1 of evset, any node that has inputs via the same gate as **gt$** is replaced by the addresses of the branches in **evset**. The process of evaluating the *posteriori* probabilities of nodes in **evset** which have not yet been evaluated is repeated until every node in **evset** does not has inputs via the same

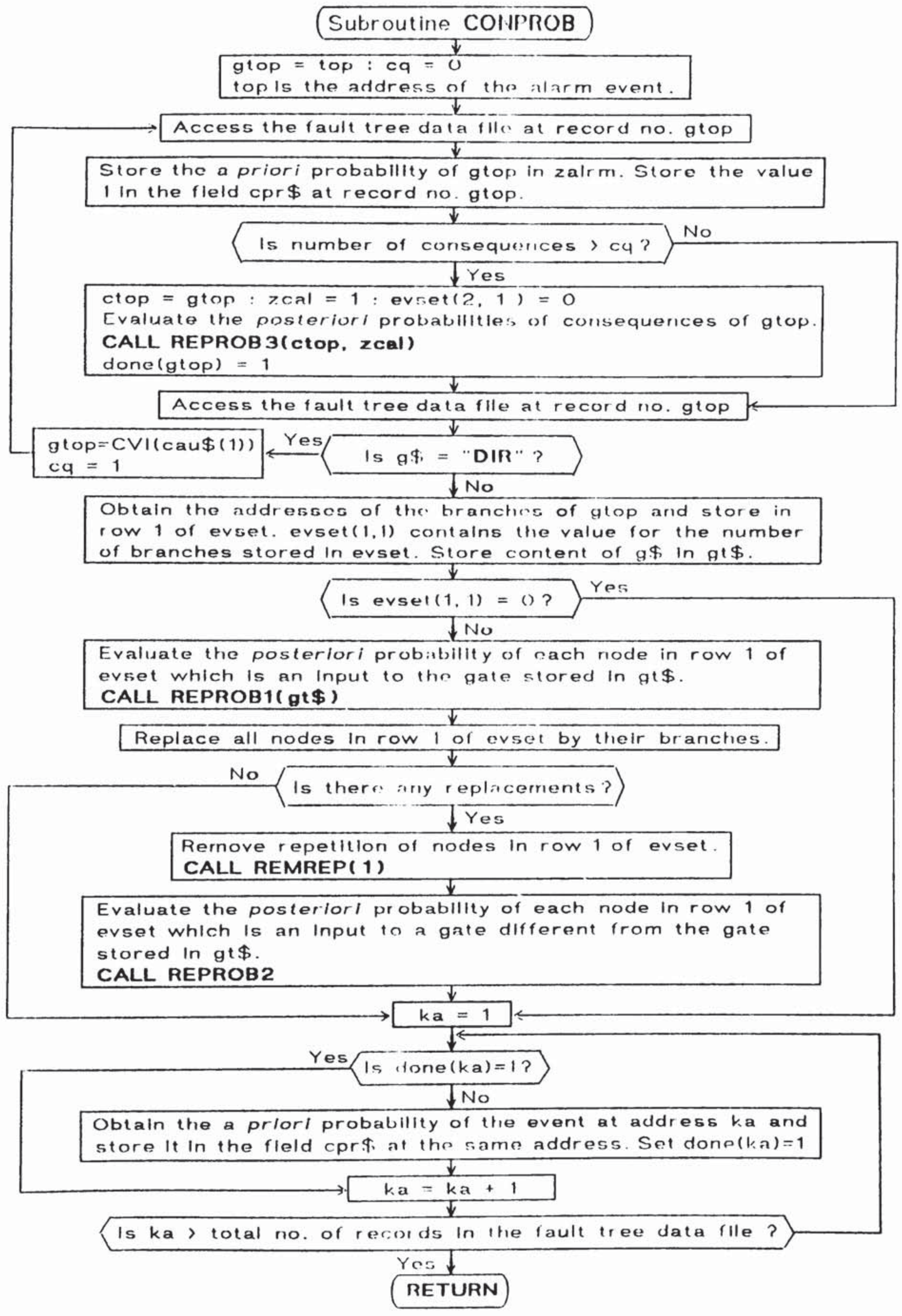

Subroutine CONPROB
gtop = top : cq = 0
top is the address of the alarm event.
Access the fault tree data file at record no. gtop
Store the a priori probability of gtop in zalrm. Store the value 1 in the field cpr$ at record no. gtop.
Is number of consequences > cq ?
No
Yes
ctop = gtop : zcal = 1 : evset(2, 1) = 0
Evaluate the posteriori probabilities of consequences of gtop.
CALL REPROB3(ctop, zcal)
done(gtop) = 1
Access the fault tree data file at record no. gtop
gtop=CVI(cau$(1))
cq = 1
Yes
Is g$ = "DIR" ?
No
Obtain the addresses of the branches of gtop and store in row 1 of evset. evset(1,1) contains the value for the number of branches stored in evset. Store content of g$ in gt$.
Is evset(1, 1) = 0 ?
Yes
No
Evaluate the posteriori probability of each node in row 1 of evset which is an input to the gate stored in gt$.
CALL REPROB1(gt$)
Replace all nodes in row 1 of evset by their branches.
No
Is there any replacements ?
Yes
Remove repetition of nodes in row 1 of evset.
CALL REMREP(1)
Evaluate the posteriori probability of each node in row 1 of evset which is an input to a gate different from the gate stored in gt$.
CALL REPROB2
ka = 1
Yes
Is done(ka)=1?
No
Obtain the a priori probability of the event at address ka and store it in the field cpr$ at the same address. Set done(ka)=1
ka = ka + 1
Is ka > total no. of records in the fault tree data file ?
Yes
RETURN

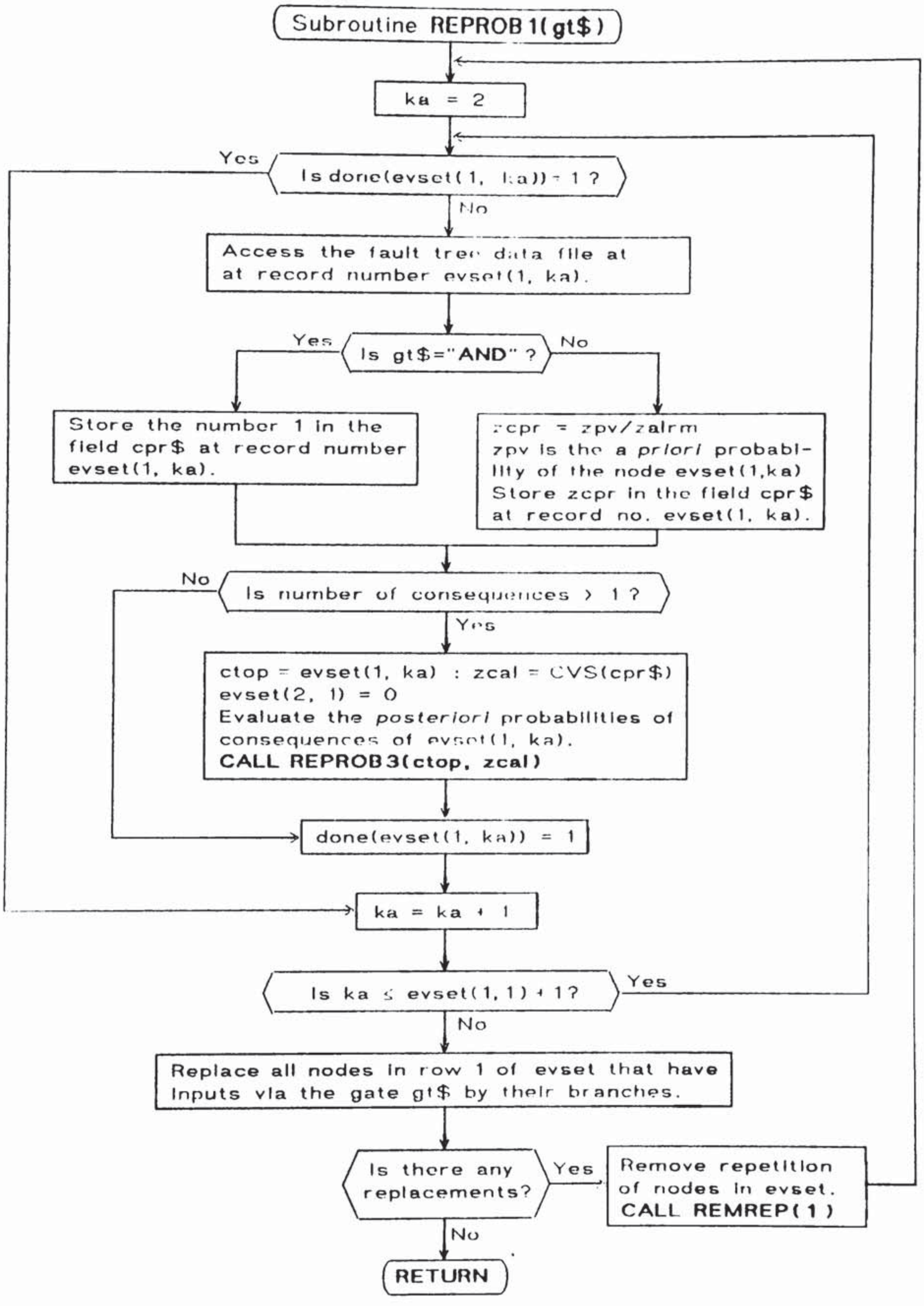
Figure 5.10: Flowchart of Subroutine CONPROB

gate as **gt$**. The flowchart for subroutine **REPROB1** is shown in figure 5.11.

Subroutine **REPROB2** evaluates the *posteriori* probabilities of nodes in row 1 of evset which are either inputs via a gate different from that contained in **gt$** or inputs to the alarm event via combinations of **AND** and **OR** gates. For each unevaluated node in row 1 of evset, first, the probability of the intersection between the alarm event and the node is evaluated by calling subroutine **PPROB**. Subroutine **PPROB** operates on 2 two dimensional arrays **s1** which is empty and **s2** containing the address of the node and the address of the alarm event. The posteriori probability of the node is determined by dividing the result obtained from the call to **PPROB** by the a priori probability of the alarm event.

When all the nodes in **evset** have been evaluated, any node that has inputs via whatever gate type is replaced by the addresses of its branches. The process of evaluating the posteriori probabilities for the new nodes in **evset** is repeated until all the nodes in row 1 of **evset** represent primary events. The flowchart for subroutine **REPROB2** is shown in figure 5.12.

Subroutine **REPROB3** is called by **CONPROB**, **REPROB1** and **REPROB2** for the purpose of evaluating the posteriori probabilities of the consequences of an event. **REPROB3** is called after the posteriori of an event has been evaluated. The parameters passed to **REPROB3** are **ctop** and **zcal** which contain the address and the *posteriori* probability of the event respectively. The function of **REPROB3** is to



Subroutine REPROB1(gt$)
ka = 2
Yes
Is done(evset(1, ka)) = 1 ?
No
Access the fault tree data file at at record number evset(1, ka).
Yes
Is gt$="AND" ?
No
Store the number 1 in the field cpr$ at record number evset(1, ka).
zcpr = zpv/zalrm
zpv is the a priori probability of the node evset(1,ka)
Store zcpr in the field cpr$ at record no. evset(1, ka).
No
Is number of consequences > 1 ?
Yes
ctop = evset(1, ka) : zcal = CVS(cpr$)
evset(2, 1) = 0
Evaluate the posteriori probabilities of consequences of evset(1, ka).
CALL REPROB3(ctop, zcal)
done(evset(1, ka)) = 1
ka = ka + 1
Is ka ≤ evset(1, 1) + 1?
Yes
No
Replace all nodes in row 1 of evset that have inputs via the gate gt$ by their branches.
Is there any replacements?
Yes
Remove repetition of nodes in evset.
CALL REMREP(1)
No
RETURN

Figure 5.11 : Flowchart of Subroutine REPROB1

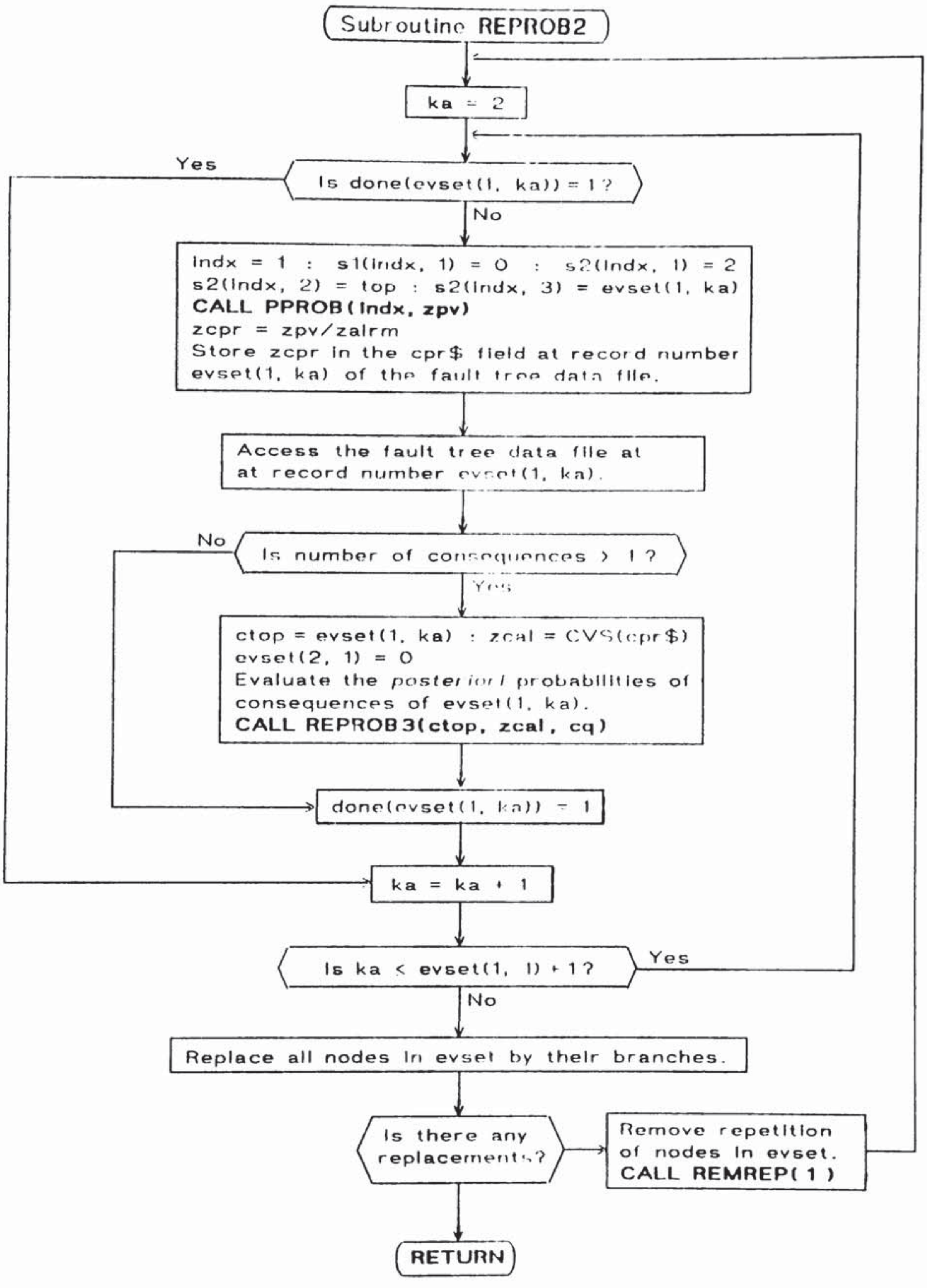

Subroutine REPROB2
ka = 2
Is done(evset(1, ka)) = 1 ?
Yes
No
indx = 1 : s1(indx, 1) = 0 : s2(indx, 1) = 2
s2(indx, 2) = top : s2(indx, 3) = evset(1, ka)
CALL PPROB(indx, zpv)
zcpr = zpv/zalrm
Store zcpr in the cpr$ field at record number evset(1, ka) of the fault tree data file.
Access the fault tree data file at at record number evset(1, ka).
Is number of consequences > 1 ?
No
Yes
ctop = evset(1, ka) : zcal = CVS(cpr$)
evset(2, 1) = 0
Evaluate the posteriori probabilities of consequences of evset(1, ka).
CALL REPROB3(ctop, zcal, cq)
done(evset(1, ka)) = 1
ka = ka + 1
Is ka < evset(1, 1) + 1?
Yes
No
Replace all nodes in evset by their branches.
Is there any replacements?
Remove repetition of nodes in evset.
CALL REMREP(1)
RETURN

Figure 5.12: Flowchart of Subroutine **REPROB2**

obtain the addresses of the consequences of **ctop** and store them in row 2 of the array **evset**. Then subroutine **REPROB4** is called to calculate the *posteriori* probability of every node in row 2 of **evset**. The parameters passed to **REPROB4** are the row number of **evset** where the nodes to be evaluated are stored, and **zcal**. At this stage, the row number passed to **REPROB4** is 2. On returning from **REPROB4**, the flag **done(ctop)** is set equal to 1.

Subroutine **REPROB3** then evaluates the *posteriori* probabilities of the consequences of the nodes in row 2 of evset. In fact, when **REPROB3** returns to its caller, every consequent event which has **ctop** as an input, either directly via an **AND**, **OR** or a **DIR** gate; or through a chain of events via combinations of the three gate types, would have been evaluated. The way **REPROB3** achieve this is shown in a flowchart in figure 5.13.

Subroutine **REPROB4** does the actual determination of the *posteriori* probability of consequent events. The method of determining the *posteriori* probability of a consequent event depends on its gate type obtained from the fault tree data file. If the gate type is a **DIR**, or an **OR** with **zcal** equal to 1, then its *posteriori* probability is set equal to **zcal**. Otherwise, subroutine **PPROB** is called to calculate the probability of the intersection between the alarm event and the consequent event. The result obtained is then divided by the *a priori* probability of the alarm event to give the *posteriori* probability of the consequent event. The flowchart for subroutine **REPROB4** is shown in figure 5.14.

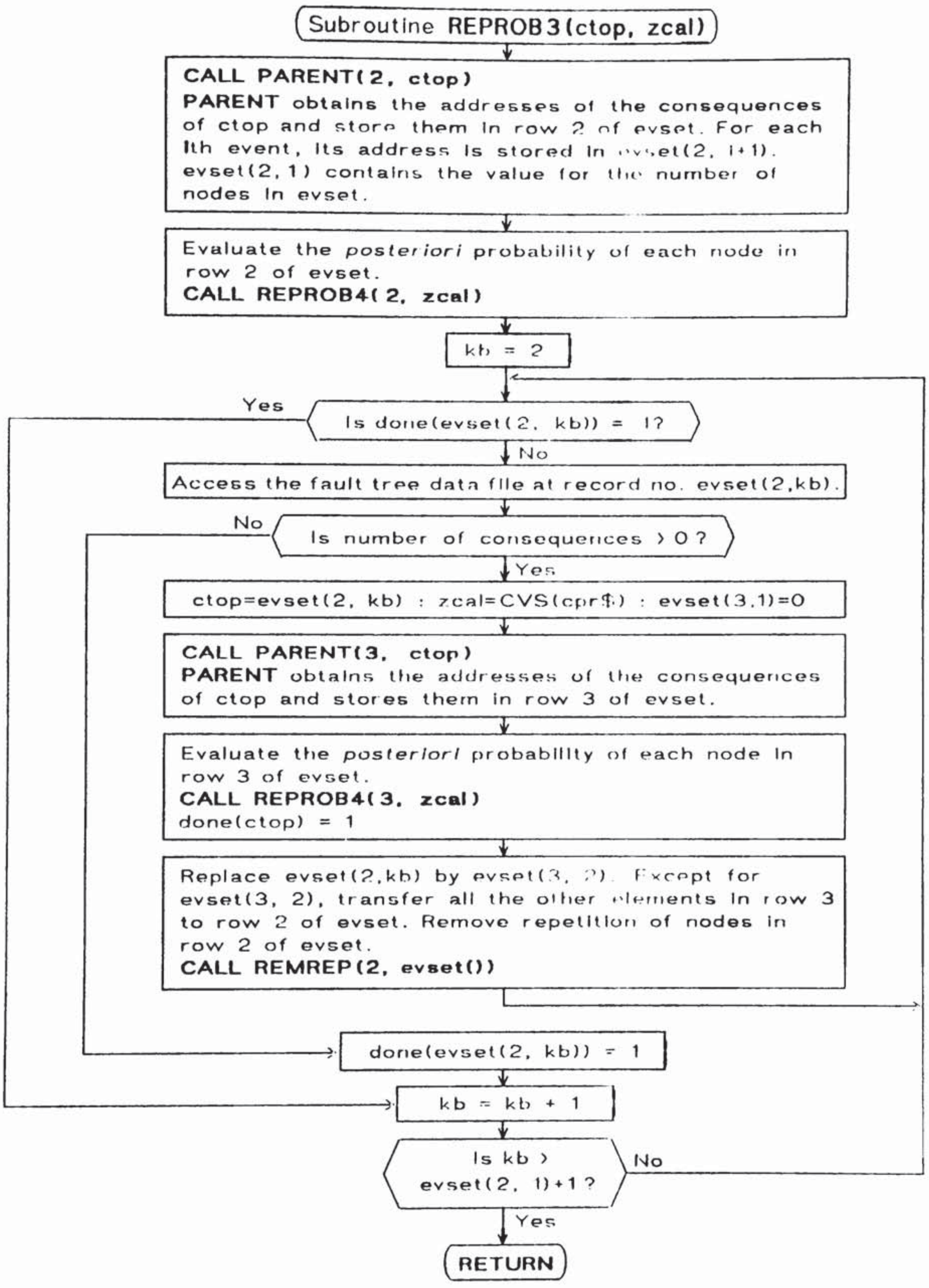

Subroutine REPROB3(ctop, zcal)
CALL PARENT(2, ctop)
PARENT obtains the addresses of the consequences of ctop and store them in row 2 of evset. For each Ith event, its address is stored in evset(2, i+1). evset(2,1) contains the value for the number of nodes in evset.
Evaluate the posteriori probability of each node in row 2 of evset.
CALL REPROB4(2, zcal)
kb = 2
Yes
Is done(evset(2, kb)) = 1?
No
Access the fault tree data file at record no. evset(2,kb).
No
Is number of consequences > 0 ?
Yes
ctop=evset(2, kb) : zcal=CVS(cpr$) : evset(3,1)=0
CALL PARENT(3, ctop)
PARENT obtains the addresses of the consequences of ctop and stores them in row 3 of evset.
Evaluate the posteriori probability of each node in row 3 of evset.
CALL REPROB4(3, zcal)
done(ctop) = 1
Replace evset(2,kb) by evset(3, 2). Except for evset(3, 2), transfer all the other elements in row 3 to row 2 of evset. Remove repetition of nodes in row 2 of evset.
CALL REMREP(2, evset())
done(evset(2, kb)) = 1
kb = kb + 1
Is kb > evset(2, 1)+1 ?
No
Yes
RETURN

Figure 5.13: Flowchart of Subroutine REPROB3

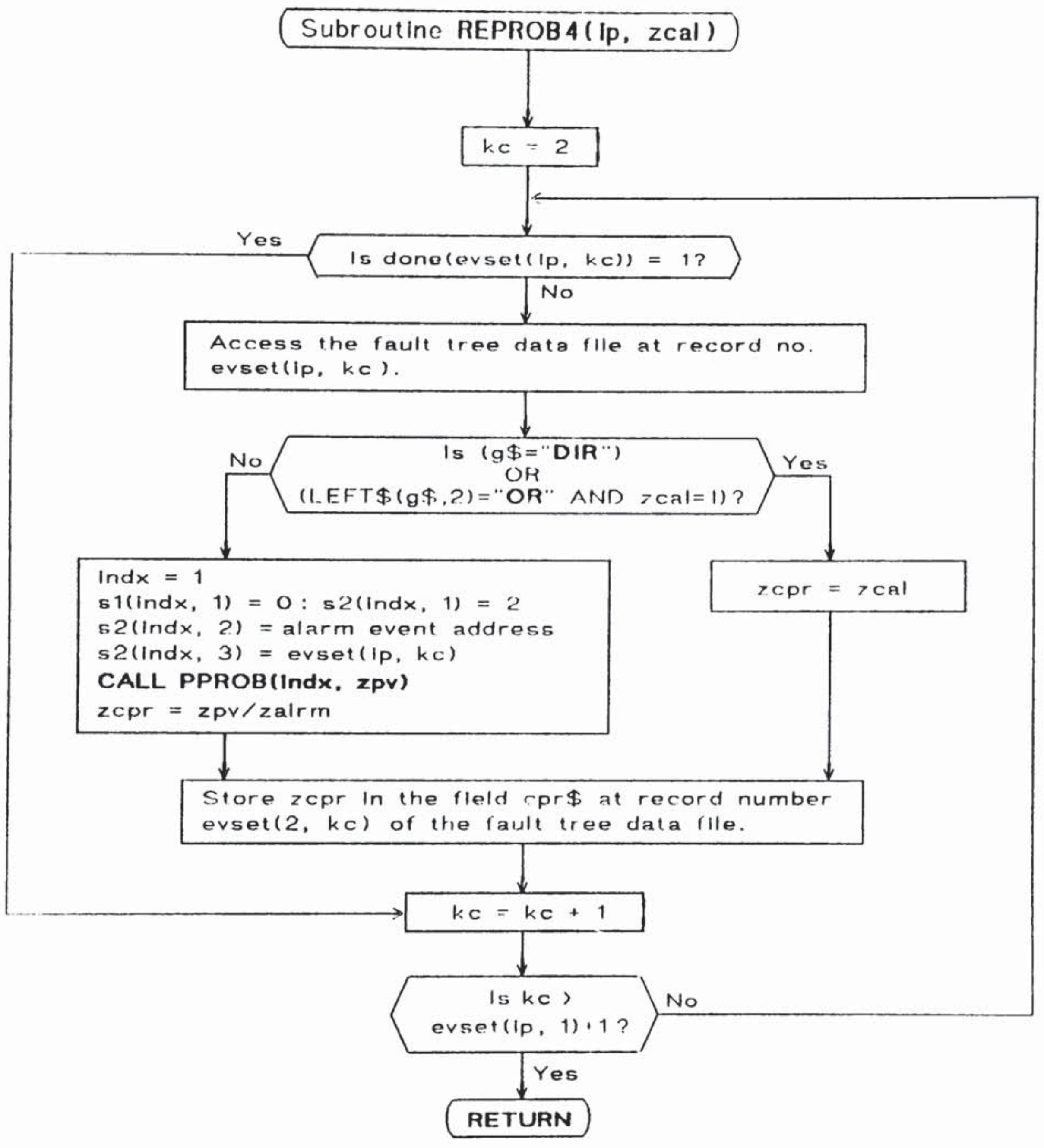
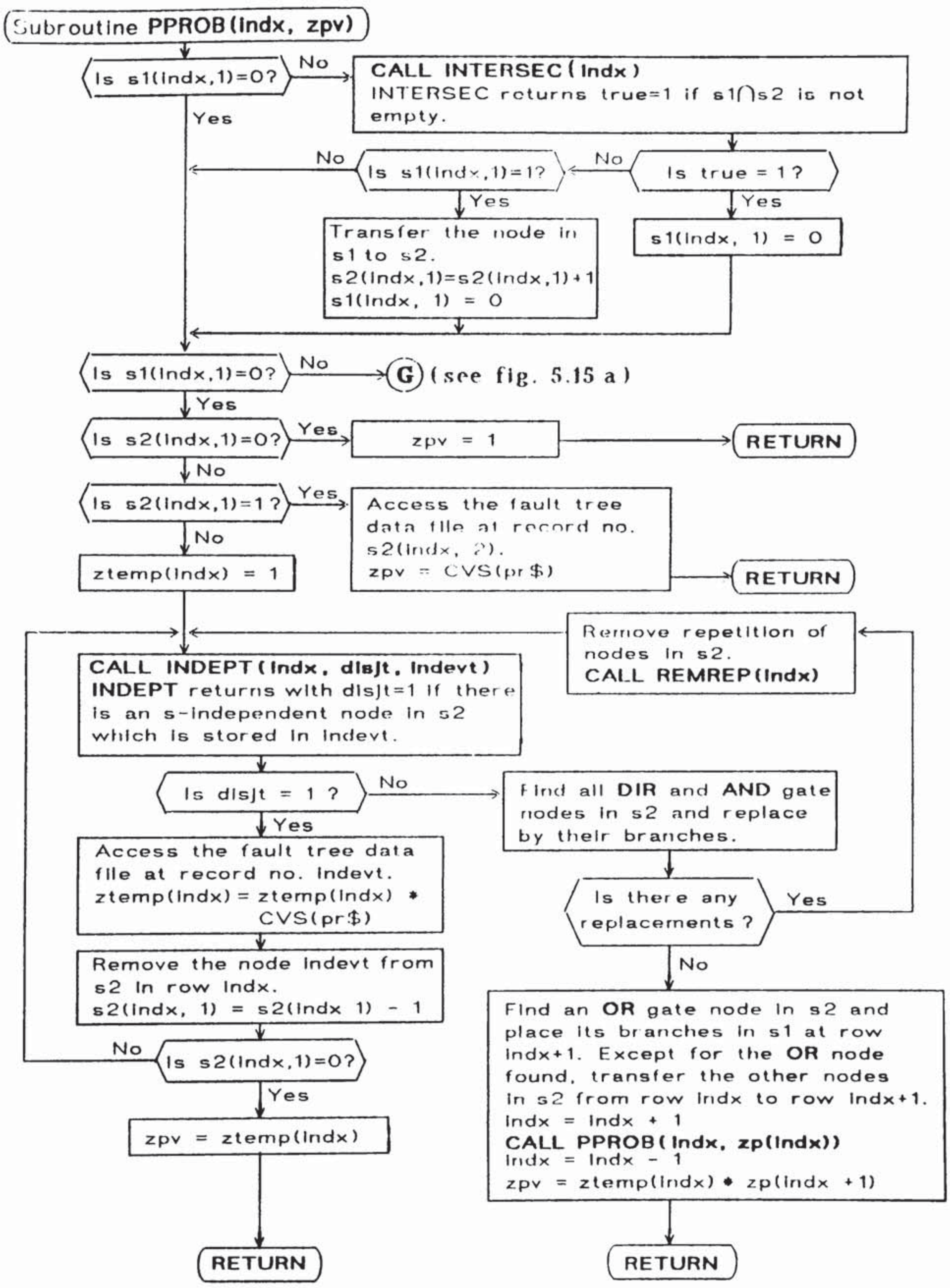

Subroutine REPROB4(lp, zcal)
kc = 2
Is done(evset(lp, kc)) = 1?
Yes
No
Access the fault tree data file at record no. evset(lp, kc).
Is (g$="DIR") OR (LEFT$(g$,2)="OR" AND zcal=1)?
No
Yes
Indx = 1
s1(Indx, 1) = 0 : s2(Indx, 1) = 2
s2(Indx, 2) = alarm event address
s2(Indx, 3) = evset(lp, kc)
CALL PPROB(Indx, zpv)
zcpr = zpv/zalrm
zcpr = zcal
Store zcpr in the field cpr$ at record number evset(2, kc) of the fault tree data file.
kc = kc + 1
Is kc > evset(lp, 1)+1?
No
Yes
RETURN

Figure 5.14: Flowchart of Subroutine REPROB4

5.4.4 : Description of Subroutine PPROB

The algorithm used in subroutine **PPROB** is similar to **PROB** described earlier. The only difference is when an independent secondary event is encountered during the recursion, there is no need to check if its *a priori* probability has been evaluated, which is a requirement in **PROB**. If the *a priori* probability of the secondary event is required, it is obtained from the fault tree data file. Apart from this difference, the rest of the **PPROB** program is similar to **PROB**. The flowchart for subroutine **PPROB** is shown in figure 5.15.



Subroutine PPROB(Indx, zpv)
Is s1(Indx,1)=0?
No
Yes
CALL INTERSEC(Indx)
INTERSEC returns true=1 if s1∩s2 is not empty.
No
Is s1(Indx,1)=1?
No
Is true = 1?
Yes
Yes
Transfer the node in s1 to s2.
s2(Indx,1)=s2(Indx,1)+1
s1(Indx, 1) = 0
s1(Indx, 1) = 0
Is s1(Indx,1)=0?
No
G (see fig. 5.15 a)
Yes
Is s2(Indx,1)=0?
Yes
zpv = 1
RETURN
No
Is s2(Indx,1)=1?
Yes
Access the fault tree data file at record no. s2(Indx, 2).
zpv = CVS(pr$)
RETURN
No
ztemp(Indx) = 1
Remove repetition of nodes in s2.
CALL REMREP(Indx)
CALL INDEPT(Indx, disjt, Indevt)
INDEPT returns with disjt=1 if there is an s-independent node in s2 which is stored in Indevt.
Is disjt = 1 ?
No
Find all DIR and AND gate nodes in s2 and replace by their branches.
Yes
Access the fault tree data file at record no. Indevt.
ztemp(Indx) = ztemp(Indx) * CVS(pr$)
Is there any replacements ?
Yes
No
Remove the node Indevt from s2 in row Indx.
s2(Indx, 1) = s2(Indx 1) - 1
No
Is s2(Indx,1)=0?
Yes
zpv = ztemp(Indx)
Find an OR gate node in s2 and place its branches in s1 at row Indx+1. Except for the OR node found, transfer the other nodes in s2 from row Indx to row Indx+1.
Indx = Indx + 1
CALL PPROB(Indx, zp(Indx))
Indx = Indx - 1
zpv = ztemp(Indx) * zp(Indx + 1)
RETURN
RETURN

Figure 5.15: Flowsheet of Subroutine PPROB

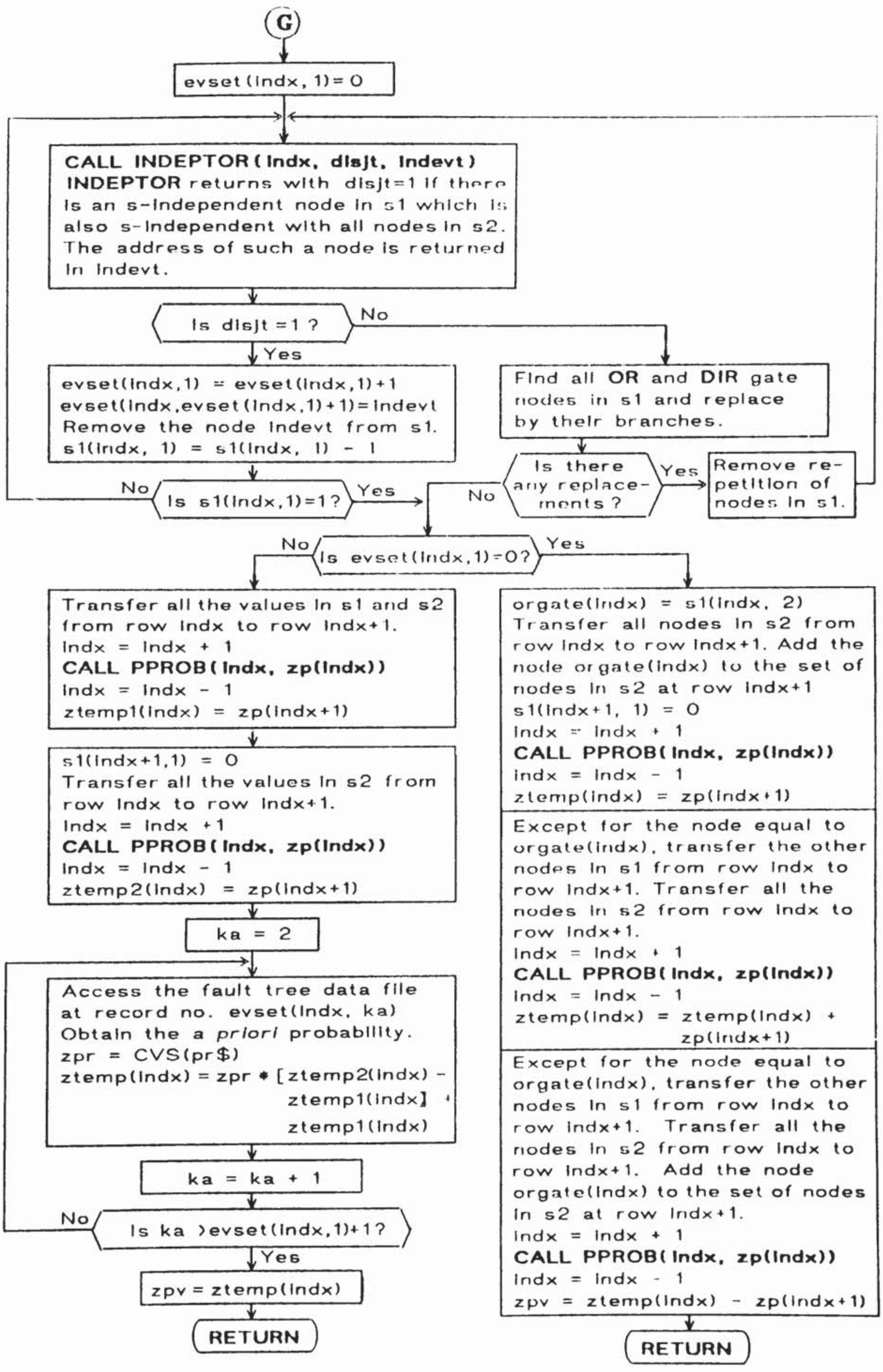

G
evset(indx, 1)= 0
CALL INDEPTOR(indx, disjt, indevt)
INDEPTOR returns with disjt=1 if there is an s-independent node in s1 which is also s-independent with all nodes in s2. The address of such a node is returned in indevt.
Is disjt =1 ?
No
Yes
evset(indx,1) = evset(indx,1)+1
evset(indx,evset(indx,1)+1)=indevt
Remove the node indevt from s1.
s1(indx, 1) = s1(indx, 1) - 1
Find all OR and DIR gate nodes in s1 and replace by their branches.
Is there any replacements ?
Yes
No
Remove repetition of nodes in s1.
No
Is s1(indx,1)=1?
Yes
No
Is evset(indx,1)=0?
Yes
Transfer all the values in s1 and s2 from row indx to row indx+1.
indx = indx + 1
CALL PPROB(indx, zp(indx))
indx = indx - 1
ztemp1(indx) = zp(indx+1)
s1(indx+1,1) = 0
Transfer all the values in s2 from row indx to row indx+1.
indx = indx +1
CALL PPROB(indx, zp(indx))
indx = indx - 1
ztemp2(indx) = zp(indx+1)
ka = 2
Access the fault tree data file at record no. evset(indx, ka)
Obtain the a priori probability.
zpr = CVS(pr$)
ztemp(indx) = zpr * [ztemp2(indx) - ztemp1(indx)] + ztemp1(indx)
ka = ka + 1
No
Is ka)evset(indx,1)+1?
Yes
zpv = ztemp(indx)
RETURN
orgate(indx) = s1(indx, 2)
Transfer all nodes in s2 from row indx to row indx+1. Add the node orgate(indx) to the set of nodes in s2 at row indx+1
s1(indx+1, 1) = 0
indx = indx + 1
CALL PPROB(indx, zp(indx))
indx = indx - 1
ztemp(indx) = zp(indx+1)
Except for the node equal to orgate(indx), transfer the other nodes in s1 from row indx to row indx+1. Transfer all the nodes in s2 from row indx to row indx+1.
indx = indx + 1
CALL PPROB(indx, zp(indx))
indx = indx - 1
ztemp(indx) = ztemp(indx) + zp(indx+1)
Except for the node equal to orgate(indx), transfer the other nodes in s1 from row indx to row indx+1. Transfer all the nodes in s2 from row indx to row indx+1. Add the node orgate(indx) to the set of nodes in s2 at row indx+1.
indx = indx + 1
CALL PPROB(indx, zp(indx))
indx = indx - 1
zpv = ztemp(indx) - zp(indx+1)
RETURN

Figure 5.15 a (continuation of fig. 5.15)

CHAPTER SIX

6. MODEL FOR TESTING THE DIAGNOSIS PACKAGE

6.1 : Introduction

The fault diagnosis package was tested on a model of a pilot distillation plant. During the period of this research work, the pilot plant was used by T.O. Folami, a research colleague, who was investigating the application of advance control methods on a distillation column. With his help and others [116], a HAZOP study was made on the pilot plant to provide data for the diagnosis package. A description of the pilot plant and a discussion on the HAZOP study is given in this chapter.

6.2 : The Pilot Distillation Plant

The pilot distillation plant was donated to the Department of Chemical Engineering at the University of Aston by IBM UK Limited. It has been used in a number of research projects including Daie [112] Shafie [113] and Folami [114]. Folami has made some modifications to the plant to overcome problems faced by Daie and Shafie, so as to suit the requirements for his research work. A schematic diagram of the pilot plant is shown in figure 6.1. The material that Folami used for the distillation is a binary mixture of Trichloroethylene and Tetrachloroethylene.

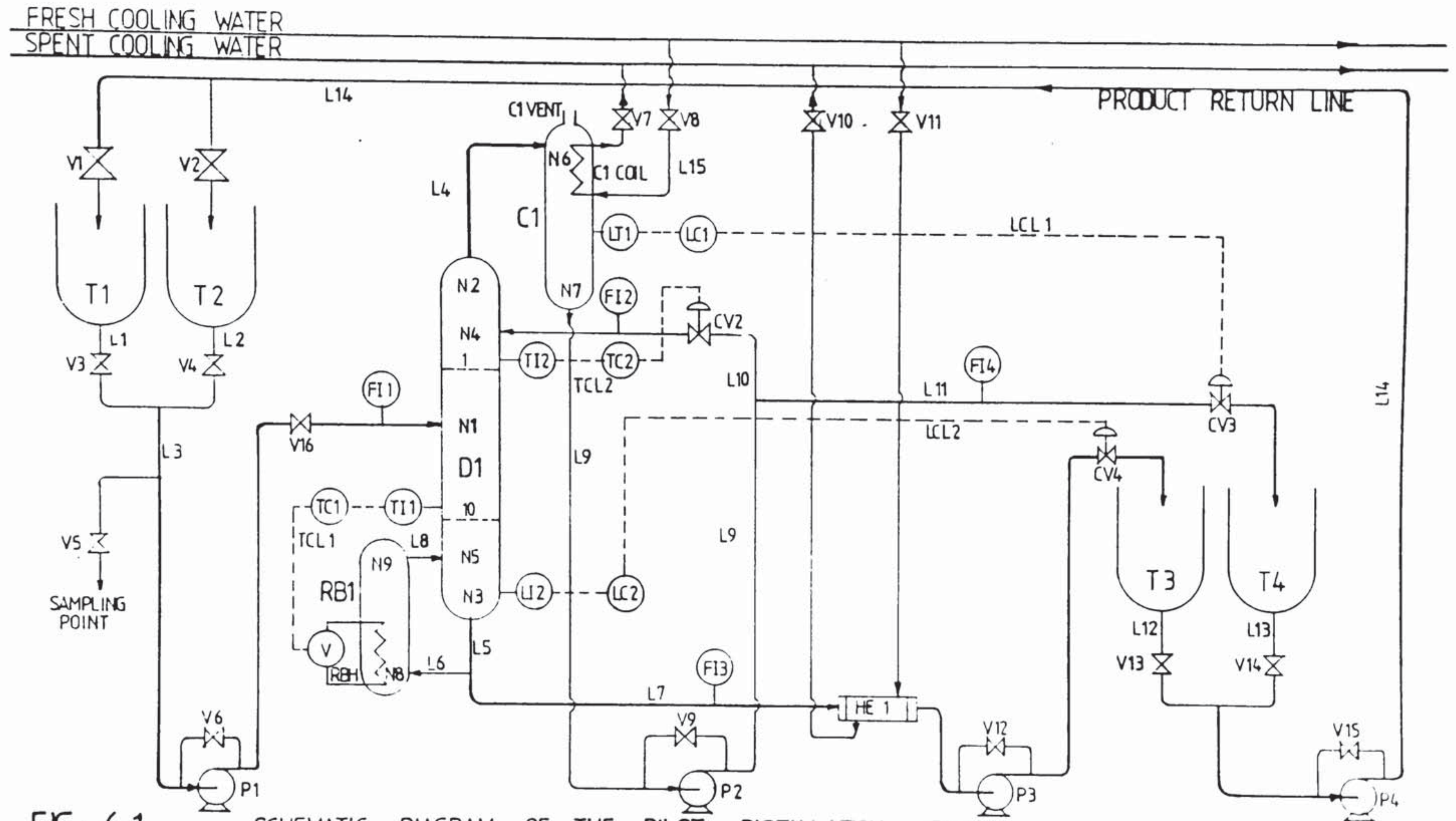


FIG: 6·1 SCHEMATIC DIAGRAM OF THE PILOT DISTILLATION PLANT

The distillation column is made of a 0.0762 m. (3 inch) outside diameter glass tube with 10 sieve trays. The length of the column is 1.2 m. of which the enriching and stripping sections are 0.65 m. and 0.55 m. long respectively. The feed is introduced into the column at the seventh tray. Each tray has 145 holes of 0.00011 m. diameter and the spacing between each tray is 0.08 m. The height of the weir in each tray is 0.0003 m. and the diameter of the downcomer is 0.0105 m.

