Binary Decision Diagrams for Fault Tree Analysis

by

Roslyn Mary Sinnamon

A Doctoral Thesis
submitted in partial fulfilment of the requirements for the award of
Doctor of Philosophy of Loughborough University

October 1996

Abstract

This thesis develops a new approach to fault tree analysis, namely the Binary Decision Diagram (BDD) method. Conventional qualitative fault tree analysis techniques such as the "top-down" or "bottom-up" approaches are now so well developed that further refinement is unlikely to result in vast improvements in terms of their computational capability. The BDD method has exhibited potential gains to be made in terms of speed and efficiency in determining the minimal cut sets. Further, the nature of the binary decision diagram is such that it is more suited to Boolean manipulation. The BDD method has been programmed and successfully applied to a number of benchmark fault trees.

The analysis capabilities of the technique have been extended such that all quantitative fault tree top event parameters, which can be determined by conventional Kinetic Tree Theory, can now be derived directly from the BDD. Parameters such as the top event probability, frequency of occurrence and expected number of occurrences can be calculated exactly using this method, removing the need for the approximations previously required.

Thus the BDD method is proven to have advantages in terms of both accuracy and efficiency. Initiator/enabler event analysis and importance measures have been incorporated to extend this method into a full analysis procedure.

Acknowledgements

I would like to give my sincere thanks to my supervisor Dr John Andrews who has provided me with invaluable help, guidance and friendship throughout the course of my PhD work. I would also like to acknowledge Richard Pullen of Isograph Ltd. who supplied the FAULTREE+ software which provided me with important comparative results. Also thanks goes to my colleagues in the mathematics department for their kind friendship and good humour. Lastly I would like to thank Nigel for his support during my PhD.

Contents

Notation

C_i	Minimal cut set i
dt	Infinitesimally small interval
du	Infinitesimally small interval
$f(t)$	Failure probability density function
$G_i(\mathbf{q})$	Criticality function for component i (Birnbaums measure of importance)
$g(t)$	Repair probability density function
I_i	Importance measure for component i
nc	Number of minimal cut sets
P	Probability; probability of an event
$Q_{sys}(t)$	System unavailability
Q_{AV}	Average unavailability
$Q(t)$	Unavailability function
q_i	Component unavailability
t	Time; aggregate system life
t_0	Initial time, start time
t_1	Smallest ordered age at failure
$v(t)$	Unconditional repair intensity
$w(t)$	Unconditional failure intensity
λ	Constant failure rate
μ	Constant repair rate
$\lambda(t)$	Conditional failure intensity
$\mu(t)$	Conditional repair intensity
r	Mean time to failure (MTTF)
τ	Mean time to repair (MTTR)
θ	Test or inspection interval

CHAPTER 1

INTRODUCTION TO FAULT TREE ANALYSIS

1.1 Introduction

Reliability engineering techniques are extremely important for industrial safety systems which are designed to protect against hazardous events. If the safety features of an industrial system fail the consequences may be catastrophic, as in the case of the Piper Alpha disaster in 1988 which caused 167 deaths. Reliability is also an important factor affecting the economic viability of industrial processes.

The most common technique used for system reliability assessment is fault tree analysis. Fault tree analysis is a formal deductive procedure for determining combinations of component failures and human errors that could result in the occurrence of a specified undesired system failure mode. The undesired system failure is referred to as the 'top event' and the fault tree represents how this top event can be caused by individual or combined lower level events. Once constructed the fault tree diagram can be analysed to yield reliability parameters concerning the system failure.

The fault tree provides a diagrammatic description of the way in which a system can fail in a specific mode. The importance of the fault tree for safety system analysis is that it yields a complete description of the various causes of the system failure. Hence the engineers can identify and rectify any problem areas in the design.

1.1.1 Background

Fault tree analysis was first developed by H. A. Watson 1961-62 at Bell Telephone Laboratories during a study of the launch control system of the Minuteman, Intercontinental ballistic missile.

The quantification techniques known as Kinetic Tree Theory was supplied nearly ten years later by Vesely (1). Initiator and enabler event theory was then developed by Dunglinson and Lambert (2) and importance measures were supplied by Birnbaum (3) and Barlow and Proschan (4). Industries which have made extensive use of fault tree analysis include the nuclear industry, power generation industries, the chemical process

industry and aerospace industry. Guidelines to construct the fault tree in a methodical manner have been developed by Hassl et al. (5) (Nuclear Regulatory Commission).

A full description of fault tree analysis can be found in Andrews and Moss (6) and Henley and Kumamoto (7).

1.2 General Description of the Fault Tree Analysis Method

1.2.1 Fault Tree Symbols and Construction

Fault tree analysis is a deductive logic approach used to identify the causal relationships leading to a specific system failure mode. The initial step in the fault tree analysis of a system is to identify the system failure mode of concern. This becomes the top event of the fault tree. One system may have many different failure modes that need to be analysed, in this case a separate fault tree is required for each one. The top event of the fault tree is developed by branches leading down from this event to other sub-events which represent its possible causes. These sub-events are continually redefined in terms of lower resolution events until the branches are terminated with component failures. The terminating events in the development are classified as basic events.

Each fault tree is built up from gates and events, the gates link the events together depending on their causal relationships. The main two types of gates used are the 'AND' and 'OR' gate. Other gate types exist (6) but these only reduce the size of the fault tree diagram and have to be expressed in terms of 'AND', 'OR' and 'NOT' logic prior to the analysis. The minimal set of gates to represent all logic operators which are used in the fault tree analysis methods described in this thesis are shown in table 1.1.

Also a restricted minimal set of the event type symbols used in the fault tree analysis are shown in table 1.2. Again a more detailed list of symbols in general use can be found in reference (6). Basic events, represented by a circle, indicate the limit of resolution of the fault tree. For a quantitative analysis it is these events for which data are required.

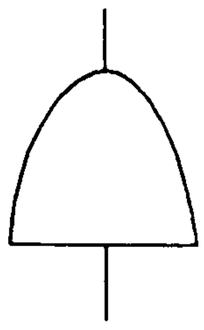
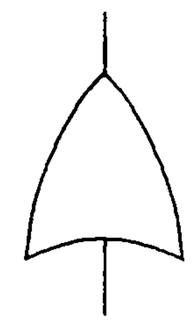
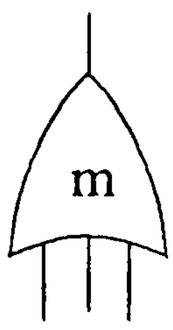
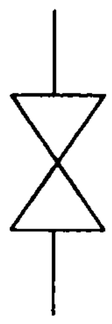
Gate Symbol	Gate Type	Causal Relation
	AND Gate	Output event occurs if all input events occur simultaneously
	OR Gate	Output event occurs if at least one of the input events occur
 n inputs	m-out-of-n Gate (Vote Gate)	Output event occurs if at least m-out-of-n input events occur
	NOT Gate	Output event occurs if the input event does not

Table 1.1 Common Gate Symbols and Types

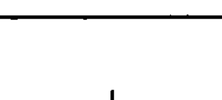
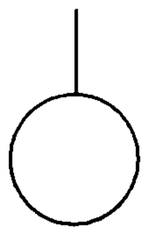
Event Symbol	Meaning of Symbol
	Intermediate event further developed by a gate
	Basic event

Table 1.2 Common Event Symbols

The OR gate, the AND gate and the NOT gate combine events in exactly the same way as the Boolean operations of 'disjunction', 'conjunction' and 'negation'. There is therefore a one-to-one correspondence between Boolean algebraic expressions and the fault tree structure.

Once the system has been defined and a particular system failure mode selected as the top event, the fault tree is developed by determining the **immediate**, **necessary** and **sufficient** causes for its occurrence.

Fault tree analysis occurs in two stages, a qualitative stage and a quantitative stage, both are discussed below.

1.3 Qualitative Results of Fault Tree Analysis

Each unique way that system failure can occur is a **system failure mode** and will involve the failure of individual components or combinations of components. When system failure modes are defined in terms of component failures only, the system is referred to as a **coherent** system. A fault tree where basic events represent component failure and which contains only AND and OR logic gates is called a **coherent** fault tree and the system failure modes are defined by the concept of a **cut set**.

A **cut set** is a collection of basic events such that if they all occur the top event also occurs.

A **minimal cut set** is the smallest combination of component failures, which if they all occur will cause the top event to occur.

The dual concept of a cut set in the success space is called a **path set**. This is a collection of basic events in which, providing each failure event does not occur, the top event will not occur (i.e. a list of components whose functioning ensures successful system operation for the particular top event considered).

Implicant sets are combinations of both failure and success events that cause the top event and **prime implicants** are implicants where this combination is minimal.

A system which has failure modes involving both failure and success events is referred to as a **non-coherent** system. Non-coherent fault trees are fault trees that contain AND, OR and NOT logic gates.

1.3.1 Minimal Cut Sets of Coherent Fault Trees

Once a fault tree has been constructed a qualitative assessment which produces minimal cut sets can be performed. The conventional approach to obtain the minimal cut sets is to take the Boolean logic expression for the top event and transform it into disjunctive normal form (sum of products form). One way of doing this is to use a "top-down" approach, which is demonstrated by its application to the fault tree shown in figure 1.1.

To obtain the sum of products form for the top event in figure 1.1, the inputs to the gates in the fault tree are represented as logic equations. The dot or product is used to represent AND gates whilst the sum is used to represent OR gates.

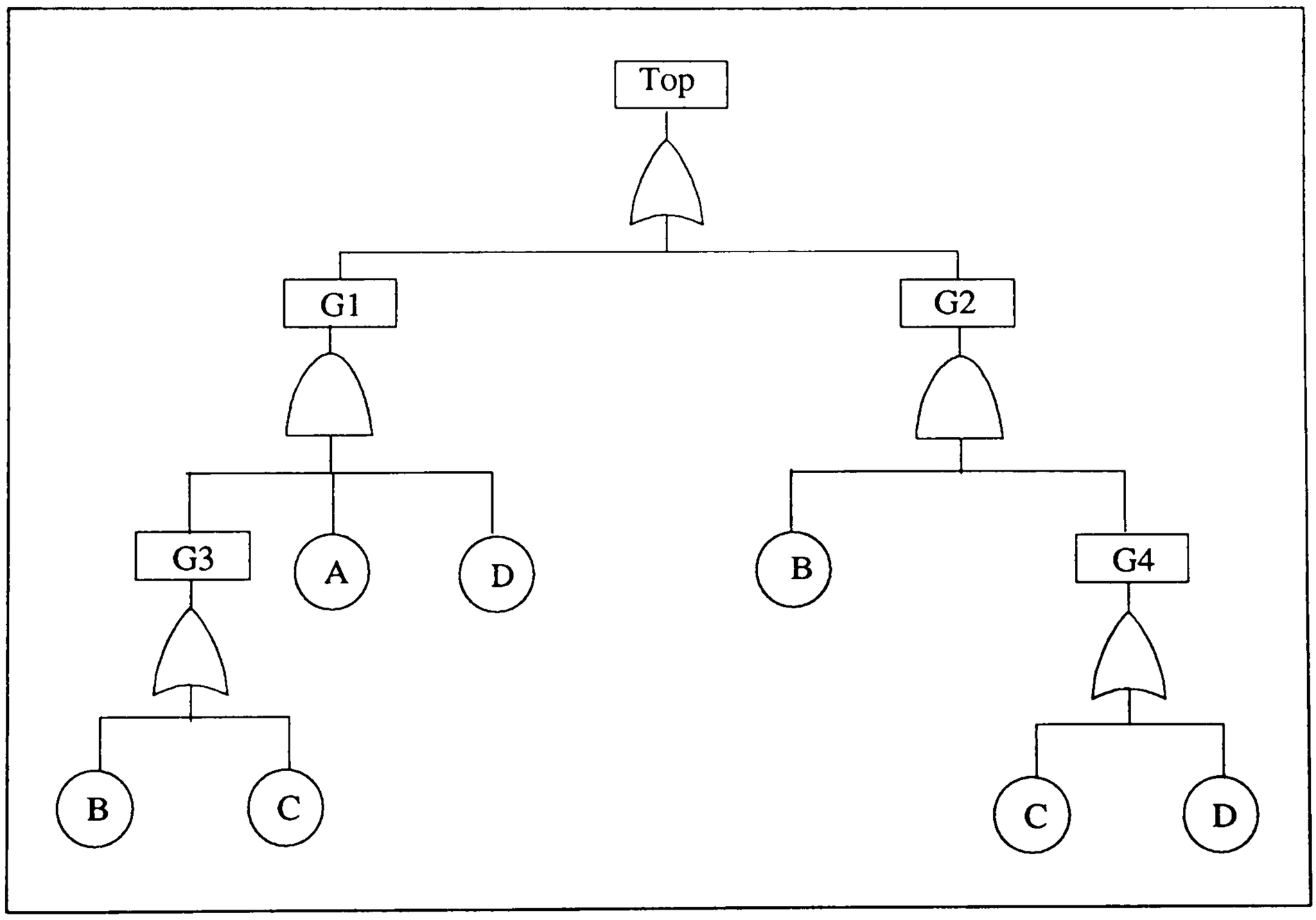


Figure 1.1 Example Fault Tree for the Calculation of the Minimal Cut Sets

The top-down approach starts with the top event and expands this event by substituting in the Boolean variables appearing lower down in the tree. Boolean

variables are assigned to represent the occurrence of each basic event. Then the laws of Boolean algebra given below are used to remove redundancies in the expressions.

1. Commutative Laws

   $A+B=B+A, \quad A.B=B.A$

2. Associative Laws

   $(A+B)+C=A+(B+C), \quad (A.B).C=A.(B.C)$

3. Distributive Laws

   $A+(B.C)=(A+B).(A+C), \quad A.(B+C)=A.B+A.C$

4. Identities

   $A+0=A, \quad A.1=A$

5. Idempotent Laws

   $A+A=A, \quad A.A=A$

6. Absorption Laws

   $A+A.B=A, \quad A.(A+B)=A$

7. Complementation

   $\overline{A} = 1 - A, \quad A.\overline{A} = 0, \quad \overline{(\overline{A})} = A$

8. De Morgans Laws

   $\overline{(A+B)} = \overline{A}.\overline{B}, \quad \overline{(A.B)} = \overline{A} + \overline{B}$

To find the minimal cut sets for the example fault tree in figure 1.1 begin at the top gate called Top. Top is an OR gate with two inputs, G1 and G2, therefore Top can be expressed as G1+G2. Next dealing with gate G1 which is an AND gate with inputs G3, A and D, Top can now be expressed as G3.A.D+G2. G2 is then represented in terms of its inputs to give B.G4, therefore Top becomes G3.A.D+B.G4. Continuing in this way and substituting in the inputs for gates G3 and G4 completes the expression for the top gate defined in terms of basic events.

$$Top=(B+C).A.D+B.(C+D)$$

Expanding according to the distributive laws and then applying the absorption law gives:

$$\text{Top}=B.A.D+C.A.D+B.C+B.D \tag{1.1}$$

$$=C.A.D+B.C+B.D \tag{1.2}$$

Expression (1.1) provides the "Boolean Indicated Cut Sets" (BICS) of the fault tree. The BICS are simply the cut sets of the fault tree expressed in sum of products form. Expression (1.2) is the minimal disjunctive form of the logic equation, each term of which is a minimal cut set. For this example fault tree there are three minimal cut sets, one of order three (containing three basic events) and two of order two (containing two basic events). The minimal cut sets are {C, A, D}, {B, C} and {B, D}. Other commonly applied methods of deriving the minimal cut sets are based on "bottom-up" manipulations of the fault tree. A bottom-up approach starts at the last level or basic event level of the fault tree and proceeds up through the tree, substituting each gate in terms of its basic event inputs, until the top gate is reached.

1.3.2 Prime Implicants of Non-Coherent Fault Trees

Obtaining the prime implicants of non-coherent fault trees requires some additional work. Consider the non-coherent fault tree illustrated in figure 1.2. Initially it can be converted to an equivalent fault tree shown in figure 1.3 by using De Morgans Laws. The NOT gate or NOT operator has been removed by allowing the complementing of basic events. In this way the NOT operator has been "pushed down" the fault tree to complement the basic events.

By using the same top-down procedure as for coherent fault trees, the sum of products expression for the top event of the non-coherent fault tree shown in figure 1.3 can be obtained as follows:

$$\text{Top}=G1+G2$$

$$=G3.G4+A.B$$

$$=(\overline{A}.\overline{C}).(\overline{D}+\overline{E})+A.B$$

$$\text{Top}=\overline{A}.\overline{C}.\overline{D}+\overline{A}.\overline{C}.\overline{E}+A.B \tag{1.3}$$

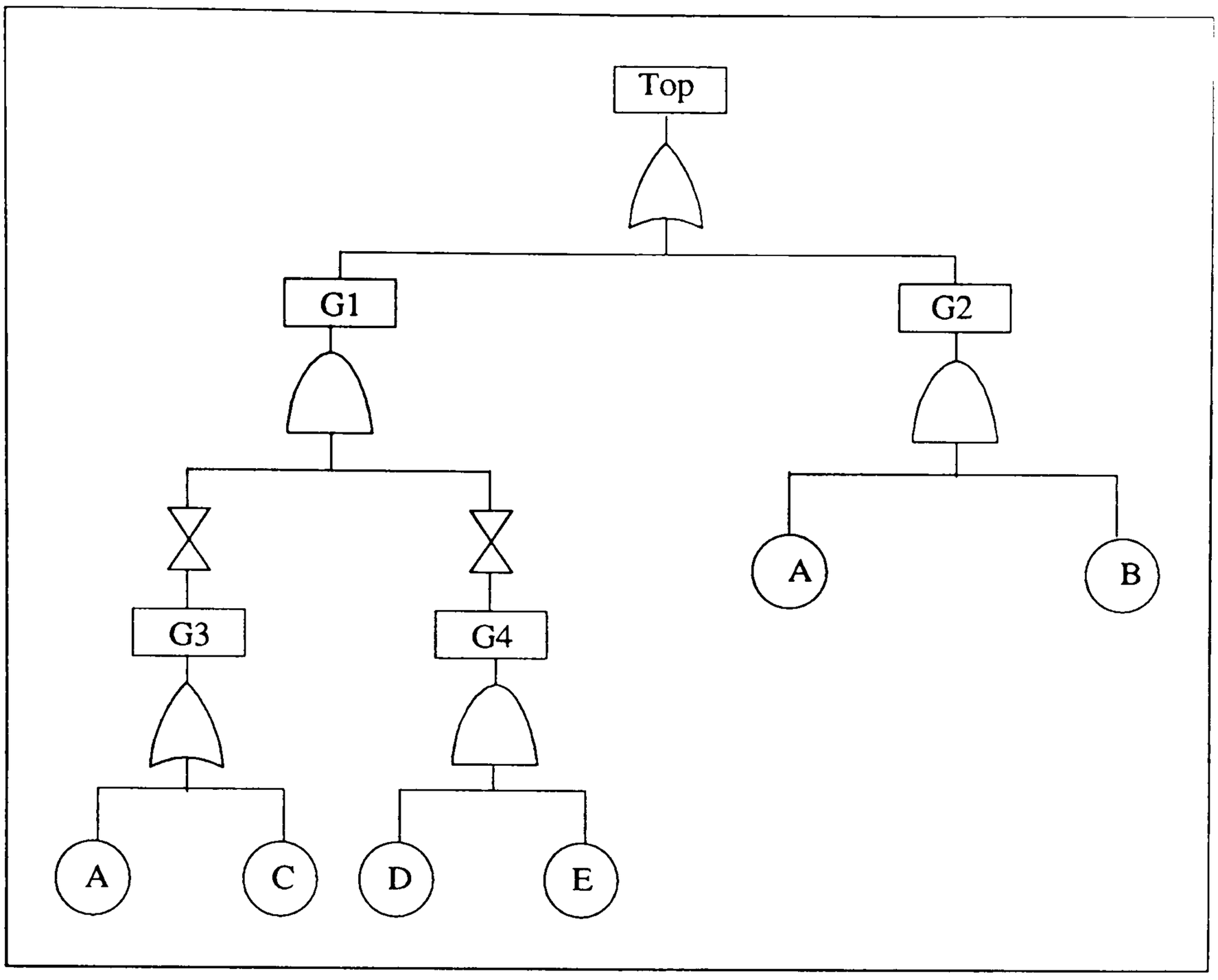


Figure 1.2 Example Non-Coherent Fault Tree

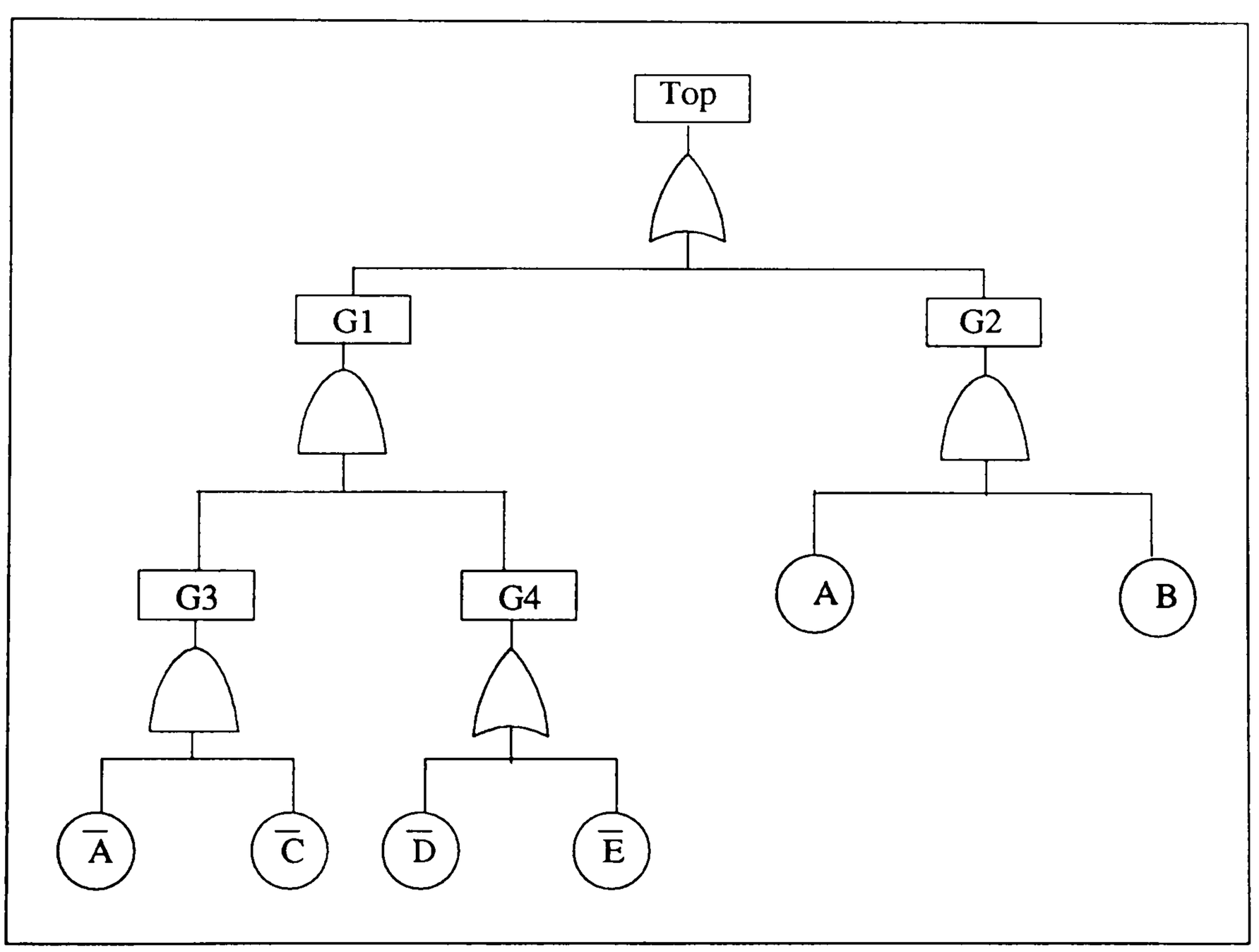


Figure 1.3 Equivalent Fault Tree to Figure 1.2

Now comes the additional work to obtain all the prime implicants. Within expression (1.3) there may be "hidden" prime implicants. These are revealed by techniques such as the consensus theorem (15).

The consensus theorem is as follows:

Given two fundamental conjunctions (products) $\psi 1$ and $\psi 2$. If there is precisely one literal p which occurs negated in one of $\psi 1$ and $\psi 2$ and un-negated in the other, then the fundamental conjunction obtained from $\psi 1.\psi 2$ by deleting p and $\overline{p}$ and omitting repetitions of any other literals is called the *consensus* of $\psi 1$ and $\psi 2$, e.g. the consensus of $A.\overline{B}.C$ and $A.B.D$ is $A.C.D$.

Applying the above theorem to the expression (1.3):

the consensus of $\overline{A}.\overline{C}.\overline{D} + A.B$ is $B.\overline{C}.\overline{D}$
the consensus of $\overline{A}.\overline{C}.\overline{E} + A.B$ is $B.\overline{C}.\overline{E}$

Therefore the 'full' Boolean expression of (1.3) will be:

$$Top = \overline{A}.\overline{C}.\overline{D} + \overline{A}.\overline{C}.\overline{E} + A.B + B.\overline{C}.\overline{D} + B.\overline{C}.\overline{E}$$

which provides the five prime implicants of the non-coherent fault tree shown in figure 1.3.

1.4 Quantitative Results of Fault Tree Analysis

1.4.1 Probability of Top Event Occurrence

The method used in Kinetic Tree Theory to calculate the probability of the top event utilises the previously determined minimal cut sets.

If a fault tree has nc minimal cut sets C_i, $i = 1,\ldots\ldots,nc$ then the top event exists if at least one minimal cut set exists.

i.e.

$$Top = C_1 + C_2 + \ldots\ldots + C_{nc} = \bigcup_{i=1}^{nc} C_i \qquad (1.4)$$

since $P(Top) = P(\bigcup_{i=1}^{nc} C_i)$ this gives:

$$P(Top)=\sum_{i=1}^{nc}P(C_i)-\sum_{i=2}^{nc}\sum_{j=1}^{i-1}P(C_i\cap C_j)+.....(-1)^{nc-1}P(C_1\cap C_2\cap....C_{nc}) \qquad (1.5)$$

This expansion is known as the inclusion-exclusion expansion. The full evaluation of each term in the inclusion-exclusion expansion for calculating the probability of the top event is not practical for fault trees with many minimal cut sets. Therefore approximations that produce acceptably accurate results are required. The inclusion-exclusion expansion adds successive odd numbered terms and subtracts successive even numbered terms, where each term is numerically less significant than the proceeding term. Therefore truncating the series at an odd numbered term will provide an upper bound and truncating the series after an even numbered term will provide a lower bound for the exact probability.

Upper and Lower Bounds for System Unavailability

Consider the first two terms in the inclusion-exclusion expansion. This gives:

$$\sum_{i=1}^{nc}P(C_i)-\sum_{i=2}^{nc}\sum_{j=1}^{i-1}P(C_i\cap C_j)\le Q_{sys}(t)\le\sum_{i=1}^{nc}P(C_i) \qquad (1.6)$$

Lower bound **Exact** **Upper bound**

The upper bound for the top event probability used here is known as the "Rare Event Approximation" since it is itself accurate if the component failure events are rare.

Minimal Cut Set Upper Bound

A more accurate upper bound is the "Minimal Cut Set Upper Bound" which is developed as follows:

P(system failure)=P(at least one minimal cut set occurs)

=1-P(no minimal cut sets occur)

since

$$P(\text{no minimal cut sets occur})\ge\prod_{i=1}^{nc}P(\text{minimal cut set i does not occur})$$

(equality being when no event appears in more than one minimal cut set in which case the minimal cut sets are independent).

Therefore:

$$\text{P(system failure)} \leq 1 - \prod_{i=1}^{nc} \text{P(minimal cut set i does not occur)}$$

i.e.

$$Q_{sys}(t) \leq 1 - \prod_{i=1}^{nc} (1 - P(C_i)) \tag{1.7}$$

It can be shown that:

$$Q_{sys}(t) \leq 1 - \prod_{i=1}^{nc} (1 - P(C_i)) \leq \sum_{i=1}^{nc} P(C_i) \tag{1.8}$$

Exact	**Min Cut Set Upper bound**	**Rare Event Approximation**

1.4.2 Unconditional System Failure Intensity

The reliability of a system (or component) is defined as the probability that the system operates (functions under stated conditions for a stated period of time), Ansell and Phillips (8). For some systems it is the unreliability which is required for the top event i.e., the probability it will **not** work **continuously** over a given time period. An upper bound for this is the expected number of top event occurrences W(0, t):

$$W(0,t) = \int_0^t w_{sys}(t)dt \tag{1.9}$$

where $w_{sys}(t)$ is the system unconditional failure intensity at time t, i.e. the probability that the top event occurs at t per unit time.

$$w_{sys}(t) = \sum_i G_i(\mathbf{q}).w_i(t) \tag{1.10}$$

where $G_i(\mathbf{q})$ is the criticality function for each component and $\mathbf{q}$ is a function of the component failure probabilities.

The criticality function $G_i(\mathbf{q})$ is defined as the probability that the system is in a critical state with respect to component i and that the failure of component i will then cause the system to go from the working to the failed state, i.e., the probability that the system fails only if component i fails. Therefore:

$$G_i(\mathbf{q}) = Q(1_i, \mathbf{q}) - Q(0_i, \mathbf{q}) \tag{1.11}$$

where:

$Q(1_i, \mathbf{q})$ - is the probability function of system failure with $q_i = 1$.
$Q(0_i, \mathbf{q})$ - is the probability function of system failure with $q_i = 0$.

q_i is the unavailability of component i.

1.4.3 Importance Measures

An importance analysis is a sensitivity analysis which can be used to identify weak areas of the system design. For each component or minimal cut set its importance is a numerical value which signifies the role that it plays in either causing or contributing to the occurrence of the top event.