There are two cylindrical glass tanks each of 30 litres capacity situated above the feed entry point to hold the feed. The feed comes from either one of the feed tanks when the column is being operated. Two product tanks are available to hold the top and bottom products from the column. To enable continuous operation, four 60W Stuart Turner centrifugal pumps are used to deliver

1) the feed into the column,

2) the distillate back to the top of the column as reflux and to the top product tank,

3) the raffinate to the bottom product tank, and

4) mixture from the top and bottom product tanks to the feed tank.

The feed, reflux, bottoms and distillate flowrates required for the operation of the column range from 1 to 15 litres per hour. However, each of the pumps could deliver in excess of 140 litres per hour across nominal heads of 2 to 3 metres. Since the required flowrates are much smaller, each pump is fitted with a re-circulation line with a valve to prevent build up of pressure in the valve downstream of the pump and to reduce mechanical strain on the pump.

A 0.0762 m. outside diameter standard glass condenser is arranged to condense the vapour from the top of the column into a 0.0762 m. outside diameter glass reflux drum connected directly below it. The condenser is connected to the top of the column by a 1 metre long by 0.0762 m. outside diameter glass tube called the vapour line.

The reboiler is of the thermosyphon type arrangement where an electrical firerod cartridge is used to vapourize the bottom liquid. The reboiler drum is made of a 3.25 inch outside diameter, 10 SWG stainless steel pipe. The length of the reboiler drum is 0.38 m. The nominal holdup of the reboiler drum during operation of the column is 1.5 litres. The vapouriser or the heating arm of the reboiler is made of a 2 inch outside diameter, 16 SWG stainless steel pipe and is 0.44 m long. Liquid flows from the bottom of the reboiler drum to the heating arm via a 3/8 inch outside diameter 16 SWG stainless steel pipe. Vapour produced from the heating arm flows back into the column via a 0.5 inch outside diameter 16 SWG stainless steel pipe. The firerod cartridge heater which is 8 inches long and 0.5 inch diameter is located centrally in the heating arm in a 0.46 m long by 5/8 inch outside diameter 16 SWG long stainless steel pipe. The bottom liquid is heated in the annular space between the pipe holding the heater and the heating arm pipe.

During operation of the pilot plant, it is essential that the level of liquid in the heating arm as well as in the reboiler drum must cover the whole length of the firerod cartridge heater so as to prevent the heater being burnt out. It is also essential that the level of liquid in the reboiler drum must not be above the vapour entry point which may prevent the vapour from the heating arm entering the distillation column. The blockage of the line carrying the vapour to the column will increase the pressure in the heating arm.

The piping around the column such as the feed line, the reflux line, the top product line and the product line is made of 1/2 inch outside diameter 16 SWG stainless steel pipes.

6.3 : Instrumentation on the Pilot Plant

The pilot distillation plant was used by Folami to study the applicability of advanced digital control techniques for real-time process control. For this purpose, a computer has been used to develop the control algorithms and also to control the plant. The computer used is a System96 Level II Microcomputer from Measurement Systems Limited. The central processing unit of the System96 is the Motorola 6809, an 8 bit microprocessor with an operating speed of 2 Mhz. The operating system used is OS9 which is modelled after the UNIX operating system of Bell Laboratories. The interface by which the System96 acquires data from the pilot plant and controls its operation is called the Monolog. The Monolog is a separate unit which houses the analogue to digital converters, the digital to analogue converters, signal conditioners and digital input/output devices. The connection between the Monolog and System96 is via an RS323 serial line. The Monolog is driven by the host computer, using its fixed internal program.

There are four control loops on the pilot plant. From figure 6.1, the control loops are labelled LCL1, LCL2, TCL2 and TCL1. Their respective functions are:

1) to control the liquid level in the reflux drum located below the condenser,

2) to control the liquid level in the reboiler drum,

3) to control the reflux flowrate into the distillation column, and

4) to control the rate of heat output to the reboiler.

The level in the reflux and the reboiler drums are monitored by two differential pressure tranducers. Each transducer measures the pressure difference between two vertical points which is proportional to the liquid level in the drum. The output of the transducer is an analoque voltage between 0 to 5 volts which corresponds to a level between 0 to 20 inches of water. Assuming that the relative density of the mixture in either the reflux or the reboiler drum is 1.5, the equivalent range of liquid level measured by the transducer is between 0 to 13.3 inches.

The flowrates of the feed, reflux, distillate and bottom products are measured by turbine flowmeters. The body and rotor of each turbine flowmeter is made of acetal rubber. On the rotor is fitted with 3 stainless ceramic magnets. The metering principle is velocity counting where the output is electrical pulses. The frequency of the electrical pulses determines the flowrate which is linear in their relationship. The output frequency is converted to an analogue voltage signal between the range 0 to 5 volts by a frequency to voltage converter before being converted to a digital signal by the Monolog.

The control valves used to automatically control the reflux flowrate and the levels in the reflux and reboiler drums are miniature, pneumatic, air to open types which can be fitted on 1/4 or 1/2 inch outside diameter pipes. An air pressure of 15 psig. will fully open the control valve. However, for the valves to operate satisfactorily, an air supply of 20 psig. is required. The signal from the Monolog to manipulate the valve stem position is in digital form. In between the Monolog and each valve, there are two converters before the actual adjustment of the control valve is done. First, the digital signal is converted to a current signal in the range 4 to 20 mA. The current signal is then converted to a pressure signal in the range of 3 to 15 psig.

The firerod cartridge heater in the heating arm of the reboiler is operated by a solid state switch and a digital control timer. When the switch is on, the output power of the heater is 2 KW. The digital control timer sets the amount of time the heater is to be switched on. The heater operates in cycles with periods of 4 seconds. If it is required for the heater to output a constant 2 KW of heat, the control timer will switch on the voltage supply to the heater for a full 4 seconds after which the voltage supply will be switched off and on again instantaneously to begin the next cycle. Thus to output a mean value of X KW of heat, where X is less than 2, the amount of time the heater will be switched on and off in one cycle is 2X and 4-2X seconds respectively.

On each tray and the reboiler are thermometer wells for insertion of thermometers to measure the temperatures of the liquid during operation. The thermometers are Nickel-Chromium Aluminium thermocouples which have an accuracy of 0.01°C. The temperatures that are measured during operation of the column are:

1) the reflux into the column,

2) the liquid on the top tray,

3) the liquid on the second tray,

4) the feed,

5) the liquid on the botom tray, and

6) the liquid in the reboiler drum.

It is assumed that the liquid temperatures measured are their saturated points particular to the composition of the liquid. Since the liquid is a binary mixture, the measured temperatures are used as concentration monitors of the components in the distillation column.

6.4 : The Physical and Hazardous Properties of the Distillation Components

The liquid used by Folami in his study of advanced control strategies on a distillation column is a binary mixture of Trichloroethylene and Tetrachloroethylene which are colourless liquids that emit a characteristic odour of chloroform. Their boiling points are 87°C and 121°C respectively at 1 atmosphere pressure. The liquid specific gravities are 1.46 and 1.63 at 20°C respectively. The flash point of Trichloroethylene is 32°C but is considered as practically non-flammable. Tetrachloroethylene on the other hand is non-flammable, non-explosive and will not support combustion.

Both Trichloroethylene and Tetrachlorothylene are toxic when inhaled, ingested by mouth or after prolonged contact with the skin and mucous membranes [111]. Exposures of the vapour at concentrations of more than 200 ppm cause irritation and burning of the eyes and irritation of the nose and throat. There may be vomiting, nausea, drowsiness, an attitude of irresponsibility, and even an appearance resembling alcoholic intoxication. They can also act as an anesthetic through the inhalation of excessive amounts within a short time. They can cause dermatitis when repeated or prolonged contact with the skin. The dermatitis is preceded by a reddening and burning of the skin. The skin becomes dry and rough, due largely to the removal of skin oils by the material. The skin then cracks easily and is readily susceptible to infection.

The recommended threshold limit value of vapour in air for both material is 50 ppm [111]. The short term inhalation limits for Trichloroethylene and Tetrachloroethylene should be 200 ppm for 30 minutes and 100 ppm for 60 minutes respectively.

6.5 : The HAZOP Study on the Pilot Plant

6.5.1 : Introduction

A HAZOP study on the pilot distillation plant was made with the main objective of obtaining data for testing the diagnosis package developed by the author of this research work. The study also provides an insight on the operability of the pilot plant that Folami has modified for his research work. The following sections discuss what are the serious malfunctions that must be avoided during the operation of the plant and also the outcome of some of the HAZOP study on the plant.

6.5.2 : Malfunctions and Hazards To Be Avoided During Operation of Pilot Plant

One serious malfunction which must be avoided while operating the pilot plant is for the reboiler drum to go dry or the liquid level to drop below a certain limit. When this happens, the firerod cartridge heater will burn out since there is no load to take off the heat output. The burn out may damage the casing holding the firerod cartridge. The safe limit for operation is for the liquid level in the reboiler drum to be 6 cm. above the top of the firerod cartridge heater.

The liquid in the reboiler drum must not be at a such a level as to cover the entry point of the vapour to the column. This will restrict the flow of vapour from the heating arm, resulting in some of it condensing in the reboiler drum, leaving less vapour to flow upwards to the trays to effect mass transfer. The effective limit of operation is for the level in the reboiler to be 6 cm. below the vapour entry point to the column.

Another malfunction which must be avoided is to allow the reflux drum to go dry. The reflux drum may go dry if there is no condensation in the condenser, i.e. all the vapour is escaping to the atmosphere. If this happens, there will be no liquid flowing back into the column as reflux to effect the distillation process in the enriching section. There will, however, be contact between liquid and vapour in the stripping section below the feed tray. The temperature of the vapour from the feed tray going to the malfunctioning condenser will be high since there is no liquid flowing downwards from the top tray to cool it down. Such escaping vapour is a health hazard to the people around the pilot plant. The symptoms of exposure to the vapour have been discussed previously.

When the pumps are running, they must not go dry, i.e. there must be always liquid flowing in the pipes served by the pumps. If there is air in the pipes, the pumps may cavitate and produce inconsistent flow and eventually become damaged.

One hazard that must be avoided is the leakage of boiling liquid from the pilot plant. As discussed earlier, the liquid mixture is harmful to the skin.

6.5.3 : Outcome From the HAZOP Study

The HAZOP study was done to record the cause and symptom equations of the pilot plant as discussed in chapter 3. The labels for lines, vessels, instruments, valves and nodes are those shown in figure 6.1. The index numbers to represent the guide words, property words and equipment failure mode are similar to those shown in tables 3.4 and 3.5. Table 6.1 shows the index to represent the components present in the operation of the pilot plant. Table 6.2 shows single and two letter symbols to represent certain plant equipment failure modes. Presented in this section, are the outcomes of the HAZOP study on the pilot distillation plant.

Table 6.1 : Indices To Represent Components in the Pilot Plant Model

Index	Component
1	Trichloroethylene
2	Tetrachloroethylene
3	Air
4	Bottoms liquid
5	Top vapour

Table 6.2 : Single and two letter symbols to represent certain plant equipment failure mode.

Symbol	Failure Mode
BO	"Burn Out" - applied to the reboiler heater.
OV	"Overflow" applied to the condenser vent.
VR	"Vapour Release" - applied to the condenser vent.
F	"Fouling" applied to the heat exchanger.
L	"Leaking" applied to pipes, vessels, valves and pumps.

Distillation column D1

Purpose: To distil a mixture of Tichloroethylene and Tetrachloroethylene into two separate products.

Lines into vessel D1

Line L3

Purpose : To feed a mixture of 40% Trichloroethylene and 60% Tetrachloroethylene into vessel D1 at a rate of approximately 12 litres per hour. Feed is at ambient temperature. During normal operation, vessel T1 supplies L3.

Cause equations for flow deviations in L3

1) L3(11) = L1(11) + V16(0) + P1(0) * V6(0) + L3(0) + FI1(1)

2) L3(12) = L1(12) + P1(0) + P1(-1) + P1(L) + L3(L) + L3(-1) + V16(-1) +
   V5(1) + FI1(1) + V6(L) + V16(L) + V5(L)

3) L3(13) = V16(1) + FI1(-1)

4) L3(16) = D1(33) * P1(0)

5) L3(144) = P1(-1) + V4(1)*T2(41)

Symptom equations for flow deviations in L3

6) L3(11) — N1(11) * N2(12) * N3(11)

7) L3(12) — N1(12) * N2(12) * N3(12)

8) L3(13) — N1(13) * N2(13) * N3(13)

9) L3(144) — N1(12) * N2(12) * N3(12)

Cause equation for temperature deviation in L3

The only feed temperature deviation that can seriously affect the performance of the distillation column is when the temperature of the feed is so low such that the vapour rising to the feed tray is condensed resulting in no vapour rising to the top tray.

10) L3(22) = T1(22)

Symptom equation for temperature deviation in L3

11) L3(22) — N1(22) * N2(11) * N3(13) * N3(531)

Cause equations for concentration deviations in L3

12) L3(521) = L1(521) + L2(521) * V4(1)

13) L3(531) = L1(531) + L2(531) * V4(1)

Symptom equations for concentration deviations in L3

14) L3(521) — N2(12) * N3(13)

15) L3(531) — N2(13) * N3(12)

Line L8

Purpose: To feed vapour at approximately 110°C from the heating arm of reboiler RB1 into the vessel D1.

Cause equations for flow deviations in L8

16) L8(11) = N9(11) + L8(0)

17) L8(12) = N9(12) + L8(-1) + L8(L)

18) L8(13) = N9(13)

19) L8(145) = N9(145)

Symptoms equations for flow deviations in L8

20) L8(11) – N5(11) * N2(11) * N3(13) * N3(531)

21) L8(12) – N5(12) * N2(12) * N3(13) * N3(531)

22) L8(13) – N5(13) * N2(13)

23) L8(145) – N3(531)

Cause equation for pressure deviation in L8

24) L8(33) = D1(33)

Symptom equation for pressure deviation in L8

25) L8(33) – N9(33)

Line L10

Purpose: To feed the distillate as reflux to the vessel D1. Flowrate of reflux is controlled by a temperature control loop, TCL2, so that temperature of the liquid on the top tray, i.e. TRAY1, is about 95°C.

Cause equations for flow deviations in L10

26) L10(11) = CV2(0) + L9(11) + L10(0) + FI2(1)

27) L10(12) = CV2(-1) + L10(-1) + L10(L) + L9(12) + FI2(1)

28) L10(13) = CV2(1) + FI2(-1)

29) L10(16) = D1(33)

Symptom equations for flow deviations in L10

30) L10(11) – N4(11) * N2(521)

31) L10(12) – N4(12) * N2(521)

32) L10(13) – N4(13) * N2(12)

Cause equation for pressure deviation in L10

33) L10(33) = D1(33)

Pipe junction between L9, L10 and L11

Purpose : To split flow of distillate in L9 as reflux flowing in L10 and the top product flowing in L11.

Cause equations for flow deviations in line L9

34) L9(11) = C1(41) + P2(0) * V9(0) + L9(0)

35) L9(12) = L9(-1) + L9(L) + P2(0) + P2(-1)

36) L9(13) = L11(13) + V9(0)

37) L9(16) = L10(16) * P2(0) * (L11(0) + CV3(0))

Symptom equation for flow deviation in L9

38) L9(16) — N7(16)

Cause equation for pressure deviation in line L9

39) L9(33) = (L10(0) + CV2(0)) * (L11(0) + CV3(0)) + L10(33) + V9(0)

Cause equation for concentration deviation in line L9

40) L9(521) = N7(521)

Vessel under consideration - distillation column D1

Cause equations for deviations in column D1

41) D1(41) = D1(L) + RB1(L) + N1(11) * N4(11)

42) D1(42) = D1(L) + RB1(L) + LCL2(-1)

43) D1(43) = N1(13) * N4(13) + LCL2(1)

44) D1(33) = L4(0) + C1(33)

Cause equations for temperature deviations on TRAY1 and TRAY10 in column D1.

45) TRAY1(22) = N1(22) + N4(13) + N5(11) + TCL2(-1)

46) TRAY1(23) = N4(11) + N4(12) * N1(11)) + TCL2(1)

47) TRAY10(22) = N1(22) + N5(11) + N1(13) + N4(13) + N5(12) + TCL1(-1)

48) TRAY10(23) = N5(13) + N1(11) + N4(12)) + TCL1(1)

Temperature control loop TCL1

Purpose : To control the temperature of fluid on the bottom tray in D1 such that it is around 115°C by regulating the heat output from the reboiler heater, RBH.

Cause equations for deviations in TCL1

49) TCL1(-1) = TI1(1) + TC1(-1)

50) TCL1(1) = TI1(-1) + TC1(1)

Temperature control loop TCL2

Purpose : To control the temperature of liquid on the top tray in column D1 by varying the reflux flowrate into the column through line L10. When temperature rises above the set point, it indicates that the composition of Trichloroethylene on the tray has decreased. The controller will increase the reflux flowrate to compensate for the increase in temperature.

Cause equations for deviations in TCL2.

51) TCL2(-1) = TI2(1) + TC2(-1)

52) TCL2(1) = TI2(-1) + TC2(1)

Level control loop LCL2

Purpose : To control the level in the reboiler drum as well as the level in the heating arm of the reboiler so as not to cover the vapour inlet from the heating arm and also not to expose the firerod cartridge heater. The level is controlled by opening or shutting the control valve CV4. The reboiler drum level is measured by LI2.

Cause equations for deviations of the level control loop LCL2

53) LCL2(-1) = LI2(1) + LC2(-1) + CV4(1)

54) LCL2(1) = LI2(-1) + LC2(1) + CV4(-1)

The reboiler RB1

Purpose : To vapourize part of the bottom product to be fed back into column D1. Heat input is via a firerod cartridge heater, RBH, which has a maximum output of 2 KW.

Lines to RB1

Line L6

Purpose : To feed part of the bottom product to RB1. During normal operation, it does not matter what flow rate is in line L6, as long as there is flow to keep the level of liquid in the reboiler about the same as the liquid level in column D1.

Cause equations for flow deviations in L6

55) L6(11) = L5(11) + L6(0)

56) L6(16) = RB1(33)

Symptom equations for flow deviations in L6

57) L6(11) — N8(11)

Cause equation for pressure deviation in L6

58) L6(33) = RB1 (33)

Pipe junction between L5, L6 and L7

Purpose : To splitthe bottom liquid flowing in L5 into reboiler liquid flowing in L6 and the bottom products flowing in L7.

Cause equation for flow deviation in line L5

59) L5(11) = L5(0)

Vessel under consideration - the reboiler RB1

Cause equations for deviations in RB1

60) RB1(41) = N8(11) + D1(41)

61) RB1(42) = D1(42)

62) RB1(43) = D1(43)

63) RB1(33) = N9(33)

64) RB1(71) = RBH(0)

65) RB1(72) = RBH(-1)

66) RB1(73) = RBH(1)

Symptom equations for deviations in RB1

67) RB1(41) — RBH(BO)

68) RB1(42) — RBH(BO)

69) RB1(43) – N9(145)

70) RB1(71) – N9(11)

71) RB1(72) – N9(12)

72) RB1(73) – N9(13)

The condenser and reflux drum C1

Purpose : To condense vapour coming from the top of column D1. The distillate falls into a reflux drum below the condenser where the liquid level is controlled by the level control loop LCL1.

Lines to C1

Line L4

Purpose : To convey vapour from top of D1 into C1

Cause equations for flow deviations in L4

73) L4(11) = N2(11) + L4(0)

74) L4(12) = N2(12) + L4(-1) + L4(L)

75) L4(13) = N2(13)

Symptom equations for flow deviations in L4

76) L4(11) – N6(11)

77) L4(12) – N6(12)

78) L4(13) – N6(13)

Cause equation for concentration deviation in L4

79) L4(521) = N2(521)

Symptom equation for concentration deviation in L4

80) L4(521) – N7(521)

Cause equation for pressure deviation in L4

81) L4(33) = C1(33)

Vessel under consideration - the condenser and reflux drum, C1

Cause equations for deviations in C1

82) C1(41) = L4(0) + C1(71) + C1(L)

83) C1(42) = N6(12) + C1(72) + C1(L) + LCL1(-1)

84) C1(43) = C1COIL(L) + N7(16) + LCL1(1)

85) C1(33) = C1VENT(0)

86) C1(71) = L15(11)

87) C1(72) = L15(12) + C1COIL(F)

Symptom equations for deviations in C1

88) C1(43) – C1VENT(OF)

89) C1(71) – C1VENT(VR)

90) C1(72) – C1VENT(VR)

Level control loop LCL1

Purpose : To control level in the reflux drum of C1 by varying the top product flow in L11.

Cause equations for deviations of LCL1

91) LCL1(-1) = LI1(1) + LC1(-1)

92) LCL1(1) = LI1(-1) + LC1(1)

Other lines and vessels

Line L15

Purpose: To convey cooling water into the cooling coil C1COIL in the condenser C1.

Cause equations for flow deviations in line L15

93) L15(11) = V7(0) + V8(0) + L15(0) + C1COIL(0)

94) L15(12) = V7(-1) + V8(-1) + L15(-1) + L15(L) + C1COIL(-1)

Lines L1 and L2

Purpose: To convey feed mixture from tank T1 or T2 to feed line L3.

Cause equations for deviations in L1 and L2

95) L1(11) = V3(0) + L1(0) + T1(41)

96) L1(12) = V3(-1) + L1(L) + L1(-1)

97) L1(521) = T1(521)

98) L1(531) = T1(531)

99) L2(521) = T2(521)

100) L2(531) = T2(531)

Vessels T1 and T2

Purpose : To hold the feed mixture.

Cause equations for deviations in T1 and T2

101) T1(521) = L14(521)

102) T1(531) = L14(531)

103) T2(521) = L14(521)

104) T2(531) = L14(531)

Line L14

Purpose : To convey mixture of top and bottom products of the distillation column D1 to the feed tank T1. The products are held in tanks T4 and T3 respectively.

Cause equations for deviations in line L14

105) L14(521) = V13(1) + V14(-1)

106) L14(531) = V13(-1) + V14(1)

Line L7

Purpose : To convey the bottom product form column D1 to holding tank T3.

Cause equations for deviations in line L7

107) L7(11) = N3(11) + P3(0) * V9(0) + CV4(0) + L7(0) + HE1(0) + FI3(1)

108) L7(12) = N3(12) + CV4(-1) + L7(-1) + L7(L) + HE1(-1) + HE1(L)
+ FI3(1)

109) L7(13) = N3(13) + V12(0) + CV4(1) + L6(16) + D1(33) + FI3(-1)

110) L7(531) = N3(531)

<u>Line L11</u>

Purpose : To convey the top product from the column D1 to holding tank T3.

<u>Cause equation for deviations in line L11</u>

111) L11(11) = N6(11) + L9(11) + L11(0) + CV3(0)

112) L11(12) = N6(12) + L11(-1) + L11(L) + CV3(-1)

113) L11(13) = N6(13) + CV3(1) + L10(16)

114) L11(521) = L9(521)

6.5.4: Assigning Probability Values to Primary Events

Primary events can be either be due to mechanical failures or human error. For events due to mechanical failure, the failure probabilities are obtained by using equation 4.15 which is reprinted below.

$$F(t) = 1 - \exp(-\lambda t) \qquad .....4.15$$

The diagnosis package was tested using an arbitrary value of t equals to 1 year. The failure rate, λ, for a particular plant item is obtained from table C.1 in Appendix C. If λ is 0.1 per year or less, then from equation 4.18 in chapter 4, the failure probability is approximately the same as the failure rate, i.e.

$$F(1) = \lambda \qquad .....6\ 1$$

The probability data for primary events due to human error are obtained from table C.2 in Appendix C.

From the result of the HAZOP study of the pilot distillation plant, the primary events are those involving:

1) manual valves not at their actual settings;

2) control valves stuck open or close;

3) pumps not working or cavitating;

4) lines fully or partially blocked;

5) lines and vessels leaking;

6) flowmeters, level meters and thermometers not indicating and transmitting to the controller the correct values; and

7) wrong setting of set-points.

Table 6.3 shows the probability values assigned to the various primary events.

Table 6.3 : 1-Year Probabilities For Primary Events of The Pilot Distillation Plant

Events	Remarks	1 year/event probability.
V3(0), V3(-1), V6(0), V7(0), V7(-1), V8(0), V8(-1), V9(0), V12(0)	During normal operation, these valves should be fully open. Failure to open these valves fully is an error on the part of the operator. Thus, the probability value assigned to these events is that of failure to follow instructions.	0.065
V4(1), V5(1)	These valves should be fully closed. The probability value assigned to these events is that of failure to close valve completely.	0.0018
V13(-1), V13(1), V14(-1), V14(1), V16(1), V16(0), V16(-1)	These valves are adjusted manually so that the required output flowrates are achieved. Thus, the probability value assigned to these events is that of the general error rate under high stress.	0.25
V5(L), V6(L), V9(L), V12(L) V16(L)	These events constitute the mechanical failure of the valves. The failure rate for manually operated valve is λ = 0.1 /year.	0.1
L1(0), L1(-1), L3(0), L3(-1), L4(0), L4(-1), L5(0), L5(-1), L6(0), L7(0), L7(-1), L8(0), L8(-1), L9(0), L9(-1), L10(0), L10(-1), L11(0), L11(-1), L15(-1)	These events are due to dirt or deposits in the small diameter steel pipelines which restrict the proper fluid of fluid. From table C.1, there is no failure rate data for such events. However, one can assume that these events are smilar to air supply line being blocked or crushed for which the failure rate is λ = 0.01/yr.	0.01
L1(L), L3(L), L4(L), L5(L), L6(L), L7(L), L8(L), L9(L), L9(L), L10(L), L11(L), L15(L)	Leakage of fluid to the environment via the pipelines is due to the operator incorrectly tightening of nuts at the joints.	0.0047

Table 6.3 (a) : continuation of table 6.3

Events	Remarks	1 year/event probability.
CV1(1), CV1(0), CV1(-1), CV1(L), CV2(1), CV2(0), CV2(-1), CV2(L), CV3(1), CV3(0), CV3(-1), CV3(L) CV4(1), CV4(0), CV4(-1), CV4(L)	The failure rate for pnuematic control valves is λ = 0.3 /year.	0.259
P1(0), P2(0), P3(0)	The failure rate of pumps stopping is $\lambda = 10^{-4}$/day.	0.036
P1(L), P2(L), P3(L), P1(-1), P2(-1), P3(-1)	Leakage to the environment via the pumps is due to catastrophic failure. The failure rate is $\lambda = 10^{-4}$/year.	1×10^{-4}
TI1(1), TI1(-1), TI2(1), TI2(-1)	The fluid temperatures are measured by thermocouples. The failure rate is λ = 0.17 /year.	0.156
LI1(1), LI1(-1), LI2(1), LI2(-1)	The level meter is assumed to be of the DP cell type. The failure rate is λ = 0.43 /year.	0.349
FI1(-1), FI1(0), FI1(1), FI2(-1), FI2(0), FI2(1), FI3(-1), FI3(0), FI3(1)	There is no failure rate data for flowmeters. However, since the flowmeter used output electrical pulses of which the frequency is linear to flowrate, one can assume that its failure rate is the same as the failure rate of the temperature transducer where λ = 0.29 /year.	0.252
RBH(0)	This event is due to loss of electrical supply where λ=0.005/yr.	0.005
RBH(-1), RBH(1)	The rate of heat output from RBH depends on the signal from the controller. Thus, it can be assumed that the occurrence of these events is similar to failure of temperature transducer where λ = 0.29 /year.	0.252
D1(L), C1(L), RB1(L), HE1(L) C1COIL(L)	The failure rate of atmospheric vessels due to serious leak is $\lambda = 10^{-4}$ /year.	1×10^{-4}

Table 6.3 (b) : continuation of table 6.3 (a)

Events	Remarks	1 year/event probability
C1COIL(F)	There is no data for failure rate due to fouling of the condenser coil. A reasonable assumption for failure rate is $\lambda = 0.01$ / year.	0.01
C1VENT(0)	Blocking of the condenser vent is due to failure to follow instructions that tells the operator to remove the cover of the vent prior to operating the pilot plant.	0.065
LC1(1), LC1(-1), LC2(1), LC2(-1), TC1(1), TC1(-1), TC2(1), TC2(-1)	The failure rate for set point is $\lambda = 0.14$ /year.	0.131
T1(41)	Tank T1 goes empty is due to human error who fails to fill the tank when its level has depleted.	
T2(41)	Tank T2 is never used when operating the pilot plant and is always empty.	1
T1(22)	This event denotes extraordinary cooling of liquid in tank T1 by cold wind or ice on the surface of the tank. The probability for this event is assumed to be 0.1 per event	0.1
HE1(-1), HE1(0) C1COIL(-1), C1COIL(0)	Assume that these events are similar to air supply line being blocked where $\lambda = 0.01$ /year.	0.01

CHAPTER SEVEN

7. RESULTS OF TESTING THE DIAGNOSIS PACKAGE

7.1 : Introduction

The diagnosis package was tested using the cause and effect data obtained from the HAZOP study of the pilot distillation plant described in chapter 6. The HAZOP data, in the form of cause and symptom equations, have first to be translated into a data structure so that the probability of every event in the cause and effect relationship can be evaluated. The data structure is also needed for displaying the causes and consequences of any particular event, in the form of a fault tree, on the VDU of the computer. The data structure has already been described in chapter 5.

The test consisted of two stages. The first stage is to test the data preparation program **TRANSLAT**. The second stage is to test the display of fault trees on the VDU of the computer. In this case, the program to be tested is **DISFAULT**.

7.2 : Translation of Cause and Symptom Equations into the Data Structure

Translation of the cause and symptom equations is done by the program **TRANSLAT**. The list of cause and symptom equations from chapter 6 were stored in a text file called **HAZOP.TXT**. The information contained in this file is translated by **TRANSLAT** into a data structure which is stored in a file named **HAZOP.COD**. The procedure **TRANSLAT** executes the translation task and has been described in chapter 5. The contents of **HAZOP.COD** is shown in Appendix G.

As an example, consider the following set of cause and symptom equations obtained from the list in chapter 6:

L10(12) = L9(12) + CV2(-1) + L10(-1) + L10(L) + FI2(1) 7.1

L10(12) - N4(12) * N2(521) 7.2

L9(12) = L9(-1) + L9(L) + P2(0) + P2(-1) 7.3

L9(521) = N7(521) 7.4

L4(521) = N2(521) 7.5

L4(521) - N7(521) 7.6

L11(521) = L9(521) 7.7

Table 7.1 shows the data structure containing the translated information of the cause and symptom equations 7.1 to 7.7. The data are extracted from table G.1 in Appendix G.

The result in table 7.1 shows that **TRANSLAT** has successfully translated equations 7.1 to 7.7 into a structured form so that the information about the cause and effect relationships are easily extracted for both probability evaluation and fault tree display. For example, looking at the event L10(12) in table 7.1, shows that it has 5 causes connected by an **OR** gate. The cause equation for L10(12), i.e., equation 7.1, has also 5 causal events, in which the occurrence of any one of the events causes L10(12) to occur. In table 7.1, the record

Table 7.1 : Data structure containing the translated information of equations 7.1 to 7.7.

Rec.	e$	g$	bc$	can$				bq$	con$			
no.				(1)	(2)	(3)	(4)		(1)	(2)	(3)	(4)
:												
76	FI2(1)	PRI	0	0	0	0	0	2	72	77	0	0
77	L10(12)	ORC	5	78	79	80	82	2	89	88	0	0
78	L9(12)	ORC	4	97	98	93	99	1	77	0	0	0
79	CV2(- 1)	PRI	0	0	0	0	0	1	77	0	0	0
80	L10(-1)	PRI	0	0	0	0	0	1	77	0	0	0
81	L10(L)	PRI	0	0	0	0	0	1	77	0	0	0
82	& 11	CON	2	81	76	0	0	0	0	0	0	0
:												
88	N2(521)	ORS	2	72	77	0	0	1	174	0	0	0
89	N4(12)	DIR	1	77	0	0	0	2	127	132	0	0
:												
93	P2(0)	PRI	0	0	0	0	0	3	94	78	102	0
:												
97	L9(-1)	PRI	0	0	0	0	0	1	78	0	0	0
98	L9(L)	PRI	0	0	0	0	0	1	78	0	0	0
99	P2(-1)	PRI	0	0	0	0	0	1	78	0	0	0
:												
111	L9(521)	DIR	1	112	0	0	0	1	246	0	0	0
112	N7(521)	DIR	1	174	0	0	0	1	111	0	0	0
:												
174	L4(521)	DIR	1	88	0	0	0	1	112	0	0	0
:												
246	L11(521)	DIR	1	111	0	0	0	0	0	0	0	0

numbers stored in the branch address fields at the record where L10(12) is stored point to the same events as the causal events of equation 7.1.

Since the results in table 7.1 are correct, implies that all the cause and effect information stored in **HAZOP.COD** are also correct.

TRANSLAT also produced a random access file called **HAZOP.PRI** containing the set of primary events for each event stored in the file **HAZOP.COD**. The information contained in **HAZOP.PRI** is used to check for independence when evaluating both the *a priori* and *posteriori* probabilities of every event in the file HAZOP.COD.

7.3 : Probability Evaluation

Two types of probabilities are evaluated in the diagnosis package. One is the *a priori* probability evaluation to provide the probability data needed for the fault tree display part of the diagnosis package. The *a priori* probability evaluation of every event in the data file is done within the program **TRANSLAT**. The other probability evaluation is the determination of *posteriori* probabilities of every event in the data file, based on an event which has occurred. The *posteriori* probability evaluation is done in the **DISFAULT** program, after receiving the alarm event and before displaying the fault tree on the VDU of the computer.

Before the *a priori* probabilities can be evaluated, every primary event has to be assigned probability values. This is done within

TRANSLAT where the program asks the user to input probability values of each primary event in the HAZOP.COD file.

To verify the validity of the probability evaluation algorithms implemented in **TRANSLAT** and **DISFAULT**, a sample cause and effect relationships were taken and the probabilities were calculated manually. These values were then compared with those obtained from the computer programs.

For the purpose of the test, cause equations 7.1 and 7.2 were used. Since both equations contains **OR** gates only, equation 4.35 was used to manually calculate the *a priori* probabilities of the top events. The *posteriori* probabilities were calculated based on the occurrence of L10(12). Equation 4.69 was used to calculate the *posteriori* probabilities. Table 7.2 shows the result of the probabilities calculated manually. For primary events, their *a priori* probabilities has already been assigned. The values obtained were then compared with the values displayed on the VDU of the screen during the testing of the fault tree display program.

7.4 : Display of Fault Trees

The display of a fault tree on the VDU of the microcomputer is obtained by executing the program **DISFAULT**. The user has to input the name of the file containing the fault tree information, e.g. **HAZOP.COD** and the name of the fault event. After that, **DISPLAY** will evaluate the *posteriori* probabilities of every event in the data file before displaying the fault tree on the VDU of the computer.

Table 7.2 : Results of Manual *A Priori* and *Posteriori* Probability Evaluations

Event	Gate Type	*A Priori* Probability	*Posteriori* Probability
L10(12)	OR	0.48128	1.0
FI2(1)	PRI	0.252	0.52360
L9(12)	OR	0.05022	0.10435
CV2(-1)	PRI	0.259	0.53815
L10(-1)	PRI	0.01	0.02078
L10(L)	PRI	0.0047	0.009766
L9(-1)	PRI	0.01	0.02078
L9(L)	PRI	0.0047	0.009766
P2(0)	PRI	0.036	0.07480
P2(-1)	PRI	0.0001	0.0002078

The following sections describe the various displays that are obtained on the VDU of the microcomputer. The colour hardcopy of the screen shots are obtained using a Tektronix 4693D colour thermal printer. Two different alarm events were used to show how the displays can be operated on to give their various effects. In one, the event is less than normal flow in line L10, i.e. **L10(12)**, and the other event was no liquid in the condenser/reflux drum C1, i.e., **C1(41)**.

7.4.1 : Displays of Causes and Consequences of L10(12)

The displays shown in this section are those when **L10(12)** is input as the alarm event. When the *posteriori* probabilities have been evaluated, the initial display that shows the causes and consequences of **L10(12)** is shown in Plate 7.1. In the Cause column, the events displayed with the green background are primary events and the event displayed on a blue background is a secondary event which is caused by other events not shown on the VDU. In the consequence column, the events displayed on blue backgrounds indicates that they are not the top most symptoms of the fault **L10(12)**.

Pressing the F1 and F2 function keys switch on the the display of the *a priori* and *posteriori* probabilities of every event displayed on the VDU. This is shown on Plate 7.2. The values above the event names and written in red are the *posteriori* probabilities and those below the event names and written in blue are the *a priori* probabilities. Pressing either of the F1 and F2 keys again will switch off the display of the *a priori* and *posteriori* probabilities respectively. The *posteriori*

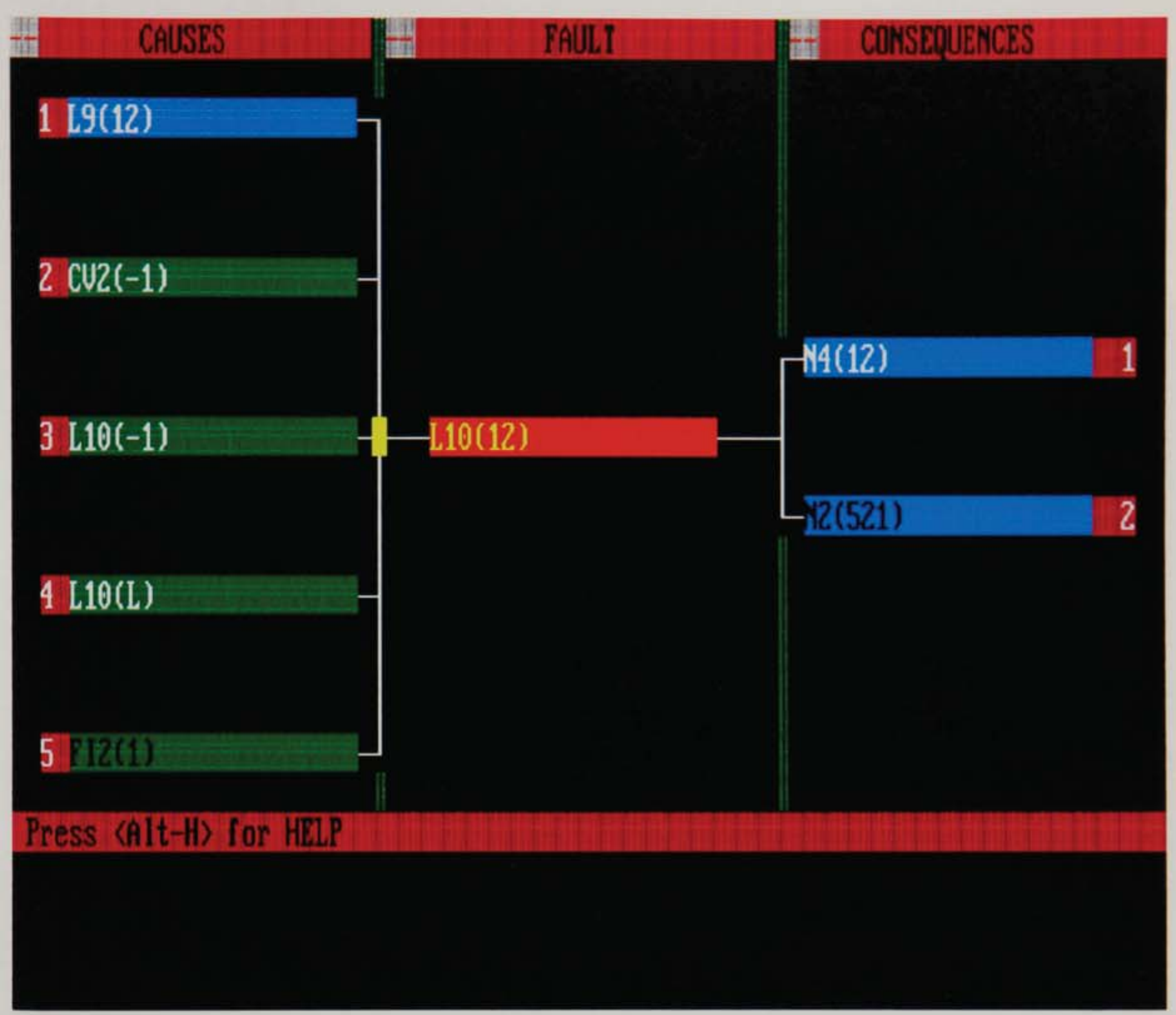


Plate 7.1 : Initial Display For The Alarm Event L10(12).

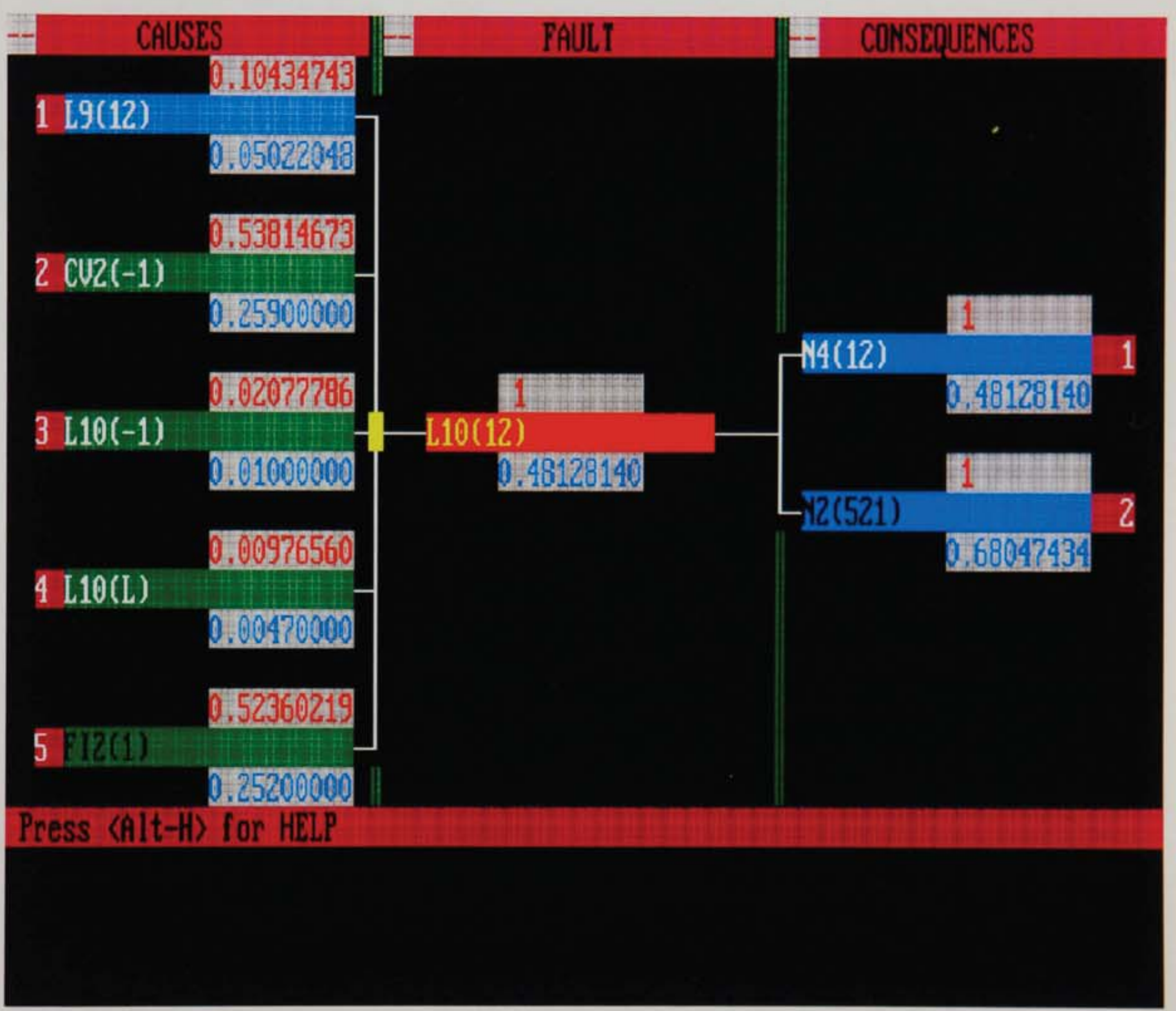


Plate 7.2 : Display After Pressing The F1 And F2 Function Keys.

probability values of 1 on the consequences indicate that these events are bound to occur or have occurred due to the occurrence of the alarm event.

The probabilities values on the display are compared with those in table 7.2. It is found that the values are in close agreement. Thus the probability evaluation algorithms implemented in **TRANSLAT** and **DISFAULT** have done their tasks successfully.

From plate 7.2, the order to check the causes of less than normal flow in line L10 would be:

1) **CV2(-1)**, i.e., the control valve CV2 insufficiently open,

2) FI2(1), i.e., flow indicator FI2 indicating higher than the actual flow in line L10,

3) **L9(12)**, i.e., there is less than normal flow in line L9,

4) **L10(-1)**, i.e., there is partial blockage in line L10,

5) **L10(L)**, i.e., there is leakage to the environment from line L10.

This order is determined by the values of the *posteriori* probabilities shown above each of the causal events on the display.

The event **FI2(1)** is not a cause of **L10(12)**. However, it is included in the fault tree to show that the measuring instrument, FI2, may have failed high and does not show the true value of the flowrate in line L10.

The causes of the secondary event **L9(12)** can be shown by directing the cursor to that event and pressing the F9 function key.

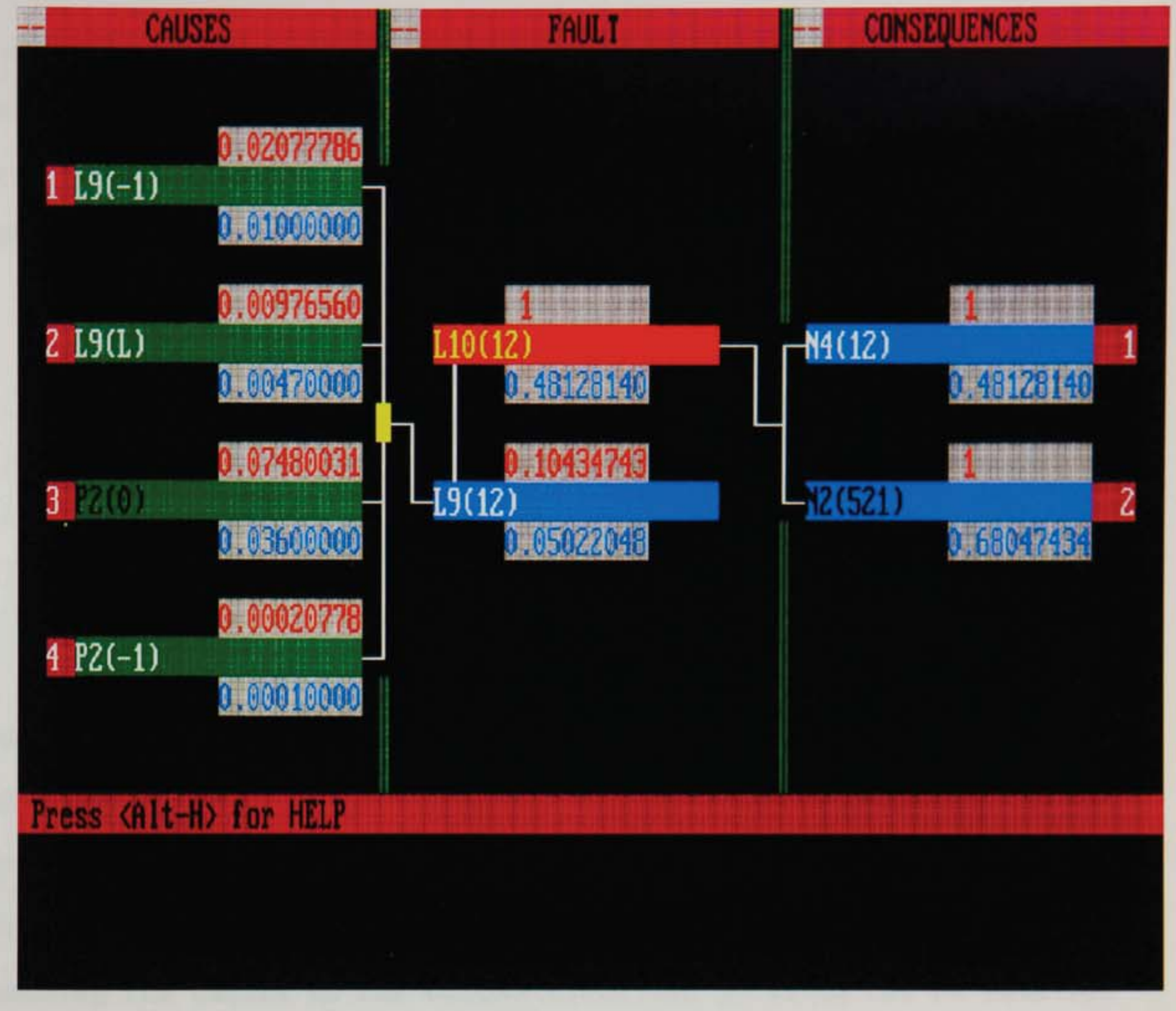


Plate 7.3 : Display After Directing The Cursor To L9(12) And Pressing The F9 Function Key.

The effect of this operation is shown in Plate 7.3. The cursor is represented by the blinking character of the first letter of an event. Movement of the cursor is effected by pressing the cursor direction keys on the numeric key pad. Pressing the F10 key will display the initial fault tree as shown in Plate 7.2.

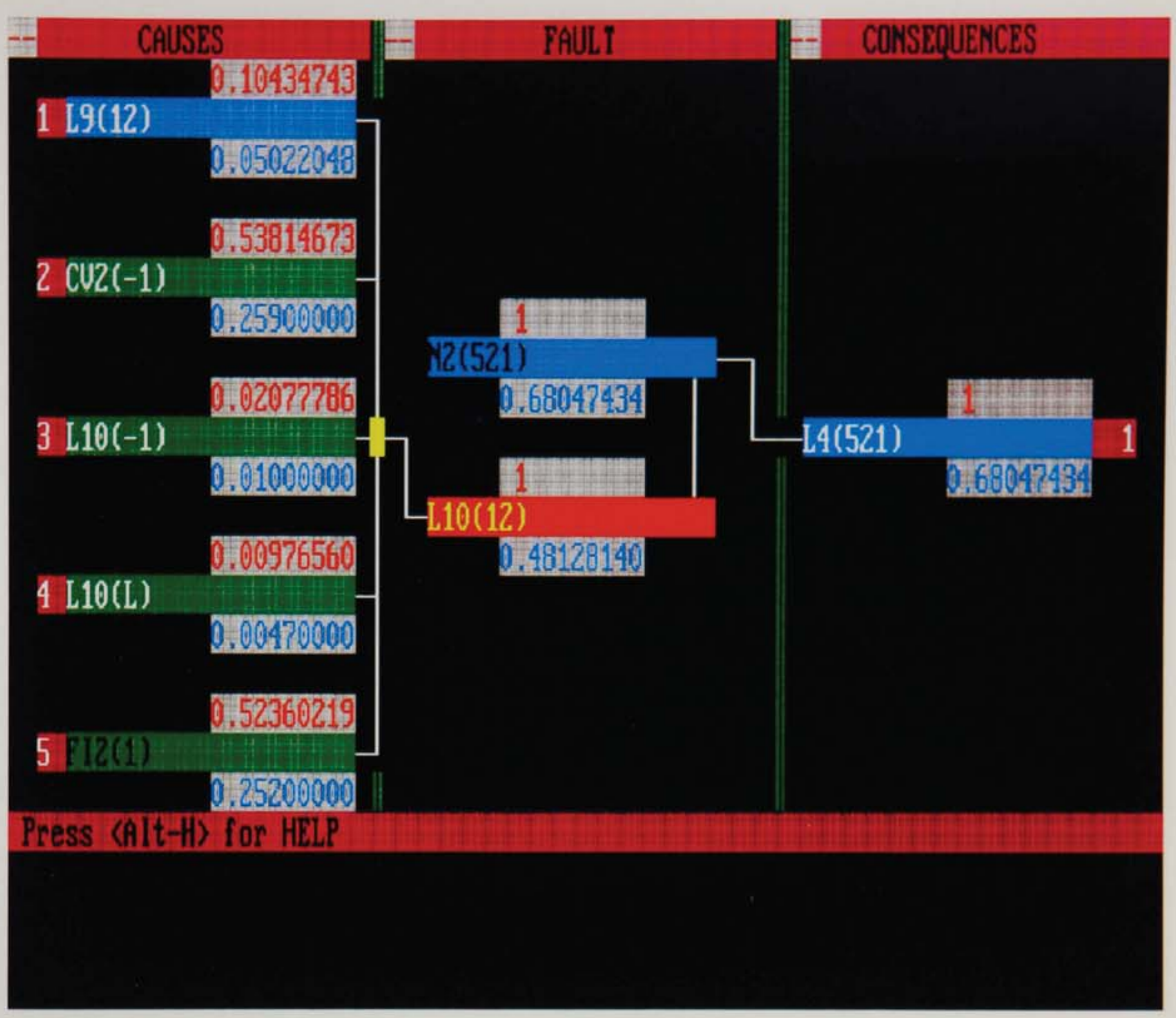
From plate 7.2, are two consequences of **L10(12)**, which are:

1) **N4(12)**, i.e., less flow at node N4,

2) **N2(521)**, i.e., less concentration of trichloroethylene at node N2.

The consequences are shown with blue background to indicate that these events can develop to other consequences not shown on the screen of the VDU. The operator can choose to see what are the other consequences of the events displayed in the consequence column by directing the cursor to the chosen event and pressing the F9 function key. For example, the display in plate 7.4 is obtained from the display shown in plate 7.2 by directing the cursor to the event **N2(521)** and pressing the F9 fucntion key.

From plate 7.4, the event **L4(521)** still has other consequences from its occurrence. These can be shown by repeating the same procedure as described above. Plate 7.5 shows the effect obtained by such operation. The event L11(521) in plate 7.5 displayed in a green box indicates that it is one of the top most symptoms of L10(12), i.e., there is no other consequences of L11(521).


-- CAUSES

-- FAULT

-- CONSEQUENCES

0.10434743
1 L9(12)
0.05022048

0.53814673
2 CV2(-1)
0.25900000

0.02077786
3 L10(-1)
0.01000000

0.00976560
4 L10(L)
0.00470000

0.52360219
5 F12(1)
0.25200000

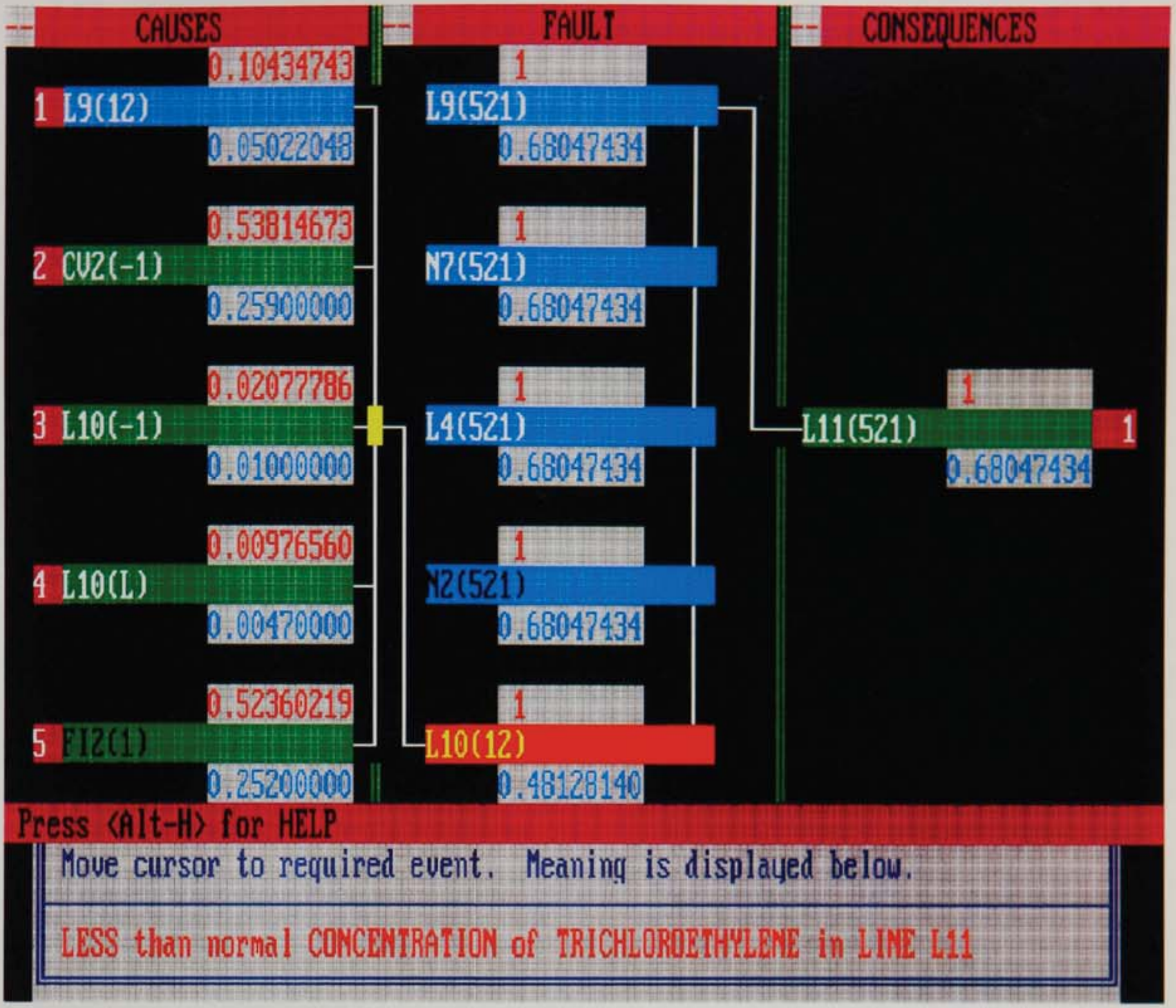
1
N2(521)
0.68047434

1
L10(12)
0.48128140

1
L4(521) 1
0.68047434

Press <Alt-H> for HELP

Plate 7.4 : Display After Directing The Cursor To N2(521) And Pressing The F9 Function Key. Prior to this display, the status of VDU screen is as shown in plate 7.2.



Press <Alt-H> for HELP

Move cursor to required event. Meaning is displayed below.

LESS than normal CONCENTRATION of TRICHLOROETHYLENE in LINE L11

Plate 7.5 : Display Showing One Of The Final Consequences of L10(12), i.e., L11(521). The display also shows the meaning of the event where the cursor is located, activated by pressing the ALT and E keys together.

The operator can obtain the meaning of the coded event by pressing the ALT and E key together. The meaning of the event where the cursor is located is shown below the fault tree display, called the message area. Plate 7.5 shows the effect of such an operation. For this case, the cursor is at the event L11(521). Moving the cursor to other events will automatically display the meaning of that event. Pressing the ALT and E keys again will switch off the meaning of the event where the cursor is located.

Plate 7.6 shows the effect of directing the cursor to L9(12) in the cause column and then pressing the F9 function key followed by consequtively directing the cursor to the events **N4(12)** and ***19** and pressing the F9 function key. The effect shows that the operator can see how the occurrence of any of the primary events shown in the cause column propagates to the final consequence, i.e., **TRAY1(23)** which means higher than normal temperature on tray 1 in the distillation column.

The event ***19** is a dummy event to indicate that it has inputs via a gate type different from the gate type of its consequence, i.e., **TRAY1(23)**. Plate 7.6 shows that in the box that displays the event ***19**, are also displayed the word AND within parenthesis, to indicate that it has inputs via an **AND** gate. This is the reason why the *posteriori* porbabilities of event ***19** and it consequence, i.e., **TRAY1(23)**, are not equal to 1.

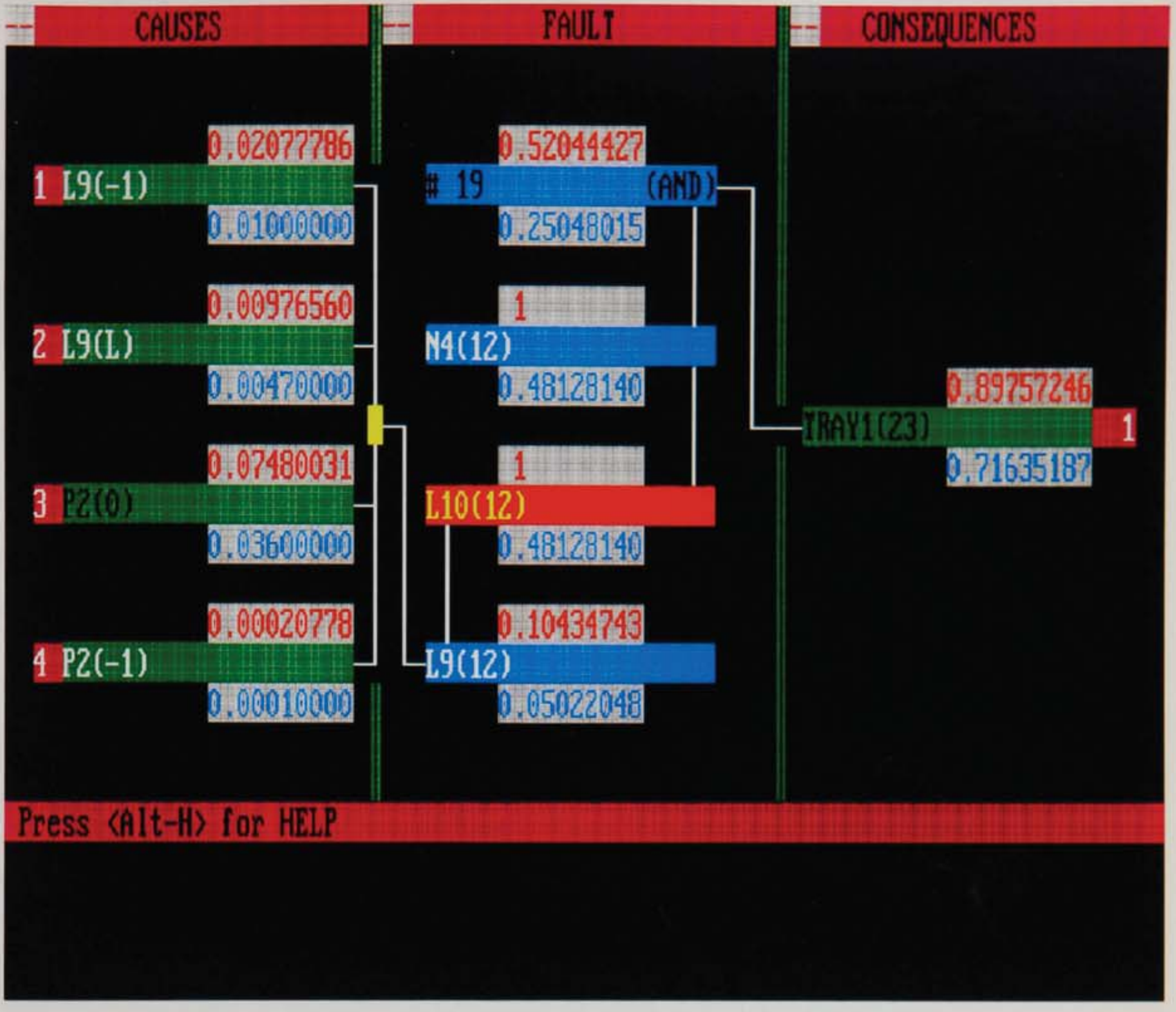


Plate 7.6 : Display Showing The Propagation Of Faults From The Primary Events To One Of The Top Most Consequences Of L10(12).

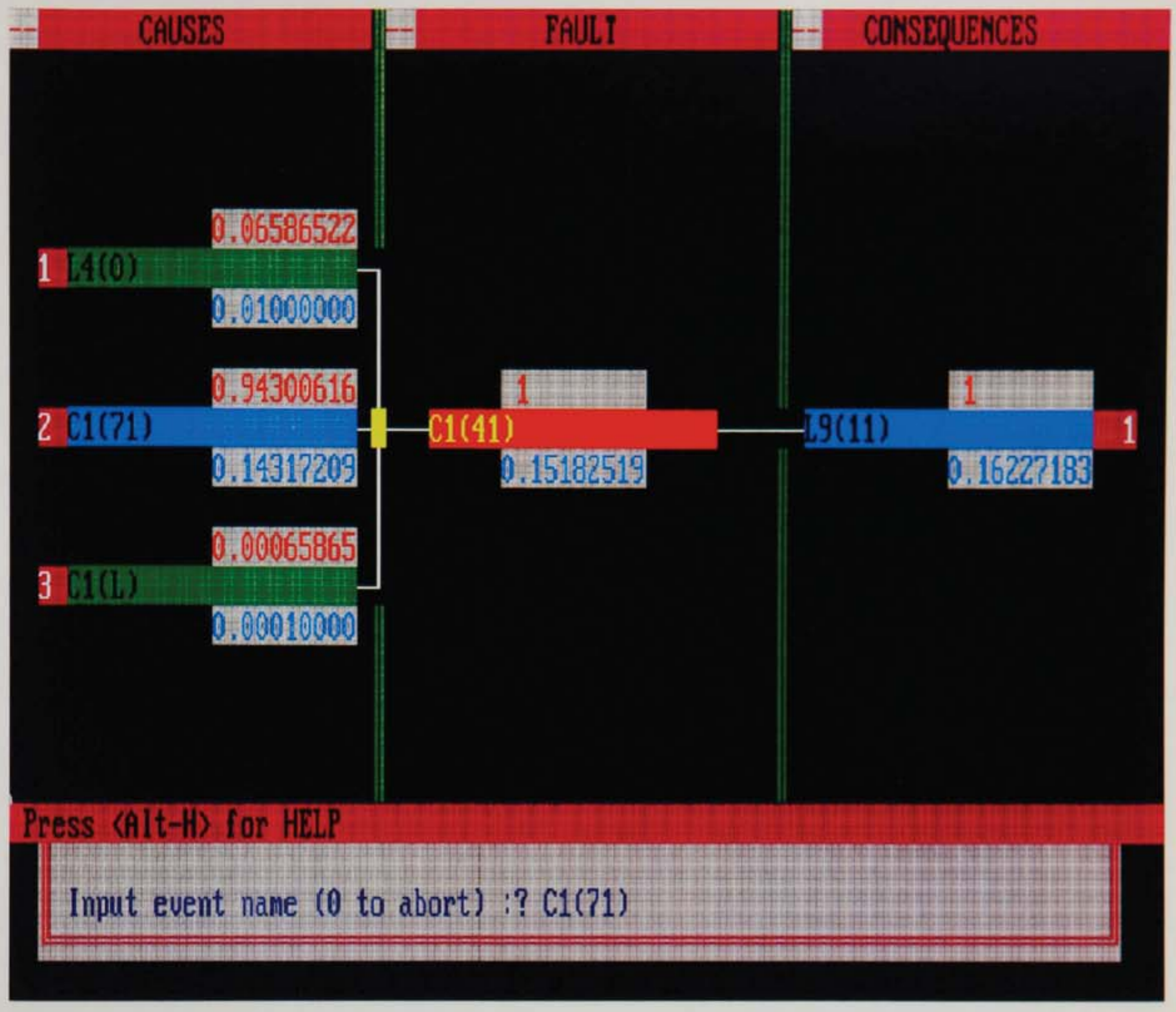
7.4.2 : Displays of Causes and Consequences of C1(41)

The following displays in this section are put forward to show what other facilities the operator has when using the diagnosis package. For this purpose, the alarm event input from the keyboard is **C1(41)**, i.e., no fluid level in the condenser/reflux drum.

In plate 7.7, ignoring the message printed in the message area at the bottom of the screen, are shown the display obtained after the *posteriori* probability evaluation has completed its task and pressing the F1 and F2 function keys. In the cause column, the causal events are printed in black foreground. This is to show that these causal events can cause other events apart from the alarm event, i.e. **C1(41)**. Also, the event in the consequence column is printed in black foreground. This is to indicate that the consequence event can be caused by other events apart from the alarm event.

The user can view what are the causes and consequences of any other event. This is done by pressing the ALT and A keys together and the message at the bottom of the screen as shown in plate 7.7, up to the question mark, is displayed. The purpose is for the user to input the name of the event from the keyboard for which its causes and consequences are to be viewed.

As an illustration, from the display in plate 7.7, the event **C1(71)**, i.e., no heat transfer in the condenser C1, is input from the keyboard. After pressing the RETURN key, the display as shown in plate 7.8 is obtained. The event just input via the keyboard is



Input event name (0 to abort) :? C1(71)

Plate 7.7 : Display showing the causes and consequences of the alarm event, C1(41), and also the message for the user to input name of an event to view its causes and consequences, activated by pressing the ALT and A keys together.

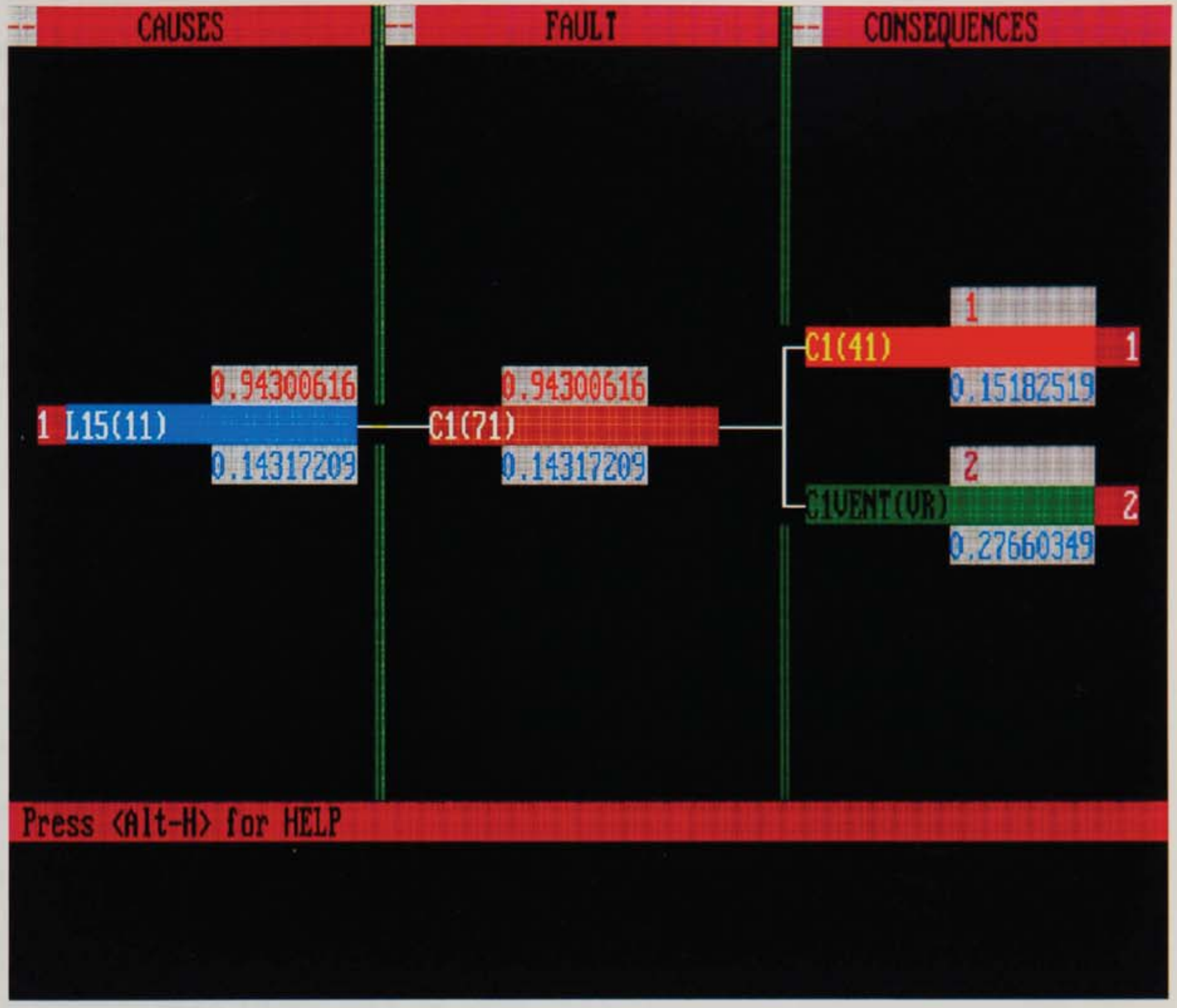


Plate 7.8 : Display of the cause and consequences of C1(71) which was input via the keyboard from the display in plate 7.7.

displayed on brown background. The display shows that **C1(71)** can cause two events, which are the alarm event, **C1(41)**, still displayed on the red background, and **C1VENT(VR)**, i.e., vapour release to the environment from the vent of the condenser C1. Thus if **C1(71)**, is found to be the cause of the alarm event, apart from the consequences emanating from **C1(41)** that either have or will occur, consequences due to **C1(71)** will also occur.

The value of 2, displayed above the event **C1VENT(VR)**, is not the posteriori probability of the event. In fact, the posteriori probabilities of consequences of causes of the alarm event are not evaluated. The reason for this omission will be discussed in chapter 8. Instead the value of 2 are assigned to the consequences of the causes of the alarm event, to indicate that any these events can occur if it is found its cause which is also the cause of the alarm event is true.

Sometimes there may be more than 5 causes or consequences of the alarm event. Since only 5 events can be displayed in each column, those that cannot fit into the screen are not shown. However, the operator can scroll up or down each individual screen to display events that are off the screen.

On the top left hand corner of each column are reserved two spaces to indicate whether the column needed to be scrolled to display events that are off the screen. Dashes indicate that there is no need to scroll the column. A downward pointing arrow tells the operator that the column needs to be scrolled downwards to display events above the top-most event in the column. An upward pointing arrow tells the operator that the column needs to scrolled upwards to display events below the bottom-most event in the column. If both the downwards and upwards arrows are displayed in any one column, then the operator has the choice to either scroll the column upwards or downwards to see the events that are off the screen. The functions keys that have been assigned for the scrolling tasks has already been described in chapter 5.

As an illustration, plate 7.9 shows the primary causes of C1(41) via the secondary events C1(71) and L15(11), and also the chain of consequences from C1(41) to RB1(41). The fault column has more than 5 events to be displayed. The downward arrow on the top left hand corner of the fault column indicates that the column needs to be scrolled down to display the events that are off the top of the screen. The dash next to the arrow indicates that there is no need for scrolling upwards. Scrolling down the fault column is achieved by pressing the F6 function key.

Plate 7.10 shows the effect when the F6 function key is pressed twice. This time the top left hand corner of the fault column has both the downward and upward arrows displayed. This is to indicate that there are events off the top and bottom of the screen not displayed. The operator can either press the F6 function key to further scroll down the fault column or the F5 function key to scroll up the fault column. This scrolling facility is also implemented on the cause and consequence column.

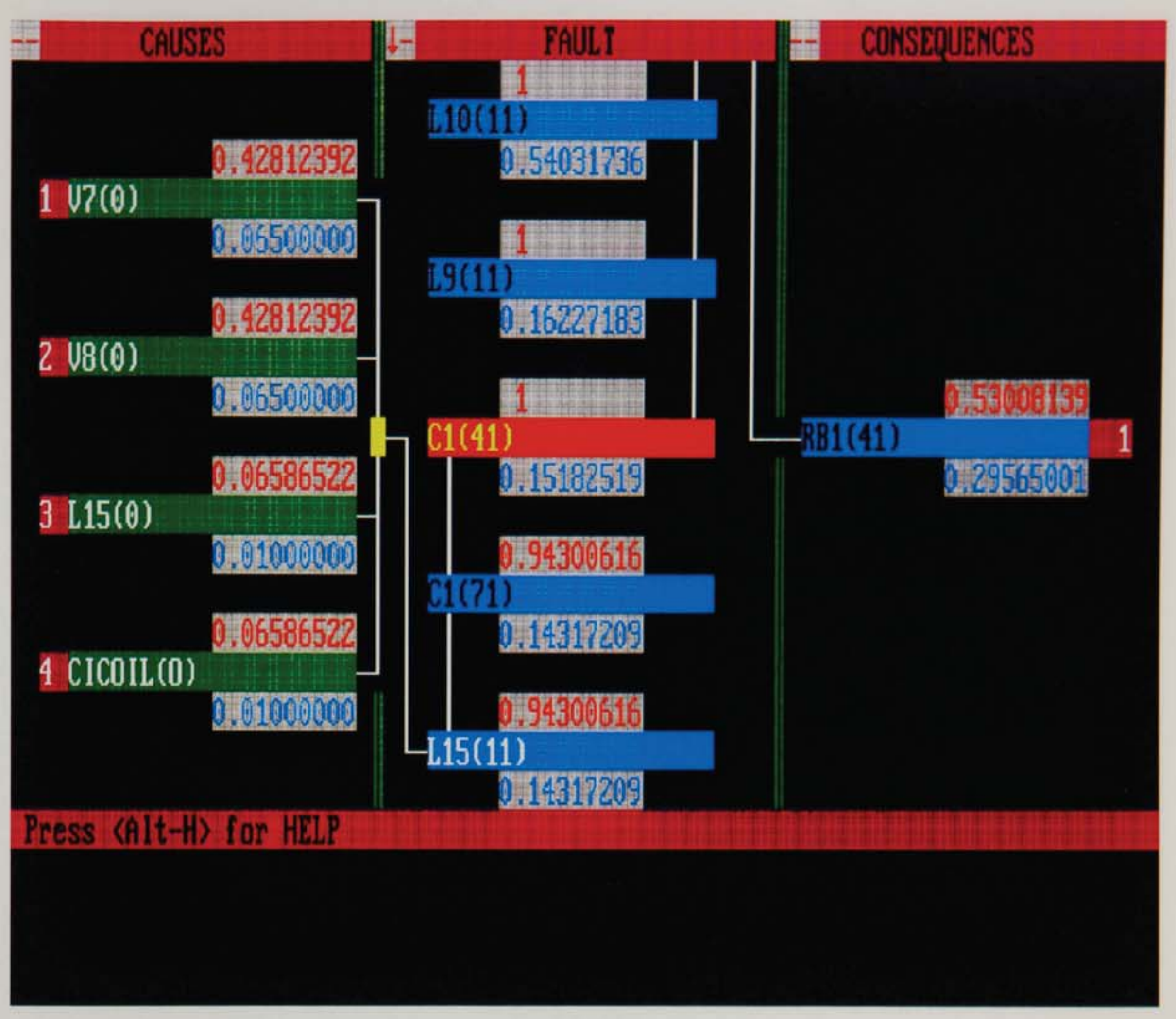


Plate 7.9 : Display showing the primary causes of C1(41) via the secondary event C1(71), and the chain of consequences to RB1(41).

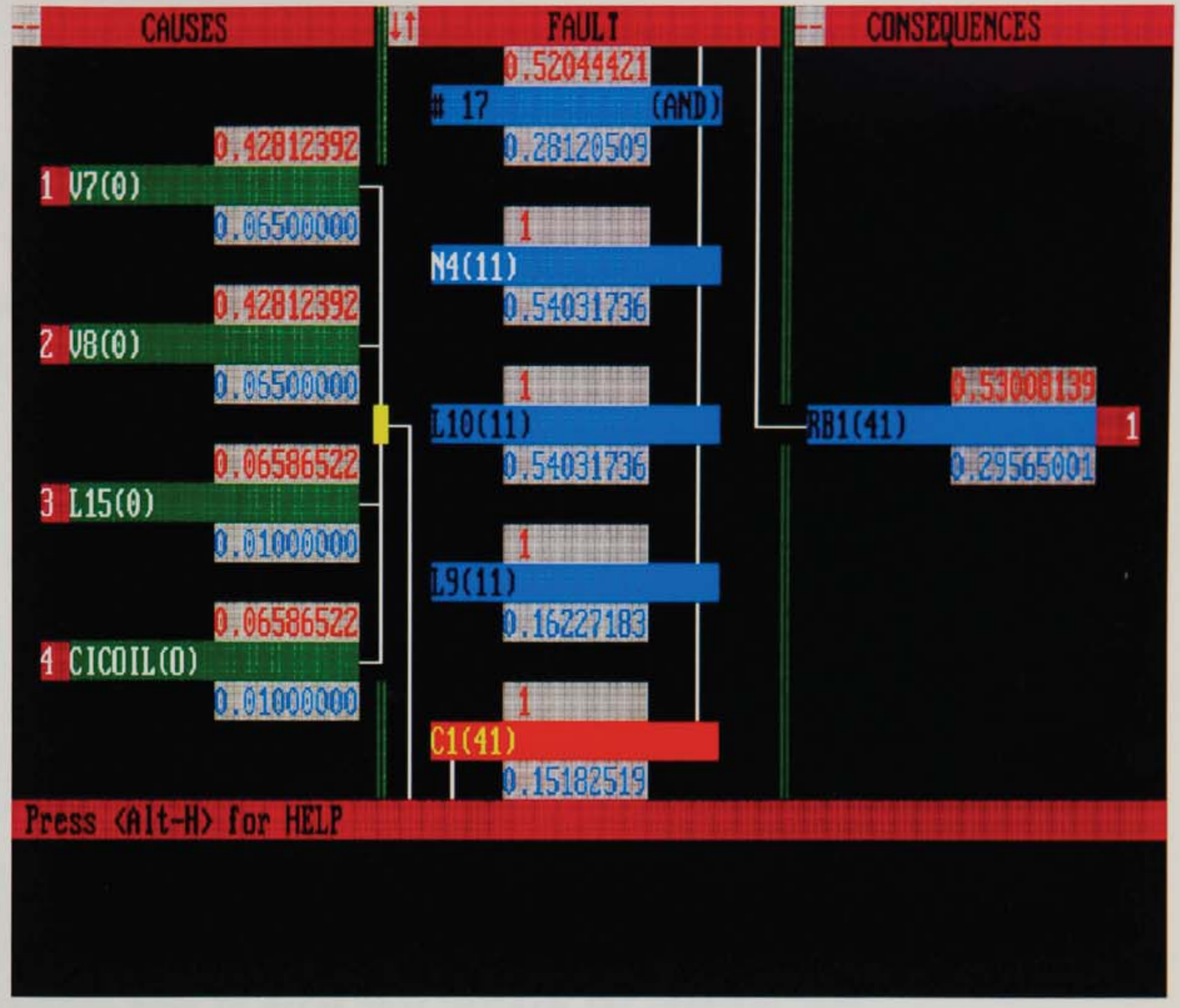


Plate 7.10 : Display showing the effect when the fault column, originally as shown in plate 7.9, is scrolled downwards by pressing the F6 function key twice.

7.5 : Discussion

The size of the screen of the VDU limits the operator to seeing only one level of causes and consequences and five events per column at a time. However, with the facilities that enable the user to view other levels of causes and consequences and the ability to scroll each column independently, all the cause and effect relationships can be viewed. Thus the diagnosis package is flexible and versatile and the user can view any part of the fault tree and can easily trace to the root cause of the alarm event.

The operations on any display can be done by single key strokes of the function keys or pressing combination of ALT key and either A or E key. Each key stroke has its particular function. If the user does not know which key performs the desired task, pressing the ALT and H keys together will display a help frame. The help frame tells the user what keys to press that perform the required task.

7.6 : Conclusion

From the results of the test, it has been demonstrated that:

1) **TRANSLAT** correctly translate the cause and symptom equations into a data structure such that the cause and effect information are easily accessible,

2) both the *a priori* and *posteriori* probability evaluations in the program **TRANSLAT** and **DISFAULT** respectively have correctly calculated probability values,

3) the fault tree display on the VDU of the computer can be easily operated on so as to view other parts of the fault tree that is off the screen.

CHAPTER EIGHT

8. DISCUSSION, RECOMMENDATION FOR FUTURE WORK AND CONCLUSION

8.1 : DISCUSSION

8.1.1 : Introduction

HAZOP studies and fault tree analysis are currently used primarily as a design tool to investigate the reliability, safety and operability of process plants. The present work has shown that the outcomes from HAZOP studies in the form of cause and symptom equations, organised in a form of a fault tree display, can be used as a diagnostic aid in finding the cause of a fault occurring whilst a process plant is in operation. The work has included the development of a package which operates in four stages, assuming that a HAZOP study is complete. These are:

1) translation of the cause and symptom equations obtained from the HAZOP study to a data structure stored in a file,
2) use of a new top-down recursive algorithm for evaluating the *a priori* probability of every non-primary event stored in the data file,
3) evaluation of the *posteriori* probability of every event in the data file on the occurrence of any event,
4) display of the fault tree on the VDU of the microcomputer, including probability information.

The usefulness of the diagnosis package depends on the ability of the HAZOP data to represent the actual process plant failure modes and the cause and effect relationships. It is also dependent on the speed of computation to evaluate the conditional probabilities on the evidence that an event has happened. The fault tree display on the VDU of the computer is important for the success of the diagnosis package, because the intention is to communicate the results to the operator. Some problems have occurred with the the development of the diagnosis package. Some of these problems have been overcome while others need further work. These are discussed in the following sections.

8.1.2 : HAZOP In Relation To Fault Diagnosis

The general objective of HAZOP study is to identify hazards and operability problems in a process plant. A general description of how a HAZOP study is carried out and limitations of HAZOP have been discussed in chapter 3. One problem of the HAZOP study is the presentation of the accumulated data so that it is easy to comprehend. Another problem is the storage of the information obtained from a HAZOP study in a computer for access by computer programs. These problems have been overcome in this work by using the Lihou [71] coding method and representing the cause and effect relationships in the form of cause and symptom equations. A cause equation relates a deviation in a process plant variable to its causes. The way process plant equipment responds to the deviation is described by a symptom equation. Thus the cause and symptom equation method of recording the HAZOP findings is both concise and comprehensive.

For the purpose of testing the diagnosis package, a HAZOP study was done on a pilot distillation plant. A problem encountered during the HAZOP study was the occurrence of cycling effects, especially when analysing recycle streams and feedback control loops. A cycling effect occurs when an event is a cause of itself. The fact that this has happened is not known until the cause and symptom equations are processed to obtain the set of primary causes of an event, during the data preparation stage of the diagnosis package.

As an illustration of a cycling effect, consider the buffer tank arrangement as shown in figure 8.1.

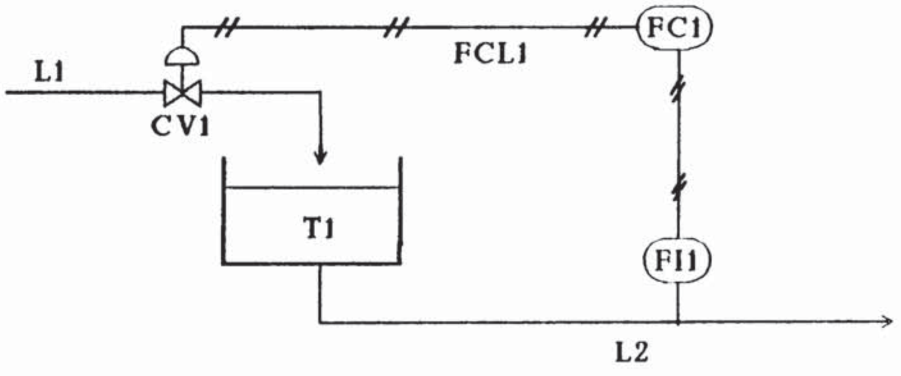


Figure 8.1 : A Schematic Diagram of a Buffer Tank

The series of cause equations which traces to the root causes of less than normal flow of fluid in line L2 are as follows:

1. L2(12) = T1(42) + L2(-1) + L2(L)
2. T1(42) = L1(12) + T1(L)
3. L1(12) = CV1(-1) + L1(-1) + L1(L) + FCL1(-1)
4. FCL1(-1) = FI1(1) + FC1(-1) + L2(13)

5. L2(13) = T1(43)

6. T1(43) = L1(13)

7. L1(13) = CV1(1) + FCL1(1)

8. FCL1(1) = FI1(-1) + FC1(1) + L2(12)

The explanation of the above symbols is found in chapter 3. If the causes of L2(12) are traced from cause equations 1 to 8, it will be found that from equation 8, L2(12) is itself a cause. When this happens, there will be an endless display of the fault tree with L2(12) as the event that causes the unbroken chain.

To avoid the cycling effect, natural responses of control loops and recycling streams are not included in the cause and symptom equations. For example, when flow of fluid in line L2 is less than normal, then the response of the controller is to increase the flow of fluid in L1 so that eventually the flow of fluid in L2 will return to normal. Thus, for a short period of time, the flow control loop will be at a state of giving high flow of fluid in L1, i.e., FCL1(1). In actual fact, FCL1(1) in response to low flow of fluid in L2 is not a fault but a natural reaction of the controller. Thus, the event L2(12) need not be included in the cause equation for the causes of FCL1(1). If this rule is followed, then the cause equations 4 and 8 will be:

4* FCL1(-1) = FI1(1) + FC1(1)

8* FCL1(1) = FI1(-1) + FC1(-1)

By applying this rule, then the cycling effect is eliminated and the fault tree will end with the primary events as the bottom events. This rule has been applied during the HAZOP study of the pilot distillation plant.

8.1.3 : The Fault Tree Data File

The cause and symptom equations are an unstructured collection of cause and effect relationships. In this form, the information required for constructing the fault tree and probability evaluations cannot be easily extracted by the diagnosis package. For easy access to the data, the cause and symptom equations were translated to a data structure and stored in a file. The format of the data file was described in chapter 5. The package has only to search for the alarm event name once from the data file and the causes and consequences for that event can be easily obtained using pointers.