Importance measures can be categorised in two ways:

1. Deterministic
2. Probabilistic

Deterministic Measures

Deterministic measures assess the importance of a component to the system operation without considering the components probability of failure. One such measure is the structural measure of importance which is defined for a component i in a system of n components as:

$$I = \frac{number\ of\ critical\ system\ states\ for\ component\ i}{total\ number\ of\ states\ for\ the\ (n-1)\ remaining\ components}$$

A critical state for component i is a state for the remaining n-1 components such that failure of component i causes the system to go from a working to a failed state.

Probabilistic Measures

These importance measures consider the component's probability of failure together with its structural contribution to failure. The most commonly used probabilistic measures for importance assessment are listed below:

1. Birnbaums Measure of Importance

Birnbaums measure of importance is also known as the criticality function. The criticality function $G_i(\mathbf{q})$ has been previously defined in equation (1.11).

2. Criticality Measure of Importance

This importance measure is defined as the probability that the system is in a critical state for component i and i has failed given that the system has failed.

$$I_{C_i} = \frac{G_i(\mathbf{q})q_i(t)}{Q_{sys}(t)} \tag{1.12}$$

3. Fussell-Vesely Measure of Importance

This measure is defined for each basic event i as the probability of the union of the minimal cut sets containing i given that the system has failed.

$$I_{FV_i} = \frac{P(\bigcup_{k/i\in k} C_k)}{Q_{sys}(t)} \tag{1.13}$$

4. Fussell-Vesely Measure of Minimal Cut Set Importance

The previously defined importance measures ranked component failures in order of their contribution to the top event. This measure provides a similar function except that the minimal cut sets are themselves ranked. This importance measure is defined simply as the probability of occurrence of minimal cut set j given that the system has failed.

$$I_j = \frac{P(C_j)}{Q_{sys}(t)} \tag{1.14}$$

1.5 Objectives of the Project

The majority of the methods developed to perform fault tree analysis work directly with the system fault tree structure. Whilst this type of logic diagram is very good to represent the system failure logic it is not necessarily the most efficient way to manipulate the resulting Boolean equation.

The task of obtaining the minimal cut sets of a fault tree can become computationally intensive if the logic equations produce many cut sets. If a large sum of product expression is obtained then applying the reduction rules can take a long time on a computer due to the number of comparisons that are needed to make the expression minimal. Also storing all the logical expressions for each gate in the tree can make extensive demands on memory space.

This thesis is concerned with the development of a fault tree analysis technique which will improve upon the efficiency of the minimal cut set algorithms previously developed. Its requirement will be to find the various causes of system failure as quickly as possible, with minimum memory requirements. The technique must also be capable of extension to derive the probability of occurrence of the top event, if possible improving on the accuracy of the approximate quantification method used in Kinetic Tree Theory.

The objectives of the work programme were to:

1. Review existing fault tree analysis algorithms for both minimal cut set evaluation and system failure probability quantification.

2. Identify the deficiencies of the current methods and also the features each method possesses which would be of benefit should these be retained in any future developed technique.

3. Determine a method which should address the deficiencies identified in (2) and retain all the desired features. The method will be formulated in a manner for efficient computer implementation.

4. Thoroughly investigate the application of a proposed fault tree analysis method manually and deduce what information can be obtained concerning the fault tree

when applying this technique. Revising the particular details and efficiency of the algorithm as required.

5. Develop the computer implementation of the proposed algorithm to qualitatively and quantitatively analyse the fault tree. Initially this method will be limited to minimal cut set derivation and top event probability.

6. Extend the algorithm to calculate all top event reliability parameters available from traditional fault tree analysis methods.

7. Extend the method to evaluate component and minimal cut set importance measures and initiator/enabler event theory.

8. Improve the efficiency and accuracy of the fault tree analysis code by developing any features which require further research.

9. Test the new features of the method by comparison with a large number of benchmark fault trees also analysed by a state of the art commercial fault tree analysis code which employs traditional Kinetic Tree Theory.

10. Investigate the potential of developing certain other aspects of the method to improve the efficiency of the technique even further. In addition, consider alternative applications of the proposed method, which may benefit other areas of research.

CHAPTER 2

METHODS FOR QUALITATIVE FAULT TREE ANALYSIS

2.1 Introduction

This chapter describes the methods which have been developed to perform a qualitative analysis of fault trees. This type of analysis will produce the minimal combinations of events which when they occur together will cause the fault tree top event. For coherent systems these failure combinations correspond to the minimal cut sets and for non-coherent fault trees, prime implicants. Methods for determining these two sets of information are described in separate sections below.

2.2 Minimal Cut Set Evaluation for Coherent Fault Trees

Qualitative fault tree analysis for coherent systems requires the determination of the minimal cut sets of a fault tree, i.e. the smallest combination of basic event failures that will result in system failure. The conventional approach to obtain the minimal cut sets is to produce a disjunctive normal form for the top event of the fault tree, as discussed in Chapter 1. One of the first approaches of doing this is to use a bottom-up procedure such as that of Semanderes (19).

Semanderes' paper presents a computer program called ELRAFT (**E**fficient **L**ogic **R**eduction **A**nalysis of **F**ault **T**rees), which obtains the minimal cut sets of a fault tree. The three steps of the algorithm are:

1. Union the input events to a gate if the gate is an OR.
2. Intersect the input events if the gate is an AND.
3. Start from the last level of the fault tree and evaluate every gate in that level in terms of basic events before proceeding up to the next level. Continue this process until the top event is expressed solely in terms of basic events.

Redundancies are eliminated in the following way:

1. In a particular intersection, a basic event can appear only once and

2. A particular intersection of basic events is unique and is not a subset of another intersection of events.

The attractive feature of the paper is the utilisation of prime numbers to reduce computer storage. This involves implementing the property of prime numbers as stated by the unique factorisation theorem.

Unique Factorisation Theorem - Every natural number greater than 1 can be expressed as a product of prime factors in one and only one way, apart from the order in which the factors are written.

By assigning a prime number to each basic event, a particular combination of basic events can be expressed uniquely as a single number. This single number is equal to the product of prime numbers corresponding to the basic events in the combination. Then, by factoring any given single number into its prime factors, the basic events which make up the combination can be determined. Refer to figure 2.1 which is used to illustrate the method.

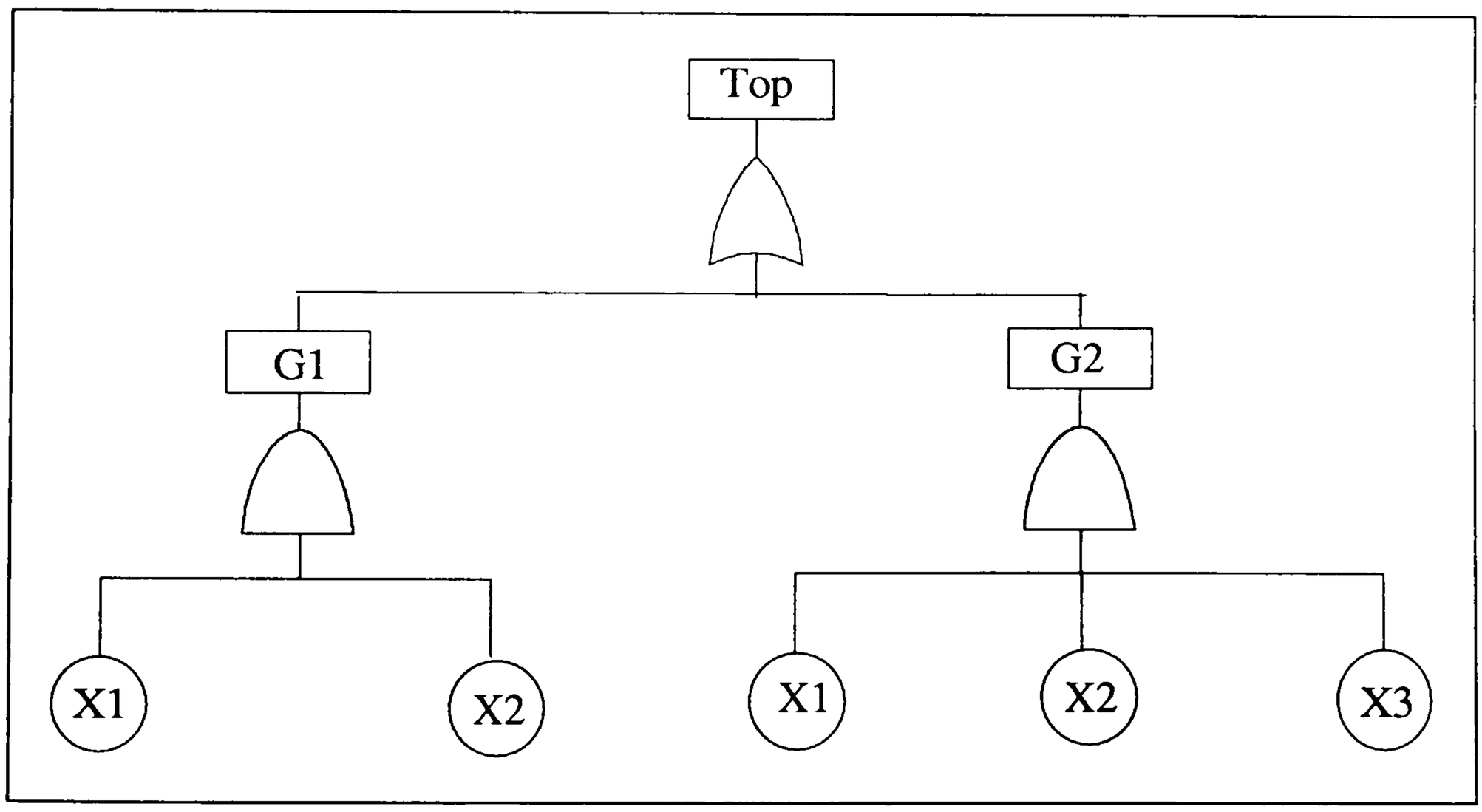


Figure 2.1 Example Fault Tree

For the fault tree shown in figure 2.1 assign the following prime numbers to each basic event, X1=2, X2=3, X3=5. Next apply the bottom-up algorithm utilising the given prime numbers:

$$G1=X1 \cap X2 \equiv 2x3=6$$
$$G2=X1 \cap X2 \cap X3 \equiv 2x3x5=30$$

$$Top=X1\cap X2\cup X1\cap X2\cap X3\equiv[6]\cup[30]$$

However 6 is a factor of 30, thus 30 is dropped and $Top\equiv 6=2x3\equiv X1\cap X2$. This indicates that the fault tree has one minimal cut set of order two which is {X1, X2}.

Semanderes states that the existing program is designed to handle fault trees which use only AND and OR logic gates. However the paper states it could be extended to handle other types of logic gates.

The method of Fussell and Vesely (20), to obtain the minimal cut sets, starts at the top event of the fault tree and proceeds to primary events in a step by step process. This technique is popularly known as 'MOCUS' and is a top-down algorithm. The key points to consider when analysing the fault tree are; AND gates increase the size of the cut sets while OR gates increase the number of cut sets. The Boolean Indicated Cut Sets (BICS) first need to be obtained (defined in Chapter 1), these are then minimised to give the minimal cut sets. It is important to note that the BICS will be precisely the minimal cut sets if the primary events are all independent. To obtain the BICS each gate in the fault tree is randomly associated with a label ω and each primary event with a label ø. (Here I believe it would be better to standardise the procedure. Rather than randomly name the gates and basic events, name the gates consecutively as they appear in the tree, from the top down and left to right on each level. Then label the primary events consecutively in the same way).

The method develops a matrix as the BICS are constructed. The matrix will hold the labels given to the gates and basic events in the fault tree. It is stressed that this method is not limited to fault trees with primary events appearing only once.

The matrix is constructed as follows:

1. Start with the gate representing the top event, this gate label is entered as the 1st element in the 1st row, 1st column of the matrix.
2. Expand the gate according to whether or not it is an AND gate or an OR gate.
   If it is an AND gate expand it by replacing the gate label by the first input to the gate, followed by the remaining inputs to that gate. Place these labels on the same row in separate columns of the matrix.
   If it is an OR gate expand it again by replacing the gate by the first input to the gate, followed by the remaining inputs to that gate. However place these input labels directly below each other on separate rows of the matrix. Duplicate all

other elements in the initial row, where the gate was found, for each of the new rows.

3. Repeat step 2 for each gate label found in the matrix. The aim is to eliminate all the gates in the matrix and finish with only primary events.

When all the entries in the matrix are primary events the BICS have been determined, they are simply the rows of the matrix. Next, a search procedure is employed to eliminate redundancies in the matrix and determine the minimal cut sets. The fault tree in figure 2.2 is used to illustrate the procedure, each gate and basic event in the fault tree has been assigned an integer for computation purposes. The matrix formulation is shown in table 2.1.

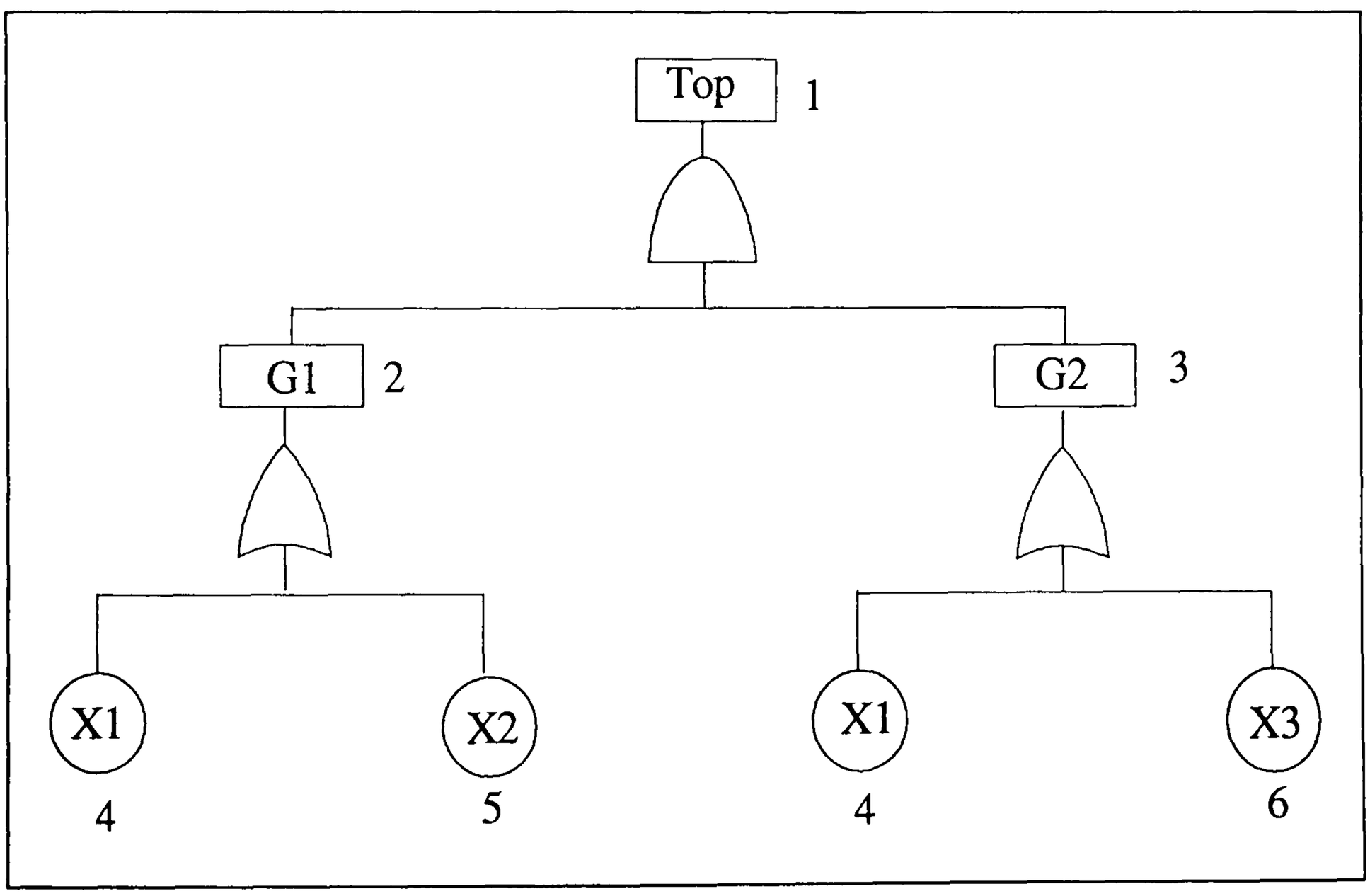


Figure 2.2 Example Fault Tree

To obtain the BICS the elements in the columns, in the final array, are selected together to correspond with ANDing and each row then corresponds to a BIC, therefore the BICS are:

$$4.4 + 4.6 + 5.4 + 5.6$$

The following reduction rules are then applied to obtain the minimal cut sets:

(1) X.X=X

(2) X+X.Y=X

As a result 4.4=4, which makes both 4.6 and 5.4 redundant, leaving the expression:

$$4 + 5.6$$

(1)	Top
(2 3)	Top is an AND gate with inputs G1 and G2.
$\begin{pmatrix} 4 & 3 \\ 5 & 3 \end{pmatrix}$	G1 is an OR gate with basic event inputs X1 and X2. Expand vertically duplicating other columns.
$\begin{pmatrix} 4 & 4 \\ 4 & 6 \\ 5 & 4 \\ 5 & 6 \end{pmatrix}$	G2 is an OR gate with basic event inputs X1 and X3. Expand vertically duplicating all other columns.

Table 2.1 Matrix Formulation to Obtain Minimal Cut Sets

Therefore the minimal cut sets for the fault tree in figure 2.2 are:

(1) {4} i.e. {X1}
(2) {5.6} i.e. {X2, X3}

The results of the paper by Benjiamin, Bowen and Schenk in 1976 (28) are very promising, however the description given of the minimal cut set method is not well explained. A new algorithm is proposed for efficiently generating the minimal cut sets of a fault tree which contains repeated basic events. It is stated that the algorithm substantially reduces both execution time and storage requirements when programmed and compared to alternative methods such as that of MOCUS. The savings are accomplished by recognising and recursively reducing the influence of the repeated events.

The algorithm proceeds in essentially three steps. First, the fault tree containing repeated events is reduced. A reduced fault tree is obtained by eliminating repeated events which are inputs to OR gates. These repeated events are eliminated by assigning each of them with a 'partner'. This partner depends on the other inputs to that gate and also on the output of that gate. This step can substantially reduce the complexity of a fault tree, containing many repeated events, and is given in detail in the

paper. In step two, the cut sets of the reduced tree are then obtained by a conventional technique such as MOCUS. These cut sets are referred to as the Group 1 cut sets. Finally, the Group 1 cut sets are then further processed to yield the Group 2 cut sets. The cut sets of Group 2 are obtained from Group 1 by reinserting the previously eliminated repeated events. As the cut sets of Group 2 are generated they are compared with the cut sets of Group 1 and any non-minimal cut sets are eliminated. When all non-minimal cut sets have been eliminated the cut sets of Group 1 and Group 2 are the desired result.

Fault tree examples are not provided which cover all the cases that may be encountered during the algorithm, as a result those parts of the algorithm are less clear. I feel that the complexity of the computation of this method may outweigh its effectiveness. Also, the nature of the algorithm suggests that it would not be effective for all types of fault trees which contain repeated events, it would only be suitable for fault trees which contain repeated events which are inputs to OR gates. Such an example fault tree is given in the paper. Applying the algorithm to this fault tree produces a gain in computational efficiency over conventional techniques. Using the proposed algorithm 66 cut sets were generated, of which 23 were discarded, and by the MOCUS method 169 were generated, of which 126 were discarded.

Wheeler et al. (29) use 'Bit Manipulation' to obtain the minimal cut sets of fault trees based upon the binary coding of events. The method uses a bottom-up algorithm and deals with fault trees containing arbitrary AND and OR logic and independent basic events. The authors state that their procedure will generate minimal cut sets and top event existence probability in an exact algebraic manner.

Firstly, to obtain the minimal cut sets, three types of logical redundancy must be identified and eliminated.

1. Redundant Factors (A.A=A)
2. Subset Redundancy (A+A.B=A)
3. Term Redundancy (A.B+B.A=A.B)

The minimal cut sets are obtained by a bottom-up sequential substitution procedure, similar to that of Semanderes (19), but at each substitution stage redundant factors are automatically eliminated using bit manipulation. The initial ordering of input information for coding requires that gates having only primary event inputs be listed first and that the remaining gates follow with a listing of their input gates. Each

primary event is assigned a single 1 in a unique position in a sequence of binary digits e.g.

A→100000
B→010000
C→001000
D→000100

Each individual cut set is then represented by the presence of the appropriate events e.g.

AB→110000

By using this coding, space required for the storage of intermediate results is minimised. In general, for a tree containing N primary events, any cut set representing any combination of these events requires a maximum of N bits for its storage. This is contrasted with word representation of events in which provision for the storage of N words is required. For a first order cut set the saving is in the ratio of bits per word depending on the computer used - for example, 36 (Honeywell) or 32 (IBM) to 1. 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 by FORTRAN. The output of an OR gate is the list of non-duplicated inputs.

With all the cut sets in binary word form, a simple logical test will identify redundant relationships among them. In addition to the OR operation , the Exclusive OR (EOR) operation can be used to compare the word bits (i.e. one or more bit positions of the words being compared must be different to be non-redundant). For the remaining types of redundancy (subset and term duplication) the following logic can be used.

For each cut set, beginning with the most dominant (i.e. the fewest primary event factors), use the selected cut set as a reference and test all others against it. For each pair, generate the logical OR combination and then the EOR combination of that result and the test cut set. If the result is zero, then the test term is redundant and is eliminated. To illustrate, let the first reference term be {A} and the test cut set be {A, B} (which is a redundant subset of A):

Reference (A)	1000		
Operation		OR	
Test (AB)	1100		
Result	1100		
Operation		EOR	
Test (AB)	1100		
Result	0000		Therefore AB is redundant

From this example it can be seen that testing a duplicate term such as AC against the reference term AC will also give a zero result. For other terms, which are neither a duplicate or a subset of the reference term, a result other than zero will be obtained, for example.

Reference (A)	1000		
Operation		OR	
Test (CD)	0011		
Result	1011		
Operation		EOR	
Test (CD)	0011		
Result	1000		Therefore CD is not redundant

The actual FORTRAN implementation uses the following logic for the task of comparing every term to all other terms:

(1) If "OR(TERM1, TERM2).EQ.TERM1" is true, TERM1 is redundant.

If the above test is false then:

(2) If "OR(TERM1, TERM2).EQ.TERM2" is true, TERM2 is redundant.

If neither test is true, no redundancy is detected.

By use of this logic, sorting of terms is avoided and testing is more efficient. Applying this procedure to all the cut sets for the top event results in the final minimal cut sets for the fault tree. The procedure has been implemented in a program called FAULTRAN. The data in table 2.2 is obtained using FAULTRAN implemented on a Honeywell 635 computer.

Number of Events	Number of Gates	Number of Cut Sets	Number of Minimal Cut Sets	Execution Time (Sec.)
17	21	256	112	4
34	43	512	224	16
51	64	768	335	40
68	85	1024	448	60

Table 2.2 Results of Four Example Fault Trees for FAULTRAN

Unfortunately the efficiency of neither the algorithm nor the program has been formally compared to alternatives. The authors state differences in computer processing speeds, types of computation performed, output provided and other factors make the comparison difficult. A limitation of this method may be that for large numbers of basic events the binary bits allocated to each event used by the computer program may be difficult to organise.

Rasmuson and Marshall (30) feel that computer implementation of algorithms that determine minimal cut sets for logic models prove less efficient than is desirable. Therefore, they produced an alternative method for determining the minimal cut sets of a fault tree, which makes more efficient use of computer memory. The gates of the fault tree are resolved in a deterministic manner, which provides a good standardised procedure. However, this deterministic method may be a disadvantage as more sorting is required to find the relevant gates. They state that the use of dynamic storage makes the program more flexible. Similar to Fussell and Vesely (20) they stress that the main goal of a fault tree algorithm is to obtain the minimal cut sets as quickly as possible, in the smallest amount of main core memory, without having to go to external devices. They believe that the efficient use of the main computer memory is the most important requirement for an efficient algorithm on contemporary computers. However, main core memory may not be so important on modern day computers where memory is cheap and available in larger quantities. The method proposed by Rasmuson and Marshall is called FATRAM (FAult Tree Reduction AlgorithM) and it is a top-down algorithm like MOCUS. The computer core requirements are minimised by careful selection of the gates to be resolved:

1. AND gates and OR gates with gate inputs are resolved first.
2. OR gates with only basic event inputs are resolved last.

The steps of the algorithm are:

1. Resolution begins with the top event. If the top event is an AND gate, all inputs are listed as one set. If it is an OR gate, inputs are listed as separate sets (same as MOCUS).
2. Iterate until all OR gates with gate inputs and all AND gates are resolved. OR gates with only basic event inputs are not resolved at this time.
3. Remove any non-minimal cut sets.
4. Process any repeated basic events remaining in the unresolved OR gates. For each repeated event do the following:
   a) The repeated event replaces all unresolved gates of which it is an input, to form new sets.
   b) These new sets are added to the collection.
   c) This event is removed as an input from the appropriate gates.
   d) Non-minimal cut sets are removed using the usual laws, i.e. X.X=X and X+X.Y=X.
5. Resolve the remaining OR gates. All sets are minimal cut sets.

Rasmuson and Marshall also discuss a 'weeding process', which is more commonly referred to as culling. Culling reduces computation by only evaluating the most important minimal cut sets, usually those up to a certain order (which is determined by the user), all other minimal cut sets are ignored. The justification for doing this is that cut sets of a high order tend to have a low probability of occurrence, and therefore do not make a significant contribution to the top event probability. The weeding process can be applied as the gates are resolved.

The paper by Rasmuson and Marshall compares FATRAM with MOCUS, the results of which are quite impressive. An example fault tree with 25 gates and 36 basic events (1 repeated), took 7.36s to evaluate 1,184 minimal cut sets and used 8,288 core (words) with MOCUS. However, FATRAM took 0.547s and used 112 core (words), clearly showing that FATRAM is more effective than MOCUS for this fault tree. To illustrate this method the minimal cut sets for the fault tree in figure 2.3 are determined.

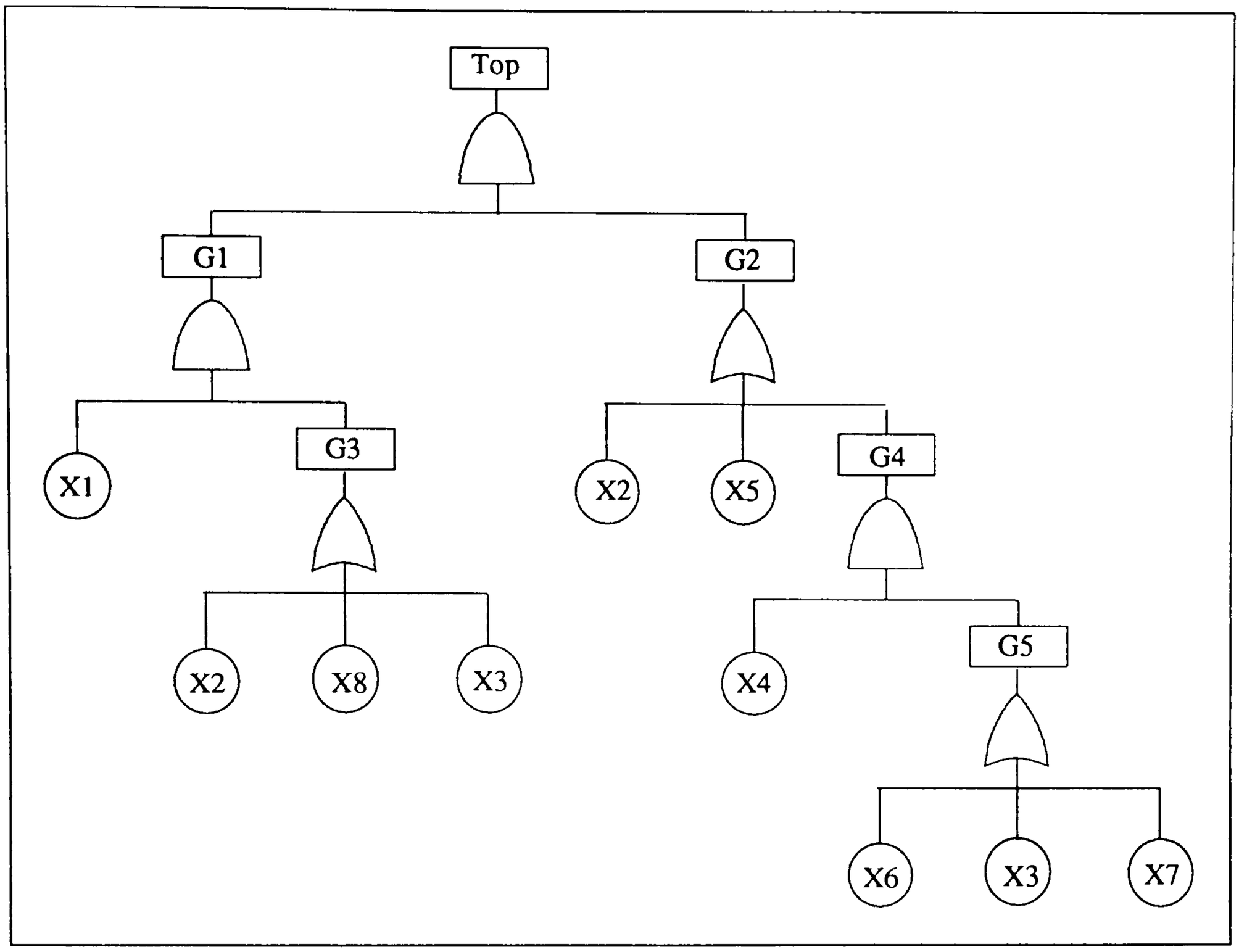


Figure 2.3 Example Fault Tree with Several Repeated Events

1) The Top Event is an AND gate, therefore all the inputs are listed as one set i.e. {G1, G2}.

2) G1 is an AND gate with basic event input X1 and gate input G3, thus G1 is resolved to give {X1, G3, G2}.

3) Both G3 and G2 are OR gates, but G3 has only basic event inputs therefore it is not yet resolved. Resolving G2, which has inputs X2, X5 and G4, each input creates new sets giving {X1, G3, X2}, {X1, G3, X5}, {X1, G3, G4}.

4) G4 is an AND gate with inputs X4 and G5. Resolving produces {X1, G3, X2}, {X1, G3, X5}, {X1, G3, X4, G5}.

5) The remaining gates, G3 and G5, are both OR gates with only basic event inputs. If any non-minimal cut sets existed they would be removed now. Repeated events are now handled. Basic event X2 is repeated and it is an input to G3. Therefore anywhere G3 occurs it is replaced by X2 and these additional sets added, i.e. existing already are {X1, G3, X2}, {X1, G3, X5} {X1, G3, X4, G5} and the additional ones are {X1, X2, X2} {X1, X2, X5} {X1, X2, X4, G5}. In {X1, X2,

X2} the redundant event X2 is removed giving {X1, X2}. Next, using step 4(c) of the algorithm G3 is altered by removing X2 to give an OR gate with only 2 inputs X3 and X8.

6) Non-minimal cut sets are deleted to give: {X1, G3, X5}, {X1, G3, X4, G5}, {X1, X2}.

7) Basic event input X3 is also a repeated event, it is an input to G3 and G5. Replace G3 and G5 in the sets by X3, creating additional sets to obtain {X1, G3, X5}, {X1, G3, X4, G5}, {X1, X2}, {X1, X3, X5}, {X1, X3, X4, X3}. The set {X1, X3, X4, X3} is reduced to {X1, X3, X4}. The gate definitions into which X3 is an input are altered like before. Thus, G3 has only one input X8, and G5 has inputs X6 and X7.

8) As there are no non-minimal cut sets and all repeated events have been handled, then the remaining OR gates, G3 and G5, are resolved. All the minimal cut sets have now been obtained:

   1) {X1, X8, X5} 2) {X1, X8, X4, X6} 3) {X1, X8, X4, X7}
   4) {X1, X2} 5) {X1, X3, X5} 6) {X1, X3, X4}.

A criticism of this paper may be that if a small simple fault tree is being analysed this method takes longer than MOCUS, however there is an advantage when it comes to analysing larger trees using a computer.

Zipf in his 1984 paper (31) provides a brief description of three different methods used by other authors to obtain the minimal cut sets of a fault tree. Two of the methods are analytical, one is a bottom-up algorithm and the other one is a top-down algorithm. The third method uses simulation. The author then proposes a new analytical algorithm, for the evaluation of the minimal cut sets, based on the findings of the three methods.

The simulation method uses Monte-Carlo Simulation which computes the most probable minimal cut sets of a fault tree. Zipf has implemented the algorithm in the CRESSC program, part of the code package RALLY by Güldner et al. (32). Since simulation reflects the actual failure events of the system, it can be assumed that those minimal cut sets which make the largest contribution to the average unavailability are obtained very quickly. The following advantages are given for the simulation method when comparing it to analytical programs:

1) A simulation program is relatively simple to develop.
2) In a simulation program, the possibility of a restart can be achieved easily, i.e. it is possible to continue the program at the point of interruption without the loss of any data.

Additionally Zipf states that the processing times, in comparison to the analytical programs, are shorter for the simulation program. This is contradictory to the general experience of simulation programs and to the later findings of this paper, hence this statement is questionable.

The disadvantages of a simulation program are also given as:

1) There is no guarantee that all the important minimal cut sets will be obtained.
2) For low failure probabilities ($<10^{-5}$) of a highly redundant system, an estimate of the error due to the non-considered minimal cut sets is difficult to achieve.

The description given of the top-down algorithm is the same as that of MOCUS and in addition a cut-off procedure is also discussed. A cut-off algorithm recognises and eliminates unimportant cut sets during substitution. The easiest cut-off method is to eliminate minimal cut sets whose size (number of events) is greater than some pre specified number, n. However a more precise approach is to calculate the significance of a minimal cut set using the failure probability of the components of this set. The great advantage of this algorithm is that only the important cut sets have to be developed completely. Further an approximation of the error caused by the cut-off procedure can be obtained.

The bottom-up algorithm described by Zipf is the one implemented in the SALP-3 (Astolfi et al. (9)) and SALP-MP (Astolfi et al. (55)) programs. On comparing the three methods, the author found the analytical methods to be more efficient than the simulation method, i.e. took shorter computation time on a computer to calculate the essential minimal cut sets. The final part of the paper gives the basic ideas of a new analytical algorithm which is a combination of the two analytical algorithms described with some additional improvements. The basic ideas of this new algorithm are:

1) Modularisation of the fault tree and a special fault tree optimisation for the algorithm (the author does not clarify on this modularisation or the optimisation procedure).

2) Convert the fault tree to an alternating AND-OR sequence of gates, i.e. each AND gate has only OR gates as inputs and vice versa.
3) Generate a subroutine representing the Boolean logic of the fault tree.
4) Determine a cut-off level for the top event, ξ_a.
5) Calculate the (maximum) unavailability of each gate.
6) Use a top-down algorithm as discussed earlier (MOCUS), where all gates of an undeveloped set are replaced by their inputs.
7) Eliminating all (undeveloped) cut sets which have an assigned value lower than cut off level ξ_a.
8) If a gate is replaced by one or more components, we use the Boolean subroutine to decide whether the components of that undeveloped set are already a cut set.

As this 'new' approach has not been fully developed by Zipf its efficiency cannot be compared with other methods.

Liminios and Ziani (10) state, like many other fault tree researchers, that the main goal of a fault tree algorithm is to obtain the minimal cut sets as efficiently as possible. Their method for cut set reduction is based on the partition of the cut sets into two families; those with repeated events and others (everything else). The algorithm has been implemented as a computer program which has been used in conjunction with MOCUS. An advantage of the method by Liminios and Ziani is that it can be combined with other reduction algorithms. In techniques, such as MOCUS, it has been recognised that deleting non-minimal cut sets is a time consuming task. Further, when a fault tree does not contain repeated events, no reduction is required. Therefore the minimal cut sets are obtained by a simple development of the top event Boolean function. The Liminios and Ziani approach deals with fault trees that contain repeated events. It improves the MOCUS top-down algorithm by obtaining all the minimal cut sets faster. The improvement is based on reducing the number of set comparisons required to find the minimal cut sets. The assumption is that the real system has been modelled by a coherent fault tree.

A summary of the algorithm is given below:

1) Evaluate the cut sets using MOCUS.
2) Split the cut sets into two families, K1 and K2.
   K1 is the family of cut sets containing repeated events and K2 is the family of all other cut sets, i.e. the cut sets containing no repeated events.

K2 is already minimal but K1 is not, therefore reduce K1 using the Boolean properties: X.X=X and X+X.Y=X. K1 is now minimal and is called K1*.

3) Combine the two families, i.e. K1*∪K2, which gives all the minimal cut sets of the fault tree.

A small example is given in reference (10) where 36 cut sets are obtained for a fault tree, using MOCUS, 20 of these cut sets belong to K1 and 16 belong to K2. Therefore the minimal number of comparisons in the K1 family is the sum of an arithmetic progression of the first n-1 natural numbers, where n is the number of cut sets in K1. Therefore 190 comparisons are made, whereas 630 comparisons would be needed if the cut sets were not split into two families. Additionally the algorithm can be used with FATRAM (30) and with the algorithm of Benjiamin et al. (28) to increase their efficiency.

This algorithm is very useful when combined with a conventional fault tree analysis technique such as MOCUS. It can clearly reduce the number of comparisons that need to be made simply by finding the repeated events in the fault tree.

2.3 Modules of Coherent Fault Trees

In order to simplify the fault tree analysis process, research has been undertaken into reducing the complexity of the fault tree by the use of modules. Modules can enable the fault tree to be 'decomposed' or reduced into a more manageable form. Chatterjee 1975 (59) presents in his paper an algorithm to obtain the 'finest modular fault tree representation' which is an equivalent representation of a given fault tree in terms of modular trees. The method determines whether or not the system is decomposable and requires of the order of n steps for a tree with n basic events. Chatterjee defines a module as a set of components which behave as a 'super component', i.e., knowledge of the state of the super components will determine the state of the system. Using this definition a single component and the whole system are always modules. Additionally in the paper any other module of a system is called a proper module and a system that does not have a proper module is called a prime system (i.e. non-decomposable).

The finest modular representation for a fault tree is an equivalent tree with the following properties:

1. All subtrees are independent.

2. The logic function associated with each gate is either a) non-decomposable, b) AND with no inputs from an AND gate immediately below it, or c) OR with no inputs from an OR gate immediately below it.

Note that point (2) implies that connecting gates with the same logic are coalesced.

The algorithm to modularise the fault tree and obtain the minimal cut sets is described as follows:

Consider any AND or OR gate. The non-replicated basic event inputs to that gate form a module. Condense these to form a super component and proceed. The super component is henceforth considered as a basic event. Continue this process until no further condensation is possible.

To illustrate the method by Chatterjee consider the gate G1 in figure 2.4. Assume this gate has been taken from a larger fault tree where basic events X1, X2 and X3 are not replicated but X4 is repeated elsewhere in the fault tree. As X1, X2 and X3 are not repeated then {X1, X2, X3} forms a modular set. Therefore the corresponding module is condensed to a super component, say S, and the tree re-drawn as shown in figure 2.5.

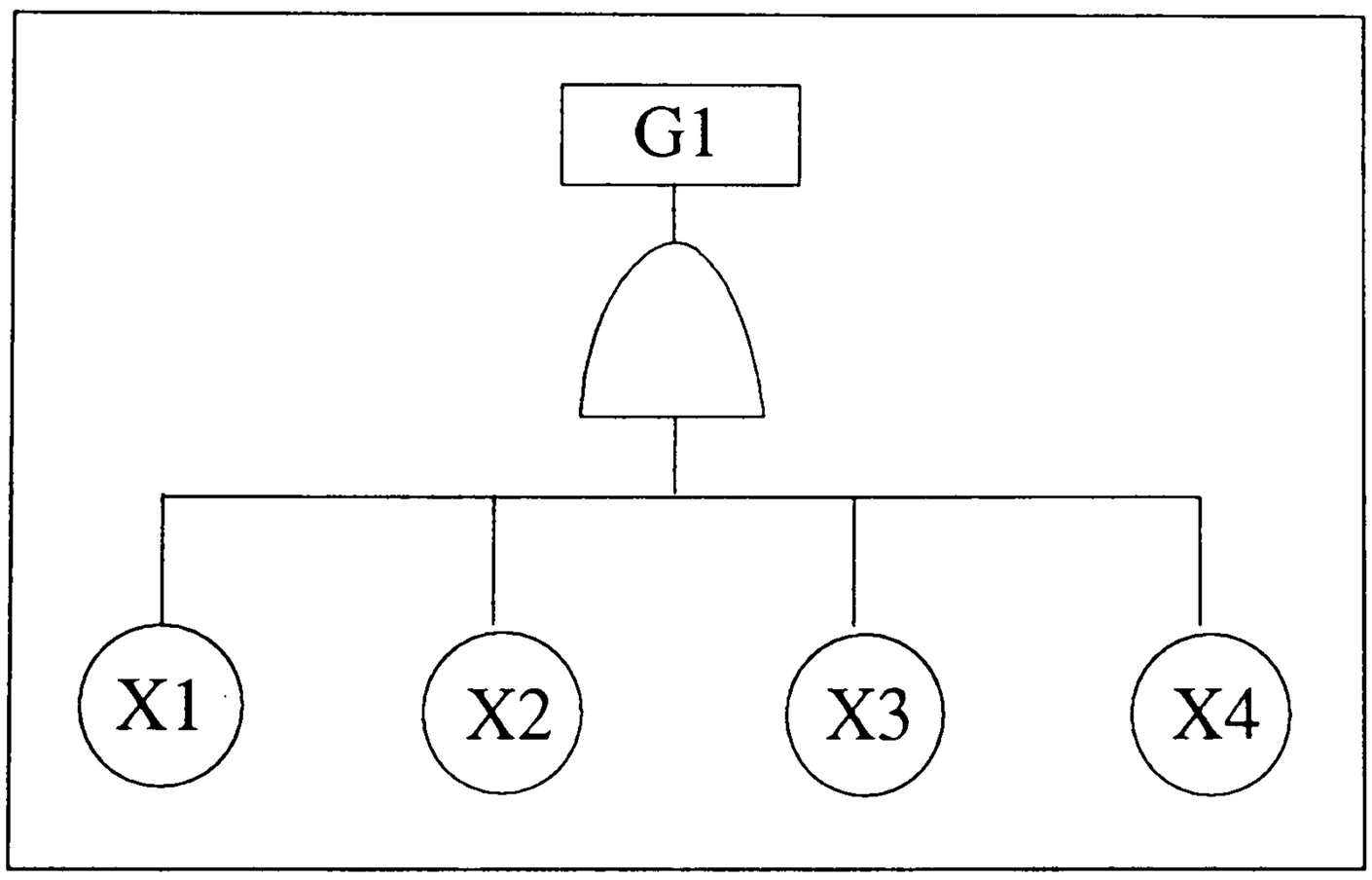


Figure 2.4 An Example Gate which can be Modularised

When further identification of modules is not possible in the fault tree, the minimal cut sets for each module are determined by a method such as MOCUS. The minimal cut sets of the modularised fault tree can then be determined and expanded into the minimal cut sets of the original system.

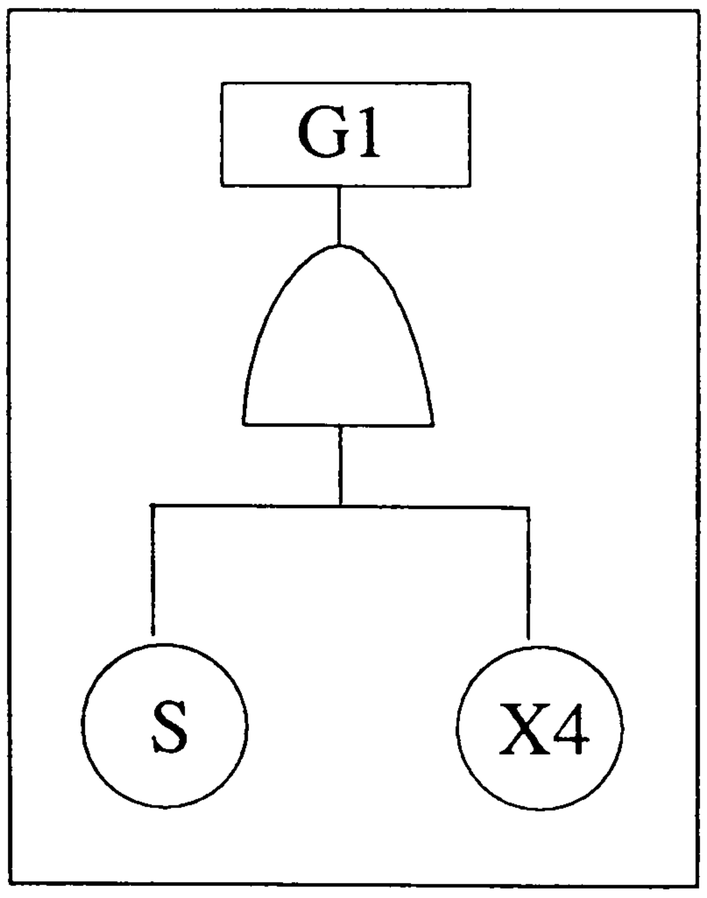


Figure 2.5 Modularised Gate

G1

The "FAUNET" modularisation approach of Platz and Olsen (44) occurs in 3 stages:

Stage 1 Contraction

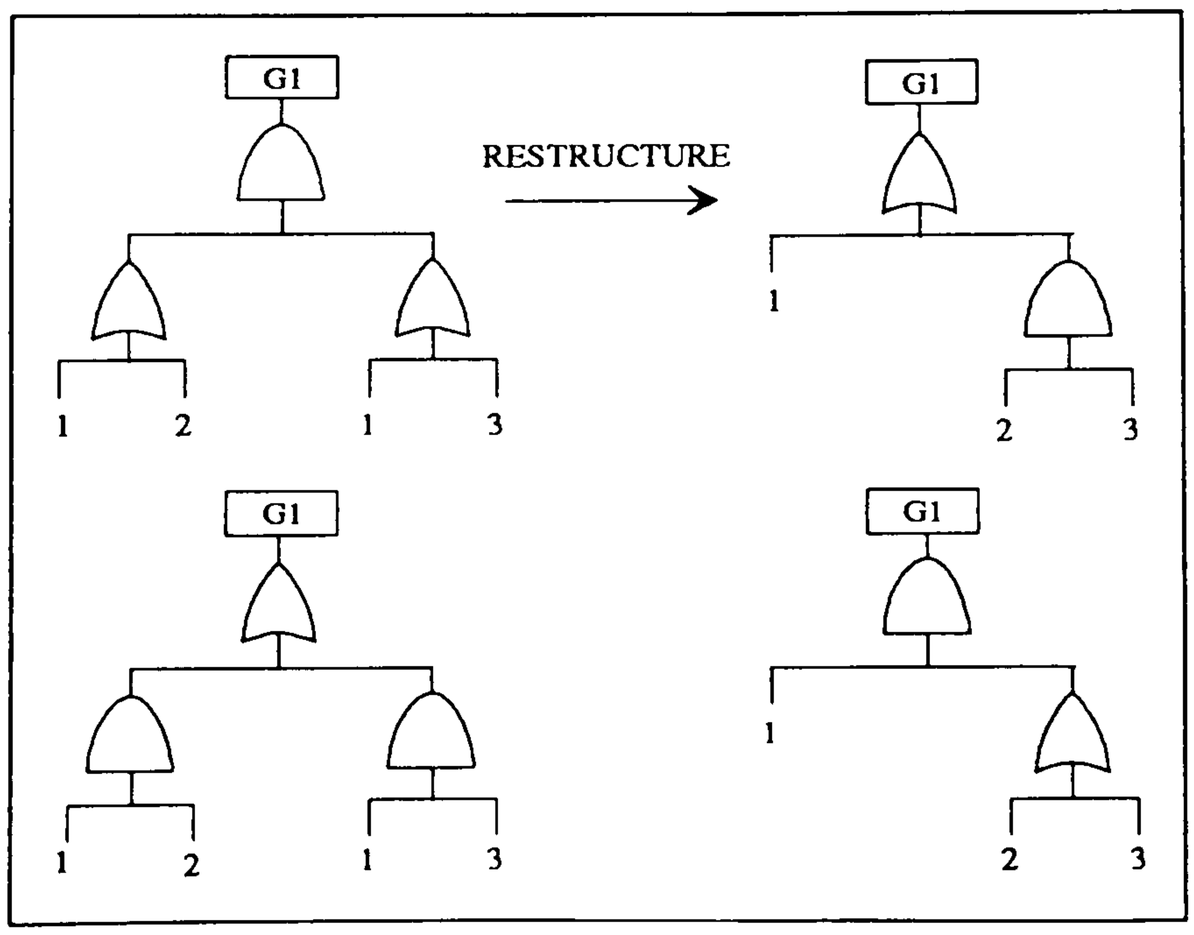
Subsequent gates of the same type are contracted to form a single gate. The tree structure then becomes an alternating sequence of OR and AND gates.

Stage 2 Factorise

Identification of primitive factors:- pairs of basic events which will always occur together in the same gate type. Replace with a complex event.

Stage 3 Extraction

Searches for structures:



The above 3 stages are repeated until the fault tree cannot be reduced any further. The minimal cut sets of the modularised fault tree are then evaluated in terms of complex events. Finally these complex events need to be expanded back in terms of the original basic events using a MOCUS type approach. The technique of Platz and Olsen is a good methodical approach to modularising the fault tree which can reduce both memory and time requirements.

Rosenthal (46) in 1980 also deals with decomposition methods for fault tree analysis. He defines a module of a fault tree as a set of at least two events which has only one output to the rest of the tree and no input to the rest of the tree. Rosenthal states that the single output event of a module is called a module top and proper modules, if there are any, can be used to split the analysis problem effectively into disjoint pairs. Once a module has been analysed, either by calculating the probability of the module or computing the cut sets, a reduced system can be defined in which the module is treated as a basic event.

The cut sets of each module can be enumerated separately from the cut sets of the reduced system. The analyst is then presented with a list of cut sets composed of higher level events plus elaboration of each module event in terms of lower level events.

In addition to using modules to decompose the fault tree Rosenthal describes the use of a 'split candidate'. A 'split candidate' is an output event such that some but not all of its input events can be grouped together to create a 'super component'.

The decomposition method of Rosenthal is best described by the use of an example. By this method the fault tree in figure 2.6 can be reduced to the fault tree in figure 2.7. In this example gate G2 is a candidate for splitting.

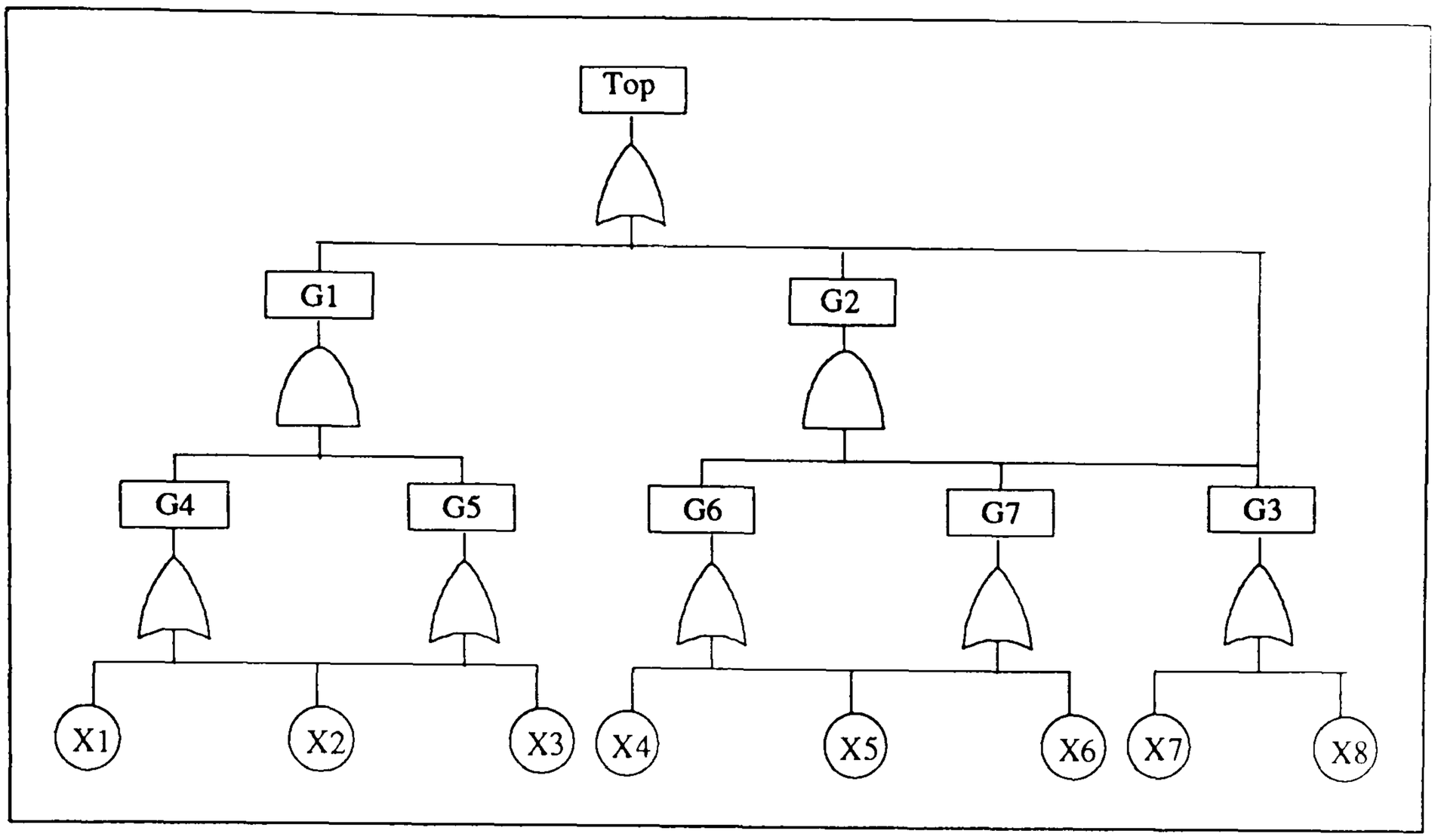


Figure 2.6 Example Fault Tree which can be Modularised

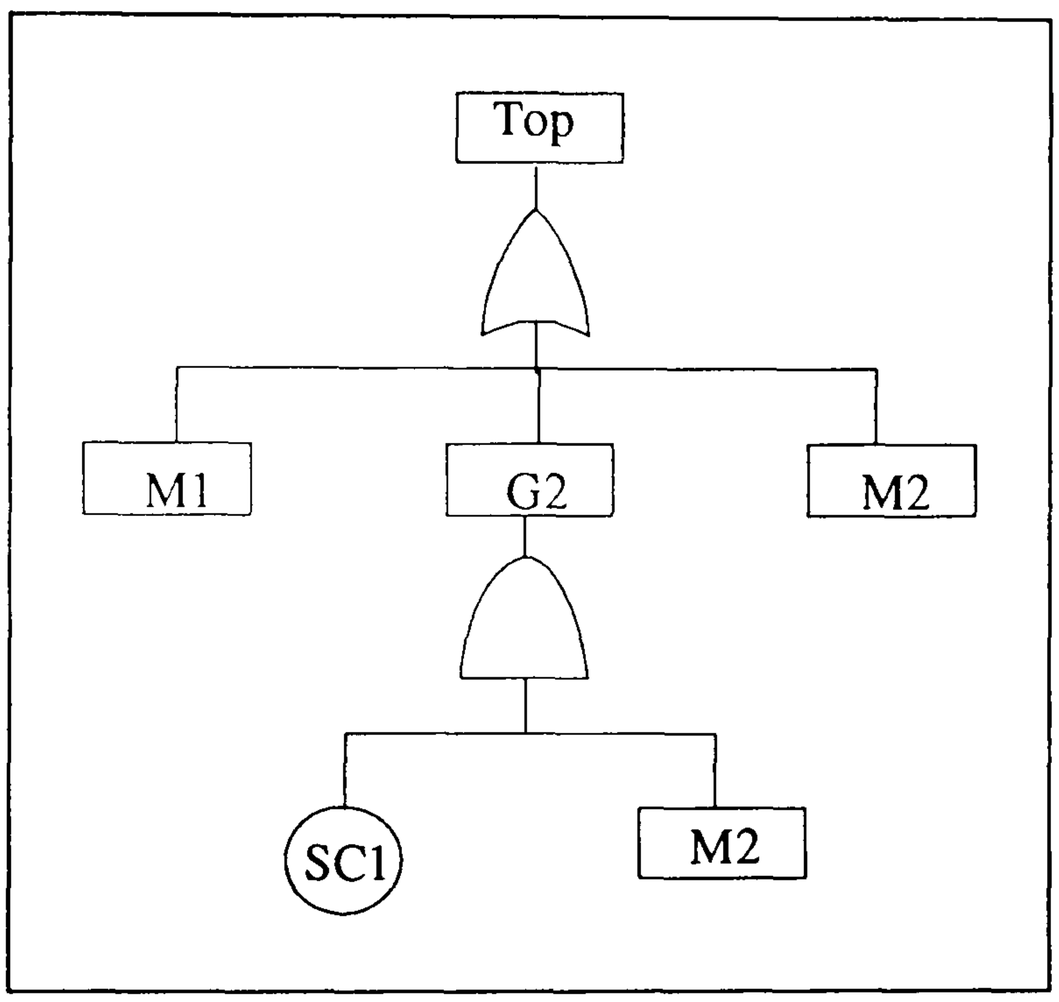


Figure 2.7 The Decomposed Fault Tree of Figure 2.6

where:

M1 (module 1) is an AND gate and has gate inputs G4 and G5
SC1 (super component 1) is an AND gate and has gate inputs G6 and G7
M2 (module 2) is an OR gate and has basic event inputs X7 and X8

Completing the analysis of the reduced fault tree illustrated in figure 2.7 using a top-down approach such as MOCUS one obtains:

Top=M1+G2+M2

=M1+(SC1.M2)+M2

Top=M1+M2 (note redundancies have been eliminated)

where

M1=G4.G5

=(X1+X2).(X2+X3)

M1=X1.X3+X2 (redundancies have been eliminated)

M2=X7+X8

Therefore the minimal cut sets of the original fault tree shown in figure 2.6 are:

(1) {X1, X3}
(2) {X2}
(3) {X7}
(4) {X8}

It can be seen that as a result of modularising the fault tree the analysis process has been simplified. In this example redundancies were eliminated at the super component level which eradicates the need to substitute all the gates in terms of basic events.

2.4 Minimal Cut Set Evaluation for Non-Coherent Fault Trees

When fault tree structures feature working states i.e., NOT failed states, then they may be non-coherent. The qualitative analysis of this type of fault tree produces prime implicants. The methods which can be used to address this type of problem are described in this section.

Nelsons method (11) was initially developed to manipulate logic functions to obtain all the prime implicants. This method was then applied to non-coherent fault trees by performing the operation d(d(F)), where d(F) corresponds to the dual of the Boolean expression F, for the top event of the fault tree. The method can be described more formally in the following two steps.

Step 1: Complement F to give $\overline{F}$, expand $\overline{F}$ into disjunctive normal form (sum-of-products), drop zero products ($p.\overline{p}=0$), repeated literals (p.p=p) and subsuming products (p+p.q=p), and call the result $\overline{\phi}$.

Step 2: Complement $\overline{\phi}$ to give ϕ, expand ϕ into disjunctive normal form, drop zero products, repeated literals and subsuming products and call the result Θ.

Nelson proved that Θ is the sum of all, and only, the prime implicants of F. To illustrate the method refer to the non-coherent fault tree shown in figure 2.8.