The number of causal events on the right hand side of a cause equation varies from one equation to another. Also, the number of causes for each particular symptom depends on the number of symptom equations having the same event as a symptom. A benefit of the format of the data structure is that the user does not have to worry about the variability of these numbers. Theoretically, the data structure can handle any number of causes of an event. The only limitation is the amount of available space in the permanent storage media of the computer to store the data file. Another advantage of the data structure is that the number of consequences of an event is easily known. This information is not easily extracted from the collection of cause and symptom equations.

8.1.4 : Computation of Fault Tree Probabilities

Most of the published work on fault tree synthesis and analysis has been based on the use of large computers. These machines typically have large memories and fast execution times. On the IBM PC compatible microcomputers, using the MS DOS operating system, the amount of memory that can be used for loading and executing a program is limited to less than 640 kilobytes. This limitation was taken into account when designing the fault diagnosis package.

During the development of the diagnosis package, an algorithm was chosen for evaluating the *a priori* and *posteriori* probabilities of every event in the fault tree data file. The algorithm had to be small enough for implementation on an IBM PC compatible computer. In chapter 4, various fault tree probability calculation algorithms were reviewed and evaluated. Most of the algorithms were developed for design purposes and were used on large main-frame or mini computers. The algorithms were unsuitable for implementation for real-time application on the IBM PC compatibles where the size of memory is limited.

The algorithm that had been successfully implemented on a microcomputer was the top-down recursive algorithm known as TDPP developed by Page and Perry [109, 110]. The use of recursion makes the code for TDPP small and compact. TDPP also takes care the problem of repeated events quite easily. The other algorithms reviewed either address this problem in complex way or not at all.

The depth of recursion in TDPP when evaluating the probability of a non-primary event depends on the number of primary events connected to that event by **OR** gates and the number of repeated primary events which are inputs to that event. For example, if the event has **n** primary events as inputs via an **OR** gate, then the depth of recursion by TDPP is **n**+2 and the total number of calls to the probability evaluation subroutine is (3***n**)-1.

However, there is a limit to the depth of recursion on the IBM PC compatibles. When a subroutine calls itself, the same subroutine is loaded somewhere else in memory before execution of the called subroutine begins. If there are repeated nested calls of the same subroutine, there may not be enough memory to complete the recursion. This is a weakness when using the recursive method, which could be overcome using recent methods that enable IBM PC compatible computers to access more memory.

The fault tree probability evaluation developed for the diagnosis package in this research work is based on TDPP and is called TDRA. A few modifications were made to overcome the limitations of TDPP. These modifications were described in chapter 5. An advantage of TDRA is that the depth of recursion is greatly reduced. If an event is caused by any one of **n** independent primary events, the depth of recursion using TDRA to evaluate the probability of the event is only 3. The total number of calls to the probability evaluation subroutine is also 3. TDRA is thus much faster than TDPP. Theoretically, TDRA can handle any number of primary events connected to an **OR** gate. The only limitation is the declaration of the sizes of the arrays **s1** and **s2** which hold the addresses of events which TDRA operates on.

8.1.5 : The Fault Tree Display

A fault tree can be very large, with several levels of secondary events between the top event and the primary events. Normally, it is drawn as an upside-down tree with the top event as the root, the secondary events as the branches and the primary events as the leaves of the tree. It is not possible to display such a fault tree on the VDU of the computer due to the small size of the screen. This limitation has been discussed in chapter 5.

An alternative method of displaying a fault tree was developed so that enough information can be displayed without confusing the user. In the new method, the fault tree is displayed on the VDU sideways. The screen is divided into 3 columns. The alarm event is displayed in the centre column, the causes are displayed in the column to the left of the alarm column and the consequences are displayed in the column to the right of the alarm column. Only one level of causes and consequences can be displayed and a maximum of 5 events per column. However, the user can operate on the display to view other parts of the fault tree not shown on the VDU. The method of operating on the display was described in chapter 5. Examples of the display are shown in chapter 7.

8.1.6 : Testing of The Diagnosis Package

The diagnosis package was tested using data obtained from the HAZOP study of the pilot distillation plant. The diagnosis testing was done off-line. The fault tree display program is called **DISFAULT**. Input of the alarm event is from the keyboard. The *posteriori* probability of every event in the fault tree data file is then evaluated before the fault tree is displayed on the VDU of the computer. From then on, the user can switch on the display of the probability values of the events displayed on the screen. These values can be used as a guide for finding the most probable cause of the alarm event.

One problem faced during the testing of the diagnosis package is the relatively slow completion of the *posteriori* probability calculation, depending on the type of alarm event input to the package. If the alarm event is near the bottom of the fault tree with only primary events as inputs, then the time needed for completion of the probability evaluation is relatively fast. However, if the alarm event is the top event or near the top of a fault tree and there are several levels of secondary events, then the probability evaluation may take quite some time. This is due to the slow rate of completing the evaluation of the conditional probabilities of the consequences of the causal events.

The procedure for evaluating the *posteriori* probability of a consequence of a causal event is to first determine the probability of that event **ANDED** with the alarm event. Since there are several primary events common to both the consequence event and the alarm event, this makes the probability evaluation relatively slow. The problem is not serious if the common primary events are inputs to **OR** gates or **AND** gates in both the consequence event and the alarm event. The problem becomes more serious if the common primary events are inputs to **OR** gates in one event as well as inputs to **AND** gates in the other event. This problem is not due to any fault of the probability evaluation algorithm. It is due to the complexity of the evaluation when events which are not s-independent exist.

To make the probability evaluation process complete its task at a faster rate, an alternative *posteriori* probability evaluation subroutine was written which is very similar to the original subroutine. The only difference is that the conditional probability of the consequences of the causal events are not evaluated. Instead, a value of 2 is assigned for each of such events to show that the consequence will occur if it is found that the causal event is actually the cause of the alarm event.

An alternative fault tree display program called **DISALT** was produced which includes the alternative *posteriori* probability evaluation subroutine. The user has a choice of which program to use as an aid for diagnosing a fault in the plant. If it is found that the *posteriori* probability evaluation takes too long to complete its task when using **DISFAULT**, the user can break off the program execution and use **DISALT** instead to get the results more quickly.

8.1.7 : Significance of the Diagnosis Package

The package developed is of use in diagnosing both the cause and consequence of any single event. Using the package, the user can view which event is the most likely cause of the alarm event and what are the consequences that can occur if no remedial action is taken. The probability values can be used as a guide for tracing the most probable line of causes for the alarm event. Together with measurements of process plant variables, the user can discount the events that are false and proceed to find the causes of events that are true.

There are two particular aspects in usage of the fault diagnosis package that need to be discussed. They are observability and instrument failure. These are discussed below.

8.1.7.1 : Observability

Observability in process control is concerned with the identification of the process variables which can be observed and those which cannot. By analogy, the observability in fault trees may be said to be concerned with the identification of which deviations or events that are observable.

In the use of fault trees for design, all the deviations or events are, in theory, observable. In real-time applications, only the deviations of measurements and status indicators are directly observable by the computer. Thus the use of a fault tree in a process computer as a diagnostic tool is limited by the measurements available which can confirm whether the observable events are true or false. It can be noted that a human operator will observe a much greater amount of information.

Martin-Solis et al [41] proposed that in displaying a fault tree for alarm analysis, which is another term for fault diagnosis, failure paths which have observable events which are false can be eliminated. This will produce a concise fault tree since inactive branches have been removed. However, such a display can be misleading because too much trust is being placed on the measuring instruments giving the true state of the variables. Instruments may fail normal, that is, shows the normal value, whereas the actual state of the variables measured may have deviated. Thus, the observable events that are supposed to be true and active are not shown when the fault tree is displayed by using Martin-Solis's method.

The fault tree display developed in this research work does not eliminate any of the branches, whether active or inactive. This is to show that observable events, whether true or false cannot be trusted. An observable event which is active, i.e. true, is used to narrow down the search space for determining the actual cause of the occurring fault. If the primary events that cause the active observable event are found to be false, it indicates that there is an instrument malfunction. However, only after checking the other inactive branches and finding that all the primary events are false, can the instrument malfunction be considered as the actual cause of the alarm being raised and it is a false alarm.

8.1.7.2 : Instrument Failure

Instrument failure is a common type of failure on process plants. Instruments which are not in control loops which fail high or low may contribute to false alarms being raised. This will incur a high load on the operator to search exhaustively through the plant items to find the cause of the alarm, when in actual fact, the process plant is running normally.

The most misleading response of process plant variables is when instruments fail normal. For this type of failure, the process computer and the operator would not know that the process variable measured by the failed instrument is in a deviant state. This type of failure can only be diagnosed when data gathered from measurements of other process variables show that the process plant is behaving abnormally.

The problem of instrument failure has similarities and interactions with the observability problem. The diagnosis package does not show events that indicate failed normal instruments. However, the operator can infer the existence of a failed normal instrument when it is proved that the inactive branch associated with the instrument is actually active.

8.1.8 : Limitations of the Diagnosis Package

The package has been written and tested for off-line fault diagnosis. It can be used whenever there is an alarm to aid the operator to find the cause of the alarm. To confirm that the events shown in the fault tree display are true or false, the operator has to depend on the readings of the measuring instruments.

One important aspect the diagnosis package has not overcome is the problem of time lag for operators to take remedial actions. This is discussed below.

8.1.8.1 : Time Lag for Fault Diagnosis

The objective of real-time fault diagnosis is to take preventive measures before any hazardous accidents could occur when there is an abnormality in the process system. To achieve this, the fault diagnosis system has to exploit the time lags in the fault propagation. In this sense, the existence of time lags is fundamental for such application.

The fault tree is a pictorial representation of a set of system states. Effectively, this representation assumes that time is not a parameter in the fault propagation being considered.

In practice, all types of system exhibit some time lags as faults propagate. The time lags may or may not be significant in analysing how faults occur in the system.

In electro-mechanical systems, including most trip systems, the system behaviour can be adequately modelled without considering the time lags which exist. For this reason, fault tree analysis for such systems are valid. This is not to say that the time lags do not affect the system behaviour, but rather that if the lags are recognised by the analyst and are effectively invariant, they will contribute little more to an understanding of the failure mechanism.

In process systems, the time lags are more complex and vary with the plant conditions. This is due to the variability of process variables within a range of values. Thus the time for the responses to take effect will also vary. If several primary faults have the same consequence, the time for each fault to propagate to that consequence will be different. Also, if a primary fault has several consequences, the time for the consequences to take effect will be different. Thus, an event may have several time lags due to having several different causes that can activate it.

As an illustration, from the HAZOP study of the pilot distillation plant in chapter 6, the causes and consequence of a high liquid level in the reflux drum is shown in figure 8.2.

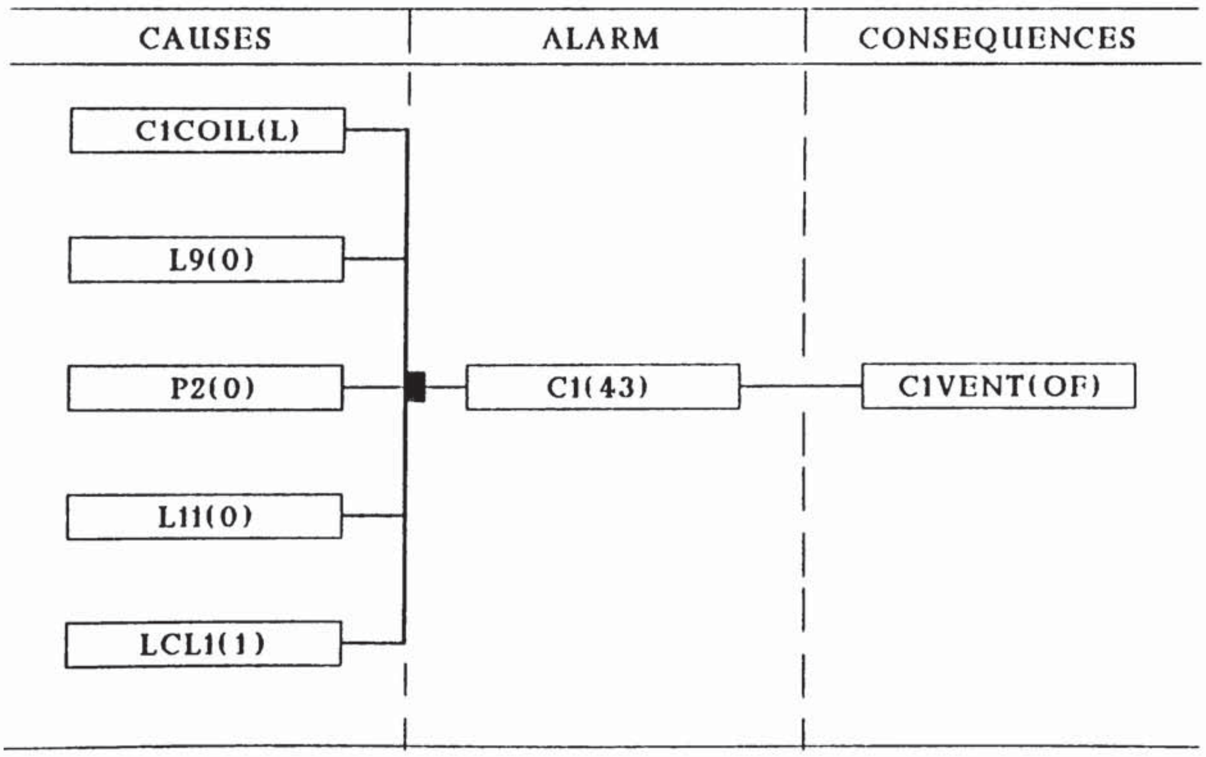


Figure 8.2 : Fault tree display showing causes and consequences of high liquid level in vessel C1.

The consequence of the high level in the reflux drum if no remedial action is taken is that liquid may rise higher until there is an overflow via the vent of the condenser. This is shown as C1VENT(OF) in the consequence column in figure 8.2. If the cause of the high level is due to leakage of water from the condenser cooling coil, shown as C1COIL(L), the time for the overflow to occur will depend on the rate of leakage and the input and output flow to the condenser. If the cause is due to either a blockage in the exit line, shown as L9(0), or the pump on the exit line not working, shown as P2(0), then the time for the overflow to occur will depend on the rate of input to the condenser. If the controller that is supposed to maintain the liquid level in the condenser at its set-point fails and instead acts to maintain the liquid at the high level, shown as LCL1(1), then the overflow would not occur. Thus, different causes of the high liquid level in the reflux drum will give different time lags for the overflow of liquid via the vent of the condenser.

There are two ways in which a hazardous accident can occur once an alarm has been raised. One way is when the accident is a direct consequence of the alarm event itself. For this case, measures can be taken quite easily to prevent the accident from occurring, if sufficient time lag is present. Another way is when the accident is a consequence of any one of the causes of the alarm. For this case, the operator has to diagnose which event caused the alarm before any preventive measures can be taken. Moreover, the time lapse between the causal event happening and the occurrence of the alarm event has to be known. This is needed for the operator to judge whether there is enough time for remedial action to be taken from the time the causal event has been found.

The fault diagnosis package developed in this research work does not show the time lags for the consequences to occur on the onset of an alarm. The reason is the complexity in obtaining the time lags.

One way of obtaining the time lags in real-time is to solve an adequate dynamic model of the process for the various failure modes using the current values of the measured variables as inputs. However, this would take quite a lot of the computer time and thus would defeat the purpose of using the diagnosing package, in that, this is an extra lag. The solution of the dynamic model is more complicated if the time lag for the occurrence of a consequence of a cause of the alarm is to be determined. This is due to the fact that the time when the causal event has occurred in the past is not known.

In the absence of actual time lag data, the operator has to estimate when hazardous incidents could occur based on the occurrence of a particular fault in the process. Sound estimation of the time lags depend upon the experience and engineering judgement of the operator. The experience can only be gained if the operator has faced similar circumstances previously or has undergone training for fault diagnosis. Since faults which can bring about hazardous incidents seldom occur, the training of operators so that they can gain such experiences can only be done using simulators. Marshall et al [116] and Yuan-Liang Su et al [117] have developed such training simulators for process plants to train operators in fault diagnosis. The diagnosis package developed in this work can be implemented for such training purposes.

8.2 : OTHER USES OF THE DIAGNOSIS PACKAGE

8.2.1 : Introduction

The use of the diagnosis package developed in this research work can be adapted for other uses apart from finding causes and consequences of an alarm event. It is envisaged that other uses of the diagnosis package are:

1) operator training,
2) determination of reliability of safety devices and measuring instruments,
3) preparation of operating manuals,

8.2.2 : Operator Training

A process plant that behaves abnormally will put a heavy mental load on the operators to find the causes so that remedial action can be taken to put the plant back to its normal operation. Errors in decision making and slow responses by the operators play the major part in most disastrous accidents [118, 119]. The importance of training operators to face unforeseen circumstances has already been discussed earlier. The diagnosis package can be used to train operators to realise how a specific cause can lead to other causes and what action to take to prevent occurrence of major disasters.

8.2.3 : Reliability of Safety Devices and Measuring Instruments

Safety devices that are installed on the process plant may fail when required to act, resulting in hazardous consequences. Measuring instruments that fail high or low will output false alarms. Instruments which fail normal are dangerous since the process computer or the operator would not know from such instruments that the plant is behaving abnormally. It is essential that safety devices and instrumentation on a process plant should be reliable so that the veracity of the measurements can be trusted and the frequency of failure should be very small. The diagnosis package can be used as a design tool to investigate and improve the reliability of the measurements. With the help of the fault tree display and the probability values, the designer can determine how reliable a safety device or measuring instrument for a particular process variable should be to justify the cost of installation.

8.2.4 : Operating Manuals

One of the difficult tasks for any plant management or process engineer is the writing of start up and shut down procedures [120]. The start up procedure is normally easier than the shut down procedure. An improper shut down can cause expensive wastes, incomplete products and a long delay. The diagnosis package can be used as an aid in understanding the sequences of events that arise from specific actions. With the aid of the package, the appropriate action which can be carried out more safely and quickly can be evaluated.

8.3 : RECOMMENDATION FOR FUTURE WORK

As it stands now, the diagnosis package is initiated by the user input of the alarm event via the keyboard. There are no on-line process variable measurements to the computer. This is not a major setback. The user can still use the diagnosis package with operators checking observable measurements to verify whether events displayed on the VDU are true or false so as to lead to the primary cause of the alarm.

The diagnosis package can be extended for use with on-line measurements so that the fault tree display can be initiated automatically. With the on-line measurements, verification of events can be done automatically. This will reduce the load of operators to check on the measuring instruments.

A common feature when a process plant is behaving abnormally is that there will be several alarms going off. Usually, most of the alarms are interelated. The package can be extended to show at the bottom of the VDU screen what alarms have been raised. The user can then choose to view the cause and effect relationships of any one of the alarm events. The fault tree display will then show which alarms are interelated that have a common cause. Also alarms that are inconsistent can be checked to see which one is true or to verify that the inconsistent alarms are false.

The present package only calculates the *posteriori* probability of every event in the fault tree data file once, based on the alarm event input by the user. With on-line information, the status of events which are false at one time may become true and vice versa as time proceeds. There is a need to update the *posteriori* probability information for such a situation. This facility can be included in future work of extending the diagnosis package.

In the present diagnosis package, the choice of which line of events that is to be traced to find the cause of the alarm is determined by the operator, guided by the *posteriori* probability values. A useful extension of this work is to include a facility where the package guides the operator of which line of events to search for the primary cause of the alarm. This can be done using an expert system technique with the information contained in the fault tree and the on-line measurements as the knowledge base for the system. Such application has been done by Kumamoto and Henley [54]. The extra facility of the fault tree display can be used by the operators to verify the findings of the expert system.

Another recommendation for future extension of the diagnosis package is to include time lag data so that operators would know how long they have to remedy an abnormal situation. The problem of obtaining the time lag for reponses of certain magnitudes of process variables has been discussed earlier. A general method of obtaining the time lag information will be very useful for fault diagnosis purposes.

8.4 : CONCLUSION

This work set out to:

1) investigate the use of the outcome of a HAZOP study for real-time fault diagnosis,
2) generate a fault tree display on the VDU of a computer using the information from the HAZOP study,
3) use the probability of occurrences of events as a guide for the fault diagnosis.

The major result of the work has been the development of a diagnosis package which can be used to diagnose the causes and consequences of any single fault in a process plant.

The achievements of the work can be summarised as follows:

1) A method of storing the HAZOP information in the computer has been developed so that the diagnosis package can easily extract the information for probability evaluations and generating the fault tree on the VDU of the computer.

2) A new fault tree probability evaluation algorithm has been developed based on the top-down recursive method developed by Page and Perry [109, 110] called TDPP . The new algorithm is called TDRA which evaluates the probabilities of every independent event in a fault tree in one traversal from the top event to the primary events. The number of recursions in TDRA is much less when compared to TDPP.

3) A new method of displaying the fault tree on the VDU of the computer has been developed which takes into account of the limited space available on the screen of the VDU.

The diagnosis package has been wriiten using Microsoft Quickbasic and is intended for use on any IBM PC compatible microcomputer.

The use of HAZOP studies to investigate the safety, reliability and operability of process plant is now accepted in the process industries. The result from the HAZOP studies is not wasted if it is used for real-time fault diagnosis as shown from the achievements of this work.

REFERENCES

1. Marshall, V.C.; "Major Chemical Hazards", Ellis Harwood Ltd., London, 1987, p. 6.

2. ibid reference 1, p. 7.

3. Health and Safety Executive; "Advisory Committee on Major Hazards - First Report", HMSO, London, 1976.

4. Health and Safety Executive; "Advisory Committee on Major Hazards - Second Report", HMSO, London, 1979.

5. Health and Safety Executive; "Advisory Committee on Major Hazards - Third Report", HMSO, London, 1980.

6. Lees F.P.; "Loss Prevention in the Process Industries", Vol. 1, Butterworths, London, 1980, p. 2.

7. Berenblut, B.J.; Menashe, J.; "Ranking Risks in Commerce and Industry", Developments '82, I.Chem.E. Symposium Series No. 73, I.Chem.E., London, 1982.

8. Health and Safety Executive; "A Guide to the HSW Act", HMSO, London, 1980.

9. ibid reference 1, pp. 442-445.

10. Anon.; "The Flixborough Disaster - Report of the Court of Inquiry", HMSO, London, 1975.

11. European Community Directive, "On the Major Accident Hazards of Certain Industrial Activities", 82/501/EEC, 1982.

12. ibid reference 1, p. 446.

13. Lawley, H.G.; "Operability Studies and Hazard Analysis", Loss Prevention, CEP, A.I.Ch.E., Vol. 8, 1974, pp.105-116.

14. Kletz, T.A.; "HAZOP and HAZAN", I.Chem.E., London, 1983.

15. Roach, J,; Lees, F.P.; "Some Features of and Activities in Hazard and Operability (HAZOP) Studies"; The Chem. Engineer, I.Chem.E., Oct. 1981, pp. 456-462.

16. Jones, M.C.; Lihou, D.A.; "CAFOS - The Computer Aid for Operability Studies", I.Chem.E. Symposium Series No. 97, I.Chem.E., 1986, pp. 249 - 260

17. Lihou, D.A.; "Computer Aided Operability Study", Loss Prevention Bulletin No. 051, I.Chem.E., 1983.

18. Browning, R.L.; "Use of Fault Tree to Check Safeguards", Loss Prevention, Vol. 12, CEP, A.I.Ch.E., 1979

19. Anon., "The Technical Lessons of Flixborough", A Symposium in Dec. 1975, I.Chem.E., 1976

20. Edwards, E., Lees, F.P.; "Man and the Computer in Process Control", I.Chem.E., 1972, p.118

21. Rasmussen, J.; "Notes on Diagnositc Strategies in Process Plant Environment", RISO-M-1983, RISO National Laboratorym, Denmark, Jan. 1978.

22. Lees, F.P.; "Process Computer Alarm and Disturbance Analysis: A Review of the State of the Art", Computers and Chem. Engg., Vol. 7, No. 6, 1983, pp. 669 - 694

23. Andow, P.K., Lees, F.P.; "Process Plant Alarm Systems: General Considerations", in Loss Prevention and Safety Promotion in the Process Industries, Buschman, C.H. (Ed.), 1974, pp. 299 - 307

24. Kramer, M.A. "Malfunction Diagnosis Using Quantative Models with Non-Boolean Reasonong in Expert Systems", A.I.Ch.E., Vol. 33, No. 1, 1987, pp. 130 - 140

25. Himmelblau, D.M. "Fault Detection and Diagnosis in Chemical and Petrochemical Processes", Elsevier Scientific Publishing Co., Amsterdam, 1978, p. 13.

26. Isermann, R.; "Process Fault Detection Based on Modelling and Estimation Methods - A Survey", Automatica, Vol. 20, No. 4, 1984, pp. 387 - 404

27. ibid reference 25, pp. 168 - 273

28. ibid reference 25, p. 184

29. ibid reference 25, pp. 273 - 342

30. Berenblut, B.J.; Whitehouse, H.B.; "A Method for Monitoring Process Plant Based on a Decision Table Analysis", The Chem. Engr., March 1977, pp. 175 181.

31. Munday, G.; "On-Line Monitoring and Analysis of Hazard of Chemical Plant", Loss Prevention and Safety Promotion,

32. ibid reference 25, p. 293

33. Rasmussen, J.; Jensen, A.; "Mental Procedures in Real Life Tasks: A Case Study of Electronic Trouble Shooting", Ergonomics, Vol. 17, 1974, p. 293

34. Edwards, F.; Lees, F.P.; "Man and Computer in Process Control", I.Chem.E., London, 1972

35. Kay, P.C.M.; "On-Line Computer Alarm Analysis", Ind. Electron., Vol. 4, 1966, p. 50

36. Welbourne, D.; "Alarm Analysis and Display at Wyfla Nuclear Power Station", Proc. IEE, Vol. 115, 1968, p. 1726

37. Patterson, D.; "Application of Computerised Alarm Analysis System To Nuclear Power Station", Proc. IEE, Vol 115, 1968, p. 1858

38. Baarth, J.; Maarleveld, A.; "Operational Aspects of a D.D.C. System", I.Chem.E. Symp. Series No. 24, 1967, p. 23

39. Andow, P.K.; "Real-Time Analysis of Process Plant Using a Mini Computer", Computers and Chem. Engg., Vol 5, 1980, pp. 143 - 155

40. Andow, P.K.; Lees, F.P.; "Process Computer Alarm Analysis: Outline of a Method Based on List Processing", Trans. I.Chem.E., Vol. 53, 1975, p. 195

41. Martin-Solis, G.A.; Andow, P.K.; Lees, F.P.; "Fault Tree Synthesis for Design and Real-Time Applications", Trans. I.Chem.E., Vol. 60, 1982, pp. 14 - 25

40. Andow, P.K.; Lees, F.P.; "Process Computer Alarm Analysis: Outline of a Method Based on List Processing", Trans. I.Chem.E., Vol. 53, 1975, p. 195

41. Martin-Solis, G.A.; Andow, P.K.; Lees, F.P.; "Fault Tree Synthesis for Design and Real-Time Applications", Trans. I.Chem.E., Vol. 60, 1982, pp. 14 - 25

42. Tsuge, Y.; Shiozaki, J.; Matsyama, H.; "Fault Diagnosis Algorithms Based on the Signed Directed Graph and Its Modifications", I.Chem.E. Symp. Series, No. 92, 1985, p. 133.

43. Tsuge, Y.; Matsuyama, H.; "Equivalence Between The Two Formulations of the Problems of Fault Diagnosis of the Chemical Process", Memoirs of the Faculty of Engineering, Kyushu University, Japan, Vol. 44, No. 3, 1984, p. 360

44. Iri, M.; Aoki, K.; "A Graphical Approach to the Problem of Locating the Origin of the System Failure", J. of the Operation Research Society of Japan, Vol. 23, No. 24, 1980, p. 295

45. Tsuge, Y.; Shiozaki, J.; Matsuyama, H.; O'shima, E.; Iguchi, Y.; Fuchigami, M.; Matshushita, M.; "Feasibility Study of a Fault Diagnosis System For Chemical Plants", International Chem. Engg., Vol. 25, No. 4, 1985, p.660.

46. Shiozaki, J.; Matsuyama, H.; Tano, K.;. O'shima, E.; "Fault Diagnosis of Chemical Processes By The Use of Signed Directed Graphs - Extension To Five-range Patterns of Abnormality", International Chem. Engg., Vol. 25, No. 4, p. 651.

47. Umeda, T.; Kuriyama, T.; O'shima, E.; Matsuyama, H.; "A Graphical Approach to Cause and Effect Analysis of Chemical Processing Systems", Chem. Engg. Sci., Vol. 35. 1980, p. 2379

48. Lind, M.; "The Use of Flow Models for Automated Plant Diagnosis" in Human Detection and Diagnosis of System Failures, Rasmussen, J.; Rouse, W.B. (Ed.), Plenum Press, 1980, p. 411

49. Andow, P.K.; "Fault Diagnosis Using Intelligent Kwnoledge Based Systems", Chem. Engg. Res. & Design, Vol. 63, 1985, p. 368

50. Lees, F.P.; "Computer Support for Diagnostic Tasks in Process Industries", in Human Detection and Diagnosis of System Failures, Rasmussen, J.; Rouse, W.B.; (Ed.), Plenum Press, 1980

51. Lapp, S.A.; Powers, G.J.; "Computer-Aided Synthesis of Fault Trees", IEEE Trans. on Reliability, April 1977, pp. 2 - 13

52. Galluzo, M.; Andow, P.K.; "Expert Systems in Chemical Engineering", EFCE XVIII Congress: The Use of Ccomputer in Chemical Engineering, Giardini, Nuxos (Italy), April 1987, pp. 661 - 667

53. Klemes, J.; Krus, A.; "Strategy of Development of Expert Systems for Prompt Solution of Failures", EFCE XVIII Congress: The Use of Computer in Chemical Engineering, Giardini, Nuxos (Italy), April 1987, pp. 851 - 857

54. Kumamoto, H,; Ikenji, K.; Inoue, K.; Henley, E.J.; "Application of Expert System TEchnique to Fault Diagnosis", The Chem. Engg. Journal, Vol. 29. 1984, pp. 1 - 9

55. Niida, K.; Itoh J.; Umeda, T.; Kobayashi, S.; Ichikawa, A.; "Some Expert System Experiments in Process Engineering", Chem. Engg. Res. & Des., Vol. 64, 1986, pp 372 - 382

56. Chester, D.; Lamb, D.; Dhurjati, P.; "Rule-Based Computer Alarm Analysis in Chemical Process Plants", Conference on Computer Techniques, Micro Delcon 84, IEEE, March 1984, pp. 22 - 29

57. Yamada, N.; Motoda, H,; "A Diagnosis Method of Dynamic System Using Knowledge on System Description", Proc. 8th Int. Joint Conf. on Artificial Intelligence, Karsluhe, W. Germany, 1983, pp. 225 - 229

58. Kramer, M.A.; Palowitch Jr., B.L.; "A Rule-Based Approach to Fault Diagnosis Using the Signed Directed Graph", A.I.Ch.E. J., Vol. 33, No. 7, 1987, pp. 1067 - 1078

59. Nelson, W.R., "Reactor: An Expert System For Diagnosis and Treatment of Nuclear Reactor Accidents", Proc. of the National Conf. on Artificial Intelligence, 1982, pp 296 - 301

60. Sgurev, V.; Dochev, D.; Dichev, C.; Agre, G.; Markov, Z.; "An Approach to Building Technical Diagnostic Expert Systems", Computers & Artificial Intelligence, Vol. 5, No. 2, 1986, pp 103 - 115

61. Fox, M.S.; Lowenfield, S.; Kleinosky, S.; "TEchniques for Sensor-Based Diagnosis", Proc. 8th, Int. Joint Conf. on Artificial Intelligence, Karsluhe, W. Germany, 1983, pp.. 158 - 163

62. Lihou, D.A.; "Aiding Process Plant Operators in Fault Finding and Corrective Action", in Human Detection and Diagnosis of System Failures, Rasmussen, J.; Rouse, W.B.; (Ed.), Plenum Press, 1980, pp. 501 - 521

63. Bainbridge, L.; "Analysis of Verbal Protocol From A Process Control Task", in The Human Operator in Process Control, Edwards, E.; Less, F.P.; (Ed.), Taylor and Francis, London, 1974, p. 146

64 Edwards, E.; Lees, F.P.; (Ed.) "The Human Operator in Process Control", Taylor and Francis, London, 1974

62. Lihou, D.A.; "Aiding Process Plant Operators in Fault Finding and Corrective Action", in Human Detection and Diagnosis of System Failures, Rasmussen, J.; Rouse, W.B.; (Ed.), Plenum Press, 1980, pp. 501 - 521

63. Bainbridge, L.; "Analysis of Verbal Protocol From A Process Control Task", in The Human Operator in Process Control, Edwards, E.; Less, F.P.; (Ed.), Taylor and Francis, London, 1974, p. 146

64 Edwards, E.; Lees, F.P.; (Ed.) "The Human Operator in Process Control", Taylor and Francis, London, 1974

65. Michie, D.; (Ed.), "Introductory Readings in Expert Systems", Gordon and Breach, New York, 1983

66. Chemical Industries Safety & Health Council, "A Guide for Hazard and Operability Studies", Chemical Industries Association Ltd., London, 1977, p. 7

67. Elliott, D.M.; Owen, J.M.; "Critical Examination in Process Design", The Chem. Engr., Nov. 1968, pp. 377 - 383

68. ibid reference 14, p. 8

69. ibid reference 66, p. 6

70. Taylor, J.R.; "Evaluation of Costs, Quality and Benefits for Six Risk Analysis Procedures", Internal Report, Electronics Dept., RISO N-14-82, Denmark, May 1982

71. Lihou, D.A.; "Computer Aided Operability Studies For Loss Control", 3rd, Int. Symp. in Loss Prevention and Safety Promotion in the Process Industries, Basle, Switzerland, Vol. 2, Sept. 1980, p. 579

72. ibid. reference 25, p. 346

73. Fussell, J.B.; "Fault Tree Analysis - Concepts and Techniques", NATO Adv. Study, Inst. on Generic Techniques of System Reliability Assessment, Nordhoff Publishing Co., Liverpool, July 1973, pp. 133 - 162

74. Spiegelman, A.; "Risk Evaluation of Chemical Plants", Loss Prevention, CEP, A.I.Ch.E., Vol. 3, 1969, pp. 1 - 10

75. Browning, R.L., "Use of Fault Tree To Check Safegaurds", Loss Prevention, CEP, A.I.Ch.E., Vol. 12, 1979, pp. 20 - 26

76. Lee, K.K.; "A Compilation Technique For Exact System Reliability", IEEE Trans. on Reliability, Vol R-30, No. 3, 1981, pp. 284 - 288

77. U.S. Nuclear Regulatory Commission; "Reactor Safety Study - An Assessment of Accident Risk in U.S. Commercial Nuclear Power Plants", WASH-1400 (NUREG - 75/10/014)

78. Fussell, J.B.; "A Formal Methodology For Fault Tree Construction", Nuclear Sci. & Engg., Vol. 52, 1973, pp. 421 - 432

79. Powers, G.J.; Tompkins, F.C.; "Computer Aided Synthesis of Fault Tree For Complex Processing", NATO Adv. Study, Inst. on Generic Techniques of System Reliability Assessment, Nordhoff Publishing Co., Liverpool, July 1973, pp. 307 - 314

80. Powers, G.J.; Tompkins, F.C.; "Fault Tree Synthesis for Chemical Processes", A.I.Ch.E. J., Vol. 20, 1974. pp. 376 - 387

81. Rudd, D.F.; Watson, C.C.; "Strategy of Process Engineering"; Wiley, New York, 1968

82. Salem, S.L.; Apostolakis, G.E.; Okrent D.; "A New Methodology For The Computer-Aided Construction of Fault Trees", Annals of Nuclear Energy, Vol. 4, 1977, pp. 417 - 433

83. Henley, F.J.; Kumamoto H.; "Reliability Engineering and Risk Assessment", Prentice Hall Inc., New Jersey, U.S.A., 1981, p. 310

84. Hastings, N.A,; Peacock, J.B.; Statistical Distributions", Butterworths, London, 1974.

85. Lihou, D.A.; "Estimation/Calculation of Probabilities", Short Course on Hazard Evaluation and Control, Dept. of Chem. Engg., University of Aston, Sept. 1982

86. Fussell, J.B.; Marshall, N.H.; "MOCUS - A Computer Program To Obtain Minimal Sets From Fault Trees", ANCR-1156, Aerojet Nuclear Co., Idaho Falls, USA, March 1974

87. Semanderes, S.N.; "ERLAFT - A Computer Program For The Efficient Logic Reduction Analysis of FAult Trees", IEEE Trans. on Nuclear Science, Vol. NS-18, No. 1, 1971, pp. 481 - 487

88. Richardson, M.; "College Algebra", Prentice Hall, 1958

89. Wheeler, D.B.; Hsuan, J.S.; Duersch, R.R.; Roe, G.M.; "Fault Tree Analysis Using Bit Manipulation", IEEE Trans. on Reliability, Vol. R-26, June 1977, pp. 95 - 99

90. Caldarola, L.; Wickenhouser, W.; "The Karlsruhe Computer Program For The Evaluation Of The Availability And Reliability Of Complex Repairable Systems", Nuclear Engg. & Design, Vol. 43, 1977, pp 463 - 470

91. Bengiamin, N.W.; Brown, B.A.; Schenck, K.F.; "An Efficient Algorithm For Reducing The Complexity Of Computation In Fault Tree Analysis", IEEE Trans. on Nuclear Science, Vol. NS-23, Oct. 1976, pp. 1442 - 1446

92. Fussell, J.B.; Vesely, W.E.; "A New Methodology For Obtaining Cut Sets For Fault Trees", Trans. Am. Nuclear Society, Vol. 15, No. 1, 1972, p. 263

93. Rasmusson, D.M.; Marshall, N.H.; "FATRAM - A Core Efficient Cut Set Algorithm", IEEE Trans. on Reliability, Vol. R-37, Oct. 1978, pp. 250 - 253

94. Bennetts, R.G.; "On The Analysis OF Fault Trees", IEEE Trans. on Reliability, Vol. R-24, Aug. 1975, pp. 175 - 185

95. Nakashima, K.; Hattori, Y.; "An Efficient Bottom-up Algorithm For Enumerating Minimal Cut Sets Of Fault Trees", IEEE Trans. on Reliability, Vol. R-28, Dec. 1986, pp. 559 - 561

96. Limnios, N.; Ziani, R.; "An Algorithm For Reducing Cut Sets In Fault Tree Analysis", IEEE Trans. on Reliability, Vol. R-35, Dec. 1986, pp. 559 - 561

97. Garibba, S.; Mussio, P.; Naldi, F.; Reina, G.; Volta, G.; "Efficient Construction Of Minimal Cut Sets From Fault Trees", IEEE Trans. on Reliability, Vol. R-26, June 1977, pp. 88 - 93

98. Kumamoto, H.; Henley, E.J.; "Top-down Algorithm For Obtaining Prime Implicant Sets Of Non-Coherent Fault Trees", IEEE Trans. on Reliability, Vol. R-27, Oct. 1978, pp 242 - 249

99. Magee, D.; Refsum, A.; "RESIN, A Desktop Computer Program For Findiong Cut Sets", IEEE Trans. on Reliability, Vol. R-30, Dec. 1981, pp. 407 - 410

100. Pullen, R.A.; "AFTP - Fault Tree Analysis Program", IEEE Trans. on Reliability, Vol. R-33, June 1984, p. 171

101. Jasmon, G.B.; Kai, O.S.; "A New Technique in Minimal Path And Cut Set Evaluation", IEEE Trans. on Reliability, Vol. R-34, June 1985, p. 136

102. Vesely, W.E.; "Reliability and Fault Tree Applications at the NTRS", IEEE Trans. on Nuclear Sci., Vol. NS-18, No. 1, 1971, pp. 472 - 479

103. Vesely, W.E.; "A Time-dependent Methodology For Fault Tree Evaluation", Nuclear Engg. & Design, Vol. 13, 1970, p. 337

104. Bennetts, R.G.; "Analysis of Reliability Block Diagrams By Boolean Technique", IEEE Trans. on Reliability, Vol. R-31, June 1982, pp. 159 - 166

105. Jiongsheng, L.; "A New Approach For Fault Tree Analysis", Scientia Simica Series A, Vol. 25, Sept. 1982, pp. 983 - 992

106. Ramadaan, S.Y.; "Reliability Analysis For Hazard and Operability Studies", PhD Thesis, University ofAston, Birmingham, UK, 1987

107. Feo, T.; "PAFT 77 - Program For the Analysis of Fault Trees", IEEE Trans. on Reliablity, Vol. R-35, April 1986, pp. 48 - 50

108. Koen, B.; Carnino, A.; "Reliability Calculations With A List Processing Technique", IEEE Trans. on Reliability, Vol. R-23, April 1974, pp. 43 - 50

109. Page, L.B.; Perry J.E.; "A Simple Approach To Fault Tree Probabilities", Computers & Chem. Engg., Vol. 10, No. 3, 1986, pp. 249 - 257

110. Page, L.B.; Perry J.E.; "An Algorithm For Exact Fault Tree Probabilities Without Cut Sets", IEEE Trans. on Reliability, Vol. R-35, Dec. 1986, pp. 544 - 558

111. Sax, N.I.; "Dangerous Properties of Industrial Materials", 6th Edition, Van Nostrand Reinhold, 1984

112. Dale, S.; "Computer Control of Chemical Process Plant with Special Reference to Distillation", PhD Thesis, University of Aston, Birmingham, United Kingdom, 1980.

113. Shafii, A.F.; "The use of Microprocessors in the Control of Chemical Process Plant with special reference to the use of Distributed Processors", PhD Thesis, University of Aston, Birmingham, U.K., 1983.

114. Folami, T.O.; "Application of Modern Control Techniques To Distillation", PhD Thesis, University of Aston, Birmingham, United Kingdom, 1989.

115. Anon., "Microsoft QuickBASIC Compiler for IBM Personal Computers and Compatibles - Version 3.0", Microsoft Corporation, 1986.

116. Marshall, E.C.; Scanlon, K.E.; Shepherd, A.; Duncan K.D.; "Panel Diagnosis Training for Major-Hazard Continuous-Process Installations", The Chem. Engr., February 1981, pp. 66 - 69.

117. Su, Yuan-Liang; Govindaraj, T.; "Fault Diagnosis in a Large Dynamic System: Experiments on a Training Simulator"; IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-16, No. 1, 1986, pp. 129 - 141.

118. Lees, F.P.; "Loss Prevention in the Process Industries", Vol. 2, Butterworths, London, 1980, Appendix 1, pp. 863 - 881.

119. Lihou, D.; "Bhopal and Beyond"; The Chem. Engr., No. 414, I.Chem.E.; U.K., May 1985, p. 15.

120. Nisenfield, A.E.; "Shutdown Features of In-Line Process Control", Loss Prevention, CEP, A.I.Ch.E., Vol. 6, 1972, pp. 1 - 3.

APPENDICES

APPENDIX A

RULES FOR FORMULATING CAUSE EQUATIONS

The following tables give the rules for formulating cause equations devised by Lihou [17] to avoid illogical fault trees.

Table A.1 : Rules for Pipelines

Index	Meaning	Cause
11	FLOW NO	a) No flow in the line(s) immediately upstream. b) No flow at the node where the line leaves an equipment. c) The supply tank empty. d) A valve shut in the line. e) A filter fully blocked. f) A pump in the line stopped. g) A blockage in the line.
12	FLOW LESS	a) Less flow in the supply line(s) immediately upstream. b) Less flow at the node where the line leaves an equipment. c) Vent blocked on a storage tank. d) More flow in a branch line (judged by the relative magnitude of the normal flows). e) A valve insufficiently open downstream of a pressure control. f) A pressure or flow controller set too low. g) A pressure or flow transmitter indicating too high. h) A pnuematic trip valve leaking to vent. i) A flow control valve fully open **AND** a valve in the line insufficiently open or other blockage. j) A valve insufficiently open; filter or exchanger tubes partly blocked in a line without flow or pressure control. k) Bypass around a pump open or leaking. Leak(s) downstream of a flow transmitter. m) Pump delivery reduced by cavitation or speed reduction.

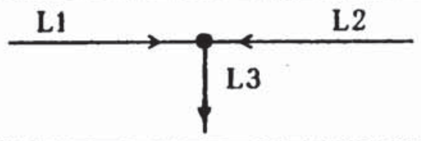
Table A.1 (a) : (continued from table A.1)

Index	Meaning	Cause
13	FLOW MORE	a) More flow in line(s) downstream. b) More flow at the nodes where the line leaves or enter an equipment. c)* Leak in the line(s) downstream without flow control. d)* Leak upstream of flow controllers in downstream line(s) e) Pressure or flow controller set too high. f) Pressure or flow transmitter indicating too low. g) Control valve stuck open. h) Bypass around a control valve fully open. i) Bypass flow around a pump too low.
14	FLOW AS WELL AS	a) Contamination of inlet line. b) Contamination in storage tank. c) Tubes leaking in exchanger. d) Steam or nitrogen purge valves leaking. e) Supply tank level low allowing air plus liquid to enter the discharge pipe. f) Steam trap leaking or its bypass valve leaking.
15	FLOW FLUC-TUATING	a) On/off controller or an unstable control loop.
16	FLOW REVERSE	a) Differential pressures. b) Unequal levels **AND** no non-return valve fitted or non-return valve leaking.
17	FLOW OTHER THAN	a) Gas or vapour entering the discharge pipe from an empty supply tank.
22	TEMPERA-TURE LESS	a) Low temperature at the node where the line leaves an equipment. b) Trace heating not on or failed. c) Temperature controller set low. Temperature indicator indicating too high.
23	TEMPERA-TURE MORE	a) High temperature at the node where the line leaves an equipment. b) Trace heating on when it should be off. c) Temperature controller set high. d) Temperature indicator indicating too low.