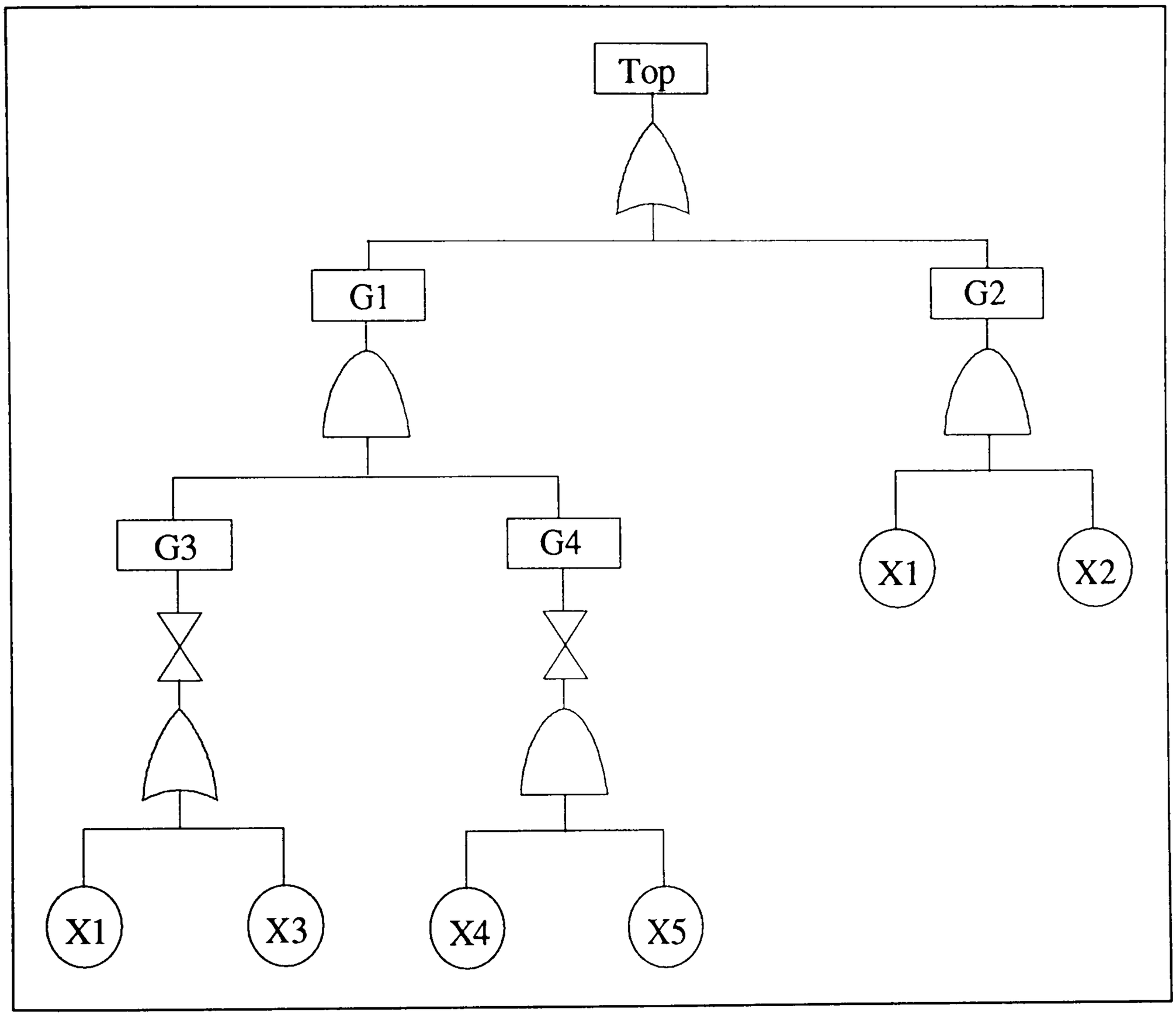


Figure 2.8 Non-coherent Fault Tree Example

The top event Boolean expression can be obtained in a top-down manner:

$$\begin{aligned}
\text{Top} &= G1 + G2 \\
&= G3.G4 + X1.X2 \\
&= (\overline{X1 + X3}).(\overline{X4.X5}) + X1.X2
\end{aligned}$$

using De Morgans Laws

$$\begin{aligned}
&= (\overline{X1}.\overline{X3}).(\overline{X4} + \overline{X5}) + X1.X2 \\
&= \overline{X1}.\overline{X3}.\overline{X4} + \overline{X1}.\overline{X3}.\overline{X5} + X1.X2
\end{aligned} \qquad (2.1)$$

Taking the dual of the top event expression:

$$d(Top)=\overline{\overline{X1}.\overline{X3}.\overline{X4}+\overline{X1}.\overline{X3}.\overline{X5}+X1.X2}$$
$$=\overline{(\overline{X1}.\overline{X3}.\overline{X4})}.\overline{(\overline{X1}.\overline{X3}.\overline{X5}+X1.X2)}$$
$$=(X1+X3+X4).(\overline{(\overline{X1}.\overline{X3}.\overline{X5})}.\overline{(X1.X2)})$$
$$=(X1+X3+X4).(X1+X3+X5).(\overline{X1}+\overline{X2})$$
$$=(X1+X3+X4.X5).(\overline{X1}+\overline{X2})$$
$$=(X1.X3+X1.X4.X5+X1.X2+X2.X3+X2.X4.X5)$$

Again taking the dual of d(Top), i.e. d(d(Top))=Top:

$$d(d(Top))=\overline{(\overline{X1}.X3+\overline{X1}.X4.X5+X1.\overline{X2}+\overline{X2}.X3+\overline{X2}.X4.X5)}$$
$$=\overline{(\overline{X1}.X3+\overline{X1}.X4.X5)}.\overline{(X1.\overline{X2}+\overline{X2}.X3+\overline{X2}.X4.X5)}$$
$$=\overline{(\overline{X1}.X3)}.\overline{(\overline{X1}.X4.X5)}.\overline{(X1.\overline{X2}+\overline{X2}.X3)}.\overline{(\overline{X2}.X4.X5)}$$
$$=(X1+\overline{X3}).(X1+\overline{X4}+\overline{X5}).\overline{(X1.\overline{X2})}.\overline{(\overline{X2}.X3)}.(X2+\overline{X4}+\overline{X5})$$
$$=(X1+\overline{X3}).(X1+\overline{X4}+\overline{X5}).(\overline{X1}+X2).(X2+\overline{X3}).(X2+\overline{X4}+\overline{X5})$$
$$=(X1+\overline{X3}.\overline{X4}+\overline{X3}.\overline{X5}).(\overline{X1}.\overline{X3}+X2).(X2+\overline{X4}+\overline{X5})$$
$$=(X1.X2+\overline{X1}.\overline{X3}.\overline{X4}+\overline{X3}.\overline{X4}.X2+\overline{X3}.\overline{X1}.\overline{X5}+X2.\overline{X3}.\overline{X5}).(X2+\overline{X4}+\overline{X5})$$
$$=X1.X2+\overline{X1}.\overline{X3}.\overline{X4}+X2.\overline{X3}.\overline{X4}+\overline{X1}.\overline{X3}.\overline{X5}+X2.\overline{X3}.\overline{X5}$$

Therefore the full set of prime implicants of Top are:

(1) $\{X1,X2\}$, (2) $\{\overline{X1},\overline{X3},\overline{X4}\}$, (3) $\{X2,\overline{X3},\overline{X4}\}$, (4) $\{\overline{X1},\overline{X3},\overline{X5}\}$,
(5) $\{X2,\overline{X3},\overline{X5}\}$.

In this way the extra prime implicants $\{X2,\overline{X3},\overline{X4}\}$ and $\{X2,\overline{X3},\overline{X5}\}$ are obtained, these were excluded in the original top-down expression (2.1)

Bennetts (12) proposes an algorithm which is given in two parts. The first part deals with the algorithm for generating a near-minimal disjunctive normal form from a description of the fault tree and discusses its computer implementation. The second part involves the necessary modification to this expression if it is to be interpreted as a probability relationship. The discussion here is concerned with the qualitative fault tree analysis method outlined in part one of Bennetts paper. The quantification technique shown in part two is described in Chapter 3.

Bennetts states that the basic strategy of the qualitative algorithm is: 'to derive an algebraic description of the output as a function of the inputs using a reverse Polish

notation and then, to "unpack" this expression into its equivalent disjunctive normal form'. A table is given (reproduced here in table 2.3) which is referred to as a 'library' that gives standard gate types together with their individual reverse Polish functions.

Number of Basic Event Inputs	Gate Type	Reverse Polish Notation
1 input	INVERT	$Z = \overline{X1}$
2 inputs	AND	Z = X1X2 AND(2)
2 inputs	OR	Z = X1X2 OR(2)
2 inputs	NAND	$Z = \overline{X1}\,\overline{X2}$ OR(2)
2 inputs	NOR	$Z = \overline{X1}\,\overline{X2}$ AND(2)
3 inputs	AND	Z = X1X2X3 AND(3)
3 inputs	OR	Z = X1X2X3 OR(3)
3 inputs	NAND	$Z = \overline{X1}\,\overline{X2}\,\overline{X3}$ OR(3)
3 inputs	NOR	$Z = \overline{X1}\,\overline{X2}\,\overline{X3}$ AND(3)
2 inputs	XOR	$Z = X1\overline{X2}$ AND(2) $\overline{X1}X2$ AND(2)OR(2)

Table 2.3 'Library' of Reverse Polish Functions

As can be seen from table 2.3, NAND and NOR gates have been re-configured using De Morgans Laws and an INVERT gate is simply eliminated by complimenting its input. Using the INVERT gate allows the tree structure to be defined in terms of AND and OR gates only, which simplifies the procedure. Also the number in brackets indicates how many preceding terms or literals are to be ANDed or ORed together.

The complete reverse Polish expression for the fault tree is derived by a bottom-up procedure. The algorithm works from the primary outputs, (i.e. those gates at the bottom of the tree with only basic event inputs) with successive substitution for gate inputs appearing higher up the tree. The procedure ends when all gates are expressed in terms of basic events. As an example consider the fault tree shown in figure 2.9.

The reverse Polish expression for the fault tree in figure 2.9 is constructed in the following fashion:

G3=X2X3 OR(2)
G1=X1G3 AND(2)=X1X2X3 OR(2) AND(2)
G2=X2X4 OR(2)

Top=G1G2 AND(2)

Top=X1X2X3 OR(2) AND(2) X2X4 OR(2) AND(2)

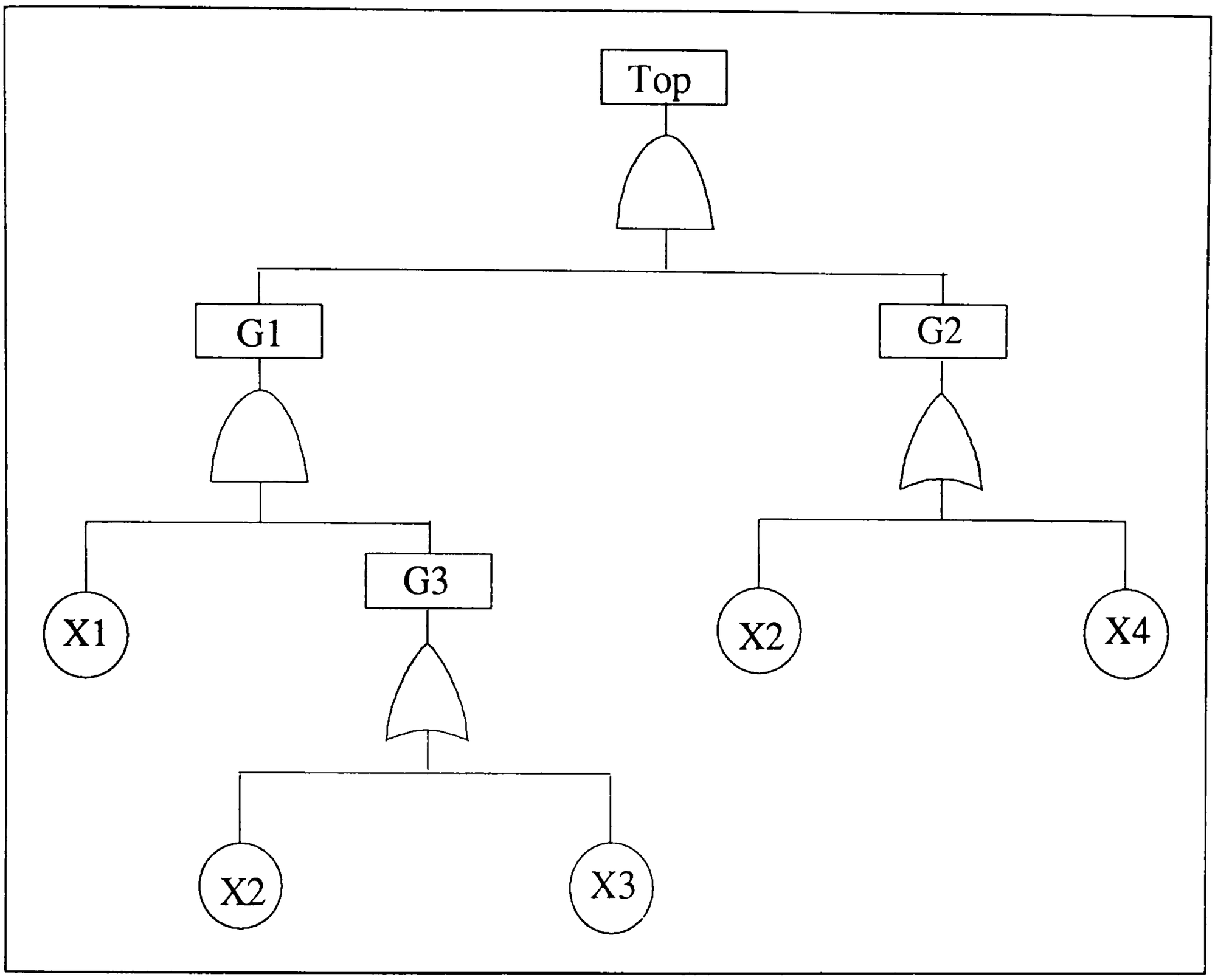


Figure 2.9 Example Fault Tree

This expression can be unpacked to yield a corresponding disjunctive normal form. To do this, the expression is read from left to right and when an operator is reached it acts on the number of preceding terms specified in the brackets. The OR operator applies the Boolean operation of disjunction (sum) to the preceding terms while the AND operator applies the Boolean operation of conjunction (product) to the preceding terms. Unpacking Top gives:

Top=X1X2X3 OR(2) AND(2) X2X4 OR(2) AND(2)

The first OR operator gives (X2+X3)

The first AND operator gives X1.(X2+X3) = X1.X2+X1.X3

The second OR operator gives (X2+X4)

The second and last AND operator gives

(X1.X2+X1.X3).(X2+X4)=X1.X2+X1.X3.X4

Top=X1.X2+X1.X3.X4

Note that intermediate results involving the AND operator are always converted into their equivalent disjunctive normal form (in the programmed version of the algorithm, this process is carried out by a routine known as MULTIPLY). An important aspect of this process is that all intermediate and final expressions are subjected to a search to eliminate various forms of logical redundancy. To do this the following rules are applied:

Rule 1 : $X+X.Y = X$

Rule 2 : $X.Y+X.Y = X.Y$

Rule 3 : $X.Y+X.\overline{Y} = X$

The programmed algorithm contains a routine (referred to as ORCHEK) that exhaustively applies these identities to the terms in a disjunctive normal form expression. In a reverse Polish expression adjacent operators of the same type can be reduced to a single operator with a suitable composite index, i. The general form of this index for n repeated operators is:

$$i = \left(\sum_{j=1}^{n} i_j\right) - (n-1) \tag{2.2}$$

By applying equation (2.2), the overall length of the reverse Polish expression is reduced and also the number of operator evaluations in the unpacking process is reduced. Bennetts used list processing programming techniques and bit manipulative facilities which enabled a very compact storage system for the Boolean data. It is stated that this reduced core store and was instrumental in increasing the execution speed. The remainder of part one of the paper describes in detail the overall algorithm and the routines ORCHEK and MULTIPLY.

I feel that two important aspects of this technique are, NOT gates do not need to be "pushed down" the tree using De Morgans Laws and XOR gates can be readily converted using the library of reverse Polish expressions. Also I believe it is worth mentioning that there is no unique reverse Polish expression for a certain fault tree. Subtle differences may be found depending on which gate inputs are written down first (i.e. the left most ones first or the right most ones first). Again consider the fault tree shown in figure 2.9, it is also possible to obtain the expression:

Top=G2G1 AND(2)=X2X4 OR(2) G3X1 AND(2) AND(2)
=X2X4 OR(2) G3X1 AND(3) (using general index formula)

=X2X4 OR(2) X2X3 OR(2) X1 AND(3)

First operator gives (X2+X4)

Second operator gives (X2+X3)

Third operator gives (X2+X4).(X2+X3).X1=X1.X2+X1.X3.X4

However it is important to note that the same resulting disjunctive normal form is obtained when the expression is unpacked.

Hulme and Worrell (14) proposed a prime implicant algorithm with factoring, which proves more efficient than Nelsons (11) algorithm, to find the prime implicants of a Boolean function. The algorithm consists solely of Boolean formula manipulation and is identical to Nelsons algorithm, with the addition of factoring just prior to each complementation. The prime implicant algorithm with factoring is:

Step 1: Factor anywhere possible in F, complement, expand into disjunctive normal form, drop zero products ($p.\overline{p}=0$), repeated literals ($p.p=p$) and subsuming products ($p+p.q=p$), and call the result $\overline{\phi}^1$.

Step 2: Factor anywhere possible in $\overline{\phi}^1$, complement, expand into disjunctive normal form, drop zero products, repeated literals and subsuming products, and call the result Θ^1.

Hulme and Worrell prove that Θ^1 is equal to the logical disjunction of all, and only, the prime implicants of F. The advantage of factoring before taking each complement is that the ensuing expansion yields fewer terms. This not only saves time during the expansion, but it also reduces the time required to check for zero products, repeated literals and subsuming products. To illustrate the difference between the algorithms of Nelson, and Hulme and Worrell consider the following example:

$$F = a.b + \overline{b}c.d + \overline{d}c$$

Following Nelsons algorithm, complement F to give:

$$\overline{F} = (\overline{a} + \overline{b}).(b + \overline{c} + \overline{d}).(d + \overline{c})$$

Expand into a disjunction of twelve terms and simplify to form:

$$\overline{\phi} = \overline{a}b.d + \overline{a.c} + \overline{b.c}$$

Repeating this same sequence for $\overline{\phi}$, complementing gives:

$$\phi = (a + \overline{b} + \overline{d}).(a + c).(b + c)$$

Expand into a disjunction of twelve terms and simplify to give the prime implicant sum:

$$\Theta = a.b + a.c + \overline{b}c + \overline{d}c$$

Next consider Hulme and Worrell's algorithm with factoring, F is first factored into:

$$F = a.b + (\overline{b}.d + \overline{d}).c$$

Complementing gives:

$$\overline{F} = (\overline{a} + \overline{b}).((b + \overline{d}).d + \overline{c})$$

Expand into a disjunction of only six terms and simplify to form:

$$\overline{\phi}^1 = \overline{a}.b.d + \overline{a}.\overline{c} + \overline{b}.\overline{c}$$

Notice the saving in the number of terms when expanding $\overline{F}$, due simply to factoring a single literal from two terms of F. Notice also that $\overline{\phi}^1 = \overline{\phi}$. Moving on to step 2, factoring gives:

$$\overline{\phi}^1 = \overline{a}(b.d + \overline{c}) + \overline{b}.\overline{c}$$

Complementing:

$$\phi^1 = (a + (\overline{b} + \overline{d})c)(b + c)$$

Expand into a disjunction of six terms and simplify to give:

$$\Theta^1 = a.b + a.c + \overline{b}c + \overline{d}c$$

Again there has been a significant saving in the computational effort.

Hulme and Worrell provide a proof of the prime implicant algorithm with factoring in their paper.

In the same year as Bennetts published his work (1975) reference (12), Worrell (13) developed the Set Equation Transformation System (SETS) which generates set equations directly, or by logical combination of other set equations through a process of substitution. It also reduces set equations by the application of set identities. The operations allowed in an equation are the set operations of intersection, union, and complement, which correspond to the Boolean operations of conjunction, disjunction, and negation respectively. Since the processing that can be accomplished using SETS is valid for any Boolean algebra, the system is useful for processing the logic equations derived from fault trees. Hence the cut sets or implicants of a fault tree can be expressed as a set equation, then after reduction by the application of set identities, the minimal cut sets or prime implicants can be obtained. The SETS algorithm is similar to that of MOCUS, although in addition to AND and OR gates it can deal with XOR and special gates. Special gates are defined as those gates that allow the use of complemented events and they also provide a way that any logical combination can be specified. When a special gate is used in a fault tree, a Boolean equation is given as the definition of the gate. The equation defines the logical combination of the input events that will produce the output event of the gate.

In the SETS algorithm, AND gates use the operation of intersection on their inputs, i.e. for the inputs X1, X2, ..., Xn the representation of the AND gate will be $X1 \cap X2 \cap \ldots\ldots\ldots\ldots\ldots \cap Xn$. OR gates use the union operation, i.e. $X1 \cup X2 \cup \ldots\ldots\ldots\ldots\ldots \cup Xn$. While for two inputs, X1 and X2, to an XOR gate the output event is given by $(X1 \cap \overline{X2}) \cup (\overline{X1} \cap X2)$.

When the fault tree structure is input to the program using the SETS method each gate is assigned a set representation of its inputs. This equation can contain intermediate or basic events, e.g. the set representation or equation of an OR gate G1, with gate inputs G2, G3 and basic event input X1 will be:

$$G1 = G2 \cup G3 \cup X1$$

Once all the gates have been dealt with in this way, the top event equation is then considered and a substitution procedure is initiated. Each intermediate event is substituted by the equation which defines that gate. This process continues until the top event set equation consists only of basic events. The resulting equation should be a "union of intersections" expression with each of the intersection terms corresponding

to the implicants of the fault tree. Once the top event equation has been formed a few reduction rules are applied to the expression, these are:

(1) $P \cap P = P$

(2) $P \cap \overline{P} = 0$

(3) $P \cup P \cap Q = P$

Having applied the above reduction rules the resulting expression is composed of a union of the sets which represent the prime implicants of the fault tree. It is important to note that when an equation is monoform in all of its variables, (i.e. if the literal X1 occurs in the equation, then the literal $\overline{X1}$ does not occur, or vice versa), then the prime implicants of the related function can be determined by applying the identities $P \cap P = P$ and $P \cup (P \cap Q) = P$ to the top event equation. The result is a union of all the prime implicants of the function. If the equation is not monoform in all its variables more complicated techniques must be used to determine the prime implicants, such as the technique of Hulme and Worrell (14). To illustrate the method consider the fault tree in figure 2.10.

$$Top = G2 \cap X1 \cap G3$$

$$G2 = X2 \cup X3$$

$$G3 = G4 \cap X4$$

$$G4 = X1 \cup X3$$

Substituting and expanding the gates in Top gives:

$$Top = (X2 \cup X3) \cap X1 \cap (G4 \cap X4)$$

$$= (X2 \cup X3) \cap X1 \cap ((X1 \cup X3) \cap X4)$$

$$= (X2 \cup X3) \cap X1 \cap (X1 \cap X4 \cup X3 \cap X4)$$

$$= (X2 \cap X1 \cup X3 \cap X1) \cap (X1 \cap X4 \cup X3 \cap X4)$$

$$= (X2 \cap X1 \cap X1 \cap X4 \cup X2 \cap X1 \cap X3 \cap X4 \cup X3 \cap X1 \cap X1 \cap X4 \cup X3 \cap X1 \cap X3 \cap X4)$$

Applying the reduction rules to this expression gives:

$$= (X2 \cap X1 \cap X4 \cup X3 \cap X1 \cap X4)$$

Therefore this fault tree has two minimal cut sets which are {X1, X2, X4} and {X1, X3, X4}.

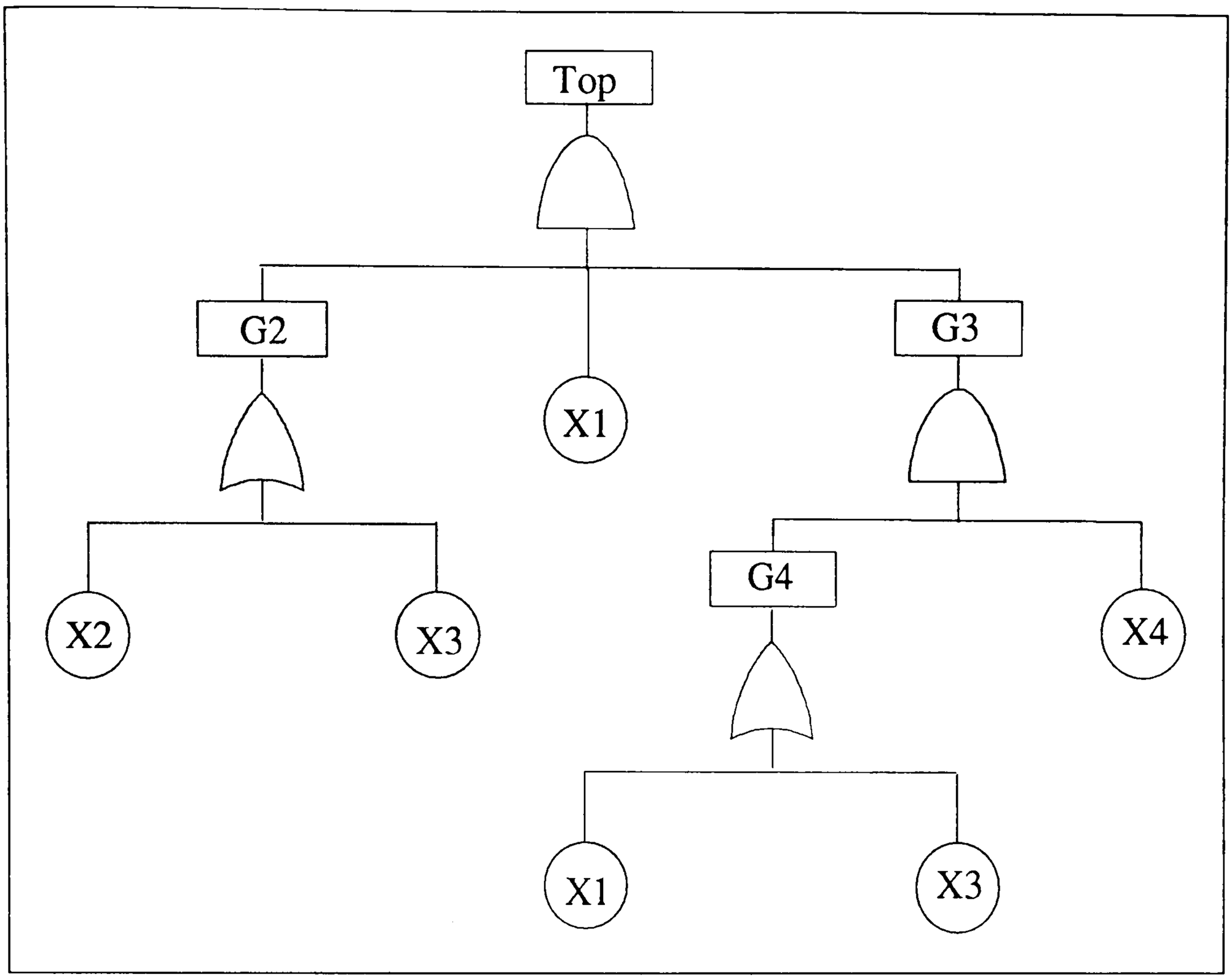


Figure 2.10 Example Fault Tree

Although this method is quite simplistic and easy to understand, an obvious disadvantage lies in the substitution and expansion procedure followed by the application of the reduction rules to the expression for the top event. When the fault tree has a large number of cut sets and repeated events this stage may take extensive computation time, therefore it may be more advantageous to expand and apply the reduction rules at each gate of the fault tree rather than at the top event stage.

The algorithm of Astolfi et al. (21) developed in 1978 simultaneously uses qualitative and quantitative information about the fault tree during the analysis. The algorithm has two main steps which are:

1) search for the most important minimal cut sets.
2) compute the availability and reliability for the minimal cut sets and top event.

The first stage of the fault tree analysis employs various algorithms which are used in the generation of the minimal cut sets. These algorithms:

1) Simplify the fault tree.

2) Use a cut-off criteria.
3) Evaluate the error introduced by the cut-off.
4) Order the cut sets in importance.
5) Optimise the analysis by reducing the number of cut sets to be minimised.
6) Lastly analyse the NOT operators.

Astolfi et al. have called their technique the "prior sensitivity analysis" method, which means that the significant minimal cut sets are produced without determining all others (i.e. a numerical cut-off is utilised). This method may cause problems when dealing with prime implicants as certain prime implicants could be neglected due to their order even though they are significant. A list processing technique, similar to that used by Bennetts (12), is employed to store and handle the fault tree information. The fault tree is simplified using the following basic steps:

1) A cascade of gates of the same type is replaced by a single gate. Pairing does not take place if the descendant gate is repeated.
2) Gates which have basic event inputs only are replaced by a compound event.

To illustrate the algorithm first consider fault trees with only AND and OR gates and no repeated events. A bottom-up approach is implemented where the probability of each gate event is calculated. The probability of an AND gate is taken as the product of the probabilities of its descendants. For an OR gate the maximum probability value of its descendants is taken. In this way when the top event is reached the resulting probability will be the probability of the most important cut set, called P_{max}.

Having established P_{max} the tree is then scanned from the top downwards to compute more restrictive thresholds for cut-off limits. A threshold probability L_{lim} is first assigned to the top gate. If the top gate is an OR gate then L_{lim} is also given to all of its descendants and likewise for all OR gates. For an AND gate say GA, with the descendants G1,.....,Gn (these can be gates or basic events), if L(GA) is the cut-off level of GA and P(Gj) is the probability of Gj then:

$$L(Gi) = \frac{L(GA)}{\prod_{\substack{j=1 \\ j \notin i}}^{n} P(Gj)} \qquad (2.3)$$

If P(Gi)<L(Gi) this gate can be deleted because it cannot contribute significantly to the system unavailability. The next step involves further reduction, again using a bottom-

up approach. In this step subtrees are replaced by their cut sets. Any cut set is neglected if its probability is less than the threshold limit of the gate above it.

Bounds for the relative error introduced by the cut-off and the choice of the cut-off limit are also discussed in the paper. Next, the paper deals with the analysis of AND-OR trees with repeated events. A minimisation procedure needs to be applied to these trees. The minimisation procedure involves the cancellation of redundant cut sets (A.B+A.B=A.B), non-minimal cut sets (A+A.B=A), and of redundant events within cut sets (A.A.B=A.B). To do this an efficient algorithm is presented which identifies the gates where the cut sets may require minimising. The algorithm is outlined below and must be applied to each repeated event (here called A) of the tree.

1. For each occurrence of A the path (sequence of gates from A towards Top) are listed.
2. The number of times, K, that a gate is crossed by the considered successive paths is determined.
3. The gates with K=1 are deleted. Gates with the same K are also deleted, except the first one in the list (the same K means that between the two gates there are no repetitions of A).

Slight modifications are needed for fault trees with repeated events to evaluate the bounds for the relative error introduced by the cut-off.

The last section of the paper extends the methodology to deal with the analysis of fault trees containing NOT gates as well as AND and OR gates. Two new problems are stated that have to be tackled which are:

1. The need to cancel prime implicants which are impossible because they contain an event and its complement.
2. The need of enlarging the cut-off criteria in order to handle prime implicants of a very high order containing many complement events.

The method produced for analysing trees of this type avoids the need to determine all of the prime implicants. Instead it determines an irredundant form for the top event which is not necessarily minimal, but it can be considered near minimal as it contains all the "essential prime implicants" common to all the irredundant forms. The "essential prime implicants" are those which contribute significantly to the top event occurrence. A disjunctive normal form is irredundant if and only if it contains no superfluous

disjuncts (see below) or literals (Mendelson (15)). A literal is called biform if, in the Boolean function, it appears in two forms, negative and affirmative, monoform if it appears in one form only.

Superfluous disjuncts and literals

If φ is an implicant and ω a sum of products form for the top event, then φ is superfluous in ω+φ when ω is logically equivalent to ω+φ.
If α is a literal, φ a product of literals and ω a sum of products form for the top event then α is superfluous in α.φ+ω, if φ+ω is logically equivalent to α.φ+ω.

The methodology uses these definitions plus Shannon's theorem (see Schneeweiss (22)) which can be stated as follows.

A Boolean function $f(\mathbf{x})$ where $\mathbf{x} = (X_1, X_2, \ldots\ldots, X_n)$ can be written as:

$$f(\mathbf{x}) = X_i . f(1_i, \mathbf{x}) + \overline{X_i} . f(0_i, \mathbf{x})$$

where:

$$\overline{X_i} = 1 - X_i$$
$$f(1_i, \mathbf{x}) = f(X_1, \ldots\ldots X_{i-1}, 1, X_{i+1}, \ldots\ldots, X_n) \qquad \text{(i.e. } X_i \text{ fails)}$$
$$f(0_i, \mathbf{x}) = f(X_1, \ldots\ldots X_{i-1}, 0, X_{i+1}, \ldots\ldots, X_n) \qquad \text{(i.e. } X_i \text{ works)}$$

$f(1_i, \mathbf{x})$ and $f(0_i, \mathbf{x})$ are called the residues of $f(\mathbf{x})$ with respect to X_i.

The implementation of the method by Astolfi et al. is articulated in two steps, expansion and reduction.

Step 1 Expansion

This step makes use of Shannon's theorem and of another function, called **min(ϕ)**. The minimisation function **min(ϕ)**, applied to the Boolean expression ϕ, deletes non-minimal implicants and applies the identity:

$$XY + \overline{X}YZ = XY + YZ \qquad (2.4)$$

The disjunctive normal form for the top event is first expanded with respect to the most repeated variable using Shannon's expansion. Then the function **min** is applied to the residues. If the residue contains at least one biform variable, it is further expanded and minimised. These operations must be applied until the residues contain only monoform events. The recursive application of expansion and minimisation transforms a disjunctive normal form into an equivalent Boolean expression containing residues in which biform events do not appear. It follows that these residues are now minimal.

As an example consider a fault tree whose disjunctive normal form for the top event is:

$$T = ABC + \overline{A}CD + B\overline{E}G + EFG + \overline{D}E\overline{G} + AE\overline{G} + CDE + \overline{C}EF + \overline{A}\,\overline{D}\,\overline{G} + B\overline{E}\,\overline{G} \qquad (2.5)$$

Applying the function **min** on T and rearranging gives:

$$T = ABC + \overline{A}CD + B\overline{E}(G + \overline{G}) + EFG + \overline{D}E\overline{G} + AE\overline{G} + CDE + \overline{C}EF + \overline{A}\,\overline{D}\,\overline{G}$$
$$= ABC + \overline{A}CD + B\overline{E} + EFG + \overline{D}E\overline{G} + AE\overline{G} + CDE + \overline{C}EF + \overline{A}\,\overline{D}\,\overline{G}$$

Next expand with respect to the most repeated variable, which here is E and minimise the residues:

$$T = E[ABC + \overline{A}CD + 0 + FG + \overline{D}\,\overline{G} + A\overline{G} + CD + \overline{C}F + \overline{A}\,\overline{D}\,\overline{G}]$$
$$+ \overline{E}[ABC + \overline{A}CD + B + 0 + 0 + 0 + 0 + 0 + \overline{A}\,\overline{D}\,\overline{G}]$$

For the first residue, $\overline{A}CD$ is made redundant by CD and $\overline{A}\,\overline{D}\,\overline{G}$ is made redundant by $\overline{D}\,\overline{G}$. For the second residue ABC is made redundant by B. This gives the reduced expression for T:

$$T = E[ABC + FG + \overline{D}\,\overline{G} + A\overline{G} + CD + \overline{C}F] + \overline{E}[\overline{A}CD + B + \overline{A}\,\overline{D}\,\overline{G}]$$

The first residue is then expanded with respect to C and the second one with respect to D (both biform variables):

$$T = E[C(AB + FG + \overline{D}\,\overline{G} + A\overline{G} + D + 0) + \overline{C}(0 + FG + \overline{D}\,\overline{G} + A\overline{G} + 0 + F)]$$
$$+ \overline{E}[D(\overline{A}C + B + O) + \overline{D}(0 + B + \overline{A}\,\overline{G})]$$

Note $D + \overline{D}\,\overline{G} = D + \overline{G}$ for the C residue and FG is made redundant by F for the $\overline{C}$ residue. Therefore:

$$T = E[C(AB + FG + D + \overline{G} + A\overline{G}) + \overline{C}(\overline{D}\,\overline{G} + A\overline{G} + F)]$$
$$+ \overline{E}[D(\overline{A}C + B) + \overline{D}(B + \overline{A}\,\overline{G})]$$

Note $A\overline{G}$ is made redundant by $\overline{G}$ and $\overline{G} + FG = \overline{G} + F$ for the C residue. Therefore:

$$T = E[C(AB + \overline{G} + F + D) + \overline{C}(\overline{D}\,\overline{G} + A\overline{G} + F)] + \overline{E}[D(\overline{A}C + B) + \overline{D}(B + \overline{A}\,\overline{G})] \quad (2.6)$$

T cannot be expanded any further as no more biform variables exist, as a result the reduction stage can be executed.

Step 2 Reduction

The reduction procedure develops one of the irredundant disjunctive normal forms for the top event. This is obtained by analysing expression (2.6) starting from the residues.

Taking $\phi(1_E, \mathbf{x})$ to be the first residue for expression (2.6) with respect to variable E and $\phi(0_E, \mathbf{x})$ the second then:

$$\phi(1_E, \mathbf{x}) = C(AB + \overline{G} + F + D) + \overline{C}(\overline{D}\,\overline{G} + A\overline{G} + F)$$
$$\phi(0_E, \mathbf{x}) = D(\overline{A}C + B) + \overline{D}(B + \overline{A}\,\overline{G})$$

The function **reduce[ϕ]** is defined as deleting both superfluous literals and superfluous prime implicants in the expression ϕ. First dealing with $\phi(1_E, \mathbf{x})$:

$$\phi(1_E, \mathbf{x}) = ABC + C\overline{G} + CF + CD + \overline{C}\,\overline{D}\,\overline{G} + \overline{C}A\overline{G} + \overline{C}F$$

note :

$$CF + \overline{C}F = F$$
$$C\overline{G} + \overline{C}\,\overline{G}\,\overline{D} = C\overline{G} + \overline{G}\,\overline{D}$$
$$C\overline{G} + \overline{C}\,\overline{G}A = C\overline{G} + \overline{G}A$$

Therefore:

$$\textbf{reduce}[\phi(1_E, \mathbf{x})] = ABC + C\overline{G} + \overline{G}\,\overline{D} + F + CD + \overline{G}A$$

Next using the 'consensus' rule $(\overline{X}K + XJ = \overline{X}K + XJ + KJ)$ to the terms $\overline{D}\,\overline{G} + DC + \overline{G}C = \overline{D}\,\overline{G} + DC$, results in the final reduced form for $\phi(1_E, \mathbf{x})$:

$$\phi(1_E, \mathbf{x}) = ABC + \overline{D}\,\overline{G} + F + CD + A\overline{G}$$

Repeating for $\phi(0_E, \mathbf{x})$ gives:

$$\phi(0_E, \mathbf{x}) = D\overline{A}C + \overline{A}\,\overline{G}\,\overline{D} + B$$

So $T = E[ABC + \overline{D}\,\overline{G} + F + CD + \overline{G}A] + \overline{E}[D\overline{A}C + \overline{A}\,\overline{D}\,\overline{G} + B]$

Now expanding T gives:

$$T = ABCE + \overline{D}\,\overline{G}E + FE + CDE + \overline{G}AE + D\overline{A}C\overline{E} + \overline{A}\,\overline{D}\,\overline{G}\,\overline{E} + B\overline{E}$$

note:

$$B\overline{E} + ABCE = B\overline{E} + ABC$$

$$CDE + D\overline{A}C\overline{E} = CDE + D\overline{A}C$$

$$\overline{D}\,\overline{G}E + \overline{A}\,\overline{D}\,\overline{G}\,\overline{E} = \overline{D}\,\overline{G}E + \overline{A}\,\overline{D}\,\overline{G}$$

Therefore:

$$T = ABC + \overline{D}\,\overline{G}E + FE + CDE + \overline{G}AE + D\overline{A}C + \overline{A}\,\overline{D}\,\overline{G} + B\overline{E}$$

Again using consensus:

$$A\overline{G}E + \overline{A}\,\overline{D}\,\overline{G} + \overline{D}\,\overline{G}E = A\overline{G}E + \overline{A}\,\overline{D}\,\overline{G}$$

Finally:

$$\textbf{reduce}\ [T] = ABC + FE + CDE + \overline{G}AE + D\overline{A}C + \overline{A}\,\overline{D}\,\overline{G} + B\overline{E} \qquad (2.7)$$

Which represents one of the irredundant forms of (2.5). If the irredundant form is unique, then it is also minimal, and constitutes the outcome of the given algorithm. This example illustrates the difficulty of obtaining the prime implicants of a disjunctive normal form. The previous extensive steps reduced the ten term expression of (2.5) to the seven term expression of (2.7). Therefore if a disjunctive normal form contained hundreds of terms then this algorithm would prove extremely computer intensive.

The top-down algorithm of Kumamoto and Henley (16) obtains the prime implicants of a non-coherent fault tree by firstly converting the fault tree to its dual tree. A local expression is given for the top event of this dual tree, in terms of its gate and basic event inputs, and it is called ψ^*. The gates are numbered in the tree from the bottom-

up in ascending order, left to right. All basic events are listed in an order corresponding to a top-down, left-right manner in the tree, with complemented events occurring last. Next, a series of operations are performed on ψ^* concerning the basic events in the list. ψ^* is operated on by considering the result of the local expression when each basic event in the list is given the value 0 and its complement is given the value 1. Next the basic event is given the value 1 and its complement the value 0.

To illustrate the method consider a fault tree whose top event is an AND gate with inputs G14 and A. G14 is an OR gate with gate inputs G12 and G13. G13 is an AND gate with inputs B and G11. G12 is also an AND gate with inputs B and G10. G11 is the final limit of resolution for this example, it is an AND gate with inputs D and G9. The list of basic events for this fault tree is {A, B, C, D, F, E, G, H, I, $\overline{F}$, $\overline{E}$, $\overline{G}$, $\overline{H}$, $\overline{I}$}:

$$\psi^*=A.G14 \tag{2.8}$$

Before changing the values of the basic events to either 0 or 1, expand G14 by its local expression which is G12+G13.

Therefore:

$$\psi^*=A.G12+A.G13 \tag{2.9}$$

Next, apply the operation A^0 which gives the basic event A the value 0:

$$A^0.\psi^*=0.G12+0.G13=0 \tag{2.10}$$

The largest gate number, which corresponds to the largest undeveloped gate in the fault tree, in ψ^* is replaced by its local expression (which here is G13) before the operation A^1 is applied. Operation A^1 gives basic event A the value 1. The local expression for G13 is B.G11.

Therefore:

$$\psi^*=A.G12+A.B.G11 \tag{2.11}$$

Applying A^1:

$$A^1.\psi^*=1.G12+1.B.G11 \tag{2.12}$$

Expanding G12 into B.G10:

$$A^1.\psi^*=1.B.G10+1.B.G11 \tag{2.13}$$

Applying B^0:

$$B^0.A^1.\psi^*=0.G10+0.G11=0 \tag{2.14}$$

Expanding G11 into D.G9:

$$A^1.\psi^*=B.G10+B.D.G9 \tag{2.15}$$

Applying B^1:

$$B^1.A^1.\psi^*=1.G10+1.D.G9 \tag{2.16}$$

Continue in this manner, expanding the gates when either a 0 or 1 results or both operations of a basic event have been considered, until all the basic events in the list have been dealt with. To obtain the prime implicants consider the operations that give a zero result and pick out the basic events on the left hand side having a zero superscript. Therefore equations (2.10) and (2.14) give A and B respectively. Each set of literals obtained in this way is a candidate of a prime implicant. By then removing redundant implicants, a complete set of the prime implicants is obtained. The method is attractive as it avoids large sum of products expressions for the top event of a fault tree.

The algorithm of Nakashima and Hattori (17) aims to improve the conventional bottom-up algorithm of Bennetts (12) by obtaining the minimal cut sets more quickly. The improvement is based on reducing the number of checks for redundant terms in the logical product of two reduced disjunctive normal forms. Therefore the algorithm is applied at each AND gate of the fault tree. Five model assumptions are made in the paper concerning the features of fault trees to which the method can be applied. These are:

1) Mutually exclusive primary events are allowed to appear, i.e. the Boolean function for the top event need not be coherent.
2) The same event can appear in several branches.

3) No complemented intermediate events can appear at any gate, i.e. only OR and AND gates are allowed. If logic gates such as NOT, XOR, NAND appear in the fault tree, then the algorithm can be applied after transforming it into an equivalent fault tree containing only OR and AND gates by using inversion operations represented by De Morgans Laws.

The algorithm begins with primary events and is repeatedly applied progressing up the tree structure, until reaching the top event. The process expands the logical product (for AND gates) or sum (for OR gates) of reduced disjunctive normal forms for the two intermediate events and yields an equivalent reduced disjunctive normal form. This is achieved using the distribution rule and then discarding redundant terms by applying the idempotence (X+X=X) and the absorption (X+X.Y=X) rules. The algorithm is called ANCHEK and to illustrate the algorithms application consider the process of obtaining a disjunctive normal form for the top event, T of a fault tree from the logical product of its sub-events T1 and T2. Let the set of minimal cut sets of T1 and T2 be C1 and C2 respectively. Each primary event appearing in the reduced disjunctive normal forms of T1 and T2 is classified as either a common primary event (event that appears in both T1 and T2) or a non-common primary event. The algorithm for obtaining C, the set of minimal cut sets of T, is based on the following principles.

Firstly let $c \in C1 \otimes C2$, where $C1 \otimes C2$ is the whole set of unions of an element of C1 and an element of C2 for two sets C1, C2 each element of which is a set of primary events.

1) If c contains non-common primary events, then c is always an element of C. Thus it is not necessary to check c at all.
2) If c contains at least one common primary event, then only elements of a subset of $C1 \cup C2$ are required to check c.

The detailed algorithm is given in the paper as a set of four explicit steps. Steps 1 and 2 are executed using the bit manipulation technique of Wheeler et al. (29) Nakashima and Hattori have constructed a computer program called BUP-CUTS (**B**ottom-**UP** algorithm for enumerating minimal **CUT** **S**ets of fault trees) written in FORTRAN. The program uses the ANCHEK algorithm, instead of the MULTIPLY and ORCHEK algorithms given by Bennetts (12), to transform the logical product of two reduced disjunctive normal forms into an equivalent reduced disjunctive normal form. The other parts of the program follow Bennetts' algorithm in principle. The paper

compares the presented algorithm (BUP-CUTS) with that of Bennetts (BEN-CUTS) for four fault tree examples. The improvement in the time to obtain the minimal cut sets can be seen in table 2.4.

Fault Tree Example	Number of Minimal Cut Sets	Computation Time (Secs) BUP-CUTS	Computation Time (Secs) BEN-CUTS
1	256	0.5	11.7
2	90	17.3	26.9
3	357	39.7	150
4	457	4.5	118

Table 2.4 Comparison of Two Techniques

Worrell et al. (18) discuss the prime implicant algorithm proposed by Henley and Kumamoto (16) and show that the same results can be obtained by using a simple prime implicant algorithm such as SETS by Worrell (13). The authors define a simple prime implicant algorithm as a method that does not use consensus for finding the prime implicants of non-coherent fault trees, i.e. write down a Boolean expression for the top event of the fault tree and expand it into a disjunctive normal form while also simplifying the result by dropping repeated literals, zero products, and subsuming. It is stated that this method does not generally work on non-coherent fault trees, however there are cases when it does work.

The important issue of the paper by Worrell et al. is the problem of deciding when the simple prime implicant algorithm will deliver a complete set of prime implicants. A 'partial' result is offered which shows that it will work for certain non-coherent trees (as well as for coherent trees) whose top events have Boolean formulae of a special type. The following theorem is given:

Theorem: If (but not "only if") a Boolean formula F has a dual d(F) which contains no zero products, then the simple prime implicant algorithm applied to F will produce the prime implicants of F.

The intuitive proof of this theorem is found in Hulme and Worrell (14). Lastly Worrell et al. state a useful corollary to their results; that when a dual formula d(F) has zero products and these are eliminated to form a new formula d(H) equivalent to d(F), then the simple prime implicant algorithm applied to H will produce all the prime implicants of H and F. The benefits that this theorem would give in executing a program is not

discussed in the paper. Finding the dual of the Boolean formula may take as long as actually expanding the formula, using consensus and then eliminating redundancies to find the prime implicants.

2.5 Modules of Non-Coherent Fault Trees

Locks (49) in 1981 has shown that the non-coherent fault tree discussed in the paper by Kumamoto and Henley (16) can also be modularised. Locks states that modules are formed by combining certain neighbouring components that have the same mutual logical dependence. If a method such as MOCUS (20) is then used to generate an expression for the top event in terms of these modules, the prime implicants can be obtained by a systematic application of the consensus theorem which has been previously discussed.

The modularising process of Wilson in 1985 (50) obtains modules from the Boolean indicator expression (a sum of the logical products of the basic events) of a fault tree. The method is based on the following consideration:

Let X_i, X_j be two basic events for which;

(1) All Boolean indicators of the fault tree which include X_i also include X_j.

(2) For each Boolean indicator of the form $\overline{X_i}.P$ in the fault tree (where P includes neither X_j nor $\overline{X_j}$) there also exists a Boolean indicator of the form $\overline{X_j}.P$ in the fault tree.

then $X_i.X_j$ can be replaced by y_i (where y_i is a basic event representing the product of basic events). In each Boolean indicator which includes $X_i.X_j$, the Boolean indicator $\overline{X_i}.P$ is replaced by $\overline{y_i}.P$ and the Boolean indicator $\overline{X_j}.P$ is deleted from the fault tree (note $(\overline{X_i}+\overline{X_j})P = \overline{X_i.X_j}.P = \overline{y_i}.P$).

The X_i or X_j can be un-negated or negated variables and so modularisation involves consideration of each of the pairs, $X_i.X_j$, $X_i.\overline{X_j}$, $\overline{X_i}.X_j$ or $\overline{X_i}.\overline{X_j}$.

Therefore the modularisation by Wilson replaces two basic events by one. Additionally the process of modularisation can be repeated for all possible pairings of basic events including basic events of type y_i.

Unfortunately the method by Wilson can become tedious for large fault trees because a Boolean indicator expression necessitates an expansion procedure for the entire fault tree. Further, it is not clear in the paper how creating modules in this way can simplify the analysis process of the fault tree.

In 1989 Kohda et al. (40) developed a very sophisticated method of identifying all possible modules of a fault tree. The procedure is called the KHIC method (Kohda, Henley, Inoue Comprehensive method) and for its purposes a module of a fault tree is defined as:

> A subtree composed of a least two events which have no inputs from the rest of the tree and no outputs to the rest except from its output event.

In addition Kohda et al. state that there are two kinds of modules;

(1) those whose output events are expressed by gate events.

(2) those whose output events are not expressed by gate events.

The latter are logical OR or AND combinations of basic events and modules. The output of the program by Kohda et al. is a hierarchical decomposition of the fault tree into modules. The order in which modules are identified corresponds to both the hierarchy established when the gates are numbered, and the order in which the modules are analysed.

Based on the closeness of events below a gate event, the KHIC method examines all gate events which can become modules in ascending order of their hierarchical levels. Note that the nearer to the top event of a gate, the higher its hierarchical level. The method can also be applied to coherent fault trees. Khoda et al. demonstrate that by modular decomposition, 100-fold reductions in computer time are obtained for a 70 gate, 67 basic events problem.

2.6 Discussion

To give an indication as to how some of the algorithms discussed in this chapter have been implemented on a computer, Randall Willie in 1978 (47) published an extensive

report on 'Computer-Aided Fault Tree Analysis'. In this report Willie describes the computer implementation of a top-down method (called MSDOWN), a bottom-up approach (called MSUP), the Nelson (11) method and a factoring approach. Additionally, the author of this thesis has successfully programmed a top-down and a bottom-up approach (48), to analyse fault trees for use on a PC, which are both available for reference. The computer implementation of these algorithms exhibits the extensive memory and processing speed problems that may be experienced when analysing complex systems.

Thus, with many commercial packages now available on PC's there is a commercial advantage to be gained in the development of an efficient algorithm to determine minimal cut sets or prime implicants. Indeed the possession of a 'fast' algorithm is a major selling point for these codes as it is an ambition to move the application of these codes into "real time" analysis. As such over the last decade publications describing new techniques have reduced and the fine details of the more successful methods are not in the public domain.

Extensive discussions have been held with the authors of some of the most popular codes and they confirmed that the approaches they have adopted are based on relatively minor refinements to the algorithms described above.

2.7 Summary

1) Most of the minimal cut set algorithms discussed in this chapter are based on Boolean Reduction methods which are inefficient for large fault trees. A complex system may produce hundreds of thousands of minimal cut sets and the determination of these cut sets can be a very time consuming process. The algorithms are not complex but are required to perform a vast number of event comparisons to reach the minimal form (i.e., the need to apply the Boolean reduction laws). Repeated events within each cut set, repeated cut sets and non-minimal cut sets all need to be removed. This requires comparison of events within each cut set and each cut set with every other cut set.

2) The algorithms described in many of the papers seem designed for special test case fault trees and do not yield the same efficiency for general fault tree analysis purposes.

3) Most of the improvements in the conventional techniques are concerned with reducing the influence of repeated basic events within the fault tree. Although savings can be made in the number of set comparisons that are needed to produce the minimal cut sets, it is not clear how much computer utilisation is required to eliminate the influence of the repeated events. Indeed in some cases the sorting procedures that are needed would appear to increase computer CPU times.

4) The improvements in computer utilisation for some of the given algorithms is related more to the improvements in computer processing capabilities over the years rather than being attributed to a marked difference in the algorithms.

5) Order cut-off procedures can leave out important minimal cut sets and therefore vital information about the system under study is lost.

6) Some of the non-coherent approaches require De Morgans Laws to be applied to the fault tree before the algorithms are executed, which would increase CPU time. The algorithms that are then applied to these trees are similar to those for coherent fault trees and therefore the same problems will be encountered for non-coherent algorithms as for coherent algorithms. These two features reduce the efficiency of the analysis process for non-coherent fault trees.

7) Modularising the fault tree can substantially reduce the analysis process and this technique is valuable when used in conjunction with conventional methods. However certain fault trees do not lend themselves to modularising and therefore their existing structure cannot be simplified in this way.

8) Current commercial packages are based on the methods described above. However due to commercial confidentiality the refinements made to improve the efficiency of the algorithms are not in the public domain. Discussions with the authors of the fastest of these codes indicate that the algorithms they use have their basis in the published literature.

9) Computerised methods, such as bottom-up or top-down approaches, to evaluate minimal cut sets, are now so well developed that further refinement is unlikely to result in vast reductions in computer time. It is felt that substantial improvement in computer utilisation will only result from a completely new approach. The use of a Binary Decision Diagram for Fault Tree Analysis described in Chapter 4 offers a promising alternative whose potential requires investigation.

CHAPTER 3

METHODS FOR QUANTITATIVE FAULT TREE ANALYSIS

3.1 Introduction

Fault tree quantification enables not only the probability of the top event to be calculated but in addition its failure rate, expected number of occurrences and also importance measures which signify the contribution each basic event makes to system failure. Due to the large number of failure combinations (minimal cut sets) which generally result from a fault tree study it is not possible using conventional techniques to calculate these parameters exactly and approximations are required. The accuracy of most of the approximations rely on the basic events having a small likelihood of occurrence. When this condition is not met it results in large inaccuracies.

Vesley (1) published a methodology known as "kinetic tree theory" which gives a time-dependent methodology for fault tree evaluation. This methodology forms the basis for the approach taken in most of the commercial fault tree packages. The procedure uses two assumptions, (i) that primary or basic events are independent and (ii) that the mode failures (critical paths) of the fault tree are known. In the quantification of top event failure characteristics it is necessary to use approximations when implementing the kinetic tree theory. Even for moderate sized problems it is not a practical proposition to evaluate all the terms in the series expansion which yields the top event probability or failure intensity.

3.2 Component Failure Parameters

Before considering the quantification of the fault tree as a whole the failure parameters of the basic events in the fault tree need to be discussed. For each component failure represented by a basic event in the fault tree the failure and repair time distributions can be assumed to have the density functions $f(t)$ and $g(t)$ respectively. Integral equations utilising these functions can be developed whose solution yields the *unconditional failure intensity* $w(t)$ and the *unconditional repair intensity* $v(t)$. These integral equations are derived in Andrews and Moss (6) and are:

$$w(t) = f(t) + \int_0^t f(t-u)v(u)du$$
$$v(t) = \int_0^t g(t-u)w(u)du \qquad (3.1)$$

where:

w(t) is the probability per unit time that a component fails at *t* given that it was working at *t=0*.

v(t) is the probability per unit time that a failed component is repaired at *t* given that it was working at *t=0*.

Depending on the distributions which define *f(t)* and *g(t)* equation (3.1) may be solved by Laplace transforms or they may require the use of numerical methods. However once *w(t)* and *v(t)* are found then *q(t)*, the probability that the component is in the failed state at *t* is obtained from:

$$q(t) = \int_0^t [w(u) - v(u)]du \qquad (3.2)$$

and the expected number of failures $W(t_1,t_2)$, of a component, over any time period (t_1,t_2) is given by:

$$W(t_1, t_2) = \int_{t_1}^{t_2} w(u)du \qquad (3.3)$$

Once *w(t)* and *v(t)* are available other parameters for the component failure and repair process can be determined.

Conditional failure intensity $\lambda(t)$: is the probability that a component fails per unit time at *t* given that it was working at time *t* and working at time zero. The difference between this and the unconditional failure intensity *w(t)* is that λ is the failure rate based on those components which are working at time *t*, whereas *w* is based on the whole population.

Conditional repair intensity $\mu(t)$*:* is the probability that a component is repaired per unit time at *t* given it is failed at time *t* and was working at time zero.

It can be shown (6) that if a component has a constant failure rate (hazard rate), λ, then the probability density function for the failure, *f(t)*, will be:

$$f(t) = \lambda e^{-\lambda t} \tag{3.4}$$

A useful parameter of a component is its mean time to failure (MTTF) which will be represented by *r*. For the exponential distribution given in (3.4):

$$r = \frac{1}{\lambda} \tag{3.5}$$

Therefore if a component has a constant failure rate then the MTTF is simply the reciprocal of its failure rate.

Similarly it can be shown that if the repair rate of a component is constant, μ, then the repair density function, *g(t)*, will be:

$$g(t) = \mu e^{-\mu t} \tag{3.6}$$

As before, the mean time to repair (MTTR), τ, will be the reciprocal of the repair rate:

$$\tau = \frac{1}{\mu} \tag{3.7}$$

If a component has constant failure and repair rates then the formula given for its unavailability, *q(t)* (equation 3.2) can be solved by Laplace transforms to give:

$$q(t) = \frac{\lambda}{\lambda + \mu}\{1 - \exp[-(\lambda + \mu)t]\} \tag{3.8}$$

When looking at systems it is common to use the steady-state component unavailability, i.e. when the component has been operable for a reasonable period of time $t \to \infty$. In this case:

$$q = \frac{\lambda}{\lambda + \mu} \tag{3.9}$$

Substituting (3.5) and (3.7) into (3.9) gives:

$$q = \frac{\tau}{\tau + r} = \frac{MTTR}{MTTR + MTTF} \tag{3.10}$$

There is also a useful relationship between $\lambda(t)$, $w(t)$ and $q(t)$:

$$w(t) = \lambda(t)[1 - q(t)] \tag{3.11}$$

Hence if a components unavailability and failure rate are known then its unconditional failure intensity $w(t)$ can be easily calculated using (3.11).