Table A.1 (b) : (continued from table A.1 (a))

Index	Meaning	Cause
32	PRESSURE LESS	a) Less pressure in the supply line. b) Less pressure at the node where the line leaves an equipment. c) A valve insufficiently open down stream of a pressure controller. d) Pressure or flow controller set low. e) Pressure of flow transmitter indicating too high. f) Pnuematic trip valve leaking to vent. g) Control valve fully open **AND** a valve in the line insufficiently open or other blockage.
33	PRESSURE MORE	a) High pressure at the beginning of the line and no pressure of flow control. b) High pressure at the end of the line and flow is controlled. c) Pressure or flow control set too high. d) Control valve stuck open. e) Bypass valve around a control valve is fully open. f) Flow is controlled and flow transmitter indicating too low. g) Pressure is controlled and pressure transmitter indicating too low. h) Pressure regulator set high or its valve stuck open. (Downstream pressure controller valve only if two are used in series)
* Leaks from a line are equated to drain or bleed valves open or leaking, filters leaking, etc.		

Table A.2 : Rules for pipeline junctions



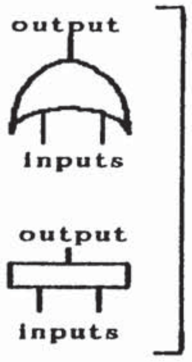
Event	Meaning	Cause
L3(11)	FLOW NO in L3	No flow in L1 **AND** no flow in L2.
L3(12)	FLOW LESS in L3	Low flow in L1 **OR** low flow in L2.
L3(13)	FLOW MORE in L3	High flow in L1 **OR** high flow in L2.
L3(22)	TEMPERATURE LESS in L2.	Low temperature in L1 **OR** low temperature in L2.
L3(23)	TEMPERATURE MORE in L3	High temperature in L1 **OR** high temperature in L2.
L3(32)	PRESSURE LESS in L3.	Low pressure in L1 or low pressure in L2.

Table A.2 : Rules for vessels

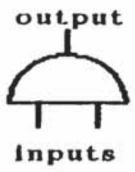
Index	Meaning	Cause
41	LEVEL NO	a) Vessel is empty.
42	LEVEL LESS	a) Level transmitter indicating too high. b) Level controller set low.. c) No flow into vessel. d) Low flow into vessel. e) Lower isolation valve on the level indicator or trasnsmitter is closed. f) Level control valve on the discharge line stuck open. g) Bypass valve around a down stream level control valve open.
43	LEVEL MORE	a) Level transmitter indicating too low. b) Level controller set high. c) Upper isolation valve on the level indicator or transmitter closed. d) Level control valve stuck open if on a supply line or stuck closed if on a discharge line. e) Bypass valve around the level control valve in a supply line open. f) Kick-back liquid flow going to a non-supply storage tank.

APPENDX B

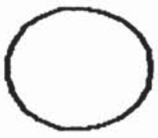
SYMBOLS COMMONLY USED IN FAULT TREE GRAPHICAL REPRESENTATION



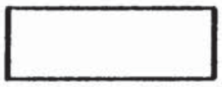
OR Gate: The **OR** gate indicates that the output event occurs if any of the input events occur.



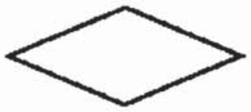
AND Gate: The **AND** gate indicates that the output event occurs only when all the input events occur.



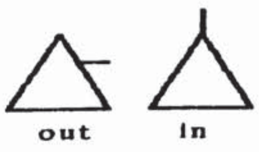
BASIC Event: The BASIC event represents a basic equipment fault or failure that requires no further development into more basic faults or failures.



INTERMEDIATE Event: The INTERMEDIATE event represents a fault event that results from the interactions of other fault events that are developed through logic gates such as those defined above.



UNDEVELOPED Event: The UNDEVELOPED event represents a fault event that is not examined further because information is unavailable or because its consequence is insignificant.



TRANSFER Symbol: The TRANSFER IN symbol indicates that the fault tree is developed further at the occurrence of the corresponding TRANSFER OUT symbol (e.g. on another page). The symbols are labelled using numbers or a code system to ensure that they can be differentiated.

APPENDIX C

FAILURE RATE AND HUMAN ERROR PROBABILITY DATA

Table C.1: Failure rates for chemical plant items

Plant Item	Remarks	Failure Rate
PUMPS AND PIPEWORK		
Pump Failures	Starting	10^{-3}/day
	Stopping	10^{-4}/day
	Abnormal running	3×10^{-5}/yr.
	Catastrophic failure	10^{-4}/yr.
Piping (catastrophic rupture)	up to 50 mm. diameter	10^{-10}/m. hr.
	50 - 150 mm. diameter	3×10^{-11}/m. hr.
	above 150 mm. diameter	10^{-11}/m. hr.
Piping (significant leakage)	up to 50 mm. diameter	10^{-9}/m. hr.
	50 - 150 mm. diameter	6×10^{-10}/m. hr.
	above 150 mm. diameter	3×10^{-11}/m. hr.
Hose rupture	Heavily stressed	4×10^{-5}/hr.
	Lightly stressed	4×10^{-6}/hr.
Loading arm	Catastrophic failure	4×10^{-5}/hr.
	Leak	3×10^{-6}/hr.
VALVES		
Pneumatic control valves	All failures	0.3 /yr.
Solenoid valves	All failures	0.3 /yr.
Manual valves	All failures	0.1 /yr.
Motor-operated valves	Fails to operate	10^{-3}/day
	Blockage	10^{-4}/day
	External leak	10^{-8}/hr.
Pressure relief valve	Blocked	0.001 /yr.
	Lifts heavy	0.004 /yr.
	Lifts light	0.06 /yr.
Vacuum relief valve	Fails to operate	0.005 /yr.

Table C.1 (a) : continuation of table C.1

Plant Item	Remarks	Failure Rate
MEASURING DEVICES (Failure to sense variable)		
Level	DP cell	0.43 /yr.
	Float type	0.41 /yr.
Temperature	Thermocouple	0.17 /yr.
	Resistance thermometer	0.14 /yr.
	Temperature transducer	0.29 /yr.
Pressure	All types	0.47 /yr.
	Guage	0.09 /yr.
Gas detectors	All types	2.5 /yr.
CONTROLLING AND TRANSMITTING DEVICES		
Controller	General failure	0.29 /yr.
	Set point	0.14 /yr.
Control loops	PIC	0.8 /yr.
	PRC	0.9 /yr.
	FIC	1.0 /yr.
	FRC	1.4 /yr.
	LIC	1.6 /yr.
	LRC	1.5 /yr.
	TIC	0.6 /yr.
	TRC	1.3 /yr.
Switch (failure to operate)	Pressure	0.15 /yr.
	Manual	10^{-5}/day
	Push button	4.4×10^{-3}/yr.
Relay (failure to operate)	Heavy duty electrical	0.44 /yr.
	Pneumatic	0.17 /yr.
Impulse Line	Blocked	0.03 /yr.
	Leaking	0.06 /yr.
Air supply line	Blocked / crushed	0.01 /yr.
	Fractured/holed	0.01 /yr.
Trip systems	General	0.5 /yr.
Audible alarm	Fails to sound	2×10^{-5}/day

Table C.1 (b) : continuation of table C.1 (a)

Plant Item	Remarks	Failure Rate
Electrical Terminal	Loss of supply	0.005 /yr.
Fire Alarm System		10^{-3}/day
GENERAL ELECTRICAL EQUIPMENT		
Standby Battery	Incorrect output	10^{-6}/hr.
Motors	Fails to start	3×10^{-4}/day
	Stops	7×10^{-6}/hr.
Emergency Diesel System	Fails to start	3×10^{-2}/day
	Stops	0.003 /hr.
VESSELS		
Pressure	Catastrophic rupture	10^{-6}/yr.
	Serious leak	10^{-5}/yr.
Atmospheric Tanks	Catastrophic rupture	6×10^{-6}/yr.
	Serious leak	10^{-4}/yr.
Double-walled refrigerated	Inner and outer wall	10^{-6}/yr.
	Inner tank	2×10^{-5}/yr.

Envionmental Effects

The rates in table C.1 are for perfect environments. Rates of up to 4 times larger are possible under poor conditions. This environmental factor corresponds to the shape parameter of the Weibull distribution.

Table C.2 : Probabilities for Human Errors

Probability Value	Event
1.0	Failure to operate second step of two closely related actions. Failure to act correctly in 60 seconds in high stress alarm condition.
0.9	Fails to respond correctly to an alarm within 5 minutes.
0.5	Fails to detect undesired position of valves during tour of plant.
0.25	General error rate under high stress conditions.
0.1	Alters status of wrong switch whilst leavingh correct switch in unchanged status. Personnel fail to check hardware on shift change unless required by checklist. Supervisor fails to recognise initial error of operator. Operator fails to respond correctly to an alram after 30 minutes, under extreme stress.
0.065	Failure to follow instruction.
0.03	Simple arithmatic errors with self-checking but without calculation.
0.016	Improperly adjusting mechanical linkage.
0.01	General errors of omission. Failure to respond correctly to an alarm after several hours under high stress.
0.007	Failure to solder joint correctly.
0.005	Incorrect reading of guage. Installs wrong size of line orifice.
0.0047	Incorrect hose connection. Incorrect tightening of nut.
0.003	General errors of omission. Misreading label and therefore selecting wrong switch. Omission of an action embedded in a procedure.
0.002	Drilling and tapping wrong size.
0.0018	Failure to close valve completely.
0.0015	Bending pins when assembling connectors.
0.001	Failure to observe. Omission of parts when assembling equipment. Failure to take correct action after observing chart reading. Selection of wrong switch which is clearly different in shape or location to the correct switch.
6 x 10	Failure to install nuts, bolts.
3 x 10	Failure to take action. Failure to observe audible alarm.
10^{-4}	Incorrect adjustment of torque on fluid lines. Selection of key operated switch rather than a non-key switch.

APPENDIX D

FEATURES OF MICROSOFT QUICKBASIC°

Microsoft Quickbasic [115] provides a complete development environment, including a program editor and enhanced debugging mode, designed to speed up the compile-debug-edit cycle. Quickbasic also has an easy-to-learn, menu-orientated user interface, in which commands can be selected from menus using the keyboard, an optional mouse or both. Once Quickbasic is started, programs can be compile, run, debug, edit, and recompile without leaving Quickbasic. During compilation, Quickbasic "remembers" error locations, so that when compilation is finished, a special search command can help find and correct the errors quickly.

Quickbasic is an extension of interpreted BASIC*, BASICA* and GW-BASIC+ Programs written in these versions of BASIC are compatible with Quickbasic. The programs have to be saved in ASCII format so that they can be loaded and run in the Quickbasic environment. There are some extra features in Quickbasic to help in writing programs that run faster and are easier to maintain and debug. These features are described below.

- Subprograms

Programs, especially large ones, are easier to maintain and debug if they are divided into smaller, more manageable parts. Quickbasic provides a way to segment programs, i.e. the subprograms. The use of subprograms allow for distinct modules to be written that perform tasks that are frequently used over and over in many different programs. Variables and arrays in a subprogram are local and will not

---

° Quickbasic is a registered trademark of Microsoft Corporation.

* BASIC and BASICA are registered trademarks of International Business Machine Corporation.

+ GW-BASIC is a registered trademark of Microsoft Corporation.

affect those in the main program which have the same name, unless:

- the variables and arrays are passed from the main program in an argument list when the subprogram is called,
- the variables and arrays are declared global at the beginning of main program.

• <u>Structured programming support</u>

There are several features that Quickbasic provides for writing a better structured programs. They are:

- multiline **IF . . . THEN . . . ELSE** which provides more flexibility than single-line **IF . . THEN . . ELSE**. It allows more complicated case structures while preserving readability of the program.
- **SELECT CASE** statements which simplify complex condition testing.
- **WHILE . . . WEND** statements which allow a series of statements to be executed in a loop as long as a given condition is true.
- **DO . . . LOOP** statements which is another form of executing a series of statements in a loop. Execution within the loop continues as long as a condition is true or until a condition becomes true.
- **EXIT** statements which provide alternative exits from **DO...LOOP** and **FOR...NEXT** statements.

• <u>Alphanumeric labels</u>

Quickbasic does not require line numbers and allows the use of alphanumeric labels. With alphanumeric labels, statements and subroutines can be given descriptive names which makes the program easier to read and can provide simpler debugging.

• <u>Flexible array dimensioning</u>

The size of arrays can be dynamically declared at run time, thus providing more efficient use of memory.

• <u>Large numeric arrays</u>

Quickbasic allows as many 64 kilobytes numeric arrays as will fit into memory.

APPENDIX E

SOURCE CODE OF THE PROGRAM TRANSLAT

```
'=================================================================='
' Main program to code CSE stored in a sequential access file     '
' into a random access file.                                       '
'=================================================================='
   CLS
'
' Define variable types
'
   DEFINT A-U
   DEFSNG V-Z
'
' Declare global variables
'
   DIM SHARED e$, g$, bc$, bq$, pr$, cpr$, maxind
   DIM SHARED rl, i, k, l, ct, g, h, ec, gate$, nu$, pri$(80)
   DIM SHARED cau$(4), con$(4), pf(50), pg(50, 2), andflg(50)
   DIM SHARED zp(50), ztemp(50), zcase(50), kpos(50), done(320)
   DIM SHARED s1(50, 80), s2(50, 80), orgate(50), andgate(50)
   DIM SHARED evset(50, 80), leaves(50, 100)
'
' Initialise variables
'
   rl = 0
   h = 1
   ct = 0
   ec = 0
   ERASE pf, pg
'
' Input name of CSE text file to be analysed
'
   flag = 1
   WHILE flag
      LOCATE 4, 1
      INPUT "Input name of cse file to be coded : "; cse$
'
' Give a name to file in which analysed CSE data is to be stored
'
      IF (INSTR(cse$, ".") = 0) AND (LEN(cse$) <= 8) THEN
         cd$ = cse$ + ".COD"
         prims$ = cse$ + ".PRI"
         flag = 0
      ELSEIF (INSTR(cse$, ".") > 0) AND (INSTR(cse$, ".") <= 9) THEN
         cd$ = LEFT$(cse$, INSTR(cse$, ".")) + "COD"
         prims$ = LEFT$(cse$, INSTR(cse$, ".")) + "PRI"
         flag = 0
         ELSE
         LOCATE 2, 1
         PRINT "Incorrect file name specification. Please input again."
      END IF
   WEND 'flag
'
' Open file to store analysed CSE
'
   OPEN cd$ FOR RANDOM AS #2 LEN = 51
   CLOSE #2
```

```
    KILL cd$
    OPEN cd$ FOR RANDOM AS #2 LEN = 51
    FIELD #2, 20 AS e$, 3 AS g$, 2 AS bc$
    ka = 25
    FOR kb = 1 TO 4
        FIELD #2, ka AS dy$, 2 AS cau$(kb)
        ka = ka + 2
    NEXT kb
    FIELD #2, ka AS dy$, 2 AS bq$
    ka = ka + 2
    FOR kb = 1 TO 4
        FIELD #2, ka AS dy$, 2 AS con$(kb)
    ka = ka + 2
    NEXT kb
    FIELD #2, ka AS dy$, 4 AS pr$, 4 AS cpr$
'
' Open file containing CSE
'
    OPEN "I", #1, cse$
    PRINT
    PRINT "Please wait while coding is being done."
    FOR ka = 1 TO 78
        PRINT "-";
    NEXT ka
'
' Analyse CSE
'
    ec = 0
    WHILE NOT EOF(1)
        LINE INPUT #1, eq$
        ct = ct + 1
        l = 1
        g = 1
        pf(g) = 1
        IF (INSTR(eq$, "=") > 0) THEN
            i = INSTR(eq$, "=")
            k = i
            CALL CAUSE(eq$)
        ELSEIF (INSTR(eq$, "-") > 0) THEN
            i = INSTR(eq$, "-")
            k = i
            CALL SYMPTOM(eq$)
        END IF
    WEND 'EOF(1)
    PRINT
    PRINT "Finished coding CSE file."
    IF ec = 0 THEN
        PRINT
        PRINT "No errors."
    ELSE
        PRINT
        PRINT "There are"; ec; " errors."
    END IF
    PRINT
    PRINT "Name of coded CSE file is "; cd$
    FOR ka = 1 TO 78
        PRINT "-";
    NEXT ka
    IF ec = 0 THEN
        PRINT
        PRINT "Now getting all the primary events connecting to top and"
        PRINT "secondary events and storing into file "; prims$; " ."
        CALL LEAF(prims$)
        PRINT
        FOR ka = 1 TO 78
            PRINT "-";
        NEXT ka
        PRINT
        PRINT "Now assigning probabilities values to primary events and"
        PRINT "storing them in "; cd$; "."
        PRINT
        CALL SETPROB
        PRINT
        PRINT "Now evaluating probabilities of top and secondary events."
        start$ = TIME$
        CALL EVALPROB
        fin$ = TIME$
    ELSE
        PRINT
        PRINT "Probability evaluation not done."
    END IF
    PRINT
    PRINT "Finished !"
    PRINT "Probability evaluation started at  "; start$
    PRINT "Probability evaluation finished at "; fin$
    PRINT "Maximum indx used = "; maxind
    CLOSE
    END
'
'=================================================================='
' Subroutine to process data file due to AND gate found after ).     '
'=================================================================='
 SUB ANDGATE1 (TP, ca) STATIC
'
    GET #2, TP
    IF (CVI(bc$) > 4) THEN
        itp = CVI(cau$(4))
        fg3 = 1
        WHILE fg3
            GET #2, itp
            IF NOT (LEFT$(e$, 1) = "&") THEN
                ca = itp
                fg3 = 0
            ELSE
                IF CVI(bc$) = 4 THEN
                    itp = CVI(cau$(4))
                ELSE
                    ca = CVI(cau$(CVI(bc$)))
                    fg3 = 0
                END IF
            END IF
        WEND 'fg3
    ELSE
        ca = CVI(cau$(CVI(bc$)))
    END IF
    LSET bc$ = MKI$(CVI(bc$) - 1)
    PUT #2, TP
    GET #2, ca
    LSET bq$ = MKI$(CVI(bq$) - 1)
    PUT #2, ca
    pg(g, 2) = TP
    pf(g) = 2
    jca = ca
    en$ = "#" + STR$(h)
    h = h + 1
    CALL DUMMY2(en$)
```

```
   ca = rl
   CALL STORE(TP, ca)
   TP = ca
   ca = jca
   CALL STORE(TP, ca)
   GET #2, TP
   LSET g$ = gate$
   PUT #2, TP
   END SUB
'
'=================================================================='
' Subroutine to create dummy event when AND gate encountered.      '
'=================================================================='
   SUB ANDGATE2 (TP, ca) STATIC
'
   pg(g, 2) = TP
   pf(g) = 2
   jca = ca
   en$ = "#" + STR$(h)
   h = h + 1
   CALL DUMMY2(en$)
   ca = rl
   a1 = rl
   CALL STORE(TP, ca)
   TP = a1
   ca = jca
   GET #2, TP
   LSET g$ = "AND"
   PUT #2, TP
   END SUB
'
'=================================================================='
' Subroutine to assign an address to an event name.                '
'=================================================================='
   SUB ASSIGN (eq$, rt, ca) STATIC
'
   en$ = MID$(eq$, l, k - 1)
   evnam$ = SPACE$(20)
   LSET evnam$ = en$
   rt = 0
   IF NOT (rl = 0) THEN
      FOR ka = rl TO 1 STEP -1
         GET #2, ka
         IF e$ = evnam$ THEN
            IF (g$ = "PRI") THEN
               rt = 1
            ELSEIF (g$ = "ORC") OR (g$ = "AND") OR (g$ = "DIR") THEN
               rt = 2
            ELSEIF (g$ = "ORS") THEN
               rt = 3
            END IF
            ca = ka
            ka = 1
         END IF
      NEXT ka
   END IF
   IF (rt = 0) THEN
      rl = rl + 1
      ca = rl
      GET #2, rl
      LSET e$ = en$
      LSET g$ = "PRI"
      PUT #2, rl
   END IF
   END SUB
'
'==================================================================='
' Subroutine to get basic addresses of branches of an event.        '
'==================================================================='
   SUB BRANCH (indx, gadd) STATIC
'
   gtemp = gadd
   s1(indx, 1) = 0
   true = 1
   WHILE true
       GET #2, gtemp
       IF CVI(bc$) >= 4 THEN
           ka = 4
       ELSE
           ka = CVI(bc$)
       END IF
       FOR kb = 1 TO ka
           s1(indx, 1) = s1(indx, 1) + 1
           s1(indx, s1(indx, 1) + 1) = CVI(cau$(kb))
       NEXT kb
       IF CVI(bc$) < 4 THEN
           true = 0
       ELSE
           gtemp = CVI(cau$(4))
           GET #2, gtemp
           IF LEFT$(e$, 1) = "&" THEN
               s1(indx, 1) = s1(indx, 1) - 1
           ELSE
               true = 0
           END IF
       END IF
   WEND 'true
   END SUB
'
'==================================================================='
' Subroutine to process cause equation.                             '
'==================================================================='
   SUB CAUSE (eq$) STATIC
'
   evfg = 0
   CALL ASSIGN(eq$, rt, ca)
   IF (rt >= 2) THEN
       CALL ERRMSGS(eq$)
   ELSE
       TP = ca
       CALL DUMMY1(eq$, TP, ca)
       i = i + 1
       k = 1
       l = i
       p1 = 1
       WHILE p1
           IF (MID$(eq$, i, 1) = "(") THEN
               evflg = 1
               i = i + 1
               k = k + 1
           ELSEIF (MID$(eq$, i, 1) = ")") AND (evflg = 1) THEN
               evflg = 0
               i = i + 1
               k = k + 1
           ELSEIF (MID$(eq$, i, 1) = ")") AND (evflg = 0) THEN
               CALL ASSIGN(eq$, rt, ca)
```

```
            CALL STORE(TP, ca)
            CALL PREVIOUS(eq$, TP, ca, p1)
            IF p1 = 1 THEN
                CALL DUMMY1(eq$, TP, ca)
            END IF
            i = i + 1
            l = i
            k = 1
        ELSEIF (MID$(eq$, l, 1) = "+") THEN
            gate$ = "ORC"
            CALL ASSIGN(eq$, rt, ca)
            CALL STORE(TP, ca)
            CALL GATETYP1(TP, ca)
            CALL DUMMY1(eq$, TP, ca)
            i = i + 1
            l = i
            k = 1
        ELSEIF (MID$(eq$, i, 1) = "*") THEN
            gate$ = "AND"
            CALL ASSIGN(eq$, rt, ca)
            CALL GATETYP1(TP, ca)
            CALL STORE(TP, ca)
            CALL DUMMY1(eq$, TP, ca)
            i = i + 1
            l = i
            k = 1
        ELSEIF (i > LEN(eq$)) THEN
            gate$ = "DIR"
            CALL ASSIGN(eq$, rt, ca)
            CALL GATETYP1(TP, ca)
            CALL STORE(TP, ca)
            p1 = 0
        ELSE
            i = i + 1
            k = k + 1
        END IF
    WEND 'p1
END IF
END SUB
'
'==============================================================='
' Subroutine to check if dummy events are to be created.        '
'==============================================================='
SUB DUMMY1 (eq$, TP, ca) STATIC
'
WHILE MID$(eq$, i + 1, 1) = "("
    g = g + 1
    pf(g) = 1
    pg(g, 1) = TP
    i = i + 1
    en$ = "*" + STR$(h)
    h = h + 1
    CALL DUMMY2(en$)
    ca = rl
    a1 = rl
    CALL STORE(TP, ca)
    TP = a1
WEND
END SUB
'
```

```
'=================================================================='
' Subroutine to assign an address to a dummy event.                '
'=================================================================='
   SUB DUMMY2 (en$) STATIC
'
   rl = rl + 1
   GET #2, rl
   LSET e$ = en$
   IF LEFT$(en$, 1) = "&" THEN
      LSET g$ = "CON"
   ELSEIF LEFT$(en$, 1) = "#" THEN
      LSET g$ = "PRI"
   END IF
   PUT #2, rl
   END SUB
'
'=================================================================='
' Subroutine to output error messages.                             '
'=================================================================='
   SUB ERRMSGS (eq$) STATIC
'
   ec = ec + 1
   PRINT
   PRINT "ERROR at line number "; ct; " of CSE file."
   PRINT CHR$(26); ct; ": "; eq$
   PRINT
   PRINT "Attempt to redefine cause equation already processed."
   PRINT
   PRINT "Continuing processing the rest of CSE file EXCEPT eqn. no."; ct
   FOR ka = 1 TO 78
      PRINT "=";
   NEXT ka
   END SUB
'
'=================================================================='
' Subroutine to evaluate probabilities of fault trees.             '
'=================================================================='
   SUB EVALPROB STATIC
'
' Start calculating probability of top/secondary events
'
   q = 1
   WHILE q <= rl
      GET #2, q
      IF NOT ((g$ = "PRI") OR (done(q) = 2) OR_
            (LEFT$(e$, 1) = "@") OR (LEFT$(e$, 1) = "&")) THEN
         ERASE s1, s2, evset, ztemp
         indx = 1
         s2(indx, 1) = 1
         s2(indx, 2) = q
         CALL PROB(indx, zp(indx))
         GET #2, q
         LSET pr$ = MKS$(zp(indx))
         PUT #2, q
         done(q) = 2
      END IF
      q = q + 1
   WEND 'q
   END SUB
'
```

```
'===============================================================
' Subroutine to expand address space for storing causes.       '
'===============================================================
  SUB EXPCAUS (itp, ca) STATIC
'
  GET #2, itp
  itm = CVI(cau$(4))
  GET #2, itm
  IF LEFT$(e$, 1) = "&" THEN
      itp = itm
      IF CVI(bc$) = 4 THEN
          CALL EXPCAUS(itp, ca)
      ELSE
          branch = CVI(bc$) + 1
          LSET bc$ = MKI$(branch)
          LSET cau$(branch) = MKI$(ca)
          PUT #2, itm
      END IF
  ELSE
      en$ = "&" + STR$(h)
      h = h + 1
      CALL DUMMY2(en$)
      GET #2, itp
      LSET cau$(4) = MKI$(rl)
      PUT #2, itp
      GET #2, rl
      LSET bc$ = MKI$(2)
      LSET cau$(1) = MKI$(itm)
      LSET cau$(2) = MKI$(ca)
      PUT #2, rl
  END IF
  END SUB
'
'===============================================================
' Subroutine to expand address space for storing consequence.  '
'===============================================================
  SUB EXPCONS (ica, TP) STATIC
'
  GET #2, ica
  itm = CVI(con$(4))
  GET #2, itm
  IF LEFT$(e$, 1) = "@" THEN
      ica = itm
      IF CVI(bq$) = 4 THEN
          CALL EXPCONS(ica, TP)
      ELSE
          conseq = CVI(bq$) + 1
          LSET bq$ = MKI$(conseq)
          LSET con$(conseq) = MKI$(TP)
          PUT #2, itm
      END IF
  ELSE
      en$ = "@" + STR$(h)
      h = h + 1
      CALL DUMMY2(en$)
      GET #2, ica
      LSET con$(4) = MKI$(rl)
      PUT #2, ica
      GET #2, rl
      LSET bq$ = MKI$(2)
      LSET con$(1) = MKI$(itm)
      LSET con$(2) = MKI$(TP)
      PUT #2, rl
```

```
   END IF
   END SUB
'
'==============================================================='
' Subroutine to check type of gate for cause equation.           '
'==============================================================='
   SUB GATETYP1 (TP, ca) STATIC
'
   GET #2, TP
   IF g$ = "PRI" THEN
      LSET g$ = gate$
      PUT #2, TP
   ELSEIF gate$ = "DIR" THEN
      IF g$ = "ORC" OR g$ = "AND" THEN
         gate$ = g$
      END IF
   ELSEIF NOT (g$ = gate$) THEN
      IF g$ = "AND" THEN
         CALL ORGATE(TP, ca)
      ELSEIF g$ = "ORC" THEN
         CALL ANDGATE2(TP, ca)
      END IF
   END IF
   END SUB
'
'==============================================================='
' Subroutine to set type of gate for symptom equations.          '
'==============================================================='
   SUB GATETYP2 (TP) STATIC
'
   GET #2, TP
   IF g$ = "PRI" THEN
      LSET g$ = "DIR"
   ELSE
      LSET g$ = "ORS"
   END IF
   PUT #2, TP
   END SUB
'
'==============================================================='
' Subroutine to find an independent event is s2().               '
'==============================================================='
   SUB INDEPT (indx, disjt, indevt) STATIC
'
   FOR ka = 2 TO s2(indx, 1) + 1
      GET #3, s2(indx, ka)
      kz = CVI(nu$)
      disjt = 1
      FOR kb = 2 TO s2(indx, 1) + 1
         IF NOT (kb = ka) THEN
            GET #3, s2(indx, kb)
            ky = CVI(nu$)
            FOR kc = 1 TO kz
               GET #3, s2(indx, ka)
               kx = CVI(pri$(kc))
               GET #3, s2(indx, kb)
               FOR kd = 1 TO ky
                  IF kx = CVI(pri$(kd)) THEN
                     disjt = 0
                     kd = ky
                     kc = kz
                  END IF
               NEXT kd
            NEXT kc
            IF disjt = 0 THEN kb = s2(indx, 1) + 1
         END IF
      NEXT kb
      IF disjt = 1 THEN
         indevt = s2(indx, ka)
         ka = s2(indx, 1) + 1
      END IF
   NEXT ka
   END SUB
'
'==================================================================='
'  Subroutine to find an independent OR gate in s().                '
'==================================================================='
   SUB INDEPTOR (indx, disjt, indevt) STATIC
'
   FOR ka = 2 TO s1(indx, 1) + 1
      GET #3, s1(indx, ka)
      kz = CVI(nu$)
      disjt = 1
      FOR kb = 2 TO s1(indx, 1) + 1
         IF NOT (kb = ka) THEN
            GET #3, s1(indx, kb)
            ky = CVI(nu$)
            FOR kc = 1 TO kz
               GET #3, s1(indx, ka)
               kx = CVI(pri$(kc))
               GET #3, s1(indx, kb)
               FOR kd = 1 TO ky
                  IF kx = CVI(pri$(kd)) THEN
                     disjt = 0
                     kd = ky
                     kc = kz
                  END IF
               NEXT kd
            NEXT kc
         END IF
         IF disjt = 0 THEN kb = s1(indx, 1) + 1
      NEXT kb
      IF disjt = 1 THEN
         indevt = s1(indx, ka)
         FOR kb = 2 TO s2(indx, 1) + 1
            GET #3, s2(indx, kb)
            ky = CVI(nu$)
            FOR kc = 1 TO ky
               GET #3, s2(indx, kb)
               kx = CVI(pri$(kc))
               GET #3, indevt
               FOR kd = 1 TO kz
                  IF kx = CVI(pri$(kd)) THEN
                     disjt = 0
                     kd = kz
                     kc = ky
                  END IF
               NEXT kd
            NEXT kc
            IF disjt = 0 THEN kb = s2(indx, 1) + 1
         NEXT kb
      END IF
      IF disjt = 1 THEN ka = s1(indx, 1) + 1
   NEXT ka
   END SUB
'
```

```
'=================================================================='
' Subroutine to determine if s1() intersection s2() is non empty.  '
'=================================================================='
    SUB INTERSEC (indx, true) STATIC
'
    true = 0
    FOR ka = 2 TO s2(indx, 1) + 1
        FOR kb = 2 TO s1(indx, 1) + 1
            IF s2(indx, ka) = s1(indx, kb) THEN
                true = 1
                kb = s1(indx, 1) + 1
                ka = s2(indx, 1) + 1
            END IF
        NEXT kb
    NEXT ka
    END SUB
'
'=================================================================='
' Subroutine to set a file named prim$ that contain for each       '
' event a set primary events that causes that event.               '
'=================================================================='
    SUB LEAF (prim$) STATIC
'
' Open file prim$ to store array leaves.
'
    OPEN prim$ FOR RANDOM AS #3 LEN = 162
    FIELD #3, 2 AS nu$
    ka = 2
    FOR kb = 1 TO 80
        FIELD #3, ka AS dy$, 2 AS pri$(kb)
        ka = ka + 2
    NEXT kb
'
' Proceed to obtain set of primary events of an event.
'
    ERASE done
    q = 1
    WHILE q <= rl
        IF NOT (done(q) = 1) THEN
            ERASE s1, s2, pf, leaves
            indx = 1
            pf(indx) = q
            CALL PRIMARY(indx)
        END IF
        q = q + 1
    WEND 'q
'
    END SUB
'
'=================================================================='
' Subroutine to store contents of temp. buffer to new address.     '
'=================================================================='
    SUB NEWADDR (TP, ca, bux) STATIC
'
    IF bux > 4 THEN
        ka = 4
    ELSE
        ka = bux
    END IF
    GET #2, TP
    FOR kb = 1 TO ka
        LSET con$(kb) = MKI$(kpos(kb))
    NEXT kb
    LSET bq$ = MKI$(bux)
    PUT #2, TP
    fg5 = 1
    WHILE fg5
        FOR kb = 1 TO ka
            fg6 = 1
            WHILE fg6
                GET #2, kpos(kb)
                kc = CVI(bc$)
                rpc = 0
                IF kc >= 4 THEN
                    kd = 4
                ELSE
                    kd = kc
                END IF
                FOR ke = 1 TO kd
                    IF CVI(cau$(ke)) = ca THEN
                        LSET cau$(ke) = MKI$(TP)
                        PUT #2, kpos(kb)
                        rpc = 1
                        ke = kd
                    END IF
                NEXT ke
                IF rpc = 1 THEN
                    fg6 = 0
                ELSEIF kc >= 4 THEN
                    kpos(kb) = CVI(cau$(4))
                END IF
            WEND 'fg6
        NEXT kb
        IF NOT (ka < 4) THEN
            GET #2, kpos(4)
            IF LEFT$(e$, 1) = "@" THEN
                kf = CVI(bq$)
                FOR kb = 1 TO kf
                    kpos(kb) = CVI(con$(kb))
                NEXT kb
            ELSE
                fg5 = 0
            END IF
        ELSE
            fg5 = 0
        END IF
    WEND 'fg5
    END SUB
'
'=================================================================='
' Subroutine to process data file when OR gate encountered.       '
'=================================================================='
    SUB ORGATE (TP, ca) STATIC
'
    IF pf(g) = 2 THEN
        TP = pg(g, 2)
        pf(g) = 1
    ELSE
        en$ = "#" + STR$(h)
        h = h + 1
        trans = 0
        GET #2, TP
        ntm$ = e$
        LSET e$ = en$
        PUT #2, TP
        GET #2, TP
        IF CVI(bq$) > 0 THEN
            trans = 1
            bux = CVI(bq$)
            CALL TEMPSTORE(TP, bux)
        END IF
        en$ = ntm$
        CALL DUMMY2(en$)
        ca = TP
        TP = rl
        CALL STORE(TP, ca)
        GET #2, TP
        LSET g$ = gate$
        PUT #2, TP
        IF trans = 1 THEN
            CALL NEWADDR(TP, ca, bux)
        END IF
        IF g > 1 THEN CALL REPLACE(TP)
    END IF
    END SUB
'
'=================================================================='
' Subroutine to return to previous top event when ")" is found.    '
'=================================================================='
    SUB PREVIOUS (eq$, TP, ca, p1) STATIC
'
    WHILE MID$(eq$, i, 1) = ")"
        TP = pg(g, 1)
        g = g - 1
        i = i + 1
    WEND
    IF MID$(eq$, i, 1) = "*" THEN
        gate$ = "AND"
    ELSEIF MID$(eq$, i, 1) = "+" THEN
        gate$ = "ORC"
    ELSE
        p1 = 0
    END IF
    IF NOT (p1 = 0) THEN
        GET #2, TP
        IF NOT (g$ = gate$) THEN
            IF g$ = "ORC" THEN
                CALL ANDGATE1(TP, ca)
            ELSE
                CALL GATETYP1(TP, ca)
            END IF
        END IF
    END IF
    END SUB
'
'=================================================================='
' Subroutine to set up array of basic events connected to an event. '
'=================================================================='
    SUB PRIMARY (indx) STATIC
'
    GET #2, pf(indx)
    IF NOT(LEFT$(e$, 1) = "&" OR LEFT$(e$, 1) = "@") THEN
        IF g$ = "PRI" THEN
            leaves(indx, 1) = 1
            leaves(indx, 2) = pf(indx)
        ELSEIF g$ = "DIR" THEN
            pf(indx + 1) = CVI(cau$(1))
            IF NOT (done(pf(indx + 1)) = 1) THEN
                indx = indx + 1
            CALL PRIMARY(indx)
            indx = indx - 1
            leaves(indx, 1) = leaves(indx + 1, 1)
            FOR ka = 2 TO leaves(indx, 1) + 1
                leaves(indx, ka) = leaves(indx + 1, ka)
            NEXT ka
        ELSE
            GET #3, pf(indx + 1)
            leaves(indx, 1) = CVI(nu$)
            ka = 2
            FOR kb = 1 TO leaves(indx, 1)
                leaves(indx, ka) = CVI(pri$(kb))
                ka = ka + 1
            NEXT kb
        END IF
    ELSEIF (g$ = "AND") OR (LEFT$(g$, 2) = "OR") THEN
        leaves(indx, 1) = 0
        CALL BRANCH(indx, pf(indx))
        kpos(indx) = 2
        WHILE kpos(indx) <= s1(indx, 1) + 1
            pf(indx + 1) = s1(indx, kpos(indx))
            IF NOT (done(pf(indx + 1)) = 1) THEN
                indx = indx + 1
                CALL PRIMARY(indx)
                indx = indx - 1
                ka = leaves(indx, 1) + 2
                kb = 2
                leaves(indx, 1) = leaves(indx, 1) + leaves(indx + 1, 1)
                FOR kc = ka TO leaves(indx, 1) + 1
                    leaves(indx, kc) = leaves(indx + 1, kb)
                    kb = kb + 1
                NEXT kc
            ELSE
                GET #3, pf(indx + 1)
                ka = leaves(indx, 1) + 2
                kb = 1
                leaves(indx, 1) = leaves(indx, 1) + CVI(nu$)
                FOR kc = ka TO leaves(indx, 1) + 1
                    leaves(indx, kc) = CVI(pri$(kb))
                    kb = kb + 1
                NEXT kc
            END IF
            CALL REMREP2(indx)
            kpos(indx) = kpos(indx) + 1
        WEND 'kpos(indx)
    END IF
'
    GET #3, pf(indx)
    LSET nu$ = MKI$(leaves(indx, 1))
    ka = 1
    FOR kb = 2 TO leaves(indx, 1) + 1
        LSET pri$(ka) = MKI$(leaves(indx, kb))
        ka = ka + 1
    NEXT kb
    PUT #3, pf(indx)
    done(pf(indx)) = 1
'
END IF
END SUB
'
```

```
'==============================================================='
' Subroutines to handle probability calculations.               '
'==============================================================='
   SUB PROB (indx, zpv) STATIC
   IF indx > maxind THEN
       maxind = indx
   END IF
'
' Priliminary reduction when s1() not empty.
'
   IF NOT (s1(indx, 1) = 0) THEN
'
' Case P.1 : Evaluation for s1() intersection s2() non empty.
'            If this is true, then set s1() empty and s2()
'            as before.
'
       CALL INTERSEC(indx, true)
       IF true = 1 THEN
           s1(indx, 1) = 0
'
' Case P.2 : Evaluation for s1() containing a single node.
'            Add the node in s1() to s2() and set s1() empty.
'
       ELSEIF s1(indx, 1) = 1 THEN
           s2(indx, 1) = s2(indx, 1) + 1
           s2(indx, s2(indx, 1) + 1) = s1(indx, 2)
           s1(indx, 1) = 0
       END IF
   END IF
'
' Case 1 : Evaluation for s1() empty.
'
   IF s1(indx, 1) = 0 THEN
'
' Case 1.1 : PROB returns zpv = 1 if s2() is empty.
'
       IF s2(indx, 1) = 0 THEN
           zpv = 1
'
' Case 1.2 : Evaluation for s2() containing one node.
'
       ELSEIF s2(indx, 1) = 1 THEN
'
' Case 1.2.1 : Evaluation for probability of node has been
'              evaluated. Return to caller with its probability.
'
           IF (done(s2(indx, 2)) = 2) THEN
               GET #2, s2(indx, 2)
               zpv = CVS(pr$)
'
' Case 1.2.2 : Evaluation for probability of node has not been
'              evaluated.  Evaluate probability by calling PROB
'              with s1() containing the branches of the node if
'              it is an OR gate or s2() containing the branches
'              of the node if it is an AND gate.
'
           ELSE
               indx = indx + 1
               s1(indx, 1) = 0
               s2(indx, 1) = 0
               GET #2, s2(indx - 1, 2)
               IF (g$ = "DIR") OR (g$ = "AND") THEN
                   CALL SIBLING(indx, 2, s2(indx - 1, 2), s2())
```

```
            ELSE
                CALL SIBLING(indx, 2, s2(indx - 1, 2), s1())
            END IF
            CALL PROB(indx, zp(indx))
            indx = indx - 1
            zpv = zp(indx + 1)
            GET #2, s2(indx, 2)
            LSET pr$ = MKS$(zpv)
            PUT #2, s2(indx, 2)
            done(s2(indx, 2)) = 2
        END IF
'
' Case 1.3 : Evaluation for case of s2() containing more than
'            one node. First set ztemp = 1. Then determine if
'            there is a node in s2() which is disjoint from the
'            other nodes in s2().
'
    ELSE
        ztemp(indx) = 1
        andgate(indx) = 1
        WHILE andgate(indx)
            CALL INDEPT(indx, disjt, indevt)
'
' Case 1.3.1 : Evaluation for case where a member in s2() is found
'              found to be disjoint from the other members in s2().
'
            IF disjt = 1 THEN
                evset(indx, 2) = indevt
'
' Case 1.3.1.1 : Evaluation for case where the probability of the
'                independent event has been evaluated.
'
                IF (done(evset(indx, 2)) = 2) THEN
                    GET #2, evset(indx, 2)
                    ztemp(indx) = ztemp(indx) * CVS(pr$)
'
' Case 1.3.1.2 : Evaluation for case where the probability of the
'                independent event has not been evaluated.
'
                ELSE
                    indx = indx + 1
                    s1(indx, 1) = 0
                    s2(indx, 1) = 0
                    GET #2, evset(indx - 1, 2)
                    IF (g$ = "DIR") OR (g$ = "AND") THEN
                        CALL SIBLING(indx, 2, evset(indx - 1, 2), s2())
                    ELSE
                        CALL SIBLING(indx, 2, evset(indx - 1, 2), s1())
                    END IF
                    CALL PROB(indx, zp(indx))
                    indx = indx - 1
                    GET #2, evset(indx, 2)
                    LSET pr$ = MKS$(zp(indx + 1))
                    PUT #2, evset(indx, 2)
                    done(evset(indx, 2)) = 2
                    ztemp(indx) = ztemp(indx) * zp(indx + 1)
                END IF
'
' Remove evset(indx, 2) from s2(). If s2() is not empty, repeat finding
' an s-independent node in s2().
'
                FOR ka = 2 TO s2(indx, 1) + 1
                    IF s2(indx, ka) = evset(indx, 2) THEN
                    IF NOT (ka = s2(indx, 1) + 1) THEN
                        FOR kb = ka TO s2(indx, 1)
                            s2(indx, kb) = s2(indx, kb + 1)
                        NEXT kb
                        ka = s2(indx, 1) + 2
                    END IF
                    s2(indx, s2(indx, 1) + 1) = 0
                    s2(indx, 1) = s2(indx, 1) - 1
                END IF
            NEXT ka
'
' If s2() is empty, return to caller with zpv = ztemp(indx).
'
            IF s2(indx, 1) = 0 THEN
                andgate(indx) = 0
                zpv = ztemp(indx)
            END IF
'
' Case 1.3.2 : Evaluation for case where there is no node found
'              to be disjoint from the other members in s2().
'
        ELSE
'
' Find all AND gate nodes in s2() and replace the nodes in s2()
' with their branches.
'
            replc = 0
            stot = s2(indx, 1) + 1
            smem = 2
            WHILE smem <= stot
                GET #2, s2(indx, smem)
                IF (g$ = "DIR") OR (g$ = "AND") THEN
                    replc = 1
                    CALL SIBLING(indx, smem, s2(indx, smem), s2())
                END IF
                smem = smem + 1
            WEND 'smem
'
' Case 1.3.2.1 : Evaluation for case when replacements. Repetition
'                of nodes is removed. Then determine if there is a
'                node disjoint from the rest of the nodes in s2().
'
            IF replc = 1 THEN
                CALL REMREP(indx, s2())
'
' Case 1.3.2.2 : Evaluation for case of no AND gate and no disjoint
'                nodes found in s2(). An OR gate node m in s2() is
'                picked and probability is evaluated via call to PROB
'                with s1() containing the branches of m and s2()
'                containing nodes without m.
'
            ELSE
                FOR ka = 2 TO s2(indx, 1) + 1
                    GET #2, s2(indx, ka)
                    IF LEFT$(g$, 2) = "OR" THEN
                        ornode = s2(indx, ka)
                        ka = s2(indx, 1) + 1
                    END IF
                NEXT ka
                indx = indx + 1
                s1(indx, 1) = 0
                CALL SIBLING(indx, 2, ornode, s1())
                ka = 2
```

```
                        FOR kb = 2 TO s2(indx - 1, 1) + 1
                            IF NOT (s2(indx - 1, kb) = ornode) THEN
                                s2(indx, ka) = s2(indx - 1, kb)
                                ka = ka + 1
                            END IF
                        NEXT kb
                        s2(indx, 1) = s2(indx - 1, 1) - 1
                        CALL PROB(indx, zp(indx))
                        indx = indx - 1
                        zpv = ztemp(indx) * zp(indx + 1)
                        andgate(indx) = 0
                    END IF
                END IF
            WEND 'andgate(indx)
        END IF
'
' Case 2 : Evaluation for s1() non empty. First initialise evset() so
'          that it is empty. Then determine if there is a node in s1()
'          which is disjoint from the other nodes of s1() as well as s2().
'
    ELSE
        evset(indx, 1) = 0
        andgate(indx) = 1
        WHILE andgate(indx)
            CALL INDEPTOR(indx, disjt, indevt)
'
' Case 2.1 : Case where a node in s1() is found to be disjoint
'            from other nodes in s1() as well as from all nodes
'            in s2(). Save the disjoint node in evset().
'
            IF disjt = 1 THEN
                evset(indx, 1) = evset(indx, 1) + 1
                evset(indx, evset(indx, 1) + 1) = indevt
'
' Remove the disjoint node from s1.
'
                FOR ka = 2 TO s1(indx, 1) + 1
                    IF s1(indx, ka) = indevt THEN
                        IF NOT (ka = s1(indx, 1) + 1) THEN
                            FOR kb = ka TO s1(indx, 1) + 1
                                s1(indx, kb) = s1(indx, kb + 1)
                            NEXT kb
                            ka = s1(indx, 1) + 1
                        END IF
                        s1(indx, s1(indx, 1) + 1) = 0
                        s1(indx, 1) = s1(indx, 1) - 1
                    END IF
                NEXT ka
                IF s1(indx, 1) = 1 THEN
                    andgate(indx) = 0
                END IF
            ELSE
                replc = 0
                stot = s1(indx, 1) + 1
                smem = 2
                WHILE smem <= stot
                    GET #2, s1(indx, smem)
                    IF (LEFT$(g$, 2) = "OR") OR (g$ = "DIR") THEN
                        replc = 1
                        CALL SIBLING(indx, smem, s1(indx, smem), s1())
                    END IF
                    smem = smem + 1
                WEND 'smem
```

```
'
' Case 2.2.1 : Evaluation for replacements made in s1().  Determine if
'              there is a node in s1() which is disjoint from the other
'              nodes in s1() as well as disjoint with all the nodes in s2().
'
               IF replc = 1 THEN
                   CALL REMREP(indx, s1())
                   IF s1(indx, 1) = 1 THEN
                       andgate(indx) = 0
                   END IF
               ELSE
                   andgate(indx) = 0
               END IF
           END IF
       WEND 'andgate(indx)
'
       IF NOT (evset(indx, 1) = 0) THEN
           indx = indx + 1
           s1(indx, 1) = s1(indx - 1, 1)
           FOR ka = 2 TO s1(indx, 1) + 1
               s1(indx, ka) = s1(indx - 1, ka)
           NEXT ka
           s2(indx, 1) = s2(indx - 1, 1)
           FOR ka = 2 TO s2(indx, 1) + 1
               s2(indx, ka) = s2(indx - 1, ka)
           NEXT ka
           CALL PROB(indx, zp(indx))
           indx = indx - 1
           ztemp(indx) = zp(indx + 1)
'
           indx = indx + 1
           s1(indx, 1) = 0
           s2(indx, 1) = s2(indx - 1, 1)
           FOR ka = 2 TO s2(indx, 1) + 1
               s2(indx, ka) = s2(indx - 1, ka)
           NEXT ka
           CALL PROB(indx, zp(indx))
           indx = indx - 1
           zcase(indx) = zp(indx + 1)
'
           andflg(indx) = 2
           WHILE andflg(indx) <= evset(indx, 1) + 1
               IF done(evset(indx, andflg(indx))) = 2 THEN
                   GET #2, evset(indx, andflg(indx))
                   zpr = CVS(pr$)
               ELSE
                   indx = indx + 1
                   s1(indx, 1) = 0
                   s2(indx, 1) = 0
                   GET #2, evset(indx - 1, andflg(indx - 1))
                   IF (g$ = "DIR") OR (g$ = "AND") THEN
                       CALL SIBLING(indx, 2, evset(indx-1,andflg(indx-1)),s2())
                   ELSE
                       CALL SIBLING(indx,2,evset(indx-1,andflg(indx-1)),s1())
                   END IF
                   CALL PROB(indx, zp(indx))
                   indx = indx - 1
                   GET #2, evset(indx, andflg(indx))
                   LSET pr$ = MKS$(zp(indx + 1))
                   PUT #2, evset(indx, andflg(indx))
                   done(evset(indx, andflg(indx))) = 2
                   zpr = zp(indx + 1)
               END IF
```

```
              ztemp(indx) = zpr * (zcase(indx) - ztemp(indx)) + ztemp(indx)
              andflg(indx) = andflg(indx) + 1
          WEND 'andflg(indx)
          zpv = ztemp(indx)
'
' Case 2.2 : Evaluation for case of no nodes in s1() disjoint in s1()
'            as well as in s2().  Find all OR and DIR nodes in s1() and
'            replace the nodes in s1 with their branches.
'
      ELSE
'
' Case 2.2.2 : Evaluation for case of no replacements made in s1().
'              A node is picked from s1() and saved in orgate(indx).
'
          orgate(indx) = s1(indx, 2)
'
          indx = indx + 1
          s1(indx, 1) = 0
          FOR ka = 2 TO s2(indx - 1, 1) + 1
              s2(indx, ka) = s2(indx - 1, ka)
          NEXT ka
          s2(indx, 1) = s2(indx - 1, 1) + 1
          s2(indx, s2(indx, 1) + 1) = orgate(indx - 1)
          CALL PROB(indx, zp(indx))
          indx = indx - 1
          ztemp(indx) = zp(indx + 1)
'
          indx = indx + 1
          ka = 2
          FOR kb = 2 TO s1(indx - 1, 1) + 1
              IF NOT (s1(indx - 1, kb) = orgate(indx - 1)) THEN
                  s1(indx, ka) = s1(indx - 1, kb)
                  ka = ka + 1
              END IF
          NEXT kb
          s1(indx, 1) = s1(indx - 1, 1) - 1
          FOR ka = 2 TO s2(indx - 1, 1) + 1
              s2(indx, ka) = s2(indx - 1, ka)
          NEXT ka
          s2(indx, 1) = s2(indx - 1, 1)
          CALL PROB(indx, zp(indx))
          indx = indx - 1
          ztemp(indx) = ztemp(indx) + zp(indx + 1)
'
          indx = indx + 1
          ka = 2
          FOR kb = 2 TO s1(indx - 1, 1) + 1
              IF NOT (s1(indx - 1, kb) = orgate(indx - 1)) THEN
                  s1(indx, ka) = s1(indx - 1, kb)
                  ka = ka + 1
              END IF
          NEXT kb
          s1(indx, 1) = s1(indx - 1, 1) - 1
          FOR ka = 2 TO s2(indx - 1, 1) + 1
              s2(indx, ka) = s2(indx - 1, ka)
          NEXT ka
          s2(indx, 1) = s2(indx - 1, 1) + 1
          s2(indx, s2(indx, 1) + 1) = orgate(indx - 1)
          CALL PROB(indx, zp(indx))
          indx = indx - 1
          zpv = ztemp(indx) - zp(indx + 1)
'
      END IF
```

```
   END IF
   END SUB
'
'================================================================'
' Subroutine to remove events repeated in set s().              '
'================================================================'
   SUB REMREP (indx, set(2)) STATIC
'
   FOR ka = 2 TO set(indx, 1)
       FOR kb = ka + 1 TO set(indx, 1) + 1
           IF set(indx, ka) = set(indx, kb) THEN
               FOR kc = kb TO set(indx, 1)
                   set(indx, kc) = set(indx, kc + 1)
               NEXT kc
               set(indx, 1) = set(indx, 1) - 1
           END IF
       NEXT kb
   NEXT ka
   END SUB
'
'================================================================'
' Subroutine to remove repetitions in set LEAVES(indx).         '
'================================================================'
   SUB REMREP2 (indx) STATIC
'
   FOR ka = 2 TO leaves(indx, 1)
       FOR kb = ka + 1 TO leaves(indx, 1) + 1
           IF leaves(indx, ka) = leaves(indx, kb) THEN
               FOR kc = kb TO leaves(indx, 1)
                   leaves(indx, kc) = leaves(indx, kc + 1)
               NEXT kc
               leaves(indx, 1) = leaves(indx, 1) - 1
           END IF
       NEXT kb
   NEXT ka
   END SUB
'
'================================================================'
' Subroutine to replace causal address in previous top event.   '
'================================================================'
   SUB REPLACE (TP) STATIC
'
   GET #1, pg(g, 1)
   IF CVI(bc$) > 4 THEN
       itp = CVI(cau$(4))
       fg4 = 1
       WHILE fg4
           GET #2, itp
           IF CVI(bc$) = 4 THEN
               itm = CVI(cau$(4))
               GET #2, itm
               IF LEFT$(e$, 1) = "&" THEN
                   itp = itm
               ELSE
                   GET #2, itp
                   LSET cau$(4) = MKI$(TP)
                   PUT #2, itp
                   fg4 = 0
               END IF
           ELSE
               LSET cau$(CVS(bc$)) = MKI$(TP)
               PUT #2, itp
               fg4 = 0
```

```
            END IF
        WEND 'fg4
    ELSE
        LSET cau$(CVI(bc$)) = MKI$(TP)
    END IF
    END SUB
'
'==================================================================='
' Subroutine to store coded CSE file into named random access file.  '
'==================================================================='
    SUB SETPROB STATIC
    FOR ka = 1 TO rl
        GET #2, ka
        IF g$ = "PRI" THEN
            PRINT "Input a priori probability for event "; e$; " :";
            INPUT zpr
            LSET pr$ = MKS$(zpr)
            PUT #2, ka
            done(ka) = 2
        END IF
    NEXT ka
    END SUB
'
'==================================================================='
' Subroutine to replace a member of s2() which is either a DIR       '
' or an AND gate with its child or children respectively.           '
'==================================================================='
    SUB SIBLING (ndx, setpos, evadd, set(2)) STATIC
'
    dtemp = evadd
    GET #2, dtemp
    set(ndx, setpos) = CVI(cau$(1))
    IF set(ndx, 1) = 0 THEN set(ndx, 1) = 1
    true = 1
    WHILE true
        GET #2, dtemp
        IF CVI(bc$) >= 4 THEN
            ka = 4
        ELSE
            ka = CVI(bc$)
        END IF
        FOR kb = 1 TO ka
            IF NOT (CVI(cau$(kb)) = set(ndx, setpos)) THEN
                set(ndx, 1) = set(ndx, 1) + 1
                set(ndx, set(ndx, 1) + 1) = CVI(cau$(kb))
            END IF
        NEXT kb
        IF CVI(bc$) < 4 THEN
            true = 0
        ELSE
            dtemp = CVI(cau$(4))
            GET #2, dtemp
            IF LEFT$(e$, 1) = "&" THEN
                set(ndx, 1) = set(ndx, 1) - 1
            ELSE
                true = 0
            END IF
        END IF
    WEND 'true
    END SUB
'
```

```
'================================================================'
' Subroutine to store cause events.                               '
'================================================================'
   SUB STORE (TP, ca) STATIC
'
   GET #2, TP
   branch = CVI(bc$) + 1
   LSET bc$ = MKI$(branch)
   PUT #2, TP
   IF branch > 4 THEN
       itp = TP
       CALL EXPCAUS(itp, ca)
   ELSE
       GET #2, TP
       LSET cau$(branch) = MKI$(ca)
       PUT #2, TP
   END IF
   GET #2, ca
   conseq = CVI(bq$) + 1
   LSET bq$ = MKI$(conseq)
   PUT #2, ca
   IF conseq > 4 THEN
       ica = ca
       CALL EXPCONS(ica, TP)
   ELSE
       GET #2, ca
       LSET con$(conseq) = MKI$(TP)
       PUT #2, ca
   END IF
   END SUB
'
'================================================================'
' Subroutine to proces symptom equation.                          '
'================================================================'
   SUB SYMPTOM (eq$) STATIC
'
   CALL ASSIGN(eq$, rt, ca)
   sca = ca
   p1 = 1
   i = i + 1
   k = 1
   l = i
   WHILE p1
       IF (MID$(eq$, i, 1) = "*") OR (i > LEN(eq$)) THEN
           CALL ASSIGN(eq$, rt, ca)
           TP = ca
           ca = sca
           CALL GATETYP2(TP)
           CALL STORE(TP, ca)
           i = i + 1
           l = i
           k = 1
           IF i > LEN(eq$) THEN
               p1 = 0
           END IF
       ELSE
           i = i + 1
           k = k + 1
       END IF
   WEND 'p1
   END SUB
'
```

```
'==============================================================='
' Subroutine to store consequences to temporary buffer.         '
'==============================================================='
   SUB TEMPSTORE (TP, bux) STATIC
'
   IF bux > 4 THEN
       ka = 4
   ELSE
       ka = bux
   END IF
   GET #2, TP
   FOR kb = 1 TO ka
       kpos(kb) = CVI(con$(kb))
       LSET con$(kb) = MKI$(0)
   NEXT kb
   LSET bq$ = MKI$(0)
   PUT #2, TP
   END SUB
'
```