The way in which components or systems are maintained has a large influence over the unavailability. If component failure is revealed then equation (3.8) is appropriate. When failures are dormant or unrevealed they will only be detected on inspection or testing. If a system is inspected every θ time units its average unavailability is given as:

$$q_{AV} = \lambda(\frac{\theta}{2} + \tau) \tag{3.12}$$

The equation used to specify its unavailability constitutes the component's 'Model Type'. Basic event models are further discussed in Chapter 6.

3.3 Kinetic Tree Theory

3.3.1 Top Event Quantification

Quantification of the fault tree top event probability is generally calculated using the component failure/basic event existence probabilities and the minimal cut sets. Here the top event probability or unavailability will be known as $Q_{sys}(t)$. Consider the Boolean sum of product expression for the top event of a fault tree:

$$Top = C_1 + C_2 + \ldots\ldots\ldots\ldots + C_{nc}$$

Where C_i, $i=1,\ldots\ldots nc$ are the minimal cut sets of the top event, i.e. product terms.

If $C_i = X1.X2\ldots\ldots\ldots Xp$ where Xj, $j=1,\ldots\ldots\ldots p$ are all independent then the probability of each minimal cut set is given by:

$$P(C_i) = P(X1).P(X2)\ldots\ldots\ldots P(Xp)$$

Since the top event exists when at least one of the minimal cut sets exist, the top event probability, Q_{EXACT}, (see Henley and Kumamoto (7)) is given by:

$$Q_{EXACT} = P(Top) = \sum_{i=1}^{nc} P(C_i) - \sum_{i=2}^{nc} \sum_{j=1}^{i-1} P(C_i \cap C_j) + \ldots + \ldots (-1)^{nc-1} P(C_1 \cap C_2 \cap \ldots \cap C_{nc}) \quad (3.13)$$

This is commonly known as the inclusion-exclusion formula which has already been mentioned in Chapter 1. To illustrate the calculation of the top event probability, consider the example fault tree in figure 3.1.

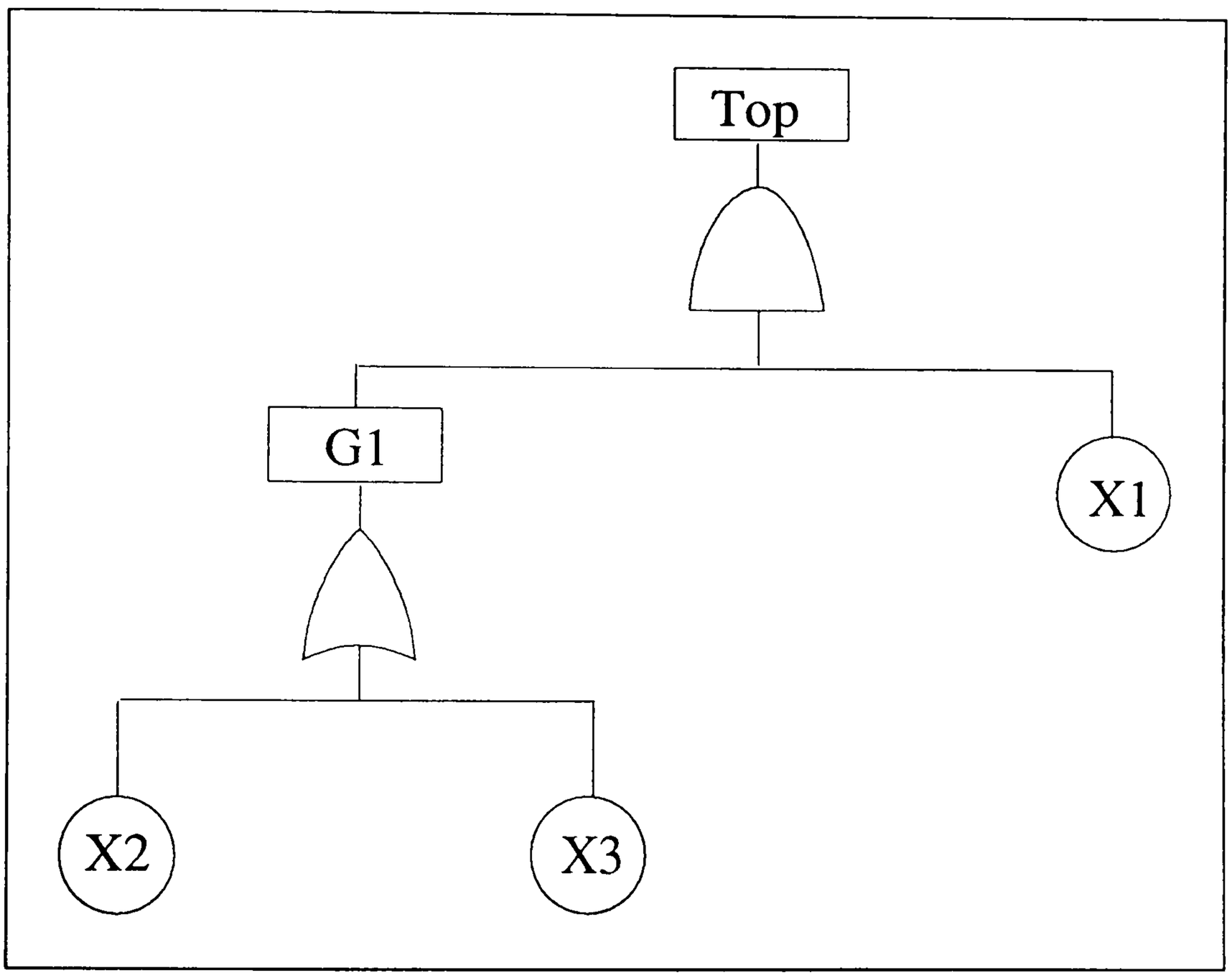


Figure 3.1 Example Fault Tree

The fault tree in figure 3.1 has the top event Boolean expression:

$$Top = X1.X2 + X1.X3$$

Therefore the minimal cut sets for this fault tree are $C_1 = X1.X2$ and $C_2 = X1.X3$.

Let the probability of each independent basic event be 0.04 and use equation (3.13) to obtain the top event probability:

$$P(Top) = P(C_1) + P(C_2) - P(C_1 \cap C_2)$$

$$=P(X1.X2)+P(X1.X3)-P(X1.X2.X1.X3)$$

note P(X1.X1)=P(X1)

$$=P(X1).P(X2)+P(X1).P(X3)-P(X1).P(X2).P(X3)$$
$$=(0.04)(0.04)+(0.04)(0.04)-(0.04)(0.04)(0.04)$$
$$P(Top)=0.003136$$

Clearly if the fault tree has many minimal cut sets then evaluating the top event probability from equation (3.13) will require extensive calculations. For all but the most simple fault tree structure the evaluation of each term in the expansion is not a practical proposition. Upper bound approximations are therefore frequently used to obtain the system failure probability. Wheeler et al. (29) also recognised the need to use approximations for the top event probability. One such approximation is the Rare Event estimation, $P_{RE}(Top)$, which takes only the first term in equation (3.13):

$$P_{RE}(Top) = \sum_{i=1}^{nc} P(C_i) \tag{3.14}$$

For the fault tree in figure 3.1 this would give:

$$P_{RE}(Top)=P(X1).P(X2)+P(X1).P(X3)$$
$$P_{RE}(Top)=0.0032$$

The reason for using this is that the higher order terms in (3.13) of simultaneous minimal cut set occurrence will have a relatively small probability if the failure of each basic event is rare and therefore will only provide a small correction to (3.14). Another approximation method is the Minimal Cut Set Upper Bound (MCSUB). This approximation is a more accurate upper bound and requires very little additional effort in its calculation:

$$P_{MCSUB}(Top) = 1 - \prod_{i=1}^{nc} (1 - P(C_i)) \tag{3.15}$$

For the fault tree in figure 3.1 this gives:

$$P_{MCSUB}(Top)=1-(1-P(X1).P(X2))(1-P(X1).P(X3))$$
$$=1-(1-(0.04)(0.04))(1-(0.04)(0.04))$$
$$P_{MCSUB}(Top)=0.003197$$

These approximations bear out the relationship:

$$Q_{EXACT} \leq 1 - \prod_{i=1}^{nc}(1 - P(C_i)) \leq \sum_{i=1}^{nc} P(C_i) \tag{3.16}$$

Exact≤Minimal Cut Set Upper Bound≤Rare Event

The problem of using the approximation occurs when basic event failures are not rare.

3.3.2 Structure Functions

The full expansion (3.13) can be obtained by first producing the structure function for the top event and then taking its expectation. The structure function $\phi(\mathbf{x})$ for the top event of a fault tree with minimal cut sets C_i, i=1,.....nc is:

$$\phi(\mathbf{x}) = 1 - \prod_{i=1}^{nc}(1 - P_i) \tag{3.17}$$

where P_i is the structure function of each minimal cut set.

For the fault tree illustrated in figure 3.1 the structure function is:

$$\phi(\mathbf{x})=1-(1-X1.X2)(1-X1.X3)$$

In the case that each C_i is statistically independent then the probability of the top event equals the expectation of the structure function i.e. $P(Top)=E[\phi(\mathbf{x})]=\phi[E(\mathbf{x})]$ and it is easy to obtain the exact top event probability:

$$\begin{aligned} E[\phi(\mathbf{x})] &= \sum_i i.P(\phi(\mathbf{x}) = i) \\ &= 0.P(\phi(\mathbf{x}) = 0) + 1.P(\phi(\mathbf{x}) = 1) \\ &= P(\phi(\mathbf{x}) = 1) \\ &= P(Top) \end{aligned}$$

Unfortunately this situation is rare since basic events are commonly shared by more than one minimal cut set. In this situation a full expansion of the logic equation is required and all powers of the indicator variables reduced ($X_i^n=X_i$) prior to taking the expectation. This provides the full series expansion given as equation (3.13). An

alternative and more efficient way of doing this is to use Shannon's theorem (22) which has been described in Chapter 2.

The structure function for the top event is first expanded or pivoted with respect to the most repeated variable using Shannon's expansion. This is continued until no repeated events occur in the residues. The expectation can then be taken to give the top event probability.

To illustrate consider again the structure function for the top event, Top, of the fault tree in figure 3.1.

$$\phi(\mathbf{x})=1-(1-X1.X2)(1-X1.X3)$$

The expectation cannot be taken without first expanding because X1 is repeated. Therefore pivot about X1, this gives:

$$\begin{aligned}\phi(\mathbf{x}) &= X1[1-(1-X2)(1-X3)]+(1-X1)[0] \\ &= X1[1-(1-X2)(1-X3)]\end{aligned}$$

Now there are no repeated events so take the expectation.

$$\begin{aligned}P(Top)=E[\phi(\mathbf{x})] &= E[X1][1-(1-E[X2])(1-E[X3])] \\ &= P(X1)(1-(1-P(X2))(1-P(X3)))\end{aligned}$$

which gives the exact probability, this equation being identical to the equation obtained by (3.13).

3.3.3 Unconditional Failure Intensity

The unconditional failure intensity of a minimal cut set, $w_{C_i}(t)$, the expected number of times the minimal cut set occurs per unit time at t, is defined as:

$$w_{C_i}(t) = \sum_{i=1}^{n} \{ w_i(t) \prod_{\substack{j \neq i \\ j=1}}^{n} q_j(t) \} \qquad (3.18)$$

To illustrate, let C_1=X1.X2.X3 be a minimal cut set of a fault tree. Then the unconditional failure intensity of C_1, $w_{C_1}(t)$ will be:

$$w_{C_1}(t) = w_{X1}(t)q_{X2}(t)q_{X3}(t) + w_{X2}(t)q_{X1}(t)q_{X3}(t) + w_{X3}(t)q_{X1}(t)q_{X2}(t)$$

The system unconditional failure intensity, $w_{sys}(t)$, is defined as the probability that the top event occurs at t per unit time. This parameter is important for the quantitative analysis of a fault tree, as we can determine the 'expected number of top event occurrences' by integrating $w_{sys}(t)$ with respect to t. Therefore $w_{sys}(t)dt$ is the probability that the top event occurs in the time interval $[t, t+dt)$. For the top event to occur between t and $t+dt$ all the minimal cut sets must not exist at t then one or more occur during t to $t+dt$. More than one minimal cut set can occur in a small time element dt since component failure events can be common to more than one minimal cut set. Therefore:

$$w_{sys}(t) = P[A\bigcup_{i=1}^{nc}\theta_i] \tag{3.19}$$

where:

A is the event that all minimal cut sets do not **exist** at time t, $A = \bigcap_{i=1}^{nc} u_i$.

u_i denotes the ith minimal cut set does not exist at t.

$\bigcup_{i=1}^{nc}\theta_i$ is the event that one or more C_i **occur** in time t to $t+dt$.

Since P(A)=1-P($\overline{A}$), equation (3.19) can be written as:

$$P[A\bigcup_{i=1}^{nc}\theta_i] = P[\bigcup_{i=1}^{nc}\theta_i] - P[\overline{A}\bigcup_{i=1}^{nc}\theta_i] \tag{3.20}$$

where:

$\overline{A}$ means at least one minimal cut set **exists** at t, i.e. $\overline{A} = \bigcup_{i=1}^{nc}\overline{u_i}$.

$\overline{u_i}$ denotes the ith minimal cut set does **exist** at t.

From equation (3.20) equation (3.19) can be expressed as:

$$w_{sys}(t)dt = P[\bigcup_{i=1}^{nc}\theta_i] - P[\overline{A}\bigcup_{i=1}^{nc}\theta_i] \tag{3.21}$$

The first term on the right hand side of expression (3.21) is the contribution from the occurrence of at least one minimal cut set and the second term is the correction contribution provided by minimal cut sets occurring while other minimal cut sets already exist (i.e. system already failed). These two terms can be denoted by $w_{sys}^{(1)}(t)$ and $w_{sys}^{(2)}(t)$ respectively, so:

$$w_{sys}(t)dt = w_{sys}^{(1)}(t)dt - w_{sys}^{(2)}(t)dt \tag{3.22}$$

Expanding the first term, the occurrence of at least one minimal cut set, gives the following series expansion:

$$w_{sys}^{(1)}(t)dt = \sum_{i=1}^{nc} P(\theta_i) - \sum_{i=2}^{nc}\sum_{j=1}^{i-1} P(\theta_i \cap \theta_j) + .. \\ +....(-1)^{nc-1} P(\theta_1 \cap \theta_2 \cap .... \cap \theta_{nc}) \tag{3.23}$$

where $\sum_{i=1}^{nc} P(\theta_i)$ is the sum of the probabilities that minimal cut set i fails in t to $t+dt$, note that this is different than equation (3.13) where the failures exist at time t (note, $P(\theta_i) = w_{C_i}(t)dt$). All other terms in equation (3.23) involve the simultaneous occurrence of two or more minimal cut sets. Since only one basic event can fail in the small time interval dt the simultaneous occurrence of more than one minimal cut set in t to $t+dt$ must result from the failure of a component common to all failed minimal cut sets. For the general term which requires m minimal cut sets to occur in t to $t+dt$ which have k common basic events:

$$\begin{aligned} &if \quad k = 0, \; P(\theta_1 \cap .....\cap \theta_m) = 0 \\ &if \quad k > 0, \; P(\theta_1 \cap .....\cap \theta_m) = w_A(t, B_1, ..., B_k)dt \prod Q_A(\underline{B}) \end{aligned} \tag{3.24}$$

where:

$w_A(t, B_1, ..., B_k)$ is the failure intensity for a set which consists of the k common components (given by equation 3.18).

$\prod Q_A(\underline{B})$ is the product of the probabilities of the remaining components.

The second term of equation (3.22), $w_{sys}^{(2)}(t)dt$ has a more involved expression:

$$w_{sys}^{(2)}(t)dt = P[\overline{A}\bigcup_{i=1}^{nc}\theta_i]$$

$$= \sum_{i=1}^{nc} P(\theta_i \cap \overline{A}) - \sum_{i=2}^{nc}\sum_{j=1}^{i-1} P(\theta_i \cap \theta_j \cap \overline{A}) + ... \quad (3.25)$$

$$.....+(-1)^{nc-1} P(\theta_1 \cap \theta_2 \cap .....\cap \theta_{nc} \cap \overline{A})$$

Since $\overline{A} = \bigcup_{i=1}^{nc} \overline{u_i}$, then each term above can be expanded again, so for a general term in (3.25):

$$P(\theta_1 \cap \theta_2 \cap .....\cap \theta_m \cap \overline{A}) = \sum_{i=1}^{nc} P(\theta_1 \cap \theta_2 \cap .....\cap \theta_m \cap \overline{u_i})$$

$$- \sum_{i=2}^{nc}\sum_{j=1}^{i-1} P(\theta_1 \cap \theta_2 \cap .....\cap \theta_m \cap \overline{u_i} \cap \overline{u_j}) + ... \quad (3.26)$$

$$..+(-1)^{nc-1} P(\theta_1 \cap \theta_2 \cap ....\cap \theta_m \cap \overline{u_1} \cap \overline{u_2} \cap ....\cap \overline{u_{nc}})$$

Where θ_i means that minimal cut set i occurs in t to $t+dt$ and $\overline{u_i}$ means that minimal cut set i exists at t. So for a general term in (3.26):

$$P(\theta_1 \cap \theta_2 \cap .....\cap \theta_m \cap \overline{u_1} \cap \overline{u_2} \cap .....\cap \overline{u_k}) = w_B(t, B_1, ..., B_r)dt \prod Q_B(\underline{B}) \quad (3.27)$$

where:

$w_B(t, B_1, ...B_r)$ is the failure intensity for a set made up of all the component failures which are common to all of the minimal cut sets $C_1, C_2, ..., C_m$ but are not in minimal cut sets $\overline{u_1}, \overline{u_2}, ..., \overline{u_k}$.

$\prod Q_B(\underline{B})$ is the product of probabilities of the remaining components.

Example

Let the basic events in the fault tree shown in figure 3.1 have the steady state failure probabilities and unconditional failure intensities summarised in table 3.1.

Basic Event	q_{xi}	w_{xi}
X1	0.04	7.2e-6
X2	0.04	3.4e-5
X3	0.04	2.9e-7

Table 3.1 Summary of Reliability Parameters for Basic Events in Figure 3.1

To calculate the unconditional failure intensity and expected number of system failures for the fault tree shown in figure 3.1 proceed as follows:

Calculating $w_{sys}^{(1)}(t)dt$ from equation 3.23 we get:

$$w_{sys}^{(1)}(t)dt = \sum_{i=1}^{2} P(\theta_i) - \sum_{i=2}^{2}\sum_{j=1}^{1} P(\theta_i \cap \theta_j) \tag{3.28}$$

Since for this fault tree C_1=X1.X2 and C_2=X1.X3 then using equation 3.18 gives:

$$\sum_{i=1}^{2} P(\theta_i) = w_{C_1} + w_{C_2} = w_{X1}q_{X2} + w_{X2}q_{X1} + w_{X1}q_{X3} + w_{X3}q_{X1}$$

$$\sum_{i=1}^{2}\sum_{j=1}^{1} P(\theta_i \cap \theta_j) = P(\theta_1 \cap \theta_2) = w_{X1}q_{X2}q_{X3}$$

Note, using equation (3.24) X1 is common to both C_1 and C_2.

Therefore:

$$w_{sys}^{(1)}(t) = w_{X1}q_{X2} + w_{X2}q_{X1} + w_{X1}q_{X3} + w_{X3}q_{X1} - w_{X1}q_{X2}q_{X3}$$

$$=(7.2e\text{-}6)(0.04)+(3.4e\text{-}5)(0.04)+(7.2e\text{-}6)(0.04)+(2.9e\text{-}7)(0.04)-(7.2e\text{-}6)(0.04)^2$$

$w_{sys}^{(1)}(t)$=1.93608e-6

Next calculating $w_{sys}^{(2)}(t)dt$:

$$w_{sys}^{(2)}(t)dt = \sum_{i=1}^{2} P(\theta_i \cap \overline{A}) - \sum_{i=2}^{2}\sum_{j=1}^{1} P(\theta_i \cap \theta_j \cap \overline{A})$$

$$= P(\theta_1 \cap \overline{A}) + P(\theta_2 \cap \overline{A}) - P(\theta_1 \cap \theta_2 \cap \overline{A}) \tag{3.29}$$

Using (3.26) to calculate the first term in (3.29):

$$P(\theta_1 \cap \overline{A}) = \sum_{i=1}^{2} P(\theta_1 \cap \overline{u_i}) - \sum_{i=2}^{2}\sum_{j=1}^{1} P(\theta_1 \cap \overline{u_i} \cap \overline{u_j})$$

$$= P(\theta_1 \cap \overline{u_1}) + P(\theta_1 \cap \overline{u_2}) - P(\theta_1 \cap \overline{u_1} \cap \overline{u_2}) \tag{3.30}$$

$$= P(\theta_1 \cap \overline{u_2}) = w_{X2}q_{X1}q_{X3}, \text{ (using equation 3.27)}$$

$P(\theta_1 \cap \overline{A}) = w_{X2}q_{X1}q_{X3}$

Note that all other terms in (3.30) are zero as a minimal cut set cannot already exist and then occur in time *dt*.

Again using (3.26) to evaluate the second term in (3.29):

$$P(\theta_2 \cap \overline{A}) = \sum_{i=1}^{2} P(\theta_2 \cap \overline{u_i}) - \sum_{i=2}^{2}\sum_{j=1}^{1} P(\theta_2 \cap \overline{u_i} \cap \overline{u_j})$$

$$= P(\theta_2 \cap \overline{u_1}) + P(\theta_2 \cap \overline{u_2}) - P(\theta_2 \cap \overline{u_1} \cap \overline{u_2}) \qquad (3.31)$$

$$= P(\theta_2 \cap \overline{u_1}) = w_{X3} q_{X1} q_{X2}$$

$$\mathrm{P}(\theta_2 \cap \overline{\mathrm{A}}) = \mathrm{w}_{\mathrm{X3}} \mathrm{q}_{\mathrm{X1}} \mathrm{q}_{\mathrm{X2}}$$

Note that all other terms in (3.31) are zero.

For the last term in (3.29) it is obvious that:

$$P(\theta_1 \cap \theta_2 \cap \overline{A}) = 0$$

Therefore:

$$w_{sys}^{(2)}(t) = P(\theta_1 \cap \overline{A}) + P(\theta_2 \cap \overline{A})$$

$$= w_{X2} q_{X1} q_{X3} + w_{X3} q_{X1} q_{X2}$$

$$=(3.4\text{e-}5)(0.04)^2+(2.9\text{e-}7)(0.04)^2$$

$$=5.4864\text{e-}8$$

Finally:

$$w_{sys}(t) = w_{sys}^{(1)}(t) - w_{sys}^{(2)}(t)$$

$$=1.93608\text{e-}6\text{-}5.4864\text{e-}8$$

$$=1.881216\text{e-}6$$

The expected number of system failures in any time period can then be obtained using equation (3.3).

It is obvious from the previous small example fault tree, that evaluating the system unconditional failure intensity is a tedious and time consuming task. This is due to the number of series expansion terms that are required in the evaluation of $w_{sys}^{(1)}(t)$ and $w_{sys}^{(2)}(t)$. Vesely (1) recognised the need for an approximation of $w_{sys}(t)$, similar to the approximations for the top event probability. The calculation of $w_{sys}^{(2)}(t)$ requires

minimal cut sets to exist and occur at the same time. If component failures are rare then the series expansion for $w_{sys}^{(2)}(t)$ can become negligible and approximated as zero. This will leave an upper bound approximation for $w_{sys}(t)$:

$$w_{sys}(t) = w_{sys}^{(1)}(t) \tag{3.32}$$

An even simpler upper bound approximation is $w_{sys}(t)_{\max}$ where:

$$w_{sys}(t)_{\max} = w_{sys}^{(1)}(t)_{\max} = \sum_{i=1}^{nc} P(\theta_i) \tag{3.33}$$

For the previous example $w_{sys}(t)_{\max}$ would be:

$w_{sys}(t)_{\max} = w_{X1}q_{X2} + w_{X2}q_{X1} + w_{X1}q_{X3} + w_{X3}q_{X1}$=1.9476e-6
(this compares to the exact answer of 1.881216e-6)

3.4 Alternative Approaches

Other researchers have dealt with the quantification of fault trees, Semanderes (19) developed the computer program ELRAFT (Efficient Logic Reduction Analysis of Fault Trees), which has been previously mentioned in Chapter 2 as a bottom-up algorithm. It was also conceived as a means of using Boolean algebra to calculate exact probabilities directly from the logic expression in an economical manner. Semanderes recognised the need to eliminate dependencies during the calculation of the top event probability. This method will be explained by means of an example. Consider the fault tree given in figure 3.2.

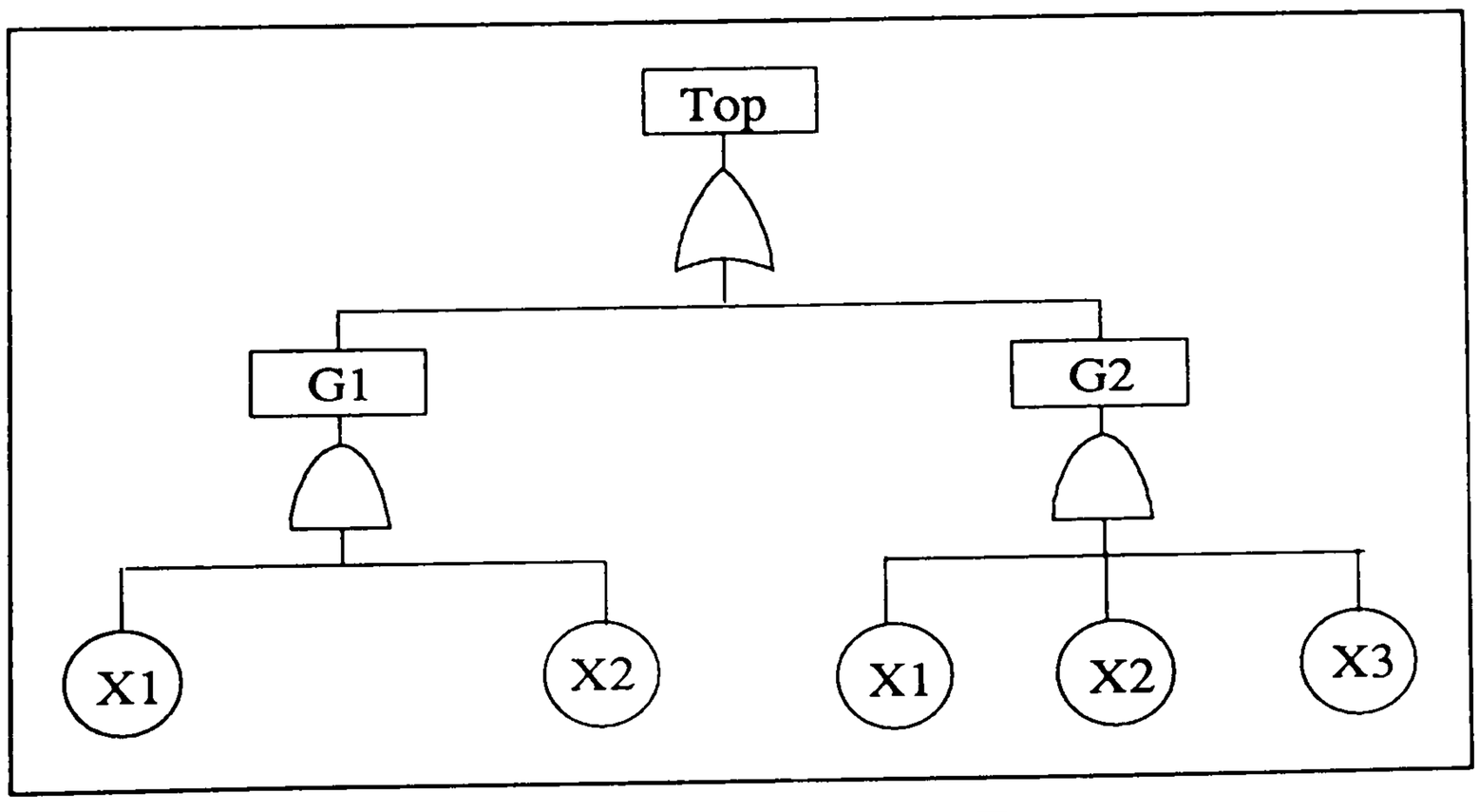


Figure 3.2 Example Fault Tree

From set theory and probability theory (23):

$$G1 = X1 \cap X2$$
$$P(G1) = P(X1).P(X2)$$

$$G2 = X1 \cap X2 \cap X3$$
$$P(G2) = P(X1).P(X2).P(X3)$$

$$\begin{aligned} Top &= G1 \cup G2 \\ P(Top) &= P(G1) + P(G2) - P(G1)P(G2) \\ &= P(X1).P(X2) + P(X1).P(X2).P(X3) - \\ &\quad P(X1).P(X2).P(X3) \\ P(Top) &= P(X1).P(X2) \end{aligned} \tag{3.34}$$

It is important to note that redundancies must be eliminated during the calculation, i.e. P(X1.X1)=P(X1) otherwise the answer will be incorrect.

Semanderes has proposed a 'Logical Model' to obtain the probability of a gate event in a fault tree.

The Logical Model

The probability of a gate event in the fault tree is obtained by:

1. Determining the possible combinations of basic events which cause the gate event.
2. Eliminating duplication of events within each combination.
3. Eliminating any combination which contains within itself some other possible combination.
4. The combinations that are left are for all intents and purposes mutually exclusive and the probability of the event is the sum of the probabilities of each combination.

It can be seen from point (4) that Semanderes employs the Rare Event approximation given in (3.14).