APPENDIX F

SOURCE CODE FOR THE PROGRAM DISFAULT

```
'==============================================================='
' Main program to output fault tree.                            '
'==============================================================='
'
' Define variable types
'
   DEFINT a - y
   DEFSNG z
'
' Declare shared variables between main program and subprogram
'
   DIM SHARED cau$(4), con$(4), p, nm$
   DIM SHARED e$, g$, bc$, bq$, pr$, cpr$
   DIM SHARED ca$(30), bc(30), fc(30), zc(30), zcp(30)
   DIM SHARED qa$(30), bq(30), fq(30), zq(30), zqp(30)
   DIM SHARED alc$(30), bac(30), fac(30), zac(30), zapc(30)
   DIM SHARED alq$(30), baq(30), faq(30), zaq(30), zapq(30)
   DIM SHARED acbr, aqbr, fxacbr, fxaqbr, av, aw, alb, alin, falin
   DIM SHARED cbr, cv, cw, clb, clin, fclin
   DIM SHARED qbr, qv, qw, qlb, qlin, fqlin
   DIM SHARED apos, atr, hnum, vnum, btemp, ftemp
   DIM SHARED flt$, alft$, aev$, zalrm, zalcn, sym$
   DIM SHARED meant, apri, post, top, zd, ze
   DIM SHARED nu$, pri$(80)
   DIM SHARED zp(50), ztemp(50), zcase(50), kpos(50)
   DIM SHARED s1(50,50), s2(50,50), orgate(50)
   DIM SHARED done1(800), done2(20), done(320), andflg(50), andgate(50)
   DIM SHARED comset(3, 50), evset(50, 50)
   DIM SHARED doneflg, etop
   DIM SHARED fxtop, fxcbr, fxqbr, zffx, zcfx
'
' Set key traps
'
   KEY 15, CHR$(&H08)+CHR$(&H1E)
   KEY 16, CHR$(&H08)+CHR$(&H12)
   KEY 17, CHR$(&H08)+CHR$(&H23)
   ON KEY(1) GOSUB TOGGLE1
   ON KEY(2) GOSUB TOGGLE2
   ON KEY(3) GOSUB CAUSEUP
   ON KEY(4) GOSUB CAUSEDN
   ON KEY(5) GOSUB FAULTUP
   ON KEY(6) GOSUB FAULTDN
   ON KEY(7) GOSUB CONSEQUP
   ON KEY(8) GOSUB CONSEQDN
   ON KEY(9) GOSUB OTHERS
   ON KEY(10) GOSUB ORIGINAL
   ON KEY(11) GOSUB CURUP
   ON KEY(12) GOSUB CURLEFT
   ON KEY(13) GOSUB CURRIGHT
   ON KEY(14) GOSUB CURDOWN
   ON KEY(15) GOSUB ANOTHER
   ON KEY(16) GOSUB EXPCODE
   ON KEY(17) GOSUB HELP
'
   SCREEN 0, 1, 0
   CLS
   COLOR 2, 0, 1
   flt$ = SPACE$(20)
   alft$ = SPACE$(20)
   aev$ = SPACE$(20)
   nm$ = SPACE$(3)
'
' Input name of file containing fault tree data
'
   LOCATE 2,1
   INPUT "Input name of data file containing coded CSE : ";cs$
'
' Open file as random access and declare variables in the
' various fields
'
   OPEN cs$ AS #1 LEN = 51
   FIELD #1, 20 AS e$, 3 AS g$, 2 AS bc$
   ka = 25
   FOR kb = 1 TO 4
       FIELD #1, ka AS my$, 2 AS cau$(kb)
       ka = ka + 2
   NEXT kb
   FIELD #1, ka AS my$, 2 AS bq$
   ka = ka + 2
   FOR kb = 1 TO 4
       FIELD #1, ka AS my$, 2 AS con$(kb)
       ka = ka + 2
   NEXT kb
   FIELD #1, ka AS my$, 4 AS pr$, 4 AS cpr$
'
   prim$ = LEFT$(cs$, INSTR(cs$, ".")) + "PRI"
   OPEN prim$ AS #2 LEN = 162
   FIELD #2, 2 AS nu$
   ka = 2
   FOR kb = 1 TO 80
       FIELD #2, ka AS dy$, 2 as pri$(kb)
       ka = ka + 2
   NEXT kb
'
' Determine no. of records in data file
'
   p = INT(LOF(1)/51)
   true = 1
   WHILE true
       GET #1, p
       IF ASC(e$) = 0 THEN
           p = p - 1
       ELSE
           true = 0
       END IF
   WEND 'true
'
' Input name of initial alarmed event
'
   true = 1
   WHILE true
       LOCATE 3,1
       INPUT "Input name of alarmed event : ";en$
       LSET flt$ = en$
       LSET alft$ = en$
       LSET aev$ = en$
'
' Find event name in data file and the first level causes
' and consequences
```

```
'
      CALL FIND.EV (fd, top, tcbr, tqbr, zd, ze)
'
' Test if event name found in data file. If not found,
' re-input event name.
'
      IF fd = 0 THEN
         PRINT "Event not found. Please input again!"
      ELSE
         fxcbr = tcbr
         fxqbr = tqbr
         fxtop = top
         zffx = zd
         zalrm = zd
         acbr = 0
         aqbr = 0
         fxaxbr = acbr
         fxaqbr = aqbr
         doneflg = 0
         ERASE alc$, bac, fac, zac, zapc, alq$, baq, faq, zaq, zapq
         CALL FORM1
         true = 0
      END IF
   WEND 'true
'
' If event name found then proceed with drawing of cause
' and consequence tree
'
   apri = 0
   post = 0
   meant = 0
   CALL DISPLAY
'
' Set variables for cursor position at fault event
'
   hnum = falin
   vnum = 30
   apos = 160 * (hnum - 1) + 2 * vnum - 1
'
' Get attribute at fault event
'
   CALL PIXATR
'
' Set cursor at fault event
'
   CALL CURSOR
'
' Switch on soft-key
'
   KEY(1) ON
   KEY(2) ON
   KEY(3) ON
   KEY(4) ON
   KEY(5) ON
   KEY(6) ON
   KEY(7) ON
   KEY(8) ON
   KEY(9) ON
   KEY(10) ON
   KEY(11) ON
   KEY(12) ON
   KEY(13) ON
   KEY(14) ON
```

```
    KEY(15) ON
    KEY(16) ON
    KEY(17) ON
'
' Clear keyboard buffer
'
    DEF SEG = 0
    POKE 1050, PEEK(1052)
    DEF SEG
'
' Single keyboard input to direct program to do certain task
'
    true = 1
    WHILE true
        kb$ = INKEY$
        IF LEN(kb$) = 2 THEN
            kb$ = RIGHT$(kb$,1)
        END IF
        IF kb$ = "" THEN
            true = 1
'
' End display and go back to system
'
        ELSEIF kb$ = CHR$(79) THEN
            true = 0
        END IF
    WEND
    COLOR 2,0 : CLS : CLOSE
    END
'
'=================================================================='
' Subroutine DISPLAY to handle screen display.                    '
'=================================================================='
    SUB DISPLAY STATIC
'
' Draw screen columns
'
    CALL FORM2
'
' Initialise fault column.
'
    CALL IN.FAULT
'
' Calculate conditional probabilites
'
    IF doneflg = 0 THEN
        COLOR 14, 5
        LOCATE 21, 42
        PRINT "Calculating conditional probabilities";
        etop = top
        CALL CONPROB
        COLOR 0, 5
        LOCATE 21, 42
        PRINT SPACE$(37);
        doneflg = 1
    END IF
'
' Get the causes of the alarm
'
    CALL GET.CAUSE(top)
'
' Get the consequences of the alarm
'
```

```
   CALL GET.CONSEQ(top)
'
' Initialise cause column.
'
   CALL IN.CAUSE
'
' Initialise consequence column.
'
   CALL IN.CONSEQ
'
' Display fault event
'
   CALL ALARM
'
' Display causes
'
   CALL CAUSES
'
' Display consequences
'
   CALL QUENCE
'
' Return to main program
'
   END SUB
'
'==============================================================='
' Subroutine DISPLAY1 to handle screen display.                  '
'==============================================================='
   SUB DISPLAY1 STATIC
'
' Clear cause column.
'
   CALL FORM3
'
' Initialise fault column.
'
   CALL IN.FAULT
'
' Initialise cause column.
'
   CALL IN.CAUSE
'
' Display fault event
'
   CALL ALARM
'
' Display causes
'
   CALL CAUSES
'
' Set variables for cursor position at fault event
'
   hnum = falin
   vnum = 30
   apos = 160 * (hnum - 1) + 2 * vnum - 1
'
' Get attribute at fault event
'
   CALL PIXATR
'
' Set cursor at fault event
'
```

```
    CALL CURSOR
'
' Return to main program
'
    END SUB
'
'=================================================================='
' Subroutine DISPLAY2 to handle screen display.                     '
'=================================================================='
    SUB DISPLAY2 STATIC
'
' Clear consequence column.
'
    CALL FORM4
'
' Initialise fault column.
'
    CALL IN.FAULT
'
' Initialise consequence column.
'
    CALL IN.CONSEQ
'
' Display fault event
'
    CALL ALARM
'
' Display consequences
'
    CALL QUENCE
'
' Set variables for cursor position at fault event
'
    hnum = falin
    vnum = 30
    apos = 160 * (hnum - 1) + 2 * vnum - 1
'
' Get attribute at fault event
'
    CALL PIXATR
'
' Set cursor at fault event
'
    CALL CURSOR
'
' Return to main program
'
    END SUB
'
'=================================================================='
' Subroutine "FIND.EV" to find event name in data file.             '
'=================================================================='
    SUB FIND.EV (ffd, ftop, ftcbr, ftqbr, zfd, zfe) STATIC
'
' Initialise flag for testing if event have been found
'
    ffd = 0
    FOR ka = 1 TO p
'
' Get data at record no. ka
'
        GET #1, ka
'
```

```
'
' Test if there is event name found in file.
'
      IF e$ = flt$ THEN
'
' Event name found in data file
'
          ftcbr = CVI(bc$)
          ftqbr = CVI(bq$)
          ftop = ka
          zfd = CVS(pr$)
          zfe = CVS(cpr$)
          ka = p
          ffd = 1
'
' Reset event name for display for cases of mixed gates
'
          IF (LEFT$(e$,1) = "*") AND (g$ = "AND") THEN
              flt$ = SPACE$(15)
              LSET flt$ = e$
              flt$ = flt$ + "(AND)"
          ELSEIF (LEFT$(e$,1) = "*") AND (LEFT$(g$,2) = "OR") THEN
              flt$ = SPACE$(16)
              LSET flt$ = e$
              flt$ = flt$ + "(OR)"
          END IF
      END IF
'
' If data at record no. nf is not the same as event name,
' proceed to next record.
'
   NEXT ka
'
' Return to main program
'
   END SUB
'
'===============================================================
' Subroutine "GET.CAUSE" to get causes.                        '
'===============================================================
   SUB GET.CAUSE(gtop) STATIC
'
' Set sym$ to relevant symbol according to type of gate
' or type of event
'
   GET #1, gtop
   cbr = CVI(bc$)
   IF g$ = "PRI" THEN
      sym$ = CHR$(32)
   ELSE
      IF cbr = 1 THEN
         sym$ = CHR$(196)
      ELSEIF LEFT$(g$,2) = "OR" THEN
         sym$ = CHR$(219)
      ELSEIF g$ = "AND" THEN
         sym$ = CHR$(16)
      END IF
'
' Initialise variables for getting causes
'
      ka = 1
      ctop = gtop
      true1 = 1
'
```

```
'
' Test no. of branches attached to event name and set kb
'
       WHILE true1
          GET #1, ctop
          IF CVI(bc$) => 4 THEN
             kb = 4
          ELSE
             kb = CVI(bc$)
          END IF
'
' Proceed with getting causes of event
'
          FOR kc = 1 TO kb
             GET #1, ctop
             kd = CVI(cau$(kc))
             GET #1, kd
             ca$(ka) = e$
'
' Get probability data
'
             zc(ka) = CVS(pr$)
             zcp(ka) = CVS(cpr$)
'
' Set display attributes
'
             IF g$ = "PRI" THEN
                bc(ka) = 2
             ELSE
                bc(ka) = 3
             END IF
             IF CVI(bq$) > 1 THEN
                fc(ka) = 0
             ELSE
                fc(ka) = 15
             END IF
             IF e$ = aev$ THEN
                bc(ka) = 4
                fc(ka) = 14
             END IF
'
' Reset event name for display for cases of mixed gates
'
             IF (LEFT$(e$,1) = "#") AND (g$ = "AND") THEN
                ca$(ka) = SPACE$(15)
                LSET ca$(ka) = e$
                ca$(ka) = ca$(ka) + "(AND)"
             ELSEIF (LEFT$(e$,1)="#") AND (LEFT$(g$,2)="OR") THEN
                ca$(ka) = SPACE$(16)
                LSET ca$(ka) = e$
                ca$(ka) = ca$(ka) + "(OR)"
             END IF
'
' Increment ka, the cause array index
'
             ka = ka + 1
'
' Get next cause
'
          NEXT kc
'
' Get record containing top event
'
          GET #1, ctop
```

```
'
' Test if no. of branches is less than 4.  If it is then set
' true1 to zero.
'
          IF CVI(bc$) < 4 THEN
             true1 = 0
'
' If no. of branches => 4 then reset top event equal to
' the fourth branch
'
          ELSE
             ctop = CVI(cau$(4))
'
' Test if new top event indicates a continuation event.  If it
' is not a continuation event then set true1 to zero.
'
             GET #1, ctop
             IF NOT(LEFT$(e$,1) = "&") THEN
                true1 = 0
'
' If continuation event then reset variables to get more causes.
'
             ELSE
                ka = ka - 1
             END IF
          END IF
       WEND 'true1
    END IF
'
' Return to caller.
'
    END SUB
'
'=============================================================='
' Subroutine "GET.CONSEQ" to get consequences.                 '
'=============================================================='
    SUB GET.CONSEQ(gtop) STATIC
'
' Test if there are any consequences attached to top event.
'
    GET #1, gtop
    qbr = CVI(bq$)
    IF qbr > 0 THEN
'
' If there are any consequences, initialise variables for
' getting them.
'
       ka = 1
       ctop = gtop
       true1 = 1
'
' Test no. of consequences attached and set kb.
'
       WHILE true1
          GET #1, ctop
          IF CVI(bq$) => 4 THEN
             kb = 4
          ELSE
             kb = CVI(bq$)
          END IF
'
' Proceed with getting the consequences of event.
'
```

```
          FOR kc = 1 TO kb
              GET #1, ctop
              kd = CVI(con$(kc))
              GET #1, kd
              qa$(ka) = e$
'
' Get probability data
'
              zq(ka) = CVS(pr$)
              zqp(ka) = CVS(cpr$)
'
' Set display attributes
'
              IF CVI(bq$) = 0 THEN
                  bq(ka) = 2
              ELSE
                  bq(ka) = 3
              END IF
              IF CVI(bc$) > 1 THEN
                  fq(ka) = 0
              ELSE
                  fq(ka) = 15
              END IF
              IF e$ = aev$ THEN
                  fq(ka) = 14
                  bq(ka) = 4
              END IF
'
' Reset event name for display for cases of mixed gates
'
              IF (LEFT$(e$,1) = "#") AND (g$ = "AND") THEN
                  qa$(ka) = SPACE$(15)
                  LSET qa$(ka) = e$
                  qa$(ka) = qa$(ka) + "(AND)"
              ELSEIF (LEFT$(e$,1) = "#") AND (LEFT$(g$,2) = "OR") THEN
                  qa$(ka) = SPACE$(16)
                  LSET qa$(ka) = e$
                  qa$(ka) = qa$(ka) + "(OR)"
              END IF
'
' Increment ka, the consequence array index
'
              ka = ka + 1
          NEXT kc
'
' Get record containing top event
'
          GET #1, ctop
'
' Test if there are less than 4 consequences attached.
' If it is then set true1 to zero.
'
          IF CVI(bq$) < 4 THEN
              true1 = 0
'
' If no. of consequences => 4 then reset ctop to the
' fourth consequence.
'
          ELSE
              ctop = CVI(con$(4))
'
' Get new top event and test if it is a continuation event.
' If it is not then set true1 to zero.
```

```
'
              GET #1, ctop
              IF NOT(LEFT$(e$,1) = "@") THEN
                  true1 = 0
'
' If continuation then reset variables to get more consequences.
'
              ELSE
                  ka = ka - 1
              END IF
          END IF
      WEND 'true1
  END IF
'
' Return to calling program
'
  END SUB
'
'===============================================================
' Subroutine "IN.FAULT" to initialise fault column.            '
'===============================================================
  SUB IN.FAULT STATIC
'
' Test no. of events to be displayed in fault column and
' set av and aw accordingly.
'
  IF (acbr = 0) AND (aqbr = 0) THEN
      aw = 0
      av = 0
      alb = 0
      fxacbr = 0
      fxaqbr = 0
  ELSEIF NOT(acbr = fxacbr) THEN
      aw = acbr
      fxacbr = acbr
      alb = 0
      IF acbr > 4 THEN
          av = acbr - 4
      ELSEIF (acbr <= 4) AND (aqbr = 0) THEN
          av = 0
      ELSEIF (acbr + aqbr) > 4 THEN
          av = acbr - 4
      ELSE
          av = -aqbr
      END IF
  ELSEIF NOT(aqbr = fxaqbr) THEN
      IF (aqbr + acbr) => 4 THEN
          aw = 4 - aqbr
      ELSE
          aw = acbr
      END IF
      IF aw < acbr THEN
          alb = 1
      ELSE
          alb = 0
      END IF
      fxaqbr = aqbr
      av = -aqbr
  END IF
'
' Calculate alin.
'
  alin = 12 - (aw - av + 1) - 2 * INT((aw - av + 1)/2)
```

```
'
' Reset alin if less than 3
'
   IF alin < 3 THEN
       alin = 3
   ELSEIF (alin > 3) AND (((aw - av + 1) = 2) OR ((aw - av + 1) = 4)) THEN
       alin = alin + 1
   END IF
'
' Store alin for future reference
'
   falin = alin
'
' Return to caller.
'
   END SUB
'
'================================================================'
' Subroutine "IN.CAUSE" to initialise cause column.              '
'================================================================'
   SUB IN.CAUSE STATIC
'
   cv = 1
'
' Test no. of causes and set cw and clb
'
   IF cbr > 5 THEN
       cw = cv + 4
       clb = 1
   ELSE
       cw = cbr
       clb=0
   END IF
'
' Calculate CLIN
'
   clin = 12 - cbr - 2 * INT(cbr/2)
'
' Reset CLIN if less than 3
'
   IF clin < 3 THEN
       clin = 3
   ELSEIF (clin > 3) AND ((cbr = 2) OR (cbr = 4)) THEN
       clin = clin + 1
   END IF
'
' Store CLIN for future reference
'
   fclin = clin
'
' Return to caller.
'
   END SUB
'
'================================================================'
' Subroutine "IN.CONSEQ" to initialise consequence column.       '
'================================================================'
   SUB IN.CONSEQ STATIC
'
   qv = 1
'
' Test no. of consequences and set QW and QLB
'
```

```
    IF qbr > 5 THEN
        qw = qv + 4
        qlb = 1
    ELSE
        qw = qbr
        qlb =0
    END IF
'
' Calculate QLIN
'
    qlin = 12 - qbr - 2 * INT(qbr/2)
'
' Reset QLIN if less than 3
'
    IF qlin < 3 THEN
        qlin = 3
    ELSEIF (qlin > 3) AND ((qbr = 2) OR (qbr = 4)) THEN
        qlin = qlin + 1
    END IF
'
' Store QLIN for future reference
'
    fqlin = qlin
'
' Return to calling program
'
    END SUB
'
'=================================================================='
' Subroutine to draw headings and footings.                        '
'=================================================================='
    SUB FORM1 STATIC
'
    CLS
    COLOR 0, 5
    LOCATE 1, 1
    PRINT SPACE$(80);
    COLOR 2,0
    LOCATE 1, 26
    PRINT CHR$(186);
    LOCATE 1, 54
    PRINT CHR$(186);
    COLOR 0, 5
    LOCATE 1, 10
    PRINT "CAUSES";
    LOCATE 1, 38
    PRINT "FAULT";
    LOCATE 1, 60
    PRINT "CONSEQUENCES";
    COLOR 0, 5
    LOCATE 21, 1
    PRINT SPACE$(80);
    COLOR 0, 5
    LOCATE 21, 2
    PRINT "Press <Alt-H> for HELP";
    COLOR 0, 0
'
' Return to calling program
'
    END SUB
'
```

```
'==============================================================='
' Subroutine to display event at fault.                         '
'==============================================================='
   SUB ALARM STATIC
'
' Output scroll indicator.
'
   COLOR 4, 7
   LOCATE 1, 27
   IF (av > -aqbr) OR (av > 0) THEN
      PRINT CHR$(25);
   ELSE
      PRINT "-";
   END IF
   LOCATE 1, 28
   IF alb = 1 THEN
      PRINT CHR$(24);
   ELSE
      PRINT "-";
   END IF
'
' Output event(s) in alarm column.
'
   IF (av <= 0) THEN
      IF -av < aqbr THEN
         COLOR 15, 0
         LOCATE 2, 48
         PRINT CHR$(179);
      ELSE
         COLOR 0, 0
         LOCATE 2, 48
         PRINT SPACE$(1);
      END IF
      COLOR 0, 0
      LOCATE 2, 31
      PRINT SPACE$(1);
   ELSE
      COLOR 0, 0
      LOCATE 2, 48
      PRINT SPACE$(1);
      COLOR 15, 0
      LOCATE 2, 31
      PRINT CHR$(179);
   END IF
   FOR ka = av TO aw
      IF ka < 0 THEN
         kb = -ka
         COLOR faq(kb), baq(kb)
         LOCATE alin, 30
         PRINT alq$(kb)
         IF NOT(apri = 0) THEN
            CALL APROB(alin+1, 35, zaq(kb), 9)
         END IF
         IF NOT(post = 0) THEN
            IF (zapq(kb) > 1) OR (zapq(kb) = 0) THEN
               fg = 5
            ELSE
               fg = 4
            END IF
            CALL APROB(alin-1, 35, zapq(kb), fg)
         END IF
         COLOR 15, 0
         LOCATE alin+1, 31
```

```
        PRINT SPACE$(1);
        LOCATE alin+1, 48
        PRINT CHR$(179);
        IF NOT(alin+2 = 21) THEN
            LOCATE alin+2, 31
            PRINT SPACE$(1);
            LOCATE alin+2, 48
            PRINT CHR$(179);
            LOCATE alin+3, 31
            PRINT SPACE$(1);
            LOCATE alin+3, 48
            PRINT CHR$(179);
        END IF
    ELSEIF ka = 0 THEN
        IF alft$ = aev$ THEN
            COLOR 14, 4
        ELSE
            COLOR 15, 6
        END IF
        LOCATE alin, 30
        PRINT alft$
        IF NOT(apri = 0) THEN
            CALL APROB(alin+1, 35, zalrm, 9)
        END IF
        IF NOT(post = 0) THEN
            IF (zalcn > 1) OR (zalcn = 0) THEN
                fg = 5
            ELSE
                fg = 4
            END IF
            CALL APROB(alin-1, 35, zalcn, fg)
        END IF
        COLOR 15, 0
        IF NOT(ka = aw) THEN
            LOCATE alin+1, 31
            PRINT CHR$(179);
            LOCATE alin+1, 48
            PRINT SPACE$(1);
            IF NOT(alin+2 = 21) THEN
                LOCATE alin+2, 31
                PRINT CHR$(179);
                LOCATE alin+2, 48
                PRINT SPACE$(1);
                LOCATE alin+3, 31
                PRINT CHR$(179);
                LOCATE alin+3, 48
                PRINT SPACE$(1);
            END IF
        END IF
    ELSE
        COLOR fac(ka), bac(ka)
        LOCATE alin, 30
        PRINT alc$(ka);
        IF NOT(apri = 0) THEN
            CALL APROB(alin+1, 35, zac(ka), 9)
        END IF
        IF NOT(post = 0) THEN
            IF (zapc(ka) > 1) OR (zapc(ka) = 0) THEN
                fg = 5
            ELSE
                fg = 4
            END IF
            CALL APROB(alin-1, 35, zapc(ka), fg)
        END IF
        COLOR 15, 0
        IF NOT(ka = aw) THEN
            LOCATE alin+1, 31
            PRINT CHR$(179);
            LOCATE alin+1, 48
            PRINT SPACE$(1);
            IF NOT(alin+2 = 21) THEN
                LOCATE alin+2, 31
                PRINT CHR$(179);
                LOCATE alin+2, 48
                PRINT SPACE$(1);
                LOCATE alin+3, 31
                PRINT CHR$(179);
                LOCATE alin+3, 48
                PRINT SPACE$(1);
            END IF
        END IF
    END IF
    IF ka < aw THEN alin = alin + 4
NEXT ka
ka = ka - 1
COLOR 15, 0
IF ka < acbr THEN
    IF ka < 0 THEN
        LOCATE 20, 31
        PRINT SPACE$(1);
    ELSE
        LOCATE 20, 31
        PRINT CHR$(179);
        LOCATE 20, 48
        PRINT SPACE$(1);
    END IF
ELSE
    LOCATE 20, 31
    PRINT SPACE$(1);
    LOCATE 20, 48
    PRINT SPACE$(1);
END IF
'
' Draw connecting line from causes to relevent event in alarm
' column.
'
IF NOT(cbr = 0) THEN
    IF alin = 11 THEN
        LOCATE 11, 27
        PRINT CHR$(196);CHR$(196);CHR$(196);
    ELSE
        LOCATE 11, 27
        PRINT CHR$(196);CHR$(191);
        FOR kb = 12 TO alin-1
            LOCATE kb, 28
            PRINT CHR$(179);
        NEXT kb
        IF ka = acbr THEN
            LOCATE alin, 28
            PRINT CHR$(192);CHR$(196);
            LOCATE 20, 28
            PRINT SPACE$(1);
        ELSE
            LOCATE 19, 28
            PRINT CHR$(179);SPACE$(1);
            LOCATE 20, 28
            PRINT CHR$(179);
         END IF
      END IF
   END IF
'
' Draw connecting line from consequence to relevent event
' in alarm column.
'
   IF NOT(qbr = 0) THEN
      IF falin = 11 THEN
         LOCATE 11, 50
         PRINT CHR$(196);CHR$(196);CHR$(196);CHR$(196);
      ELSE
         IF (-av = aqbr) THEN
            LOCATE 2, 52
            PRINT SPACE$(1);
            LOCATE falin, 50
            PRINT CHR$(196);CHR$(196);CHR$(191);
         ELSE
            LOCATE 2, 52
            PRINT CHR$(179);
            LOCATE 3, 50
            PRINT SPACE$(2);CHR$(179);
         END IF
         FOR kb = falin+1 TO 10
            LOCATE kb, 52
            PRINT CHR$(179);
         NEXT kb
         LOCATE 11, 52
         PRINT CHR$(192);CHR$(196);
      END IF
   END IF
'
' Return to calling program
'
   END SUB
'
'===========================================================================
' Subroutine to display first level causes.                                '
'===========================================================================
   SUB CAUSES STATIC
'
   COLOR 4, 7
   LOCATE 1, 1
   IF cv > 1 THEN
      PRINT CHR$(25);
   ELSE
      PRINT "-";
   END IF
   LOCATE 1, 2
   IF clb = 1 THEN
      PRINT CHR$(24);
   ELSE
      PRINT "-";
   END IF
   IF sym$ <> CHR$(32) THEN
      IF NOT(cv = 1) THEN
         COLOR 15, 0
         LOCATE 2, 26
         PRINT CHR$(179);
      ELSE
         COLOR 2, 0
         LOCATE 2, 26
```

```
            PRINT CHR$(186)
        END IF
        FOR ka = cv TO cw
            LSET nm$ = MID$(STR$(ka), 2)
            COLOR 15, 5
            LOCATE clin, 3
            PRINT nm$;
            COLOR fc(ka), bc(ka)
            LOCATE clin, 5
            PRINT ca$(ka);
            COLOR 15, 0
            LOCATE clin, 25
            PRINT CHR$(196);
            LOCATE clin, 26
            IF ka = 1 THEN
                PRINT CHR$(191);
            ELSEIF ka = cbr THEN
                PRINT CHR$(217);
            ELSE
                PRINT CHR$(180);
            END IF
            IF NOT(apri = 0) THEN
                CALL APROB (clin+1, 15, zc(ka), 9)
            END IF
            IF NOT(post = 0) THEN
                IF (zcp(ka) > 1) OR (zcp(ka) = 0) THEN
                    fg = 5
                ELSE
                    fg = 4
                END IF
                CALL APROB (clin-1, 15, zcp(ka), fg)
            END IF
            IF NOT(ka = cbr) THEN
                COLOR 15, 0
                LOCATE clin+1, 26
                PRINT CHR$(179);
                IF NOT(clin+2 = 21) THEN
                    LOCATE clin+2, 26
                    PRINT CHR$(179);
                    LOCATE clin+3, 26
                    PRINT CHR$(179);
                END IF
            ELSE
                COLOR 2,0
                LOCATE clin+1, 26
                PRINT CHR$(186);
            END IF
            IF ka < cw THEN clin = clin + 4
        NEXT ka
        COLOR 14, 0
        LOCATE 11, 26
        PRINT sym$;
    END IF
'
' Return to calling program
'
    END SUB
'
```

```
'==============================================================='
' Subroutine to display consequences.                            '
'==============================================================='
   SUB QUENCE STATIC
'
   COLOR 4, 7
   LOCATE 1, 55
   IF qv > 1 THEN
      PRINT CHR$(25);
   ELSE
      PRINT "-";
   END IF
   LOCATE 1, 56
   IF qlb = 1 THEN
      PRINT CHR$(24);
   ELSE
      PRINT "-";
   END IF
   IF NOT(qv = 1) THEN
      COLOR 15, 0
      LOCATE 2, 54
      PRINT CHR$(179);
   ELSE
      COLOR 2, 0
      LOCATE 2, 54
      PRINT CHR$(186);
   END IF
   FOR ka = qv TO qw
      RSET nm$ = RIGHT$(STR$(ka), 2)
      COLOR 15, 0
      LOCATE qlin, 54
      IF (qlin = 11) AND (qbr > 1) THEN
         PRINT CHR$(197);
      ELSEIF qbr = 1 THEN
         PRINT CHR$(196);
      ELSEIF ka = 1 THEN
         PRINT CHR$(218);
      ELSEIF ka = qbr THEN
         PRINT CHR$(192);
      ELSE
         PRINT CHR$(195);
      END IF
      LOCATE qlin, 55
      PRINT CHR$(196);
      COLOR fq(ka), bq(ka)
      LOCATE qlin, 56
      PRINT qa$(ka);
      COLOR 15, 5
      LOCATE qlin, 76
      PRINT nm$;
      IF NOT(apri = 0) THEN
         CALL APROB (qlin+1, 66, zq(ka), 9)
      END IF
      IF NOT(post = 0) THEN
         IF (zqp(ka) > 1) OR (zqp(ka) = 0) THEN
            fg = 5
         ELSE
            fg = 4
         END IF
         CALL APROB (qlin-1, 66, zqp(ka), fg)
      END IF
      IF NOT(ka = qbr) THEN
         COLOR 15, 0
```

```
          FOR kb = 1 TO 3
          IF qlin+kb <= 20 THEN
        LOCATE qlin+kb, 54
        IF qlin+kb = 11 THEN
          PRINT CHR$(180);
        ELSE
          PRINT CHR$(179);
        END IF
      ELSE
        kb = 3
      END IF
    NEXT kb
  ELSE
    COLOR 2, 0
    LOCATE qlin+1, 54
    PRINT CHR$(186);
  END IF
  IF ka < qw THEN qlin = qlin + 4
  NEXT ka
  COLOR 2,0
'
' Return to calling program
'
  END SUB
'
'================================================================'
' Subroutine to get pixel attribute.                              '
'================================================================'
  SUB PIXATR STATIC
'
  DEF SEG = &HB800
  atr = PEEK(apos)
  DEF SEG
  END SUB
'
'================================================================'
' Subroutine to set cursor at fault event.                        '
'================================================================'
  SUB CURSOR STATIC
'
  DEF SEG = &HB800
  POKE apos, atr
  apos = 160 * (hnum - 1) + 2 * vnum - 1
  atr = PEEK(apos)
  POKE apos, 128+atr
  DEF SEG
  IF meant = 1 THEN
      col = vnum
      row = hnum
      CALL EVTNAM(evt$)
      IF LEFT$(evt$, 1) = "*" THEN
          evt$ = LEFT$(evt$, 7)
      END IF
      CALL MEANING(evt$, mean$)
      COLOR 4, 7
      LOCATE 24, 5
      PRINT SPACE$(74);
      LOCATE 24, 5
      PRINT mean$;
  END IF
  END SUB
'
```

```
'================================================================'
' Subroutine to scroll fault column upwards.                     '
'================================================================'
   SUB FAUCOLUP STATIC
'
   IF (aw = acbr) THEN
       SOUND 400, 2
   ELSE
       av = av + 1
       aw = aw + 1
       IF aw = acbr THEN
           alb = 0
       ELSE
           alb = 1
       END IF
       alin = falin
       CALL ALARM
       IF vnum = 30 THEN
           apos = 160 * (hnum - 1) + 2 * vnum - 1
           CALL PIXATR
           CALL CURSOR
       END IF
   END IF
   END SUB
'
'================================================================'
' Subroutine to scroll fault column downwards.                   '
'================================================================'
   SUB FAUCOLDN STATIC
'
   IF -av = aqbr THEN
       SOUND 400, 2
   ELSE
       av = av - 1
       aw = aw - 1
       alb = 1
       alin = falin
       CALL ALARM
       IF vnum = 30 THEN
           apos = 160 * (hnum - 1) + 2 * vnum - 1
           CALL PIXATR
           CALL CURSOR
       END IF
   END IF
   END SUB
'
'================================================================'
' Subroutine to scroll cause column upwards.                     '
'================================================================'
   SUB CAUCOLUP STATIC
'
   IF (cw = cbr) OR (sym$ = chr$(32)) THEN
       SOUND 400, 2
   ELSE
       cv = cv + 1
       cw = cw + 1
       IF cw = cbr THEN
           clb = 0
       ELSE
           clb = 1
       END IF
       clin = fclin
       CALL CAUSES
```

```
        IF vnum = 5 THEN
            apos = 160 * (hnum - 1) + 2 * vnum - 1
            CALL PIXATR
            CALL CURSOR
        END IF
    END IF
    END SUB
'
'==================================================================='
' Subroutine to scroll cause column downwards.                      '
'==================================================================='
    SUB CAUCOLDN STATIC
'
    IF (cv = 1) OR (sym$ = CHR$(32)) THEN
        SOUND 400, 2
    ELSE
        cv = cv - 1
        cw = cw - 1
        clb = 1
        clin = fclin
        CALL CAUSES
        IF vnum = 5 THEN
            apos = 160 * (hnum - 1) + 2 * vnum - 1
            CALL PIXATR
            CALL CURSOR
        END IF
    END IF
    END SUB
'
'==================================================================='
' Subroutine to scroll consequence column upwards.                  '
'==================================================================='
    SUB QONCOLUP STATIC
'
    IF qw = qbr THEN
        SOUND 400, 2
    ELSE
        qv = qv + 1
        qw = qw + 1
        IF qw = qbr THEN
            qlb = 0
        ELSE
            qlb = 1
        END IF
        qlin = fqlin
        CALL QUENCE
        IF vnum = 56 THEN
            apos = 160 * (hnum - 1) + 2 * vnum - 1
            CALL PIXATR
            CALL CURSOR
        END IF
    END IF
    END SUB
'
'==================================================================='
' Subroutine to scroll consequence column downwards.                '
'==================================================================='
    SUB QONCOLDN STATIC
'
    IF qv = 1 THEN
        SOUND 400, 2
    ELSE
        qv = qv - 1
        qw = qw - 1
        qlb = 1
        qlin = fqlin
        CALL QUENCE
        IF vnum = 56 THEN
            apos = 160 * (hnum - 1) + 2 * vnum - 1
            CALL PIXATR
            CALL CURSOR
        END IF
    END IF
    END SUB
'
'=================================================================='
' Subroutine to toggle display of apriori probabilities.          '
'=================================================================='
    SUB APRIORI STATIC
'
    alin = falin
    FOR ka = av TO aw
        IF apri = 0 THEN
            COLOR 0, 0
            LOCATE alin+1, 35
            PRINT SPACE$(10);
        ELSE
            IF ka < 0 THEN
                kb = -ka
                CALL APROB (alin+1, 35, zaq(kb), 9)
            ELSEIF ka = 0 THEN
                CALL APROB (alin+1, 35, zalrm, 9)
            ELSE
                CALL APROB (alin+1, 35, zac(ka), 9)
            END IF
        END IF
        IF ka < aw THEN alin = alin + 4
    NEXT ka
    clin = fclin
    FOR ka = cv TO cw
        IF apri = 0 THEN
            COLOR 0, 0
            LOCATE clin+1, 5
            PRINT SPC(20);
        ELSE
            CALL APROB (clin+1, 15, zc(ka), 9)
        END IF
        IF ka < cw THEN clin = clin + 4
    NEXT ka
    qlin = fqlin
    FOR ka = qv TO qw
        IF apri = 0 THEN
            COLOR 0, 0
            LOCATE qlin+1, 56
            PRINT SPC(20);
        ELSE
            CALL APROB (qlin+1, 66, zq(ka), 9)
        END IF
        IF ka < qw THEN qlin = qlin + 4
    NEXT ka
'
' Return to caller.
'
    END SUB
'
```

```
'============================================================'
' Subroutine to toggle display of posteriori probabilities.  '
'============================================================'
  SUB POSTER STATIC
'
  alin = falin
  FOR ka = av TO aw
     IF post = 0 THEN
        COLOR 0, 0
        LOCATE alin-1, 35
        PRINT SPC(10);
     ELSE
        IF ka < 0 THEN
           kb = -ka
           IF (zapq(kb) > 1) OR (zapq(kb) = 0) THEN
              fg = 5
           ELSE
              fg = 4
           END IF
           CALL APROB (alin-1, 35, zapq(kb), fg)
        ELSEIF ka = 0 THEN
           IF (zalcn > 1) OR (zalcn = 0) THEN
              fg = 5
           ELSE
              fg = 4
           END IF
           CALL APROB (alin-1, 35, zalcn, fg)
        ELSEIF ka > 0 THEN
           IF (zapc(ka) > 1) OR (zapc(ka) = 0) THEN
              fg = 5
           ELSE
              fg = 4
           END IF
           CALL APROB (alin-1, 35, zapc(ka), fg)
        END IF
     END IF
     IF ka < aw THEN alin = alin + 4
  NEXT ka
'
  clin = fclin
  FOR ka = cv TO cw
     IF post = 0 THEN
        COLOR 0, 0
        LOCATE clin-1, 5
        PRINT SPC(20);
     ELSE
        IF (zcp(ka) > 1) OR (zcp(ka) = 0) THEN
           fg = 5
        ELSE
           fg = 4
        END IF
        CALL APROB (clin-1, 15, zcp(ka), fg)
     END IF
     IF ka < cw THEN clin = clin + 4
  NEXT ka
'
  qlin = fqlin
  FOR ka = qv TO qw
     IF post = 0 THEN
        COLOR 0, 0
        LOCATE qlin-1, 56
        PRINT SPC(20);
     ELSE
```

```
          IF (zqp(ka) > 1) OR (zqp(ka) = 0) THEN
             fg = 5
          ELSE
             fg = 4
                fg = 4
             END IF
             CALL APROB (alin-1, 35, zapc(ka), fg)
          END IF
       END IF
       IF ka < aw THEN alin = alin + 4
    NEXT ka
 '
    clin = fclin
    FOR ka = cv TO cw
       IF post = 0 THEN
          COLOR 0, 0
          LOCATE clin-1, 5
          PRINT SPC(20);
       ELSE
          IF (zcp(ka) > 1) OR (zcp(ka) = 0) THEN
             fg = 5
          ELSE
             fg = 4
          END IF
          CALL APROB (clin-1, 15, zcp(ka), fg)
       END IF
       IF ka < cw THEN clin = clin + 4
    NEXT ka
 '
    qlin = fqlin
    FOR ka = qv TO qw
       IF post = 0 THEN
          COLOR 0, 0
          LOCATE qlin-1, 56
          PRINT SPC(20);
       ELSE
          IF (zqp(ka) > 1) OR (zqp(ka) = 0) THEN
             fg = 5
          ELSE
             fg = 4
          END IF
          CALL APROB (qlin-1, 66, zqp(ka), fg)
       END IF
       IF ka < qw THEN qlin = qlin + 4
    NEXT ka
 '
 ' Return to caller.
 '
    END SUB
 '
 '=============================================================='
 ' Subroutine to toggle display of apriori probabilities.       '
 '=============================================================='
 TOGGLE1:
 '
    IF apri = 0 THEN
       apri = 1
    ELSE
       apri = 0
    END IF
    CALL APRIORI
    RETURN
 '
```

```
'=================================================================='
' Subroutine to toggle display of posteriori probabilities.       '
'=================================================================='
TOGGLE2:
'
   IF post = 0 THEN
      post = 1
   ELSE
      post = 0
   END IF
   CALL POSTER
   RETURN
'
'=================================================================='
' Key trap subroutine to scroll cause column up.                  '
'=================================================================='
CAUSEUP:
'
   CALL CAUCOLUP
   RETURN
'
'=================================================================='
' Key trap subroutine to scroll cause column down.                '
'=================================================================='
CAUSEDN:
'
   CALL CAUCOLDN
   RETURN
'
'=================================================================='
' Key trap subroutine to scroll fault column up.                  '
'=================================================================='
FAULTUP:
'
   CALL FAUCOLUP
   RETURN
'
'=================================================================='
' Key trap subroutine to scroll fault column down.                '
'=================================================================='
FAULTDN:
'
   CALL FAUCOLDN
   RETURN
'
'=================================================================='
' Key trap subroutine to scroll consequence column up.            '
'=================================================================='
CONSEQUP:
'
   CALL QONCOLUP
  RETURN
'
'=================================================================='
' Key trap subroutine to scroll consequence column down.          '
'=================================================================='
CONSEQDN:
'
   CALL QONCOLDN
   RETURN
'
```

```
'=============================================================='
' Key trap subroutine to show other parts of fault tree.       '
'=============================================================='
OTHERS:
'
   IF vnum = 30 THEN
       IF (acbr = 0) AND (aqbr = 0) THEN
           SOUND 600, 1
       ELSE
           CALL EVTNAM (evt$)
           LSET flt$ = evt$
           indx = 0
           ipost = 0
           DO
               FOR ka = aqbr TO 1 STEP -1
                   IF flt$ = alq$(ka) THEN
                       indx = ka
                       ipost = 1
                       ka = 1
                   END IF
               NEXT ka
               IF NOT(indx = 0) THEN EXIT DO
               FOR ka = acbr TO 1 STEP -1
                   IF flt$ = alc$(ka) THEN
                       indx = ka
                       ipost = 2
                       ka = 1
                   END IF
               NEXT ka
               EXIT DO
           LOOP
           IF ((indx = aqbr) AND (ipost = 1)) OR_
              ((indx = acbr) AND (ipost = 2)) THEN
               SOUND 600, 1
           ELSE
               IF LEFT$(evt$, 1) = "¤" THEN
                   evt$ = LEFT$(evt$, 7)
               END IF
               LSET flt$ = evt$
               CALL FIND.EV (fd, top, tcbr, tqbr, zd, ze)
               IF (ipost = 1) OR ((indx = 0) AND (acbr = 0)) THEN
                   CALL GET.CONSEQ(top)
                   aqbr = indx
                   CALL DISPLAY2
               ELSE
                   CALL GET.CAUSE(top)
                   acbr = indx
                   CALL DISPLAY1
               END IF
           END IF
       END IF
   ELSE
       CALL NUMB(jdx)
       CALL EVTNAM (evt$)
       IF LEFT$(evt$, 1) = "¤" THEN
           evt$ = LEFT$(evt$, 7)
       END IF
       LSET flt$ = evt$
       CALL FIND.EV (fd, top, tcbr, tqbr, zd, ze)
       IF vnum = 5 THEN
           acbr = acbr + 1
           alc$(acbr) = flt$
           zac(acbr) = zd
```

```
            zapc(acbr) = ze
            bac(acbr) = bc(jdx)
            fac(acbr) = fc(jdx)
            CALL GET.CAUSE(top)
            CALL DISPLAY1
        ELSE
            aqbr = aqbr + 1
            alq$(aqbr) = flt$
            zaq(aqbr) = zd
            zapq(aqbr) = ze
            baq(aqbr) = bq(jdx)
            faq(aqbr) = fq(jdx)
            CALL GET.CONSEQ(top)
            CALL DISPLAY2
        END IF
    END IF
    RETURN
'
'=================================================================='
' Key trap subroutine to show original fault tree.                 '
'=================================================================='
ORIGINAL:
'
    IF (acbr = 0) AND (aqbr = 0) AND (alft$ = aev$) THEN
        SOUND 600, 1
    ELSE
        ERASE alc$, fac, bac, zac, zapc, alq$, faq, baq, zaq, zapq
        acbr = 0
        aqbr = 0
        fxacbr = acbr
        fxaqbr = aqbr
        alft$ = aev$
        zalrm = zffx
        zalcn = zcfx
        top = fxtop
        CALL DISPLAY
'
' Set variables for cursor position at fault event
'
        hnum = falin
        vnum = 30
        apos = 160 * (hnum - 1) + 2 * vnum - 1
'
' Get attribute at fault event
'
        CALL PIXATR
'
' Set cursor at fault event
'
        CALL CURSOR
    END IF
    RETURN
'
'=================================================================='
' Subroutine to move cursor left.                                  '
'=================================================================='
CURLEFT:
'
    IF (vnum = 30) AND (cbr = 0) THEN
        SOUND 350, 1
    ELSE
        IF vnum = 30 THEN
            vnum = 5
```

```
          hnum = fclin
          CALL CURSOR
       ELSEIF vnum = 56 THEN
          vnum = 30
          hnum = falin
          CALL CURSOR
       ELSE
          SOUND 350,1
       END IF
    END IF
    RETURN
'
'==================================================================='
' Subroutine to move cursor down.                                   '
'==================================================================='
  CURDOWN:
'
  hnum = hnum + 4
  IF ((hnum > clin) AND (vnum = 5)) OR—
     ((hnum > qlin) AND (vnum = 56)) OR—
     ((hnum > alin) AND (vnum = 30)) THEN
    hnum = hnum - 4
    SOUND 350, 1
  ELSE
    CALL CURSOR
  END IF
  RETURN
'
'==================================================================='
' Subroutine to move cursor up.                                     '
'==================================================================='
CURUP:
'
    hnum = hnum - 4
    IF ((hnum < fclin) AND (vnum = 5)) OR_
       ((hnum < fqlin) AND (vnum = 56)) OR_
       ((hnum < falin) AND (vnum = 30)) THEN
        hnum = hnum + 4
        SOUND 350, 1
    ELSE
        CALL CURSOR
    END IF
    RETURN
'
'==================================================================='
' Subroutine to move cursor right.                                  '
'==================================================================='
CURRIGHT:
'
    IF (vnum = 30) AND (qbr = 0) THEN
        SOUND 350, 1
    ELSE
        IF vnum = 30 THEN
            vnum = 56
            hnum = fqlin
            CALL CURSOR
        ELSEIF vnum = 5 THEN
            vnum = 30
            hnum = falin
            CALL CURSOR
        ELSE
            SOUND 350,1
        END IF
    END IF
    RETURN
'
'=================================================================='
' Key trap subroutine to show user input fault tree.               '
'=================================================================='
ANOTHER:
'
' Save characters and attributes in message area of screen in
' array done1().
'
    ERASE done1
    DEF SEG = &HB800
    kb = 1
    FOR ka = 22 TO 24
        kc = 160 * (ka - 1) + 4
        kd = 160 * (ka - 1) + 153
        FOR ke = kc TO kd
            done1(kb) = PEEK(ke)
            kb = kb + 1
        NEXT ke
    NEXT ka
    DEF SEG
'
' Output message.
'
    COLOR 5, 7
    LOCATE 22, 3
    PRINT CHR$(186);SPACE$(73);CHR$(186);
    LOCATE 23, 3
    PRINT CHR$(186);SPACE$(73);CHR$(186);
    LOCATE 24, 3
    PRINT CHR$(200);
    FOR ka = 1 TO 73
        PRINT CHR$(205);
    NEXT ka
    PRINT CHR$(188);
    true1 = 1
    WHILE true1
        COLOR 1, 7
        LOCATE 23, 5
        PRINT "Input event name (0 to abort) :";
        INPUT evt$
        IF evt$ = "0" THEN
            true1 = 0
        ELSE
            LSET flt$ = evt$
            CALL FIND.EV(fd, top, tcbr, tqbr, zd, ze)
            IF fd = 0 THEN
                COLOR 4, 7
                LOCATE 22, 5
                PRINT "Event not found!  Please input again.";
                SOUND 600, 3
            ELSE
                ERASE alc$, bac, fac, zac, zapc, alq$, baq, faq, zaq, zapq
                acbr = 0
                aqbr = 0
                fxacbr = acbr
                fxaqbr = aqbr
                alft$ = flt$
                zalrm = zd
                zalcn = ze
                CALL DISPLAY
                true1 = 0
            END IF
        END IF
    WEND 'true1
    DEF SEG = &HB800
    kb = 1
    FOR ka = 22 TO 24
        kc = 160 * (ka - 1) + 4
        kd = 160 * (ka - 1) + 153
        FOR ke = kc TO kd
            POKE ke, done1(kb)
            kb = kb + 1
        NEXT ke
    NEXT ka
    DEF SEG
'
' Set variables for cursor position at fault event
'
    IF NOT(evt$ = "0") THEN
        hnum = falln
        vnum = 30
        apos = 160 * (hnum - 1) + 2 * vnum - 1
'
' Get attribute at fault event
'
        CALL PIXATR
'
' Set cursor at fault event
'
        CALL CURSOR
    END IF
    RETURN
'
'=================================================================='
' Key trap subroutine to display meaning of coded event.           '
'=================================================================='
EXPCODE:
'
    IF meant = 0 THEN
        meant = 1
    ELSE
        meant = 0
    END IF
    CALL EXPLAIN
    RETURN
'
'=================================================================='
' Key trap subroutine to display help screen.                      '
'=================================================================='
HELP:
'
    SCREEN 0, 1, 1
    CLS
    COLOR 14, 0
    LOCATE 1,1
    PRINT CHR$(201);
    FOR I=1 TO 78
        PRINT CHR$(205);
    NEXT i
    PRINT CHR$(187);
    LOCATE 2,1
    PRINT CHR$(186);
    COLOR 2,0
```

```
LOCATE 2,8
PRINT "HELP for operating on display.  Press ";
COLOR 3, 0
PRINT "ESC ";
COLOR 2, 0
PRINT "key to return to display.";
COLOR 14, 0
LOCATE 2,80
PRINT CHR$(186);
LOCATE 3, 1
PRINT CHR$(204);
FOR i = 1 to 10
   PRINT CHR$(205);
NEXT i
PRINT CHR$(209);
FOR i=1 TO 67
   PRINT CHR$(205);
NEXT i
PRINT CHR$(185);
LOCATE 4, 1
PRINT CHR$(186);
COLOR 2, 0
LOCATE 4, 5
PRINT "Key"
COLOR 14, 0
LOCATE 4, 12
PRINT CHR$(179);
COLOR 2, 0
LOCATE 4,40
PRINT "Function";
COLOR 14, 0
LOCATE 4, 80
PRINT CHR$(186);
LOCATE 5, 1
PRINT CHR$(199);
FOR i = 1 to 10
   PRINT CHR$(196);
NEXT i
PRINT CHR$(197);
FOR i=1 TO 67
   PRINT CHR$(196);
NEXT i
PRINT CHR$(182);
LOCATE 6, 1
PRINT CHR$(186)
COLOR 3, 0
LOCATE 6,5
PRINT "F1";
COLOR 14, 0
LOCATE 6, 12
PRINT CHR$(179);
COLOR 7, 0
LOCATE 6, 14
PRINT "Switch on/off display of a priori probabilities."
COLOR 14, 0
LOCATE 6,80
PRINT CHR$(186);
LOCATE 7, 1
PRINT CHR$(199);
FOR i=1 to 10
   PRINT CHR$(196);
NEXT i
PRINT CHR$(197);
```

```
FOR I=1 to 67
   PRINT CHR$(196);
NEXT I
PRINT CHR$(182);
LOCATE 8, 1
PRINT CHR$(186);
COLOR 3, 0
LOCATE 8, 5
PRINT "F2";
COLOR 14, 0
LOCATE 8, 12
PRINT CHR$(179);
COLOR 7, 0
LOCATE 8, 14
PRINT "Switch on/off display of posteriori probabilities."
COLOR 14, 0
LOCATE 8, 80
PRINT CHR$(186);
LOCATE 9, 1
PRINT CHR$(199);
FOR I=1 to 10
   PRINT CHR$(196);
NEXT I
PRINT CHR$(197);
FOR I=1 to 67
   PRINT CHR$(196);
NEXT I
PRINT CHR$(182);
LOCATE 10, 1
PRINT CHR$(186);
COLOR 3, 0
LOCATE 10, 4
PRINT "F3/F4"
COLOR 14, 0
LOCATE 10, 12
PRINT CHR$(179);
COLOR 7, 0
LOCATE 10, 14
PRINT "Scroll CAUSE column upwards/downwards.";
COLOR 14, 0
LOCATE 10, 80
PRINT CHR$(186);
LOCATE 11, 1
PRINT CHR$(199);
FOR I=1 to 10
   PRINT CHR$(196);
NEXT I
PRINT CHR$(197);
FOR I=1 to 67
   PRINT CHR$(196);
NEXT I
PRINT CHR$(182);
LOCATE 12, 1
PRINT CHR$(186);
COLOR 3, 0
LOCATE 12, 4
PRINT "F5/F6"
COLOR 14, 0
LOCATE 12, 12
PRINT CHR$(179);
COLOR 7, 0
LOCATE 12, 14
PRINT "Scroll ALARM column upwards/downwards.";
```

```
COLOR 14, 0
LOCATE 12, 80
PRINT CHR$(186);
LOCATE 13, 1
PRINT CHR$(199);
FOR I=1 to 10
    PRINT CHR$(196);
NEXT I
PRINT CHR$(197);
FOR i=1 to 67
    PRINT CHR$(196);
NEXT i
PRINT CHR$(182);
LOCATE 14, 1
PRINT CHR$(186);
COLOR 3, 0
LOCATE 14, 4
PRINT "F7/F8"
COLOR 14, 0
LOCATE 14, 12
PRINT CHR$(179);
COLOR 7, 0
LOCATE 14, 14
PRINT "Scroll CONSEQUENCE column upwards/downwards.";
COLOR 14, 0
LOCATE 14, 80
PRINT CHR$(186);
LOCATE 15, 1
PRINT CHR$(199);
FOR I=1 to 10
    PRINT CHR$(196);
NEXT I
PRINT CHR$(197);
FOR i=1 to 67
    PRINT CHR$(196);
NEXT i
PRINT CHR$(182);
LOCATE 16, 1
PRINT CHR$(186);
COLOR 3, 0
LOCATE 16, 5
PRINT "F9"
COLOR 14, 0
LOCATE 16, 12
PRINT CHR$(179);
COLOR 7, 0
LOCATE 16, 14
PRINT "Display causes or consequences of event at cursor."
COLOR 14, 0
LOCATE 16, 80
PRINT CHR$(186);
LOCATE 17, 1
PRINT CHR$(199);
FOR I=1 to 10
    PRINT CHR$(196);
NEXT I
PRINT CHR$(197);
FOR I=1 to 67
    PRINT CHR$(196);
NEXT i
PRINT CHR$(182);
LOCATE 18, 1
PRINT CHR$(186);
```

```
COLOR 3, 0
LOCATE 18, 5
PRINT "F10"
COLOR 14, 0
LOCATE 18, 12
PRINT CHR$(179);
COLOR 7, 0
LOCATE 18, 14
PRINT "Display original fault tree.";
COLOR 14, 0
LOCATE 18, 80
PRINT CHR$(186);
LOCATE 19, 1
PRINT CHR$(199);
FOR i=1 to 10
   PRINT CHR$(196);
NEXT i
PRINT CHR$(197);
FOR i=1 to 67
   PRINT CHR$(196);
NEXT i
PRINT CHR$(182);
LOCATE 20, 1
PRINT CHR$(186);
COLOR 3, 0
LOCATE 20, 3
PRINT "ALT-A"
COLOR 14, 0
LOCATE 20, 12
PRINT CHR$(179);
COLOR 7, 0
LOCATE 20, 14
PRINT "Display causes and consequences of event input via keyboard.";
COLOR 14, 0
LOCATE 20, 80
PRINT CHR$(186);
LOCATE 21, 1
PRINT CHR$(199);
FOR i=1 to 10
   PRINT CHR$(196);
NEXT i
PRINT CHR$(197);
FOR i=1 to 67
   PRINT CHR$(196);
NEXT i
PRINT CHR$(182);
LOCATE 22, 1
PRINT CHR$(186);
COLOR 3, 0
LOCATE 22, 3
PRINT "ALT-E"
COLOR 14, 0
LOCATE 22, 12
PRINT CHR$(179);
COLOR 7, 0
LOCATE 22, 14
PRINT "Switch on/off display of meaning of event at cursor.";
COLOR 14, 0
LOCATE 22, 80
PRINT CHR$(186);
LOCATE 23, 1
PRINT CHR$(200);
FOR i=1 to 10
```

```
        PRINT CHR$(205);
    NEXT i
    PRINT CHR$(207);
    FOR i=1 to 67
        PRINT CHR$(205);
    NEXT i
    PRINT CHR$(188);
    true1 = 1
    WHILE true1
        kb2$ = INKEY$
        IF LEN(kb2$) = 2 THEN
            kb2$ = RIGHT$(kb2$,1)
        END IF
        IF kb2$ = "" THEN
            true = 1
        ELSE
            IF kb2$ = CHR$(27) THEN
                true1 = 0
                SCREEN 0, 1, 0
                COLOR , , 1
            END IF
        END IF
    WEND 'true1
    RETURN
'=============================================================='
' Subroutine to draw columns.                                  '
'=============================================================='
    SUB FORM2 STATIC
'
    COLOR 2, 0
    FOR ka = 2 TO 20
        LOCATE ka, 1
        PRINT SPACE$(80);
        LOCATE ka, 26
        PRINT CHR$(186);
        LOCATE ka, 54
        PRINT CHR$(186);
    NEXT ka
'
    END SUB
'
'=============================================================='
' Subroutine to clear cause column.                            '
'=============================================================='
    SUB FORM3 STATIC
'
    COLOR 2, 0
    FOR ka = 2 TO 20
        LOCATE ka, 1
        PRINT SPACE$(53);
        LOCATE ka, 26
        PRINT CHR$(186);
    NEXT ka
'
    END SUB
'
'=============================================================='
' Subroutine to clear consequence column.                      '
'=============================================================='
    SUB FORM4 STATIC
'
    COLOR 2, 0
    FOR ka = 2 TO 20
```

```
      LOCATE ka, 27
      PRINT SPACE$(53);
      LOCATE ka, 54
      PRINT CHR$(186);
   NEXT ka
'
   END SUB
'
'============================================================'
' Subroutine to show meaning of coded events.                '
'============================================================'
   SUB EXPLAIN STATIC
'
   IF meant = 1 THEN
      col = vnum
      row = hnum
      CALL EVTNAM(evt$)
      IF LEFT$(evt$, 1) = "#" THEN
         evt$ = LEFT$(evt$, 7)
      END IF
      CALL MEANING(evt$, mean$)
      COLOR 5, 7
      LOCATE 22, 3
      PRINT CHR$(186);" Move cursor to required event.  ";
      PRINT "Meaning is displayed below.                ";CHR$(186);
      LOCATE 23, 3
      PRINT CHR$(199);
      FOR ka = 1 TO 73
         PRINT CHR$(196);
      NEXT ka
      PRINT CHR$(182);
      LOCATE 24, 3
      PRINT CHR$(186);SPACE$(73);
      LOCATE 24, 77
      PRINT CHR$(186);
      LOCATE 25, 3
      PRINT CHR$(200);
      FOR ka = 1 TO 73
         PRINT CHR$(205);
      NEXT ka
      PRINT CHR$(188);
      COLOR 4, 7
      LOCATE 24, 5
      PRINT mean$;
   ELSE
      COLOR 2, 0
      FOR ka = 22 TO 25
         LOCATE ka, 3
         PRINT SPACE$(75);
      NEXT ka
   END IF
'
   END SUB
'
'============================================================'
' Subroutine to get index of event                           '
'============================================================'
   SUB NUMB (jdx) STATIC
'
   evt$ = ""
   IF vnum = 5 THEN
      cpos = apos - 7
   ELSEIF vnum = 56 THEN
```

```
      cpos = apos + 39
   END IF
   DEF SEG = &HB800
   FOR ka = 1 TO 3
      cas = PEEK(cpos)
      evt$ = evt$ + CHR$(cas)
      cpos = cpos + 2
   NEXT ka
   DEF SEG
   jdx = VAL(evt$)
   END SUB
'
'=================================================================='
' Subroutine to get event name from screen display.                '
'=================================================================='
   SUB EVTNAM (evt$) STATIC
'
   evt$ = ""
   cpos = apos - 1
   DEF SEG = &HB800
   FOR ka = 1 TO 20
      cas = PEEK(cpos)
      evt$ = evt$ + CHR$(cas)
      cpos = cpos + 2
   NEXT ka
   DEF SEG
   END SUB
'
'=================================================================='
' Subroutine to output apriori probability of an event.            '
'=================================================================='
   SUB APROB (evrow, evcol, zval, fg) STATIC
'
   COLOR fg, 7
   LOCATE evrow, evcol
   PRINT SPACE$(10);
   LOCATE evrow, evcol
   IF (zval = 0) OR (zval => 1) THEN
      PRINT USING "##";zval
   ELSE
      PRINT USING "#.########";zval
   END IF
   END SUB
'
'=================================================================='
' Subroutine to calculate conditional probabilities of fault tree. '
'=================================================================='
   SUB CONPROB STATIC
'
   ERASE s1, s2, andflg, comset, done
   zalcn = 1
   zcfx = 1
   gtop = etop
   cq = 0
'
' Evaluate conditional probability of events and causes
' connected to alarmed event by DIR gate.
'
   proceed = 1
   WHILE proceed
      GET #1, gtop
      LSET cpr$ = MKS$(1.0)
      PUT #1, gtop
```

```
        done(gtop) = 1
'
' Evaluate conditional probabilities of consequences of event.
'
        GET #1, gtop
        IF CVI(bq$) > cq THEN
            ctop = gtop
            zcal = 1.0
            CALL REPROB3(ctop, zcal)
        END IF
'
' Determine if event node is a "DIR" gate.  If so set gtop to
' child of event.
'
        GET #1, gtop
        IF g$ = "DIR" THEN
            gtop = CVI(cau$(1))
            cq = 1
        ELSE
            proceed = 0
        END IF
    WEND 'proceed
'
' Check if gtop is a primary node. If not, store the gate
' type of gtop in gt$.
'
    GET #1, gtop
    IF NOT(g$ = "PRI") THEN
        gt$ = g$
'
' Store the addresses of the branches of gtop in comset().
'
        comset(1, 1) = 0
        CALL SIBLING(1, 2, gtop, comset())
'
' Evaluate the posteriori probabilities of every node in comset.
' The posteriori probabilities of the branches of the nodes in comset
' which are of gate type gt$ are also evaluated.
'
        CALL REPROB1(gt$)
'
' After returning from REPROB1, comset will contain nodes of gate
' types different from gt$.
' Replace the nodes in evset by their branches.
'
        stot = comset(1, 1) + 1
        smem = 2
        replc = 0
        WHILE smem <= stot
            GET #1, comset(1, smem)
            IF NOT(g$ = "PRI") THEN
                replc = 1
                CALL SIBLING(1, smem, comset(1, smem), comset())
            END IF
            smem = smem + 1
        WEND 'smem
'
' If there are replacements of nodes in comset, remove any repetition
' of nodes in comset and evaluate their posteriori probabilities.
'
        IF replc = 1 THEN
            CALL REMREP(1, comset())
            CALL REPROB2
        END IF
    END IF
'
' Evaluate the posteriori probabilities of other events that have
' no relation with gtop.
'
    ka = 1
    WHILE ka <= p
        IF NOT(done(ka) = 1) THEN
            GET #1, ka
            LSET cpr$ = MKS$(CVS(pr$))
            PUT #1, ka
            done(ka) = 1
        END IF
        ka = ka + 1
    WEND 'ka
'
    END SUB
'
'=================================================================='
' Subroutine to evaluate conditional probabilities of nodes        '
' in evset having gate type gt$                                    '
'=================================================================='
    SUB REPROB1(gt$) STATIC
'
    reptflg = 1
    WHILE reptflg
        ka = 2
        WHILE ka <= comset(1, 1) + 1
            IF NOT(done(comset(1, ka)) = 1) THEN
                GET #1, comset(1, ka)
                IF gt$ = "AND" THEN
                    LSET cpr$ = MKS$(1)
                ELSE
                    LSET cpr$ = MKS$(CVS(pr$)/zalrm)
                END IF
                PUT #1, comset(1, ka)
'
                GET #1, comset(1, ka)
                IF (CVI(bq$) > 1) THEN
                    ctop = comset(1, ka)
                    zcal = CVS(cpr$)
                    CALL REPROB3(ctop, zcal)
                END IF
                done(comset(1, ka)) = 1
            END IF
            ka = ka + 1
        WEND 'ka
'
        stot = comset(1, 1) + 1
        replc = 0
        smem = 2
        WHILE smem <= stot
            GET #1, comset(1, smem)
            IF (g$ = gt$) OR (g$ = "DIR") THEN
                replc = 1
                CALL SIBLING(1, smem, comset(1, smem), comset())
            END IF
            smem = smem + 1
        WEND 'smem
'
        IF (replc = 1) THEN
            CALL REMREP(1, comset())
        ELSE
            reptflg = 0
        END IF
    WEND 'reptflg
'
    END SUB
'
'==========================================================================='
' Subroutine to evaluate conditional probabilities of nodes in             '
' comset having gate types different from gt$                              '
'==========================================================================='
    SUB REPROB2 STATIC
'
    reptflg = 1
    WHILE reptflg
        ka = 2
        WHILE ka <= comset(1, 1) + 1
            IF NOT(done(comset(1, ka)) = 1) THEN
                indx = 1
                s1(indx, 1) = 0
                s2(indx, 1) = 2
                s2(indx, 2) = etop
                s2(indx, 3) = comset(1, ka)
                CALL PPROB(indx, zpv)
                GET #1, comset(1, ka)
                LSET cpr$ = MKS$(zpv/zalrm)
                PUT #1, comset(1, ka)
'
                GET #1, comset(1, ka)
                IF (CVI(bq$) > 1) THEN
                    ctop = comset(1, ka)
                    zcal = CVS(cpr$)
                    CALL REPROB3(ctop, zcal)
                END IF
                done(comset(1, ka)) = 1
            END IF
            ka = ka + 1
        WEND 'ka
'
        stot = comset(1, 1) + 1
        replc = 0
        smem = 2
        WHILE smem <= stot
            GET #1, comset(1, smem)
            IF NOT(g$ = "PRI") THEN
                replc = 1
                CALL SIBLING(1, smem, comset(1, smem), comset())
            END IF
            smem = smem + 1
        WEND 'smem
'
        IF (replc = 1) THEN
            CALL REMREP(1, comset())
        ELSE
            reptflg = 0
        END IF
    WEND 'reptflg
    END SUB
'
```

```
'==============================================================='
' Subroutine to evaluate conditional probabilities of           '
' consequences of an event.                                     '
'==============================================================='
   SUB REPROB3 (ctop, zcal) STATIC
'
' Obtain consequence of event.
'
   comset(2, 1) = 0
   CALL PARENT (2, 2, ctop, comset())
'
' Evaluate conditional probability of each node in comset.
'
   CALL REPROB4(2, zcal)
'
   kb = 2
   WHILE kb <= comset(2, 1) + 1
       IF (done(comset(2, kb)) = 1) THEN
           kb = kb + 1
       ELSE
           GET #1, comset(2, kb)
           IF (CVI(bq$) = 0) THEN
               done(comset(2, kb)) = 1
               kb = kb + 1
           ELSE
               ctop = comset(2, kb)
               zcal = CVS(cpr$)
               comset(3, 1) = 0
               CALL PARENT(3, 2, ctop, comset())
               CALL REPROB4(3, zcal)
               done(comset(2, kb)) = 1
               comset(2, kb) = comset(3, 2)
               kc = 3
               FOR kd = comset(2, 1)+2 TO comset(2, 1)+comset(3, 1)
                   comset(2, kd) = comset(3, kc)
                   kc = kc + 1
               NEXT kd
               comset(2, 1) = comset(2, 1) + comset(3, 1) - 1
               CALL REMREP(2, comset())
           END IF
       END IF
   WEND 'kb
'
   END SUB
'
'==============================================================='
' Subroutine to evaluate conditional probability of nodes in    '
' row ip of comset.                                             '
'==============================================================='
   SUB REPROB4(ip, zcal) STATIC
'
   kc = 2
   WHILE kc <= comset(ip, 1) + 1
       IF NOT(done(comset(ip, kc)) = 1) THEN
           GET #1, comset(ip, kc)
           IF (g$ = "DIR") OR (LEFT$(g$, 2) = "OR" AND zcal = 1) THEN
               LSET cpr$ = MKS$(zcal)
               PUT #1, comset(ip, kc)
           ELSE
               indx = 1
               s1(indx, 1) = 0
               s2(indx, 1) = 2
               s2(indx, 2) = comset(ip, kc)
```

```
            s2(indx, 3) = etop
            CALL PPROB(indx, zpv)
            GET #1, comset(ip, kc)
            LSET cpr$ = MKS$(zpv/zalrm)
            PUT #1, comset(ip, kc)
         END IF
      END IF
      kc = kc + 1
   WEND 'kc
'
   END SUB
'
'==============================================================='
' Subroutines to handle probability calculations.               '
'==============================================================='
   SUB PPROB(indx, zpv) STATIC
'
' Priliminary reduction when s1() not empty.
'
   IF NOT(s1(indx, 1) = 0) THEN
'
' Case P.1 : Evaluation for s1() intersection s2() non empty.
'            If this is true, then set s1() empty and s2()
'            as before.
'
      CALL INTERSEC(indx, true)
      IF true = 1 THEN
         s1(indx, 1) = 0
'
' Case P.2 : Evaluation for s1() containing a single node.
'            Add the node in s1() to s2() and set s1() empty.
'
      ELSEIF s1(indx, 1) = 1 THEN
         s2(indx, 1) = s2(indx, 1) + 1
         FOR ka = s2(indx, 1) + 1 TO 3 STEP -1
            s2(indx, ka) = s2(indx, ka-1)
         NEXT ka
         s2(indx, 2) = s1(indx, 2)
         s1(indx, 1) = 0
      END IF
   END IF
'
' Case 1 : Evaluation for s1() empty.
'
   IF s1(indx, 1) = 0 THEN
'
' Case 1.1 : PPROB returns zpv = 1 if s2() is empty.
'
      IF s2(indx, 1) = 0 THEN
         zpv = 1
'
' Case 1.2 : Evaluation for s2() containing one node.
'
      ELSEIF s2(indx, 1) = 1 THEN
         GET #1, s2(indx, 2)
         zpv = CVS(pr$)
'
'
' Case 1.3 : Evaluation for case of s2() containing more than
'            one node. First set ztemp = 1. Then determine if
'            there is a node in s2() which is disjoint from the
'            other nodes in s2().
'
```

```
        ELSE
            ztemp(indx) = 1
            andgate(indx) = 1
            WHILE andgate(indx)
                CALL INDEPT(indx, disjt, indevt)
'
' Case 1.3.1 : Evaluation for case where a member in s2() is found
'              found to be disjoint from the other members in s2().
'
                IF disjt = 1 THEN
                    GET #1, indevt
                    ztemp(indx) = ztemp(indx) * CVS(pr$)
'
' Remove evset(indx, 2) from s2(). If s2() is not empty, repeat finding
' an s-independent node in s2().
'
                    FOR ka = 2 TO s2(indx, 1) + 1
                        IF s2(indx, ka) = indevt THEN
                            IF NOT(ka = s2(indx, 1) + 1) THEN
                                FOR kb = ka TO s2(indx, 1)
                                    s2(indx, kb) = s2(indx, kb+1)
                                NEXT kb
                                ka = s2(indx, 1) + 1
                            END IF
                            s2(indx, s2(indx, 1)+1) = 0
                            s2(indx, 1) = s2(indx, 1) - 1
                        END IF
                    NEXT ka
'
' If s2() is equal to 1, get the probability of the node in s2() and
' multiply by ztemp(indx).
'
                    IF s2(indx, 1) = 1 THEN
                        GET #1, s2(indx, 2)
                        zpv = ztemp(indx) * CVS(pr$)
                        andgate(indx) = 0
                    END IF
'
' Case 1.3.2 : Evaluation for case where there is no node found
'              to be disjoint from the other members in s2().
'
                ELSE
'
' Find all AND gate nodes in s2() and replace the nodes in s2()
' with their branches.
'
                    replc = 0
                    stot = s2(indx, 1) + 1
                    smem = 2
                    WHILE smem <= stot
                        GET #1, s2(indx, smem)
                        IF (g$ = "DIR") OR (g$ = "AND") THEN
                            replc = 1
                            CALL SIBLING(indx, smem, s2(indx, smem), s2())
                        END IF
                        smem = smem + 1
                    WEND 'smem
'
' Case 1.3.2.1 : Evaluation for case when replacements are made.
'                Repetition of nodes is removed. Then determine if
'                there is a node disjoint from the rest of the nodes
'                in s2().
'
```

```
                    IF replc = 1 THEN
                        CALL REMREP(indx, s2())
                        IF s2(indx, 1) = 1 THEN
                            GET #1, s2(indx, 2)
                            zpv = ztemp(indx) * CVS(pr$)
                            andgate(indx) = 0
                        END IF
'
' Case 1.3.2.2 : Evaluation for case of no AND gate found in s2().
'                An OR gate node m in s2() is picked and probability
'                is evaluated via call to PPROB with s1() containing
'                the children of m and s2() containing nodes without m.
'
                    ELSE
                        FOR ka = 2 TO s2(indx, 1)+1
                            GET #1, s2(indx, ka)
                            IF LEFT$(g$, 2) = "OR" THEN
                                ornode = s2(indx, ka)
                                ka = s2(indx, 1)+1
                            END IF
                        NEXT ka
                        indx = indx + 1
                        s1(indx, 1) = 0
                        CALL SIBLING(indx, 2, ornode, s1())
                        ka = 2
                        FOR kb = 2 TO s2(indx-1, 1)+1
                            IF NOT(s2(indx-1, kb) = ornode) THEN
                                s2(indx, ka) = s2(indx-1, kb)
                                ka = ka + 1
                            END IF
                        NEXT kb
                        s2(indx, 1) = s2(indx-1, 1) - 1
                        CALL PPROB(indx, zp(indx))
                        indx = indx - 1
                        zpv = ztemp(indx) * zp(indx + 1)
                        andgate(indx) = 0
                    END IF
                END IF
            WEND 'andgate(indx)
        END IF
'
' Case 2 : Evaluation for s1() non empty. First initialise evset() so
'          that it is empty. Then determine if there is a node in s1()
'          which is disjoint from the other nodes of s1() as well as s2().
'
    ELSE
        evset(indx, 1) = 0
        andgate(indx) = 1
        WHILE andgate(indx)
            CALL INDEPTOR(indx, disjt, indevt)
'
' Case 2.1 : Case where a node in s1() is found to be disjoint
'            from other nodes in s1() as well as from all nodes
'            in s2(). Save the disjoint node in evset().
'
            IF disjt = 1 THEN
                evset(indx, 1) = evset(indx, 1) + 1
                evset(indx, evset(indx, 1)+1) = indevt
'
' Remove the disjoint node from s1.
'

                FOR ka = 2 TO s1(indx, 1) + 1
                IF s1(indx, ka) = indevt THEN
                    IF NOT(ka = s1(indx, 1) + 1) THEN
                        FOR kb = ka TO s1(indx, 1)
                            s1(indx, kb) = s1(indx, kb+1)
                        NEXT kb
                        ka = s1(indx, 1) + 1
                    END IF
                    s1(indx, s1(indx, 1)+1) = 0
                    s1(indx, 1) = s1(indx, 1) - 1
                END IF
            NEXT ka
'
            IF s1(indx, 1) = 1 THEN
                andgate(indx) = 0
            END IF
'
' Case 2.2 : Evaluation for case of no nodes in s1() disjoint in s1()
'            as well as in s2().  Find all OR and DIR nodes in s1() and
'            replace the nodes in s1 with their branches.
'
        ELSE
            replc = 0
            stot = s1(indx, 1) + 1
            smem = 2
            WHILE smem <= stot
                GET #1, s1(indx, smem)
                IF (LEFT$(g$, 2) = "OR") OR (g$ = "DIR") THEN
                    replc = 1
                    CALL SIBLING(indx, smem, s1(indx, smem), s1())
                END IF
                smem = smem + 1
            WEND 'smem
'
' Case 2.2.1 : Evaluation for replacements made in s1().  Determine if
'              there is a node in s1() which is disjoint from the other
'              nodes in s1() as well as disjoint with all the nodes in s2().
'
            IF replc = 1 THEN
                CALL REMREP(indx, s1())
                IF s1(indx, 1) = 1 THEN
                    andgate(indx) = 0
                END IF
            ELSE
                andgate(indx) = 0
            END IF
        END IF
    WEND 'andgate(indx)
'
' Case 2.1a : Evaluation for case of disjoint nodes found in s1().
'
    IF NOT(evset(indx, 1) = 0) THEN
        indx = indx + 1
        s1(indx, 1) = s1(indx-1, 1)
        FOR ka = 2 TO s1(indx, 1) + 1
            s1(indx, ka) = s1(indx-1, ka)
        NEXT ka
        s2(indx, 1) = s2(indx-1, 1)
        FOR ka = 2 TO s2(indx-1, 1) + 1
            s2(indx, ka) = s2(indx-1, ka)
        NEXT ka
        CALL PPROB(indx, zp(indx))
        indx = indx - 1
        ztemp(indx) = zp(indx+1)
```

```
'
          indx = indx + 1
          s1(indx, 1) = 0
          s2(indx, 1) = s2(indx-1, 1)
          FOR ka = 2 TO s2(indx, 1) + 1
              s2(indx, ka) = s2(indx-1, ka)
          NEXT ka
          CALL PPROB(indx, zp(indx))
          indx = indx - 1
          zcase(indx) = zp(indx+1)
'
          andflg(indx) = 2
          WHILE andflg(indx) <= evset(indx, 1) + 1
              GET #1, evset(indx, andflg(indx))
              zpr = CVS(pr$)
              ztemp(indx) = zpr * (zcase(indx) - ztemp(indx)) + ztemp(indx)
              andflg(indx) = andflg(indx) + 1
          WEND 'andflg(indx)
          zpv = ztemp(indx)
'
' Case 2.2.2 : Evaluation for case of no disjoint nodes found in s1()
'              even if replacements are made in s1().
'              A node is picked from s1() and saved in orgate(indx).
'
      ELSE
          orgate(indx) = s1(indx, 2)
'
          indx = indx + 1
          s1(indx, 1) = 0
          FOR ka = 2 TO s2(indx-1, 1)+1
              s2(indx, ka) = s2(indx-1, ka)
          NEXT ka
          s2(indx, 1) = s2(indx-1, 1) + 1
          s2(indx, s2(indx, 1)+1) = orgate(indx-1)
          CALL PPROB(indx, zp(indx))
          indx = indx - 1
          ztemp(indx) = zp(indx+1)
'
          indx = indx + 1
          ka = 2
          FOR kb = 2 TO s1(indx-1, 1)+1
              IF NOT(s1(indx-1, kb) = orgate(indx-1)) THEN
                  s1(indx, ka) = s1(indx-1, kb)
                  ka = ka + 1
              END IF
          NEXT kb
          s1(indx, 1) = s1(indx-1, 1) - 1
          FOR ka = 2 TO s2(indx-1, 1)+1
              s2(indx, ka) = s2(indx-1, ka)
          NEXT ka
          s2(indx, 1) = s2(indx-1, 1)
          CALL PPROB(indx, zp(indx))
          indx = indx - 1
          ztemp(indx) = ztemp(indx) + zp(indx+1)
'
          indx = indx + 1
          ka = 2
          FOR kb = 2 TO s1(indx-1, 1)+1
              IF NOT(s1(indx-1, kb) = orgate(indx-1)) THEN
                  s1(indx, ka) = s1(indx-1, kb)
                  ka = ka + 1
              END IF
          NEXT kb
          s1(indx, 1) = s1(indx-1, 1) - 1
          FOR ka = 2 TO s2(indx-1, 1)+1
              s2(indx, ka) = s2(indx-1, ka)
          NEXT ka
          s2(indx, 1) = s2(indx-1, 1) + 1
          s2(indx, s2(indx, 1)+1) = orgate(indx-1)
          CALL PPROB(indx, zp(indx))
          indx = indx - 1
          zpv = ztemp(indx) - zp(indx+1)
      END IF
  END IF
  END SUB
'
'==================================================================='
' Subroutine to remove events repeated in set s().                  '
'==================================================================='
  SUB REMREP(ndx, set(2)) STATIC
'
  ka = 2
  WHILE ka <= set(ndx, 1)
      kb = ka+1
      WHILE kb <= set(ndx, 1)+1
          remvd = 0
          IF set(ndx, ka) = set(ndx, kb) THEN
              FOR kc = kb TO set(ndx, 1)
                  set(ndx, kc) = set(ndx, kc+1)
              NEXT kc
              set(ndx, 1) = set(ndx, 1) - 1
              remvd = 1
          END IF
          IF remvd = 0 THEN
              kb = kb + 1
          END IF
      WEND 'kb
      ka = ka + 1
  WEND 'ka
'
  END SUB
'
'==================================================================='
'  Subroutine to find an independent OR gate in s().                '
'==================================================================='
  SUB INDEPTOR(ndx, disjt, indevt) STATIC
'
  FOR ka = 2 TO s1(ndx, 1)+1
      GET #2, s1(ndx, ka)
      kz = CVI(nu$)
      disjt = 1
      FOR kb = 2 TO s1(ndx, 1)+1
          IF NOT(kb = ka) THEN
              GET #2, s1(ndx, kb)
              ky = CVI(nu$)
              FOR kc = 1 TO kz
                  GET #2, s1(ndx, ka)
                  kx = CVI(pri$(kc))
                  GET #2, s1(ndx, kb)
                  FOR kd = 1 TO ky
                      IF kx = CVI(pri$(kd)) THEN
                          disjt = 0
                          kd = ky
                          kc = kz
                      END IF
                  NEXT kd
```

```
            NEXT kc
         END IF
         IF disjt = 0 THEN kb = s1(ndx, 1)+1
      NEXT kb
      IF disjt = 1 THEN
         indevt = s1(ndx, ka)
         FOR kb = 2 TO s2(ndx, 1)+1
            GET #2, s2(ndx, kb)
            ky = CVI(nu$)
            FOR kc = 1 TO ky
               GET #2, s2(ndx, kb)
               kx = CVI(pri$(kc))
               GET #2, indevt
               FOR kd = 1 TO kz
                  IF kx = CVI(pri$(kd)) THEN
                     disjt = 0
                     kd = kz
                     kc = ky
                  END IF
               NEXT kd
            NEXT kc
            IF disjt = 0 THEN kb = s2(ndx, 1)+1
         NEXT kb
      END IF
      IF disjt = 1 THEN ka = s1(ndx, 1)+1
   NEXT ka
   END SUB
'
'==================================================================='
' Subroutine to determine if s1() intersection s2() is non empty.    '
'==================================================================='
   SUB INTERSEC(ndx, true) STATIC
'
   true = 0
   FOR ka = 2 TO s2(ndx, 1)+1
      FOR kb = 2 TO s1(ndx, 1)+1
         IF s2(ndx, ka) = s1(ndx, kb) THEN
            true = 1
            kb = s1(ndx, 1)+1
            ka = s2(ndx, 1)+1
         END IF
      NEXT kb
   NEXT ka
   END SUB
'
'==================================================================='
' Subroutine to replace a member of a set with its children.         '
'==================================================================='
   SUB SIBLING(ndx, setpos, evadd, set(2)) STATIC
'
   dtemp = evadd
   GET #1, dtemp
   set(ndx, setpos) = CVI(cau$(1))
   IF set(ndx, 1) = 0 THEN
      set(ndx, 1) = 1
   END IF
   repeat = 1
   WHILE repeat
      GET #1, dtemp
      IF CVI(bc$) => 4 THEN
         db = 4
      ELSE
         db = CVI(bc$)
```

```
        END IF
        FOR dc = 1 TO db
            IF NOT(CVI(cau$(dc)) = set(ndx, setpos)) THEN
                set(ndx, 1) = set(ndx, 1) + 1
                set(ndx, set(ndx, 1)+1) = CVI(cau$(dc))
            END IF
        NEXT dc
        IF CVI(bc$) < 4 THEN
            repeat = 0
        ELSE
            dtemp = CVI(cau$(4))
            GET #1, dtemp
            IF (LEFT$(e$, 1) = "&") THEN
                set(ndx, 1) = set(ndx, 1) - 1
            ELSE
                repeat = 0
            END IF
        END IF
    WEND 'repeat
    END SUB
'
'==============================================================='
' Subroutine to get consequnces of an event.                    '
'==============================================================='
    SUB PARENT (ndx, setpos, evadd, set(2)) STATIC
'
    dtemp = evadd
    GET #1, dtemp
    IF NOT(CVI(bq$) = 0) THEN
        set(ndx, setpos) = CVI(con$(1))
        IF set(ndx, 1) = 0 THEN
            set(ndx, 1) = 1
        END IF
        true = 1
        WHILE true
            GET #1, dtemp
            IF CVI(bq$) => 4 THEN
                ka = 4
            ELSE
                ka = CVI(bq$)
            END IF
            FOR kb = 1 TO ka
                IF NOT(CVI(con$(kb)) = set(ndx, setpos)) THEN
                    set(ndx, 1) = set(ndx, 1) + 1
                    set(ndx, set(ndx, 1)+1) = CVI(con$(kb))
                END IF
            NEXT kb
            IF CVI(bq$) < 4 THEN
                true = 0
            ELSE
                dtemp = CVI(con$(4))
                GET #1, dtemp
                IF (LEFT$(e$, 1) = "@") THEN
                    set(ndx, 1) = set(ndx, 1) - 1
                ELSE
                    true = 0
                END IF
            END IF
        WEND 'true
    END IF
    END SUB
'
```

```
'================================================================'
' Subroutine to find an independent event is s2().               '
'================================================================'
    SUB INDEPT(indx, disjt, indevt) STATIC
'
    FOR ka = 2 TO s2(indx, 1)+1
        GET #2, s2(indx, ka)
        kz = CVI(nu$)
        disjt = 1
        FOR kb = 2 TO s2(indx, 1)+1
            IF NOT(kb = ka) THEN
                GET #2, s2(indx, kb)
                ky = CVI(nu$)
                FOR kc = 1 TO kz
                    GET #2, s2(indx, ka)
                    kx = CVI(pri$(kc))
                    GET #2, s2(indx, kb)
                    FOR kd = 1 TO ky
                        IF kx = CVI(pri$(kd)) THEN
                            disjt = 0
                            kd = ky
                            kc = kz
                        END IF
                    NEXT kd
                NEXT kc
                IF disjt = 0 THEN kb = s2(indx, 1) + 1
            END IF
        NEXT kb
        IF disjt = 1 THEN
            indevt = s2(indx, ka)
            ka = s2(indx, 1)+1
        END IF
    NEXT ka
    END SUB
'
'================================================================'
' Subroutine to get meaning of coded event.                      '
'================================================================'
    SUB MEANING (evt$, mean$) STATIC
'
    ka = INSTR(evt$, "(")
    IF NOT(ka = 0) THEN
        item$ = LEFT$(evt$, ka-1)
'
' Get item name
'
        IF MID$(item$, 1, 6) = "C1COIL" THEN
            sen4$ = "COOLING COIL IN CONDENSER C1 "
        ELSEIF MID$(item$, 1, 6) = "C1VENT" THEN
            sen4$ = "VENT OF CONDENSER C1 "
        ELSEIF MID$(item$, 1, 4) = "TRAY" THEN
            sen4$ = item$
        ELSEIF MID$(item$, 1, 3) = "FCL" THEN
            sen4$ = "FLOW CONTROL LOOP " + item$
        ELSEIF MID$(item$, 1, 3) = "LCL" THEN
            sen4$ = "LEVEL CONTROL LOOP " + item$
        ELSEIF MID$(item$, 1, 3) = "TCL" THEN
            sen4$ = "TEMPERATURE CONTROL LOOP " + item$
        ELSEIF MID$(item$, 1, 3) = "RBH" THEN
            sen4$ = "REBOILER HEATER " + item$
        ELSEIF MID$(item$, 1, 2) = "TC" THEN
            sen4$ = "TEMPERATURE CONTROLLER " + item$
        ELSEIF MID$(item$, 1, 2) = "LC" THEN
        sen4$ = "LEVEL CONTROLLER " + item$
    ELSEIF MID$(item$, 1, 2) = "RB" THEN
        sen4$ = "REBOILER DRUM " + item$
    ELSEIF MID$(item$, 1, 2) = "CV" THEN
        sen4$ = "CONTROL VALVE " + item$
    ELSEIF MID$(item$, 1, 2) = "FI" THEN
        sen4$ = "FLOW INDICATOR " + item$
    ELSEIF MID$(item$, 1, 2) = "LI" THEN
        sen4$ = "LEVEL INDICATOR " + item$
    ELSEIF MID$(item$, 1, 2) = "TI" THEN
        sen4$ = "TEMPERATURE INDICATOR " + item$
    ELSEIF MID$(item$, 1, 2) = "PI" THEN
        sen4$ = "PRESSURE INDICATOR " + item$
    ELSEIF MID$(item$, 1, 2) = "HE" THEN
        sen4$ = "HEAT EXCHANGER " + item$
    ELSEIF LEFT$(item$, 1) = "L" THEN
        sen4$ = "LINE " + item$
    ELSEIF LEFT$(item$, 1) = "N" THEN
        sen4$ = "NODE " + item$
    ELSEIF LEFT$(item$, 1) = "V" THEN
        sen4$ = "VALVE " + item$
    ELSEIF LEFT$(item$, 1) = "T" THEN
        sen4$ = "TANK " + item$
    ELSEIF LEFT$(item$, 1) = "P" THEN
        sen4$ = "PUMP " + item$
    ELSEIF LEFT$(item$, 1) = "C" THEN
        sen4$ = "CONDENSER " + item$
    ELSEIF LEFT$(item$, 1) = "D" THEN
        sen4$ = "DISTILLATION COLUMN " + item$
    ELSE
        sen4$ = "VESSEL " + item$
    END IF
'
    kb = INSTR(evt$, ")")
    code$ = MID$(evt$, ka+1, kb-ka-1)
    IF NOT((LEN(code$) = 1) OR (LEFT$(code$, 1) = "-")) AND_
          (ASC(LEFT$(code$, 1)) < 58) THEN
'
' Get the Property Word from index code.
'
        index$ = MID$(code$, 1, 1)
        IF index$ = "1" THEN
            sen2$ = "FLOW"
        ELSEIF index$ = "2" THEN
            sen2$ = "TEMPERATURE"
        ELSEIF index$ = "3" THEN
            sen2$ = "PRESSURE"
        ELSEIF index$ = "4" THEN
            sen2$ = "LEVEL"
        ELSEIF index$ = "5" THEN
            sen2$ = "CONCENTRATION"
        ELSEIF index$ = "6" THEN
            sen2$ = "PURGE"
        ELSEIF index$ = "7" THEN
            sen2$ = "HEAT TRANSFER"
        ELSEIF index$ = "8" THEN
            sen2$ = "REACTION"
        ELSE
            sen2$ = ""
        END IF
'
' Get the Guide Word from the index code.
'
```

```
        index$ = MID$(code$, 2, 1)
        IF index$ = "1" THEN
            sen1$ = "NO"
        ELSEIF index$ = "2" THEN
            sen1$ = "LESS than normal"
        ELSEIF index$ = "3" THEN
            sen1$ = "MORE than normal"
        ELSEIF index$ = "4" THEN
            sen1$ = "AS WELL AS"
        ELSEIF index$ = "5" THEN
            sen1$ = "FLUCTUATION"
        ELSEIF index$ = "6" THEN
            sen1$ = "REVERSE"
        ELSEIF index$ = "7" THEN
            sen1$ = "OTHER THAN"
        ELSE
            sen1$ = ""
        END IF
'
' Get the Component Word from the index code.
'
        IF LEN(code$) = 3 THEN
            index$ = MID$(code$, 3, 1)
            IF index$ = "1" THEN
                sen3$ = "of TRICHLOROETHYLENE"
            ELSEIF index$ = "2" THEN
            ELSEIF index$ = "2" THEN
                sen3$ = "of TETRACHLOROETHYLENE"
            ELSEIF index$ = "3" THEN
                sen3$ = "of WATER"
            ELSEIF index$ = "4" THEN
                sen3$ = "of AIR"
            ELSEIF index$ = "5" THEN
                sen3$ = "of BOTTOMS LIQUID"
            ELSEIF index$ = "6" THEN
                sen3$ = "of TOP VAPOUR"
            ELSE
                sen3$ = ""
            END IF
        ELSE
            sen3$ = ""
        END IF
        mean$ = sen1$ + " " + sen2$ + " " + sen3$ + " in " + sen4$
    ELSE
        IF MID$(code$, 1, 1) = "L" THEN
            sen1$ = "LEAKING"
        ELSEIF MID$(code$, 1, 1) = "F" THEN
            sen1$ = "FOULING"
        ELSEIF MID$(code$, 1, 2) = "BO" THEN
            sen1$ = "BURN OUT"
        ELSEIF MID$(code$, 1, 2) = "OF" THEN
            sen1$ = "OVERFLOW"
        ELSEIF MID$(code$, 1, 2) = "VR" THEN
            sen1$ = "VAPOUR RELEASE"
        ELSE
            IF MID$(item$, 1, 6) = "C1COIL" THEN
                IF MID$(code$, 1, 1) = "0" THEN
                    sen1$ = "FULLY BLOCKED"
                ELSEIF MID$(code$, 1, 1) = "-" THEN
                    sen1$ = "PARTLY BLOCKED"
                END IF
            ELSEIF MID$(item$, 1, 6) = "C1VENT" THEN
                IF MID$(code$, 1, 1) = "0" THEN
        sen1$ = "BLOCKED"
    END IF
ELSEIF MID$(item$, 1, 3) = "FCL" THEN
    IF MID$(code$, 1, 1) = "0" THEN
        sen1$ = "giving NO FLOW"
    ELSEIF MID$(code$, 1, 1) = "-" THEN
        sen1$ = "giving LESS than normal FLOW"
    ELSEIF MID$(code$, 1, 1) = "1" THEN
        sen1$ = "giving MORE than normal FLOW"
    END IF
ELSEIF MID$(item$, 1, 3) = "LCL" THEN
    IF MID$(code$, 1, 1) = "0" THEN
        sen1$ = "giving NO LEVEL"
    ELSEIF MID$(code$, 1, 1) = "-" THEN
        sen1$ = "giving LESS than normal LEVEL"
    ELSEIF MID$(code$, 1, 1) = "1" THEN
        sen1$ = "giving MORE than normal LEVEL"
    END IF
ELSEIF MID$(item$, 1, 3) = "TCL" THEN
    IF MID$(code$, 1, 1) = "0" THEN
        sen1$ = "acts to give NO HEAT"
    ELSEIF MID$(code$, 1, 1) = "-" THEN
        sen1$="acts to give LESS than normal TEMPERATURE"
    ELSEIF MID$(code$, 1, 1) = "1" THEN
        sen1$="acts to give MORE than normal TEMPERATURE"
    END IF
ELSEIF MID$(item$, 1, 2) = "FI" THEN
    IF MID$(code$, 1, 1) = "0" THEN
        sen1$ = "indicating NO FLOW"
    ELSEIF MID$(code$, 1, 1) = "1" THEN
        sen1$ = "indicating MORE than normal FLOW"
    ELSEIF MID$(code$, 1, 1) = "-" THEN
        sen1$ = "indicating LESS than normal FLOW"
    END IF
ELSEIF MID$(item$, 1, 2) = "TI" THEN
    IF MID$(code$, 1, 1) = "1" THEN
        sen1$="indicating MORE than normal TEMPERATURE"
    ELSEIF MID$(code$, 1, 1) = "-" THEN
        sen1$="indicating LESS than normal TEMPERATURE"
    END IF
ELSEIF MID$(item$, 1, 2) = "LI" THEN
    IF MID$(code$, 1, 1) = "1" THEN
        sen1$ = "indicating MORE than normal LEVEL"
    ELSEIF MID$(code$, 1, 1) = "-" THEN
        sen1$ = "indicating LESS than normal LEVEL"
    END IF
ELSEIF (MID$(item$, 1, 2) = "TC") OR_
       (MID$(item$, 1, 2) = "LC") THEN
    IF MID$(code$, 1, 1) = "1" THEN
        sen1$ = "set point HIGH"
    ELSEIF MID$(code$, 1, 1) = "-" THEN
        sen1$ = "set point LOW"
    END IF
ELSEIF (MID$(item$, 1, 2)) = "CV" THEN
    IF MID$(code$, 1, 1) = "0" THEN
        sen1$ = "stuck CLOSED"
    ELSEIF MID$(code$, 1, 1) = "-" THEN
        sen1$ = "insufficiently OPEN"
    ELSEIF MID$(code$, 1, 1) = "1" THEN
        sen1$ = "stuck OPEN"
    END IF
ELSEIF MID$(item$, 1, 1) = "L" THEN
    IF MID$(code$, 1, 1) = "0" THEN
```

```
                sen1$ = "FULLY BLOCKED"
            ELSEIF MID$(code$, 1, 1) = "-" THEN
                sen1$ = "PARTLY BLOCKED"
            END IF
        ELSEIF MID$(item$, 1, 1) = "P" THEN
            IF MID$(code$, 1, 1) = "0" THEN
                sen1$ = "STOPPED"
            ELSEIF MID$(code$, 1, 1) = "-" THEN
                sen1$ = "CAVITATING"
            ELSEIF MID$(code$, 1, 1) = "1" THEN
                sen1$ = "RUNNING"
            END IF
        ELSEIF MID$(Item$, 1, 1) = "V" THEN
            IF MID$(code$, 1, 1) = "0" THEN
                sen1$ = "CLOSED or BLOCKED"
            ELSEIF MID$(code$, 1, 1) = "-" THEN
                sen1$ = "INSUFFICIENTLY OPEN"
            ELSEIF MID$(code$, 1, 1) = "1" THEN
                sen1$ = "OPEN or OPEN TOO MUCH"
            END IF
        ELSE
            sen1$ = ""
        END IF
    END IF
    mean$ = sen4$ + " " + sen1$
  END IF
ELSE
  mean$ = evt$
END IF
END SUB
```