Zipf (31) implements Monte-Carlo Simulation in the CRESSC program for probability calculation. This algorithm has two options depending on whether the average (mean) top event unavailability or the failure frequency is required. The CRESSC method to calculate the average unavailability is:

1. Generate a subroutine representing the Boolean logic of the fault tree.
2. Convert components which have a time-dependent unavailability $U_j(t)$ to the average unavailability P_j.
3. For each component j a (0,1) uniformly distributed random variable Z_j is generated. If $\frac{Z_j}{P_j} < q$, then the component j will be considered as failed. It is stated that q is a parameter which is controlled in the program in such a way that a system failure occurs with approximately every other trial.
4. If system failure occurs then a maximum of two minimal cut sets is taken from the simulated failure combination (Zipf does not clarify this step further).
5. Steps (3) and (4) are repeated until the desired number of minimal cut sets are obtained or a pre-defined time limit is reached. If necessary, a restart of the program is possible.
6. The average unavailability of the system is evaluated analytically from the determined minimal cut sets. Additionally, the minimal cut sets are arranged according to their contribution to the average unavailability.
7. For control purposes the average unavailability is again calculated by simulating the up and down times of the system.

To obtain the failure frequency of the system, use the failure frequency for P_j in point (3) above instead of the average unavailability. Although a simulation program is simple to develop, there is no guarantee that all the important minimal cut sets will be obtained. Also for low failure probabilities ($<10^{-5}$) of a highly redundant system, an estimate of the error due to the non-considered minimal cut sets is difficult to achieve.

Part two of the paper by Bennetts (12) deals with interpreting a disjunctive normal form as a probability relationship. The algorithm deals with obtaining Boolean expressions for non-coherent fault trees, however the method may be applied to coherent fault trees. The procedure is based on comparing two terms taken from a disjunctive normal form (as generated by Bennetts algorithm) and determining whether they are a disjoint pair or not. If they are no further action is required, but if they are not modifications must be made. The recognition of disjoint groupings is based on the following theorem.

Theorem: Two conjunctive terms T_1 and T_2 will represent disjoint groupings if there exists at least one literal in T_1 such that the same literal occurs in its complemented form in T_2.

The procedure for identifying and modifying non-disjoint pairs can be generalised into the following.

Procedure : Let T_i and T_j be two terms whose relative complement $T_k = T_i/T_j$ is defined by the non-empty set $\{y_1, y_2, \ldots\ldots, y_n\}$. This set contains the elements that appear in T_i which do not appear in T_j. The procedure by which T_i and T_j are converted into a disjoint collection of terms is described by the following expansion:

$$\begin{aligned} T_i + T_j = T_i + \overline{y_1} T_j + y_1 \overline{y_2} T_j + y_1 y_2 \overline{y_3} T_j + \ldots \\ \ldots + (y_1 y_2 \ldots\ldots y_{r-1} \overline{y_r}) T_j + \ldots\ldots + (y_1 y_2 \ldots\ldots \overline{y_n}) T_j \end{aligned} \tag{3.35}$$

If T_i and T_j have different sizes, it is desirable to order them such that the smaller term occurs first, this will lead to a smaller relative complement. The disjunctive normal forms generated by the algorithm have already been ordered into ascending order of cardinality for reasons of efficiency. This implies that depending on the order of the expression, two different but both valid disjoint expressions can be found. To illustrate the use of equation (3.35) consider the following expression:

$$Z = A + B + C.D + D.E.\overline{F} \tag{3.36}$$

The steps to obtain a disjoint expression are:

(1) Take A, B as T_i, T_j respectively

$\therefore$ $\{y_1, y_2, \ldots\ldots\ldots, y_n\} = \{A\}$

$\therefore$ $A + B = A + \overline{A}.B$

(2) Take $\overline{A}.B$, C.D as T_i, T_j respectively

$\therefore$ $\{y_1, y_2, \ldots\ldots\ldots, y_n\} = \{\overline{A}, B\}$

$\therefore$ $\overline{A}.B + C.D = \overline{A}.B + \cancel{A}.\cancel{C}.\cancel{D} + \overline{A}.\overline{B}.C.D$

(A.C.D is made redundant by the A in step (1))

(3) Take $\overline{A}.\overline{B}.C.D$, $D.E.\overline{F}$ as T_i, T_j respectively

$\therefore$ $\{y_1, y_2, \ldots\ldots\ldots, y_n\} = \{\overline{A}, \overline{B}, C\}$

$\therefore$ $\overline{A}.\overline{B}.C.D + D.E.\overline{F} = \cancel{A}.\cancel{D}.\cancel{E}.\overline{F} + \overline{A}.\cancel{B}.\cancel{D}.\cancel{E}.\overline{F} + \overline{A}.\overline{B}.\overline{C}.D.E.\overline{F}$

(A.D.E.$\overline{F}$ is made redundant by the A in step (1) and $\overline{A}$.B.D.E.$\overline{F}$ by the $\overline{A}$.B in step (1))

Therefore the disjoint expression for (3.36) is:

$$Z = A + \overline{A}.B + \overline{A}.\overline{B}.C.D + \overline{A}.\overline{B}.\overline{C}.D.E.\overline{F}$$

The top event probability can now be calculated by summing the probability of each disjoint term in the expression.

Inagaki and Henley (24) have extended the methodology of Vesely, to obtain the probability of the top event and unconditional failure intensity, for non-coherent fault trees. In the paper they stress an important issue concerning non-coherent systems, "It is shown that repair and maintenance policies for non-coherent systems must be carefully formulated since, for example, component repair can lead to system failure."

One of the assumptions for their method is that repair of the component brings it to its like new condition. Also all the prime implicants have been determined by use of the top-down algorithm of Kumamoto and Henley (16).

For non-coherent systems, all prime implicants are not needed to quantify the top event, usually it is sufficient to use a coherent approximation. Chu and Apostolakis (25) have devised a 'minimal expression' approach, but this can lose important information regarding prime implicants which are deleted from the top event representation. Inagaki and Henley avoid this approach in their methodology. Instead they calculate the top event probability at time t using the inclusion-exclusion formula given in equation (3.13). However prime implicants are used in place of the minimal cut sets. The difference when using prime implicants is that, in the expansion procedure the probability of prime implicants that have got mutually exclusive basic events may occur (i.e. X1 and $\overline{X1}$ may reside in the same intersection). When this happens the probability will be zero.

Evaluation of $w_{sys}(t)$ for a non-coherent system is slightly more involved than Vesely's method for coherent systems. The procedure is best described by use of an example, the following notation is needed.

$\lambda_j(t)$-conditional failure intensity
$\mu_j(t)$-conditional repair intensity
$w_j(t)$-unconditional failure intensity

$v_j(t)$-unconditional repair intensity

Also note that $\lambda_j(t) = \dfrac{w_j(t)}{\overline{q_j(t)}}$.

Let a non-coherent system have the following three prime implicants.

(1) {X1, X2}
(2) {X1, X3}
(3) {X2, $\overline{X3}$}

First calculating $w_{sys}^{1}(t)$ from equation (3.23):

$$\begin{aligned} w_{sys}^{(1)}(t)dt &= \sum_{i=1}^{3} P(\theta_i) - \sum_{i=2}^{3}\sum_{j=1}^{i-1} P(\theta_i \cap \theta_j) + \ldots\ldots + (-1)^2 P(\theta_1 \cap \theta_2 \cap \theta_3) \\ &= P(\theta_1) + P(\theta_2) + P(\theta_3) - [P(\theta_1 \cap \theta_2) + \\ &\quad P(\theta_1 \cap \theta_3) + P(\theta_2 \cap \theta_3)] + P(\theta_1 \cap \theta_2 \cap \theta_3) \\ &= w_{C_1}dt + w_{C_2}dt + w_{C_3}dt - [w_{X1}q_{X2}q_{X3}dt + w_{X2}q_{X1}q_{\overline{X3}}dt + 0] + 0 \end{aligned}$$

Cancel *dt* from both sides

$$\begin{aligned} w_{sys}^{(1)}(t) &= w_{X1}q_{X2} + w_{X2}q_{X1} + w_{X1}q_{X3} \\ &\quad + w_{X3}q_{X1} + w_{X2}q_{\overline{X3}} + v_{X3}q_{X2} - w_{X1}q_{X2}q_{X3} \\ &\quad - w_{X2}q_{X1}q_{\overline{X3}} \end{aligned}$$

Next calculating $w_{sys}^{(2)}(t)$ using equation (3.25):

$$\begin{aligned} w_{sys}^{(2)}(t)dt &= \sum_{i=1}^{3} P(\theta_i \cap \overline{A}) - \sum_{i=2}^{3}\sum_{j=1}^{i-1} P(\theta_i \cap \theta_j \cap \overline{A}) + \ldots. \\ &\quad \ldots\ldots + (-1)^2 P(\theta_1 \cap \theta_2 \cap \theta_3 \cap \overline{A}) \\ &= P(\theta_1 \cap \overline{A}) + P(\theta_2 \cap \overline{A}) + P(\theta_3 \cap \overline{A}) \\ &\quad - [P(\theta_1 \cap \theta_2 \cap \overline{A}) + P(\theta_1 \cap \theta_3 \cap \overline{A}) + P(\theta_2 \cap \theta_3 \cap \overline{A})] \\ &\quad + P(\theta_1 \cap \theta_2 \cap \theta_3 \cap \overline{A}) \end{aligned} \tag{3.37}$$

$$\begin{aligned} P(\theta_1 \cap \overline{A}) &= P(\theta_1 \cap \overline{u_1}) + P(\theta_1 \cap \overline{u_2}) + P(\theta_1 \cap \overline{u_3}) \\ &\quad - [P(\theta_1 \cap \overline{u_1} \cap \overline{u_2}) + P(\theta_1 \cap \overline{u_1} \cap \overline{u_3}) + P(\theta_1 \cap \overline{u_2} \cap \overline{u_3})] \\ &\quad + P(\theta_1 \cap \overline{u_1} \cap \overline{u_2} \cap \overline{u_3}) \\ &= 0 + w_{X2}q_{X1}q_{X3} + w_{X1}q_{X2}q_{\overline{X3}} - [0 + 0 + 0] + 0 \end{aligned}$$

$$P(\theta_2 \cap \overline{A}) = P(\theta_2 \cap \overline{u_1}) + P(\theta_2 \cap \overline{u_2}) + P(\theta_2 \cap \overline{u_3})$$
$$-[P(\theta_2 \cap \overline{u_1} \cap \overline{u_2}) + P(\theta_2 \cap \overline{u_1} \cap \overline{u_3}) + P(\theta_2 \cap \overline{u_2} \cap \overline{u_3})]$$
$$+ P(\theta_2 \cap \overline{u_1} \cap \overline{u_2} \cap \overline{u_3})$$
$$= w_{X3} q_{X1} q_{X2} + 0 + w_{X3} q_{X1} q_{X2} - [0 + w_{X3} q_{X1} q_{X2} + 0] + 0$$

Here it is important to explain the calculation of $P(\theta_2 \cap \overline{u_3})$ and $P(\theta_2 \cap \overline{u_1} \cap \overline{u_3})$. For $P(\theta_2 \cap u_3)$ there are two possible ways in which this can happen:

(1) The basic events X2, X3 and $\overline{X3}$ must exist at time t and X1 occurs in $[t, t+dt)$.

(2) X1, X2 and $\overline{X3}$ must exist at time t and X3 must occur in $[t, t+dt)$.

Here (1) cannot happen since X3 and $\overline{X3}$ cannot exist at the same time. In case (2) the probability of occurrence of X3 in $[t, t+dt)$ should be a conditional probability, i.e. λ_{X3}, because $\overline{X3}$ exists at t. Thus:

$$P(\theta_2 \cap \overline{u_3}) = \lambda_{X3} q_{X1} q_{X2} q_{\overline{X3}} = w_{X3} q_{X1} q_{X2}$$

Note $(\lambda_{X3} = \dfrac{w_{X3}}{q_{\overline{X3}}})$

By a similar argument:

$$P(\theta_2 \cap \overline{u_1} \cap \overline{u_3}) = w_{X3} q_{X1} q_{X2}$$

Moving on:

$$P(\theta_3 \cap \overline{A}) = P(\theta_3 \cap \overline{u_1}) + P(\theta_3 \cap \overline{u_2}) + P(\theta_3 \cap \overline{u_3})$$
$$-[P(\theta_3 \cap \overline{u_1} \cap \overline{u_2}) + P(\theta_3 \cap \overline{u_1} \cap \overline{u_3}) + P(\theta_3 \cap \overline{u_2} \cap \overline{u_3})]$$
$$+ P(\theta_3 \cap \overline{u_1} \cap \overline{u_2} \cap \overline{u_3})$$
$$= w_{\overline{X3}} q_{X1} q_{X2} + \mu_{X3} q_{X2} q_{X1} q_{X3} + 0 - [\mu_{X3} q_{X2} q_{X1} q_{X3} + 0 + 0] + 0$$

Here the calculation of $P(\theta_3 \cap \overline{u_2})$ and $P(\theta_3 \cap \overline{u_1} \cap \overline{u_2})$, requires the conditional repair of X3, i.e. μ_{X3}, because X3 exists at t. This shows an important characteristic of non-coherent systems. Prime implicant (3) ceases to exist when X3 is repaired, however prime implicant (2) is created by this repair action, thus the top event continues to exist.

The remaining four terms in equation (3.37) are zero, as the prime implicants involved cannot exist and occur at the same time.

This gives:

$$
\begin{aligned}
w_{sys}(t) &= w_{sys}^{(1)}(t) - w_{sys}^{(2)}(t) \\
&= [w_{X1}q_{X2} + w_{X2}q_{X1} + w_{X1}q_{X3} \\
&\quad + w_{X3}q_{X1} + w_{X2}q_{\overline{X3}} + v_{X3}q_{X2} \\
&\quad - w_{X1}q_{X2}q_{X3} - w_{X2}q_{X1}q_{\overline{X3}}] \\
&\quad - [w_{X2}q_{X1}q_{X3} + w_{X1}q_{X2}q_{\overline{X3}} + w_{X3}q_{X1}q_{X2} + v_{X3}q_{X1}q_{X2}]
\end{aligned}
$$

The remainder of the paper discusses the necessary modifications to the theorem of Bennetts, to recognise disjointness in the Boolean expression, if $w_{sys}(t)$ is to be calculated.

Chu and Apostolakis (1980) (25) discuss in their paper methods for probabilistic analysis of non-coherent fault trees. The paper gives a very useful result which is that effort can be saved in the calculation of the top event, of a non-coherent fault tree, by first creating modules. Modules are individual subtrees of the fault tree which do not share common events. The NOT gates in the fault tree must be pushed to the bottom level i.e. to negate primary events, using De Morgans Laws. These modules can then be used to create more accurate upper and lower bounds for the top event probability. The purpose of the paper is to investigate the applicability of several well known methods, for the probabilistic analysis of coherent fault trees, to the analysis of non-coherent fault trees. The following methods are discussed:

(1) Inclusion-Exclusion Method
(2) Min-Max Bounds
(3) Minimal Cut Set Upper and Minimal Path Set Lower Bounds

Inclusion-Exclusion Method

Equation (3.13) given by Vesely is used for the inclusion-exclusion calculation, with the minimal cut sets (C_i) replaced by prime implicants (ξ_i) to give:

$$
\begin{aligned}
P(Top) = \sum_{i=1}^{nc} P(\xi_i) - \sum_{i=2}^{nc}\sum_{j=1}^{i-1} P(\xi_i \cap \xi_j) + \ldots \\
+ \ldots (-1)^{nc-1} P(\xi_1 \cap \xi_2 \cap \ldots \cap \xi_{nc})
\end{aligned}
\qquad (3.38)
$$

However in evaluating the products of prime implicants, conflicting literals must be deleted ($Xi\overline{Xi} = 0$) and the idempotent law must be applied ($XiXi = Xi$). As the calculation of (3.38) can be formidable for large systems, the following inclusion-exclusion bounds are given:

An odd number of terms gives an upper bound (3.39)

An even number of terms gives a lower bound (3.40)

$$Q_{sys}(t) \leq \sum_{i=1}^{nc} P(\xi_i) \tag{3.41}$$

It can be seen that equation (3.41) constitutes the Rare Event approximation given in (3.14). Although the Rare Event approximation can be a very good approximation to $Q_{sys}(t)$ for coherent systems, especially when the probabilities of primary inputs are less than 0.10, a general statement as to how good this bound is for non-coherent systems cannot be made. Therefore one must proceed to evaluate other bounds, i.e. those of (3.39) and (3.40) in order to satisfactorily bracket $Q_{sys}(t)$.

Min-Max Bounds

The min-max lower bound for the top event probability is calculated as:

$$\max_j \{\prod_{i \in \xi_j} P(X_i)\} \tag{3.42}$$

The min-max upper bound employs the minimal path sets η_i of the fault tree:

$$\min_j \{(1 - \prod_{i \in \eta_j} (1 - P(Xi)))\} \tag{3.43}$$

Chu and Apostolakis call K_j the **prime implicates**, where $(K_j = 1 - \prod_{i \in \eta_j} (1 - Xi))$.

The minimal path sets of a fault tree can be found by evaluating the prime implicants of the dual of the fault tree.

As an example consider the small fault tree shown in figure 3.3.

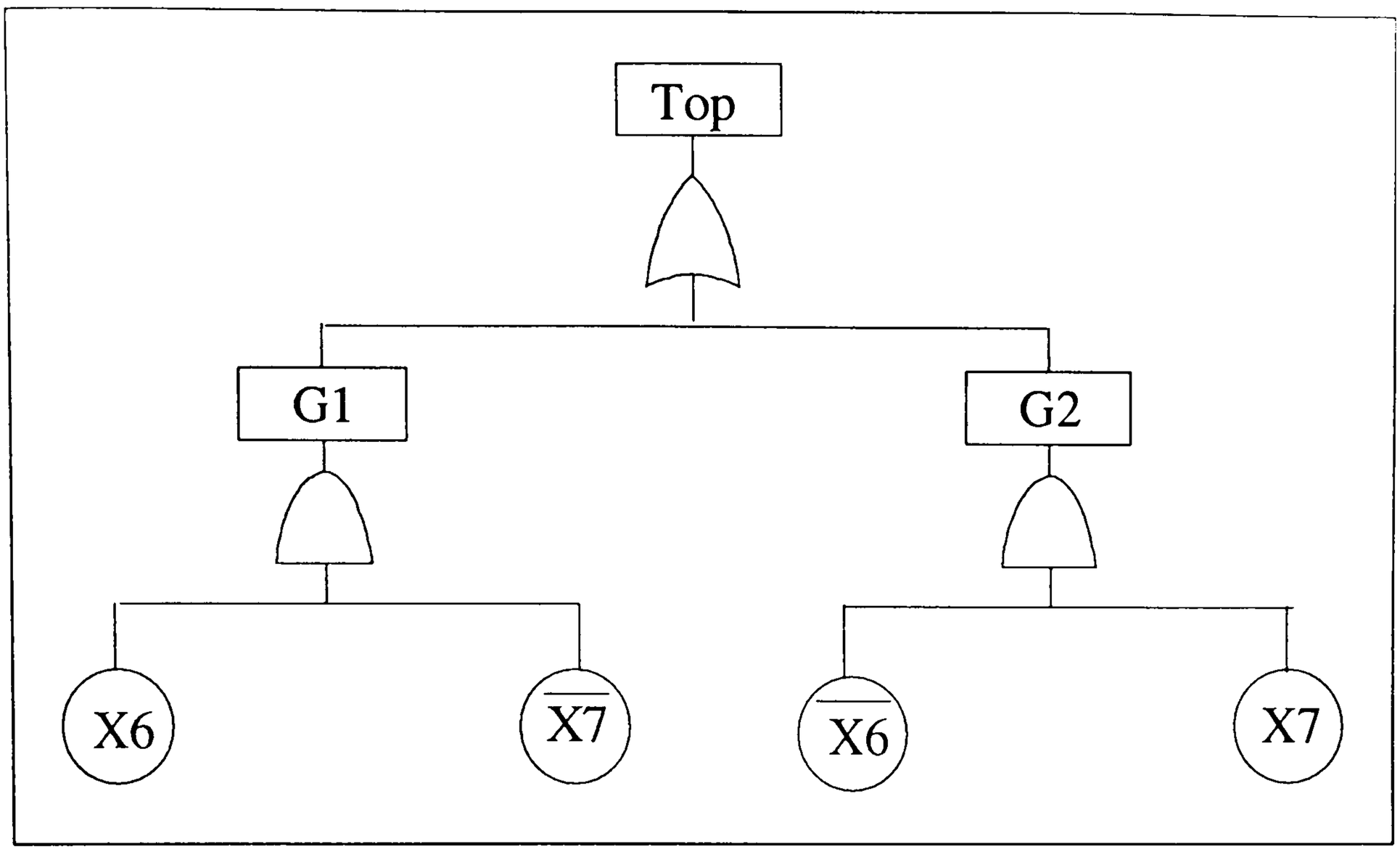


Figure 3.3 Small Non-Coherent Fault Tree

The prime implicants of the fault tree in figure 3.3 are $\xi_1 = X6.\overline{X7}$ and $\xi_2 = \overline{X6}.X7$ and the minimal path sets are $\eta_1 = X6.X7$ and $\eta_2 = \overline{X6}.\overline{X7}$. Let P(X6)=0.5 and P(X7)=0.7. To find the min-max lower bounds, we first determine the probabilities of the prime implicants.

$$P(\xi_1) = (0.5)(0.3) = 0.15$$
$$P(\xi_2) = (0.5)(0.7) = 0.35$$

The maximum of these probabilities, 0.35, is the min-max lower bound.

Similarly, the probabilities of the prime implicates are calculated.

$$\begin{aligned} P(K_1) &= 1-(1-P(X6))(1-P(X7)) \\ &= 1-(1-0.5)(1-0.7) = 0.85 \\ P(K_2) &= 1-(1-P(\overline{X6}))(1-P(\overline{X7})) \\ &= 1-(1-0.5)(1-0.3) = 0.65 \end{aligned}$$

The minimum of these probabilities, 0.65, is the min-max upper bound.

The following inequalities are always true.

$$\max_j\{\prod_{i\in\xi_j} P(X_i)\} \le Q_{sys}(t) \le \min_j\{(1-\prod_{i\in\eta_j}(1-P(Xi))\} \tag{3.44}$$

Obviously the best min-max bounds are obtained when all the prime implicants and prime implicates are known.

Minimal Cut Set Upper and Minimal Path Set Lower Bounds

For coherent structure functions the minimal cut set upper and minimal path set lower bounds are:

$$\prod_{i=1}^{m} K_i \le Q_{sys}(t) \le \coprod_{i=1}^{n} \xi_i \qquad (3.45)$$

However Chu and Apostolakis show that these bounds do not hold for non-coherent structure functions. Therefore one must be careful in the choice of bounds or approximations to be used for the quantitative analysis of a non-coherent fault tree.

Until now most of the researchers have been concerned with the probability of the top event occurrence or system failure, however Locks (26) wanted to estimate the reliability, R, of a coherent system. In the paper three different ways to estimate the reliability are compared, these are:

(1) Recursive disjoint products
(2) Recursive inclusion-exclusion
(3) Minimal cut set approximation based on partial information

Recursive Disjoint Products

The recursive disjoint products approach, as given by Abraham (27), builds the system reliability function for R by accumulating the probabilities of the m minimal path sets, η_i, one minimal path set at a time. Let R_j denote the probability accumulated at step j, the recursive formula is:

$$R_j = R_{j-1} + P(\eta_j \cap \overline{\eta_1} \cap \ldots \cap \overline{\eta_{j-1}}) \qquad (3.46)$$

by De Morgans Laws, both R_j and R_{j-1} are polynomials with all terms disjoint. Hence the accumulated probability is the sum of the probabilities of the terms.

Recursive Inclusion-Exclusion

Instead of accumulating the probabilities of disjoint terms as in the disjoint products approach, the polynomial has alternating plus and minus signs:

$$R_j = R_{j-1} + P_j - P(\eta_j \cap \bigcup_{i=1}^{j-1} \eta_i) \tag{3.47}$$

With the inclusion-exclusion method an upper limit of the number of terms is 2^{m-1}, where m is the number of minimal path sets. However the actual number of terms is usually a tiny fraction of this upper limit because of cancellation.

To illustrate the disjoint products method and inclusion-exclusion approach consider a fault tree which has the three minimal path sets, {a, b}, {a, c}, {a, d}.

First calculating disjoint products:

$$\begin{aligned}
R_1 &= R_0 + P(\eta_1) = 0 + P(a)P(b) = P(a)P(b) \\
R_2 &= R_1 + P(\eta_2 \cap \overline{\eta_1}) \\
&= R_1 + P(ac \cap \overline{ab}) = R_1 + P(ac \cap (\overline{a} \cup \overline{b})) \\
R_2 &= R_1 + P(a)P(\overline{b})P(c) \\
R_3 &= R_2 + P(\eta_3 \cap \overline{\eta_1} \cap \overline{\eta_2}) \\
&= R_2 + P(ad \cap \overline{ab} \cap \overline{ac}) \\
&= R_2 + P(ad \cap (\overline{a} \cup \overline{b}) \cap (\overline{a} \cup \overline{c})) \\
&= R_2 + P(a\overline{b}d \cap (\overline{a} \cup \overline{c})) \\
R_3 &= R_2 + P(a)P(\overline{b})P(\overline{c})P(d)
\end{aligned}$$

The final disjoint sum for the reliability is:

$$R = P(a)P(b) + P(a)P(\overline{b})P(c) + P(a)P(\overline{b})P(\overline{c})P(d)$$

The recursive build-up of the system reliability function by the inclusion-exclusion method is:

$$R_1 = R_0 + P_1 - P(\eta_1 \cap \cup \eta_0) = P(a)P(b)$$

$$R_2 = R_1 + P_2 - P(\eta_2 \cap \eta_1)$$
$$= R_1 + P(a)P(c) - P(ac \cap ab)$$
$$R_2 = R_1 + P(a)P(c) - P(a)P(b)P(c)$$
$$R_3 = R_2 + P_3 - P(\eta_3 \cap (\eta_1 \cup \eta_2))$$
$$= R_2 + P(a)P(d) - P(ad \cap ab \cup ad \cap ac)$$
$$= R_2 + P(a)P(d) - P(abd \cup acd)$$
$$= R_2 + P(a)P(d) -$$
$$[P(a)P(b)P(d) + P(a)P(c)P(d) - P(a)P(b)P(c)P(d)]$$

The final inclusion-exclusion system reliability function is:

$$R = P(a)P(b) + P(a)P(c) + P(a)P(d)$$
$$- P(a)P(b)p(c) - P(a)P(b)P(d) + P(a)P(b)P(c)P(d)$$

Locks states that it is desirable to use the disjoint products approach over the inclusion-exclusion method, as the disjoint products approach results in a smaller polynomial.

Minimal Cut Set Approximation

A variety of minimal cut set approximations can be used to estimate system reliability. The primary justification for approximating is that with high reliability, the probability of failure is small relative to the probability of success. The only problem with the Rare Event approximation $\sum P(C_i)$ is that all the minimal cut sets are needed.

If only a subset of the minimal cut sets are known and the components are highly reliable, an upper bound can be obtained, using the most important minimal cut sets. The common aspect of all importance schemes is that if only the information relevant to the most important minimal cut sets is used, enough failure probability is accounted for to provide an accurate bound. Since the components are highly reliable, the most important minimal cut sets often have the smallest number of components. Locks shows that it is not always necessary to use exact methods such as the inclusion-exclusion method or the disjoint products approach with full information, when excellent approximations can be obtained with the important minimal cut sets and partial information.

3.5 Importance Measures

A very useful piece of information which can be derived from a fault tree study is the **importance** measure for each component or each minimal cut set. An importance analysis is a sensitivity analysis which identifies weak areas of the system and can be very valuable if used at the system design stage. For each component its importance signifies the role that it plays in either causing or contributing to the occurrence of the top event. In general a numerical value is assigned to each basic event which allows it to be ranked along with other failure events according to the extent of its contribution to the occurrence of the top event.

Probabilistic importance measures can be categorised in two ways: (i) those which are appropriate for system availability assessment (top event probability) and (ii) those which are concerned with system reliability assessment (expected number of top event occurrences).

The most commonly used importance measures are described below.

(i) Top Event Probability

Birnbaum Measure of Component Importance

The Birnbaum measure of importance (I_b) was first introduced back in 1969 (3). This measure is defined as the rate at which the system failure probability improves as the failure probability of component i improves. Birnbaums measure of importance is also known as the criticality function, $G_i(\mathbf{q})$, which can be expressed in two ways:

$$\text{(i) } G_i(\mathbf{q}) = Q(1_i, \mathbf{q}) - Q(0_i, \mathbf{q}) \tag{3.48}$$

where $Q(\mathbf{q})$ is the probability that the system fails expressed as a function of the component failure probabilities where:

$$Q(1_i, \mathbf{q}) = (q_1, \ldots . q_{i-1}, 1, q_{i+1}, \ldots, q_n)$$
$$Q(0_i, \mathbf{q}) = (q_1, \ldots . q_{i-1}, 0, q_{i+1}, \ldots, q_n)$$

$$\text{(ii) } G_i(\mathbf{q}) = \partial Q(\mathbf{q}) / \partial q_i \tag{3.49}$$

This is defining the criticality function as a partial derivative which is the same as the first expression.

To illustrate, let $Q(\mathbf{q})$ for a system be $Q(\mathbf{q}) = q_{X1}q_{X2} + q_{X1}q_{X3} - q_{X1}q_{X2}q_{X3}$, using (3.48):

$$Q(1_{X1}, \mathbf{q}) = q_{X2} + q_{X3} - q_{X2}q_{X3}$$
$$Q(0_{X1}, \mathbf{q}) = 0$$
$$G_{X1}(\mathbf{q}) = Q(1_{X1}, \mathbf{q}) - Q(0_{X1}, \mathbf{q}) = q_{X2} + q_{X3} - q_{X2}q_{X3}$$

Criticality Measure of Component Importance

The criticality measure of importance (I_C) is "The probability that the system is in a state at time *t* in which component *i* is critical **and** that component *i* has failed at time *t* conditional on system failure at time *t*."

$$I_{C_i} = \frac{G_i(\mathbf{q})q_i(t)}{Q_{sys}(t)} \tag{3.50}$$

It can be seen from equation (3.50) that the criticality measure of importance for a component *i*, is the product of its criticality function and unavailability divided by the system unavailability.