APPENDIX G

CONTENT OF THE FAULT TREE DATA FILE HAZOP.COD

Table G.1 below shows the content of the file **HAZOP.COD** which is produced by the program called **TRANSLAT** which operates on the file **HAZOP.TXT** input by the user. **HAZOP.TXT** contains the list of cause and symptom equations obtained from the HAZOP study of the pilot distillation plant. The *a priori* and *posteriori* probabilities are not shown in the table.

Nomenclature

ADDR	: the record number in the data file.
EVENT	: the field **e$** containing the name of each event.
GATE	: the field **g$** containing the 3 character string which describes the type of event stored in **e$**.
BC	: the field **bc$** containing the number of branches connected to the event stored in **e$**.
C1 - C4	: the fields **cau$(1)** to **cau$(4)** containing the record numbers of events which are causes of the event stored in **e$**.
BQ	: the field **bq$** containing the number of consequences of the event stored in **e$**.
Q1 - Q2	: the fields **con$(1)** to **con$(4)** containing the record numbers of events which are consequences of the event stored in **e$**.

Table G.1 : Content of the file HAZOP.COD

ADDR	EVENT	GATE	BC	C1	C2	C3	C4	BQ	Q1	Q2	Q3	Q4
1	L3(11)	ORC	5	2	3	5	9	3	33	34	35	0
2	L1(11)	ORC	3	204	205	206	0	1	1	0	0	0
3	V16(0)	PRI	0	0	0	0	0	1	1	0	0	0
4	P1(0)	PRI	0	0	0	0	0	3	5	10	27	0
5	# 1	AND	2	4	6	0	0	1	1	0	0	0
6	V6(0)	PRI	0	0	0	0	0	1	5	0	0	0
7	L3(0)	PRI	0	0	0	0	0	1	1	0	0	0
8	FI1(1)	PRI	0	0	0	0	0	2	1	10	0	0
9	& 2	CON	2	7	8	0	0	0	0	0	0	0

Table G.1.a : continuation of table G.1

ADDR	EVENT	GATE	BC	C1	C2	C3	C4	BQ	Q1	Q2	Q3	Q4
10	L3(12)	ORC	12	11	4	12	15	3	36	34	37	0
11	L1(12)	ORC	3	207	208	209	0	1	10	0	0	0
12	P1(-1)	PRI	0	0	0	0	0	2	10	29	0	0
13	P1(L)	PRI	0	0	0	0	0	1	10	0	0	0
14	L3(L)	PRI	0	0	0	0	0	1	10	0	0	0
15	& 3	CON	4	13	14	16	19	0	0	0	0	0
16	L3(-1)	PRI	0	0	0	0	0	1	10	0	0	0
17	V16(-1)	PRI	0	0	0	0	0	1	10	0	0	0
18	V5(1)	PRI	0	0	0	0	0	1	10	0	0	0
19	& 4	CON	4	17	18	8	22	0	0	0	0	0
20	V6(L)	PRI	0	0	0	0	0	1	10	0	0	0
21	V16(L)	PRI	0	0	0	0	0	1	10	0	0	0
22	& 5	CON	3	20	21	23	0	0	0	0	0	0
23	V5(L)	PRI	0	0	0	0	0	1	10	0	0	0
24	L3(13)	ORC	2	25	26	0	0	3	38	39	40	0
25	V16(1)	PRI	0	0	0	0	0	1	24	0	0	0
26	F11(-1)	PRI	0	0	0	0	0	1	24	0	0	0
27	L3(16)	AND	2	28	4	0	0	0	0	0	0	0
28	D1(33)	ORC	2	122	123	0	0	5	27	70	86	237
29	L3(144)	ORC	2	12	31	0	0	3	36	34	37	0
30	V4(1)	PRI	0	0	0	0	0	3	31	49	53	0
31	# 6	AND	2	30	32	0	0	1	29	0	0	0
32	T2(41)	PRI	0	0	0	0	0	1	31	0	0	0
33	N1(11)	DIR	1	1	0	0	0	3	116	127	132	0
34	N2(12)	ORS	6	1	10	29	67	1	167	0	0	0
35	N3(11)	DIR	1	1	0	0	0	1	220	0	0	0
36	N1(12)	ORS	2	10	29	0	0	0	0	0	0	0
37	N3(12)	ORS	3	10	29	50	0	1	228	0	0	0
38	N1(13)	DIR	1	24	0	0	0	2	119	129	0	0
39	N2(13)	ORS	3	24	50	61	0	1	170	0	0	0
40	N3(13)	ORS	5	24	41	46	68	1	234	0	0	0
41	L3(22)	DIR	1	42	0	0	0	4	43	44	40	45
42	T1(22)	PRI	0	0	0	0	0	1	41	0	0	0
43	N1(22)	DIR	1	41	0	0	0	2	124	129	0	0
44	N2(11)	ORS	2	41	54	0	0	1	166	0	0	0
45	N3(531)	ORS	4	41	54	57	63	1	239	0	0	0
46	L3(521)	ORC	2	47	49	0	0	2	34	40	0	0
47	L1(521)	DIR	1	210	0	0	0	1	46	0	0	0
48	L2(521)	DIR	1	212	0	0	0	1	49	0	0	0
49	# 7	AND	2	48	30	0	0	1	46	0	0	0
50	L3(531)	ORC	2	51	53	0	0	2	39	37	0	0
51	L1(531)	DIR	1	211	0	0	0	1	50	0	0	0
52	L2(531)	DIR	1	213	0	0	0	1	53	0	0	0
53	# 8	AND	2	52	30	0	0	1	50	0	0	0
54	L8(11)	ORC	2	55	56	0	0	4	65	44	40	45
55	N9(11)	DIR	1	159	0	0	0	1	54	0	0	0
56	L8(0)	PRI	0	0	0	0	0	2	54	152	0	0
57	L8(12)	ORC	3	58	59	60	0	4	66	34	40	45
58	N9(12)	DIR	1	161	0	0	0	1	57	0	0	0
59	L8(-1)	PRI	0	0	0	0	0	1	57	0	0	0
60	L8(L)	PRI	0	0	0	0	0	1	57	0	0	0
61	L8(13)	DIR	1	62	0	0	0	2	69	39	0	0

Table G.1.b : continuation of table G.1.a

ADDR	EVENT	GATE	BC	C1	C2	C3	C4	BQ	Q1	Q2	Q3	Q4
62	N9(13)	DIR	1	163	0	0	0	1	61	0	0	0
63	L8(145)	DIR	1	64	0	0	0	1	45	0	0	0
64	N9(145)	DIR	1	158	0	0	0	1	63	0	0	0
65	N5(11)	DIR	1	54	0	0	0	2	124	12	0	0
66	N5(12)	DIR	1	57	0	0	0	1	129	0	0	0
67	& 9	CON	3	46	57	83	0	0	0	0	0	0
68	& 10	CON	2	54	57	0	0	0	0	0	0	0
69	N5(13)	DIR	1	61	0	0	0	1	132	0	0	0
70	L8(33)	DIR	1	28	0	0	0	1	71	0	0	0
71	N9(33)	DIR	1	70	0	0	0	1	152	0	0	0
72	L10(11)	ORC	4	73	74	75	76	2	87	88	0	0
73	L9(11)	ORC	3	92	94	96	0	1	72	0	0	0
74	CV2(0)	PRI	0	0	0	0	0	2	72	108	0	0
75	L10(0)	PRI	0	0	0	0	0	2	72	108	0	0
76	FI2(1)	PRI	0	0	0	0	0	2	72	77	0	0
77	L10(12)	ORC	5	78	79	80	82	2	89	88	0	0
78	L9(12)	ORC	4	97	98	93	99	1	77	0	0	0
79	CV2(-1)	PRI	0	0	0	0	0	1	77	0	0	0
80	L10(-1)	PRI	0	0	0	0	0	1	77	0	0	0
81	L10(L)	PRI	0	0	0	0	0	1	77	0	0	0
82	& 11	CON	2	81	76	0	0	0	0	0	0	0
83	L10(13)	ORC	2	84	85	0	0	2	90	34	0	0
84	CV2(1)	PRI	0	0	0	0	0	1	83	0	0	0
85	FI2(-1)	PRI	0	0	0	0	0	1	83	0	0	0
86	L10(16)	DIR	1	28	0	0	0	2	102	101	0	0
87	N4(11)	DIR	1	72	0	0	0	2	116	126	0	0
88	N2(521)	ORS	2	72	77	0	0	1	174	0	0	0
89	N4(12)	DIR	1	77	0	0	0	2	127	132	0	0
90	N4(13)	DIR	1	83	0	0	0	3	119	124	129	0
91	L10(33)	DIR	1	28	0	0	0	1	110	0	0	0
92	C1(41)	ORC	3	122	176	177	0	1	73	0	0	0
93	P2(0)	PRI	0	0	0	0	0	3	94	78	102	0
94	# 12	AND	2	93	95	0	0	1	73	0	0	0
95	V9(0)	PRI	0	0	0	0	0	4	94	100	110	222
96	L9(0)	PRI	0	0	0	0	0	1	73	0	0	0
97	L9(-1)	PRI	0	0	0	0	0	1	78	0	0	0
98	L9(L)	PRI	0	0	0	0	0	1	78	0	0	0
99	P2(-1)	PRI	0	0	0	0	0	1	78	0	0	0
100	L9(13)	ORC	2	101	95	0	0	0	0	0	0	0
101	L11(13)	ORC	3	173	245	86	0	1	100	0	0	0
102	L9(16)	AND	3	86	93	103	0	1	106	0	0	0
103	# 13	ORC	2	104	105	0	0	1	102	0	0	0
104	L11(0)	PRI	0	0	0	0	0	3	103	109	240	0
105	CV3(0)	PRI	0	0	0	0	0	3	103	109	240	0
106	N7(16)	DIR	1	102	0	0	0	1	181	0	0	0
107	# 16	AND	2	108	109	0	0	1	110	0	0	0
108	# 14	ORC	2	75	74	0	0	1	107	0	0	0
109	# 15	ORC	2	104	105	0	0	1	107	0	0	0
110	L9(33)	ORC	3	107	91	95	0	0	0	0	0	0
111	L9(521)	DIR	1	112	0	0	0	1	246	0	0	0
112	N7(521)	DIR	1	174	0	0	0	1	111	0	0	0
113	D1(41)	ORC	3	114	115	116	0	1	156	0	0	0

Table G.1.c : continuation of table G.1.b

ADDR	EVENT	GATE	BC	C1	C2	C3	C4	BQ	Q1	Q2	Q3	Q4
114	D1(L)	PRI	0	0	0	0	0	2	113	117	0	0
115	RB1(L)	PRI	0	0	0	0	0	2	113	117	0	0
116	¤ 17	AND	2	33	87	0	0	1	113	0	0	0
117	D1(42)	ORC	3	114	115	118	0	1	157	0	0	0
118	LCL2(-1)	ORC	3	142	143	144	0	1	117	0	0	0
119	¤ 18	AND	2	38	90	0	0	1	120	0	0	0
120	D1(43)	ORC	2	119	121	0	0	1	158	0	0	0
121	LCL2(1)	ORC	3	145	146	147	0	1	120	0	0	0
122	L4(0)	PRI	0	0	0	0	0	3	28	166	92	0
123	C1(33)	DIR	1	184	0	0	0	2	28	175	0	0
124	TRAY1(22)	ORC	4	43	90	65	125	0	0	0	0	0
125	TCL2(-1)	ORC	2	138	139	0	0	1	124	0	0	0
126	TRAY1(23)	ORC	3	87	127	128	0	0	0	0	0	0
127	¤ 19	AND	2	89	33	0	0	1	126	0	0	0
128	TCL2(1)	ORC	2	140	141	0	0	1	126	0	0	0
129	TRAY10(22)	ORC	6	43	65	38	130	0	0	0	0	0
130	& 20	CON	3	90	66	131	0	0	0	0	0	0
131	TCL1(-1)	ORC	2	134	135	0	0	1	129	0	0	0
132	TRAY10(23)	ORC	4	69	33	89	133	0	0	0	0	0
133	TCL1(1)	ORC	2	136	137	0	0	1	132	0	0	0
134	TI1(1)	PRI	0	0	0	0	0	1	131	0	0	0
135	TC1(-1)	PRI	0	0	0	0	0	1	131	0	0	0
136	TI1(-1)	PRI	0	0	0	0	0	1	133	0	0	0
137	TC1(1)	PRI	0	0	0	0	0	1	133	0	0	0
138	TI2(1)	PRI	0	0	0	0	0	1	125	0	0	0
139	TC2(-1)	PRI	0	0	0	0	0	1	125	0	0	0
140	TI2(-1)	PRI	0	0	0	0	0	1	128	0	0	0
141	TC2(1)	PRI	0	0	0	0	0	1	128	0	0	0
142	LI2(1)	PRI	0	0	0	0	0	1	118	0	0	0
143	LC2(-1)	PRI	0	0	0	0	0	1	118	0	0	0
144	CV4(1)	PRI	0	0	0	0	0	2	118	234	0	0
145	LI2(-1)	PRI	0	0	0	0	0	1	121	0	0	0
146	LC2(1)	PRI	0	0	0	0	0	1	121	0	0	0
147	CV4(-1)	PRI	0	0	0	0	0	2	121	228	0	0
148	L6(11)	ORC	2	149	150	0	0	1	153	0	0	0
149	L5(11)	DIR	1	155	0	0	0	1	148	0	0	0
150	L6(0)	PRI	0	0	0	0	0	1	148	0	0	0
151	L6(16)	DIR	1	152	0	0	0	1	234	0	0	0
152	RB1(33)	ORC	2	56	71	0	0	2	151	154	0	0
153	N8(11)	DIR	1	148	0	0	0	1	156	0	0	0
154	L6(33)	DIR	1	152	0	0	0	0	0	0	0	0
155	L5(0)	PRI	0	0	0	0	0	1	149	0	0	0
156	RB1(41)	ORC	2	153	113	0	0	1	165	0	0	0
157	RB1(42)	DIR	1	117	0	0	0	1	165	0	0	0
158	RB1(43)	DIR	1	120	0	0	0	1	64	0	0	0
159	RB1(71)	DIR	1	160	0	0	0	1	55	0	0	0
160	RBH(0)	PRI	0	0	0	0	0	1	159	0	0	0
161	RB1(72)	DIR	1	162	0	0	0	1	58	0	0	0
162	RBH(-1)	PRI	0	0	0	0	0	1	161	0	0	0
163	RB1(73)	DIR	1	164	0	0	0	1	62	0	0	0
164	RBH(1)	PRI	0	0	0	0	0	1	163	0	0	0
165	RBH(BO)	ORS	2	156	157	0	0	0	0	0	0	0

Table G.1.d : continuation of table G.1.c

ADDR	EVENT	GATE	BC	C1	C2	C3	C4	BQ	Q1	Q2	Q3	Q4
166	L4(11)	ORC	2	44	122	0	0	1	171	0	0	0
167	L4(12)	ORC	3	34	168	169	0	1	172	0	0	0
168	L4(L)	PRI	0	0	0	0	0	1	167	0	0	0
169	L4(-1)	PRI	0	0	0	0	0	1	167	0	0	0
170	L4(13)	DIR	1	39	0	0	0	1	173	0	0	0
171	N6(11)	DIR	1	166	0	0	0	1	240	0	0	0
172	N6(12)	DIR	1	167	0	0	0	3	178	240	241	0
173	N6(13)	DIR	1	170	0	0	0	1	101	0	0	0
174	L4(521)	DIR	1	88	0	0	0	1	112	0	0	0
175	L4(33)	DIR	1	123	0	0	0	0	0	0	0	0
176	C1(71)	DIR	1	185	0	0	0	2	92	189	0	0
177	C1(L)	PRI	0	0	0	0	0	2	92	178	0	0
178	C1(42)	ORC	4	172	179	177	180	0	0	0	0	0
179	C1(72)	ORC	2	186	187	0	0	2	178	189	0	0
180	LCL1(-1)	ORC	2	190	191	0	0	1	178	0	0	0
181	C1(43)	ORC	3	182	106	183	0	1	188	0	0	0
182	C1COIL(L)	PRI	0	0	0	0	0	1	181	0	0	0
183	LCL1(1)	ORC	2	192	193	0	0	1	181	0	0	0
184	C1VENT(0)	PRI	0	0	0	0	0	1	123	0	0	0
185	L15(11)	ORC	4	194	195	196	197	1	176	0	0	0
186	L15(12)	ORC	5	198	199	200	203	1	179	0	0	0
187	C1COIL(F)	PRI	0	0	0	0	0	1	179	0	0	0
188	C1VENT(OF	DIR	1	181	0	0	0	0	0	0	0	0
189	C1VENT(VR)	ORS	2	176	179	0	0	0	0	0	0	0
190	LI1(1)	PRI	0	0	0	0	0	1	180	0	0	0
191	LC1(-1)	PRI	0	0	0	0	0	1	180	0	0	0
192	LI1(-1)	PRI	0	0	0	0	0	1	183	0	0	0
193	LC1(1)	PRI	0	0	0	0	0	1	183	0	0	0
194	V7(0)	PRI	0	0	0	0	0	1	185	0	0	0
195	V8(0)	PRI	0	0	0	0	0	1	185	0	0	0
196	L15(0)	PRI	0	0	0	0	0	1	185	0	0	0
197	C1COIL(O)	PRI	0	0	0	0	0	1	185	0	0	0
198	V7(-1)	PRI	0	0	0	0	0	1	186	0	0	0
199	V8(-1)	PRI	0	0	0	0	0	1	186	0	0	0
200	L15(-1)	PRI	0	0	0	0	0	1	186	0	0	0
201	L15(L)	PRI	0	0	0	0	0	1	186	0	0	0
202	C1COIL(-1)	PRI	0	0	0	0	0	1	186	0	0	0
203	& 21	CON	2	201	202	0	0	0	0	0	0	0
204	V3(0)	PRI	0	0	0	0	0	1	2	0	0	0
205	L1(0)	PRI	0	0	0	0	0	1	2	0	0	0
206	T1(41)	PRI	0	0	0	0	0	1	2	0	0	0
207	V3(-1)	PRI	0	0	0	0	0	1	11	0	0	0
208	L1(L)	PRI	0	0	0	0	0	1	11	0	0	0
209	L1(-1)	PRI	0	0	0	0	0	1	11	0	0	0
210	T1(521)	DIR	1	214	0	0	0	1	47	0	0	0
211	T1(531)	DIR	1	215	0	0	0	1	51	0	0	0
212	T2(521)	DIR	1	214	0	0	0	1	48	0	0	0
213	T2(531)	DIR	1	215	0	0	0	1	52	0	0	0
214	L14(521)	ORC	2	216	21	0	0	2	210	212	0	0
215	L14(531)	ORC	2	218	219	0	0	2	211	213	0	0
216	V13(1)	PRI	0	0	0	0	0	1	214	0	0	0
217	V14(-1)	PRI	0	0	0	0	0	1	214	0	0	0

Table G.1.e : continuation of table G.1.d

ADDR	EVENT	GATE	BC	C1	C2	C3	C4	BQ	Q1	Q2	Q3	Q4
218	V13(-1)	PRI	0	0	0	0	0	1	215	0	0	0
219	V14(1)	PRI	0	0	0	0	0	1	215	0	0	0
220	L7(11)	ORC	6	35	222	223	226	0	0	0	0	0
221	P3(0)	PRI	0	0	0	0	0	1	222	0	0	0
222	¤ 22	AND	2	221	95	0	0	1	220	0	0	0
223	CV4(0)	PRI	0	0	0	0	0	1	220	0	0	0
224	L7(0)	PRI	0	0	0	0	0	1	220	0	0	0
225	HE1(0)	PRI	0	0	0	0	0	1	220	0	0	0
226	& 23	CON	3	224	225	227	0	0	0	0	0	0
227	FI3(1)	PRI	0	0	0	0	0	2	220	228	0	0
228	L7(12)	ORC	7	37	147	229	232	0	0	0	0	0
229	L7(-1)	PRI	0	0	0	0	0	1	228	0	0	0
230	L7(L)	PRI	0	0	0	0	0	1	228	0	0	0
231	HE1(-1)	PRI	0	0	0	0	0	1	228	0	0	0
232	& 24	CON	4	230	231	233	227	0	0	0	0	0
233	HE1(L)	PRI	0	0	0	0	0	1	228	0	0	0
234	L7(13)	ORC	6	40	235	144	236	0	0	0	0	0
235	V12(0)	PRI	0	0	0	0	0	1	234	0	0	0
236	& 25	CON	3	151	28	238	0	0	0	0	0	0
237	@ 26		0	0	0	0	0	2	91	234	0	0
238	FI3(-1)	PRI	0	0	0	0	0	1	234	0	0	0
239	L7(531)	DIR	1	45	0	0	0	0	0	0	0	0
240	L11(11)	ORC	4	172	171	104	105	0	0	0	0	0
241	L11(12)	ORC	4	172	242	243	244	0	0	0	0	0
242	L11(-1)	PRI	0	0	0	0	0	1	241	0	0	0
243	L11(L)	PRI	0	0	0	0	0	1	241	0	0	0
244	CV3(-1)	PRI	0	0	0	0	0	1	241	0	0	0
245	CV3(1)	PRI	0	0	0	0	0	1	101	0	0	0
246	L11(521)	DIR	1	111	0	0	0	0	0	0	0	0