Fussell-Vesely Measure of Component Importance

This measure of Importance is usually close in numerical value to the Criticality Measure. The Fussell-Vesely Importance (I_{FV}) is calculated as the probability of the Union of the Minimal Cut Sets which contain event *i* divided by the top event occurrence probability. I_{FV_i} therefore gives the probability that when the system fails, component *i* contributed to the failure.

$$I_{FV_i} = \frac{P(\bigcup_{k/i \in k} C_k)}{Q_{sys}(t)} \tag{3.51}$$

Consider the minimal cut sets of a fault tree to be (1) {X1.X2}, (2) {X1.X3}. Therefore $I_{FV_{X2}}$ will be:

$$I_{FV_{X2}} = \frac{P(X1.X2)}{P(X1.X2)+P(X1.X3)-P(X1.X2.X3)}$$

(ii) Top Event Expected Number of Occurrences

The two component importance measures given in this category indicate the contribution made to the expected number of occurrences of the top event. These measures are appropriate for interval reliability assessments in which the order of occurrence of component failure events is of concern. Hence the two measures rank separately those events which are **initiators** and those which are **enablers** (these are further discussed in Chapter 7). Both measures were developed by Barlow and Proschan (4).

Barlow-Proschan Measure of Initiator Importance

If components fail sequentially in time and only one component failure event can occur in a vanishingly small time element *dt* then one event must have caused the system failure – the initiating event. The Barlow-Proschan Initiator importance BPI_i is the conditional probability that initiating event *i* caused the failure, given that the system fails prior to time *t*.

$$BPI_i = \frac{\int_0^t G_i(\mathbf{q}) w_i(t)\,dt}{W(0,t)} \tag{3.52}$$

Therefore the Barlow-Proschan measure of initiator importance requires the criticality function, component unconditional failure intensity and the expected number of top event occurrences for its evaluation.

Barlow-Proschan Measure of Enabler Importance

This importance measure (BPE_i) assesses the contribution of an enabling component *i* when initiating event *j* causes the system failure. The failure of enabler *i* is only then a factor when enabler *i* and initiator *j* both occur in the same minimal cut set (C).

$$BPE_i = \frac{\sum_{\substack{j \\ i \neq j \\ i,j \in C}} \int_0^t \{Q(1_i,1_j,\mathbf{q}) - Q(1_i,0_j,\mathbf{q})\} q_i(t) w_j(t)\,dt}{W(0,t)} \tag{3.53}$$

The calculation of this importance measure is a bit more involved than the others, because of the calculation of $Q(1_i,1_j,\mathbf{q})$ and $Q(1_i,0_j,\mathbf{q})$.

3.6 Summary

Having reviewed the literature the following points summarise the current techniques available to quantify fault trees:

(1) The primary limitation of the "Kinetic Tree Theory" by Vesely is the assumption that all the minimal cut sets of the fault tree are known. For some complex fault trees obtaining all the minimal cut sets can be a formidable task.

(2) The inclusion-exclusion formula for the probability calculation is a very computer intensive calculation. For all but the simplest of systems this is not possible to compute even on modern high-speed digital computers. Acceptably accurate approximations are required.

(3) The Rare Event approximation and the Minimal Cut Set Upper Bound are only suitable to use if the failure of each basic event is rare. If this is not the case these approximations may result in large inaccuracies.

(4) The structure function technique to evaluate the top event probability can save on effort if each minimal cut set is independent, in this case the Minimal Cut Set Upper Bound is exact. However this is rare and a checking procedure would be required to determine whether the minimal cut sets are independent, which would increase computation.

(5) Applying Shannon's decomposition to the structure function is difficult to program and the choice of the pivoting variable is sometimes difficult to establish in order to increase efficiency.

(6) The various techniques such as structure functions and Shannon's decomposition serve more as an alternative method rather than an improved method to calculate the top event unavailability.

(7) If calculating the top event probability requires a total number of calculations of the order of N, calculating the unconditional failure intensity for a system requires of the order of N^2 calculations. The calculation of the unconditional failure intensity again requires approximations and these will require the same assumptions and have the same limitations as listed for the top event unavailability.

(8) Importance measures are extremely useful pieces of information from a fault tree study but can only be accurately evaluated if $Q_{sys}(t)$ and all the minimal cut sets are known.

(9) Since Kinetic Tree Theory was first developed in the early 1970's its efforts have concentrated on making minor modifications to the established method to improve accuracy and efficiency. As such research into quantitative fault tree analysis suffers from the same limitations as that for qualitative analysis and no new method has been proposed which produces dramatic improvements.

CHAPTER 4

BINARY DECISION DIAGRAMS

4.1 Introduction

Chapters 2 and 3 have described and discussed the various methods developed to qualitatively and quantitatively analyse fault trees. These techniques are not without their limitations. If, because of the complexity of the system under study, the fault tree is large then finding the causes of failure, termed minimal cut sets, can require an extensive computer processing capability. To then quantify top event parameters usually means resorting to approximations. Tackling these problems to improve computational efficiency and accuracy has been the main concern over the years for many fault tree researchers. Semanderes (19), Fussell and Vesely (20), Bennetts (12) and Benjiamin et al. (28), have all addressed these issues. New techniques are usually developed by modifying and extending the established, conventional approaches such as MOCUS (20). However these 'bottom-up' or 'top-down' approaches are now so well developed that further refinement is unlikely to result in significant reductions in computation time or increases in accuracy. Therefore it is felt that substantial improvement in computer utilisation will only result from a completely new approach.

Recent papers by Akers (33), Bryant (34) and more importantly Rauzy (35) have indicated that an alternative approach to fault tree analysis which utilises Binary Decision Diagrams has the potential to provide a faster means of analysing fault trees. Binary Decision Diagrams were first introduced by Lee (36) to represent switching circuits. Rather than analysing the fault tree directly as with conventional approaches, this new approach first converts the fault tree to a binary decision diagram, from which the minimal cut sets are obtained. This diagram specifies the failure logic equation in a form which is easier to manipulate than a fault tree. This chapter describes the use of a Binary Decision Diagram (BDD) for fault tree analysis and some ways in which it can be efficiently implemented on a computer.

4.2 Description of the Binary Decision Diagram

A BDD is a directed acyclic graph as can be seen from the example illustrated in figure 4.1. All paths through the BDD start at the root vertex and terminate in one of two

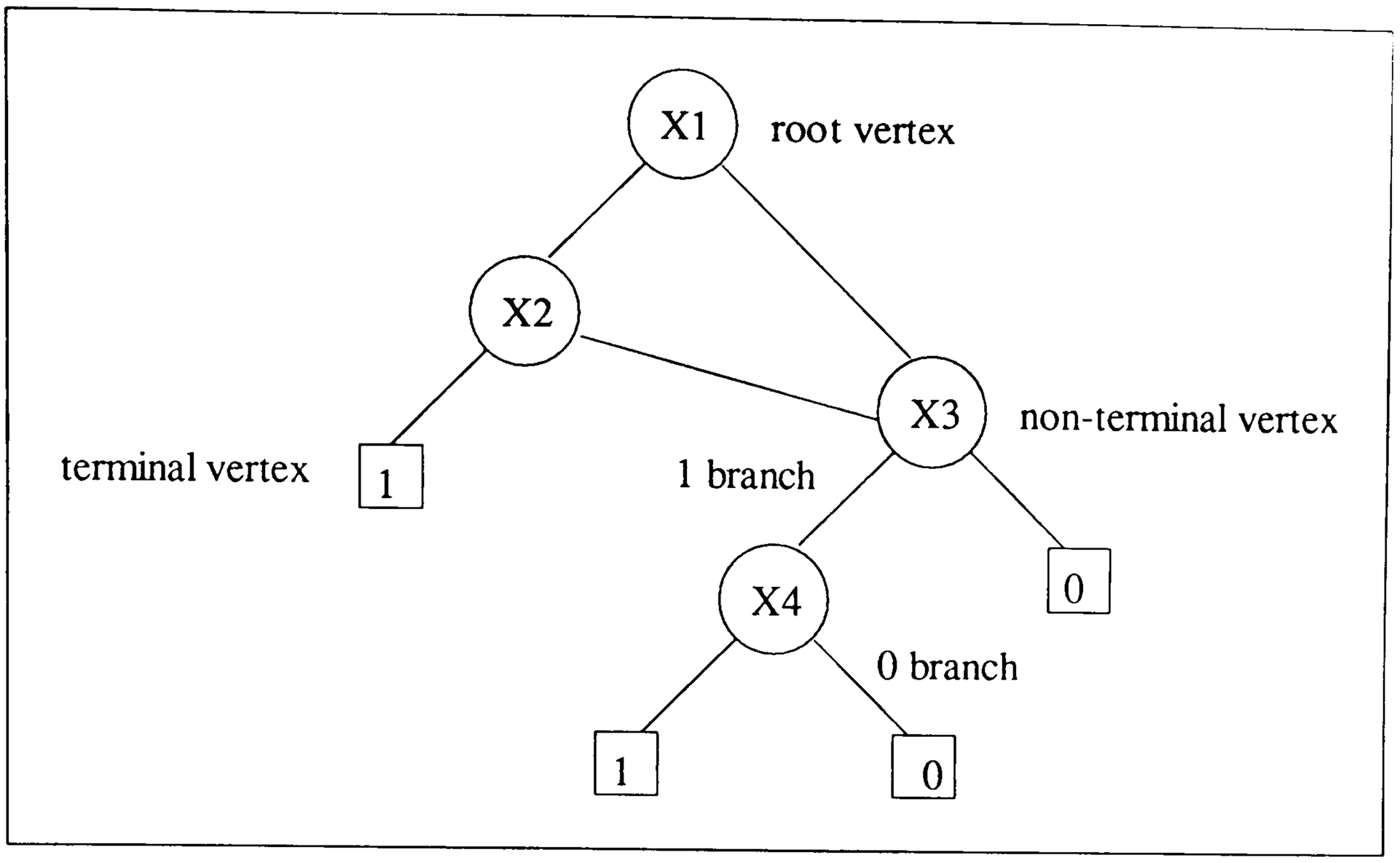


Figure 4.1 Example BDD

states, either a 1 state which corresponds to system failure, or a 0 state which corresponds to system success. All the paths terminating in a 1 state give the cut sets of the fault tree. A BDD is composed of terminal and non-terminal vertices or nodes, which are connected by branches. Terminal vertices have the value 0 or 1 and non-terminal vertices correspond to the basic events of the fault tree. Each vertex has a 0 exit branch which represents the basic event non-occurrence (works) and a 1 exit branch which represents basic event occurrence (fails).

All the left hand branches leaving each vertex are the 1 branches and all the right hand branches are the 0 branches. Notice that in figure 4.1 node X3 is shared by the right branch of X1 and the right branch of X2, the importance of this "sub-node sharing" will be discussed later.

Every path starts from the root vertex, and proceeds down through the diagram to the terminal vertices. Only the vertices that lie on a 1 branch on the way to a terminal 1 vertex are included in a path. The paths through a BDD, constructed for a particular fault tree, correspond to the cut sets of that fault tree. For example the paths, or cut sets, of the BDD shown in figure 4.1 are :

(1) X1.X2
(2) X1.X3.X4
(3) X3.X4

4.3 Constructing the Binary Decision Diagram Using Structure Functions

The BDD can be constructed from the structure function (defined in section 3.3.2) which is represented by the fault tree. This is achieved by successively substituting the value 1 (fails) and then the value 0 (works) for each node encountered in the BDD. To illustrate, consider the simple fault tree in figure 4.2.

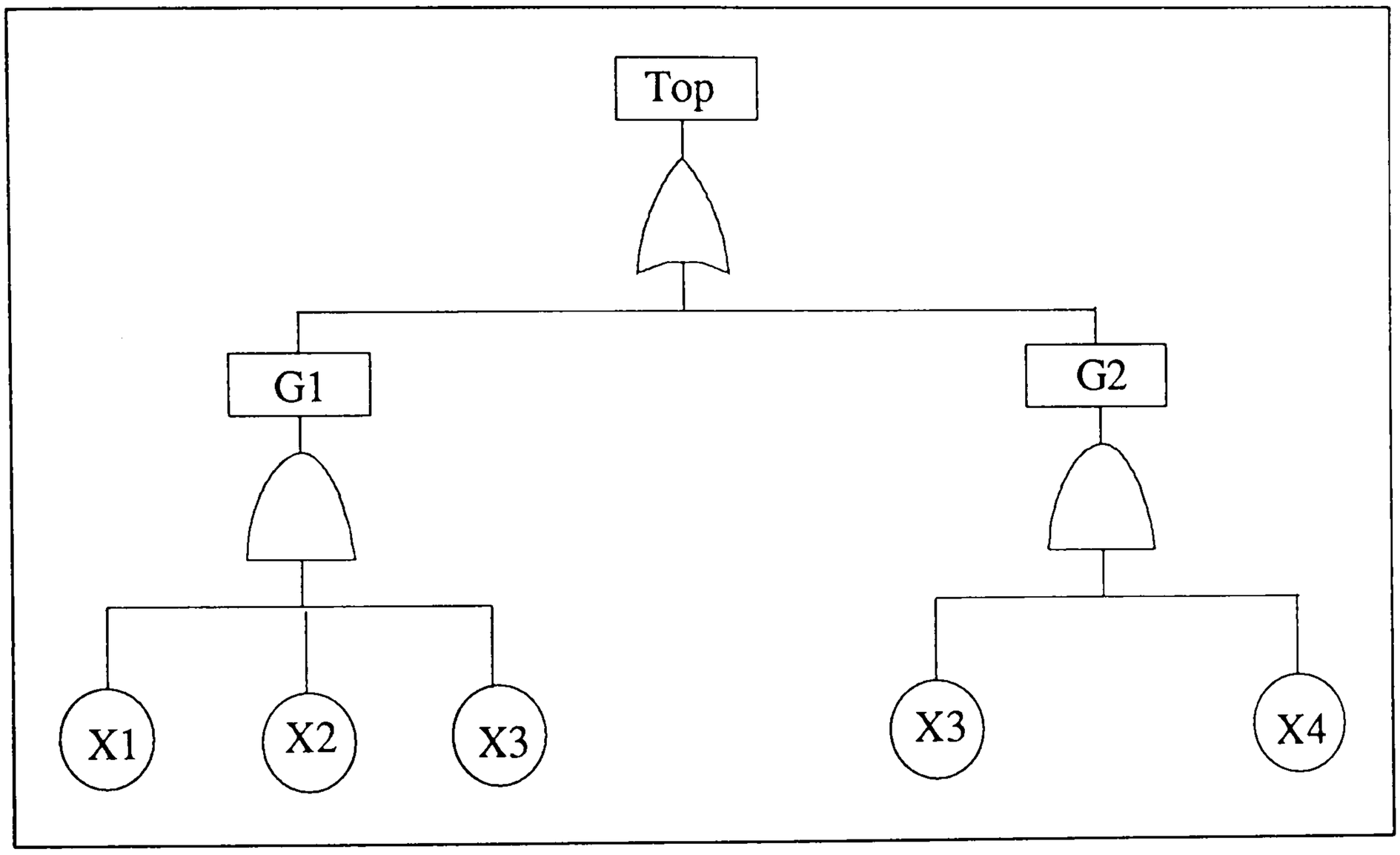


Figure 4.2 Example Fault Tree

The minimal cut sets for this fault tree are:

(1) {X1, X2, X3}
(2) {X3, X4}

Therefore its structure function, $\phi(\mathbf{x})$ is:

$$\phi(\mathbf{x})=1-(1-X1.X2.X3)(1-X3.X4) \tag{4.1}$$

We need to know the order in which to consider the basic events in the structure function before we can assign values of 0 and 1 to their indicator variable. Here a top-down, left-right ordering of basic events would give.

$$X1<X2<X3<X4 \tag{4.2}$$

which will yield the BDD shown in figure 4.3 (figure 4.3a with the Boolean equations to show its development from the structure function and its simplified form in figure 4.3b). Using this ordering node X1 is drawn first, along with its 1 and 0 branches. The structure functions or residues, resulting when X1=1 and when X1=0 are substituted into the original structure function, are indicated on its left and right branches respectively. Next node X2 is considered and its 1 and 0 branches drawn, again the residue structure functions are attached to these branches when X2 is given the value 1 and then the value 0. This process is continued for the other basic events until terminal 1/0 vertices are reached.

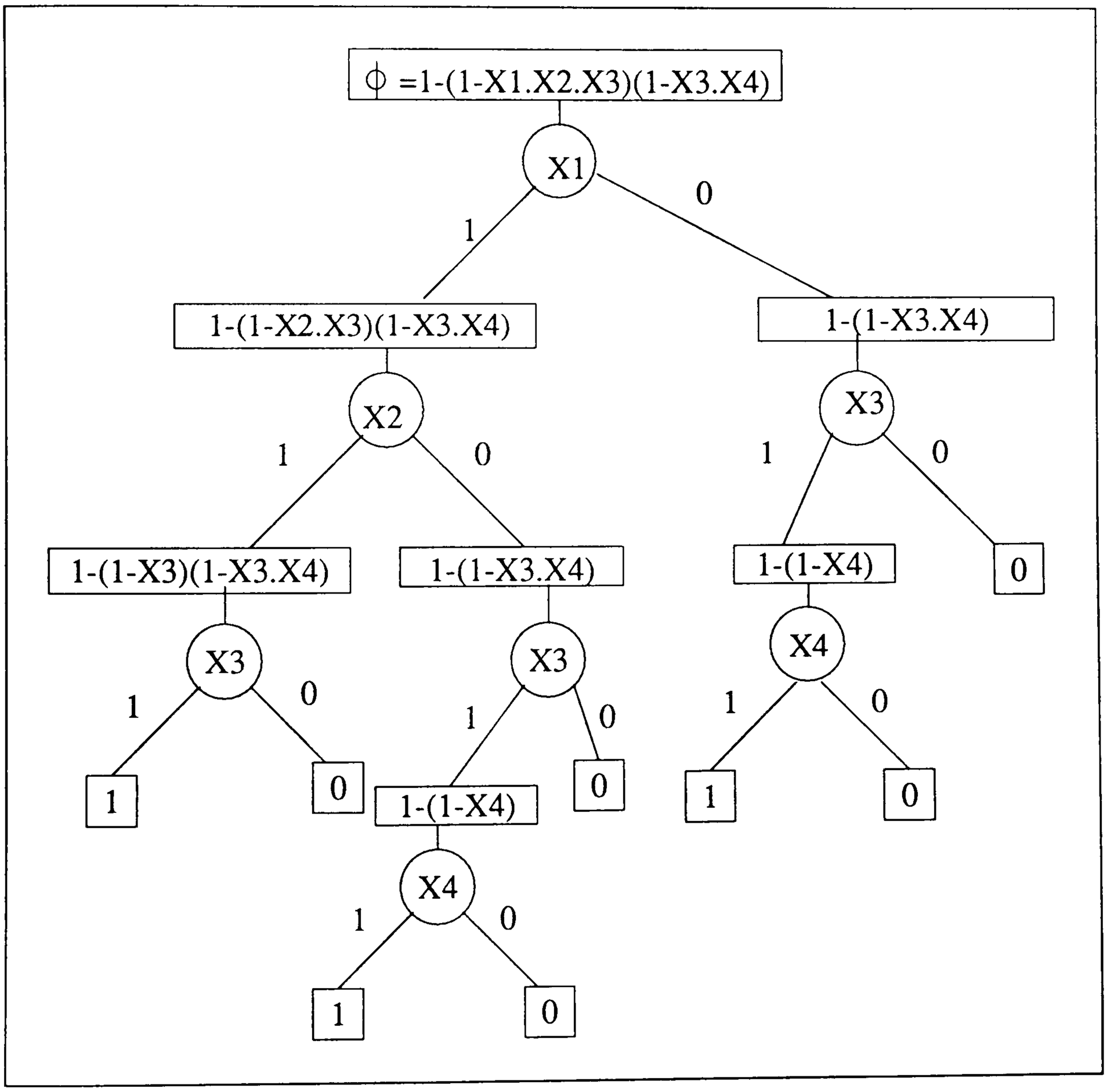


Figure 4.3a. BDD with Boolean Equations

4.4 Reducing the Binary Decision Diagram Structure

Friedman and Supowit (37) stated that a tree representing a function of n variables can have a maximum of 2^{n+1}-1 nodes, however they recognised that a BDD can be reduced in size by two 'collapsing' operations which are:

(1) If the two sons of a node *a* are equivalent, then delete node *a* and direct all of its incoming edges to its left son.
(2) If nodes *a* and *b* are equivalent, then delete node *b* and direct all of its incoming edges to *a*.

where a 'son' of a node is simply the node attached to either its 1 or 0 branch, e.g. in figure 4.3b F2 and F4 are the sons of node F1.

The above 'collapsing' operations have been applied to figure 4.3a to create the simplified BDD in figure 4.3b. Notice that the repeated X3 structure in figure 4.3a has been 'collapsed' to the shared sub-node F4 in figure 4.3b.

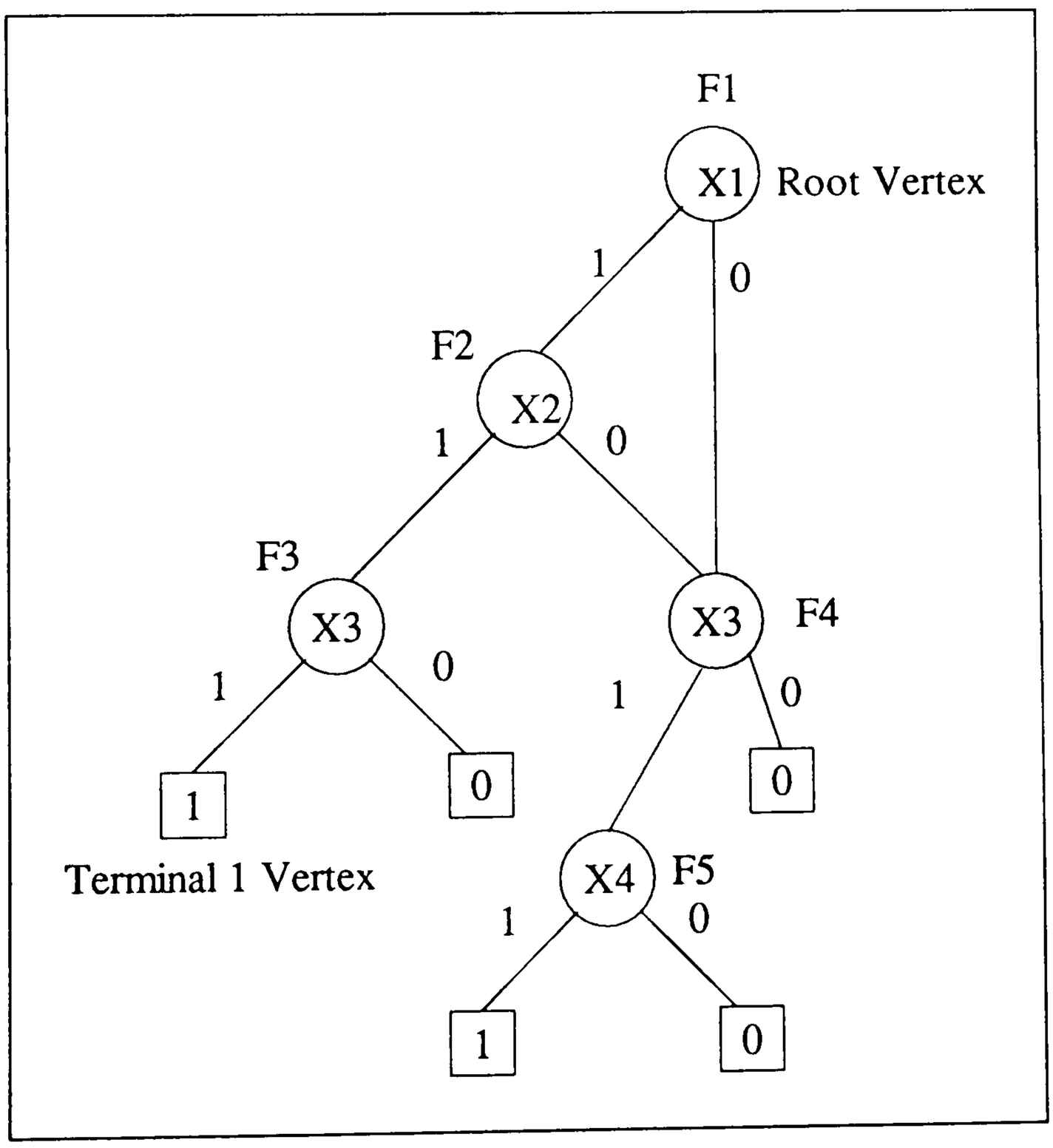


Figure 4.3b. BDD for the Fault Tree Shown in Figure 4.2

To obtain the cut sets of the fault tree the paths through the BDD are traced from the top or root vertex to a terminal 1 vertex. Remember only the basic events that lie on a 1 branch (indicating the failure of that basic event) on the way to a terminal 1 vertex are included in a path. Therefore the paths through the BDD which correspond to the cut sets of the fault tree are:

(1) X1.X2.X3
(2) X1.X3.X4
(3) X3.X4

Clearly the resulting BDD for this basic event ordering is not minimal, i.e. the paths through the BDD result in at least one redundant cut set. For this example cut set (2) above is redundant. To obtain only the minimal cut sets the BDD needs to undergo a minimisation procedure, whose details are given in section 4.6.

This method of BDD construction requires the availability of the structure function. The structure function is rarely available in a fault tree study and therefore an alternative approach is required to convert the fault tree structure to its equivalent binary decision diagram to obtain the minimal cut sets.

4.5 Constructing the Binary Decision Diagram Using an 'ite' Procedure

The BDD method developed by Rauzy (35) first converts the fault tree to a binary decision diagram which encodes an **If-Then-Else** (**ite**) structure. An attractive feature of the BDD method is that the **ite** structure derives from Shannon's formula (Chapter 2, section 2.3), such that if f(**x**) is the Boolean function for the fault tree top event then by pivoting about any variable X1 the Shannon formula can be written as:

$$f(\mathbf{x}) = X1.f1 + \overline{X1}.f2 \qquad (4.3)$$

where f1 and f2 are Boolean functions with X1=1 and X1=0 respectively and are of one order less than f. The corresponding **ite** structure is **ite**(X1, f1, f2), where X1 is the Boolean variable and f1 and f2 are logic functions. This means **if** X1 fails **then** consider function f1 **else** consider function f2. Therefore in the BDD f1 is represented by the structure lying below the 1 branch of X1 and f2 is represented by the structure lying below the 0 branch. The diagram for this is the one given in figure 4.4.

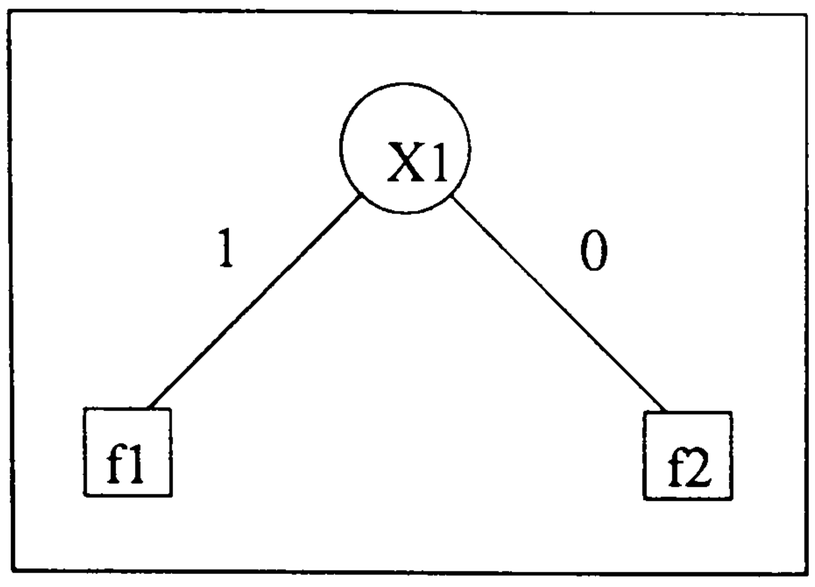


Figure 4.4 BDD for **ite**(X1, f1, f2)

Once the basic events in the fault tree have been given an ordering, the following procedure is then used to construct the BDD from the fault tree. Rauzy uses a top-down ordering i.e., the basic events which are placed higher up the tree are listed first and are regarded as being "less than" those lower down the tree. Note that in the following procedures <op> corresponds to a Boolean operation of the logic gates in the fault tree, so if the gate is an AND gate <op> will be the dot or product symbol (·), and if the gate is an OR gate <op> will be the sum symbol (+).

Procedure

(1) Assign each basic event, Xi in the fault tree the **ite** structure **ite**(Xi, 1, 0), (Xi can either fail - 1 branch or work - 0 branch).

(2) If x<y;

Let J=**ite**(x, F1, F2) and H=**ite**(y, G1, G2) then

J<op>H=**ite**(x, F1<op>H, F2<op>H)

(3) If x=y;

i.e., J=**ite**(x, F1, F2) and H=**ite**(x, G1, G2) then

J<op>H=**ite**(x, F1<op>G1, F2<op>G2)

(4) If F is a k/n vote gate with inputs F1,.....,Fn, i.e. F=*at-least*(k, F1,.....,Fn), then F is rewritten implicitly as (F1∩*at-least*(k-1, F2,.....,Fn))∪*at-least*(k, F2,.....,Fn).

These are used in conjunction with the following identities to produce the simplest **ite** structure for each gate:

1<op>H=1 if <op> is an OR gate

1<op>H=H if <op> is an AND gate

0<op>H=H if <op> is an OR gate

0<op>H=0 if <op> is an AND gate

To illustrate the application of this method consider the fault tree shown in figure 4.5.

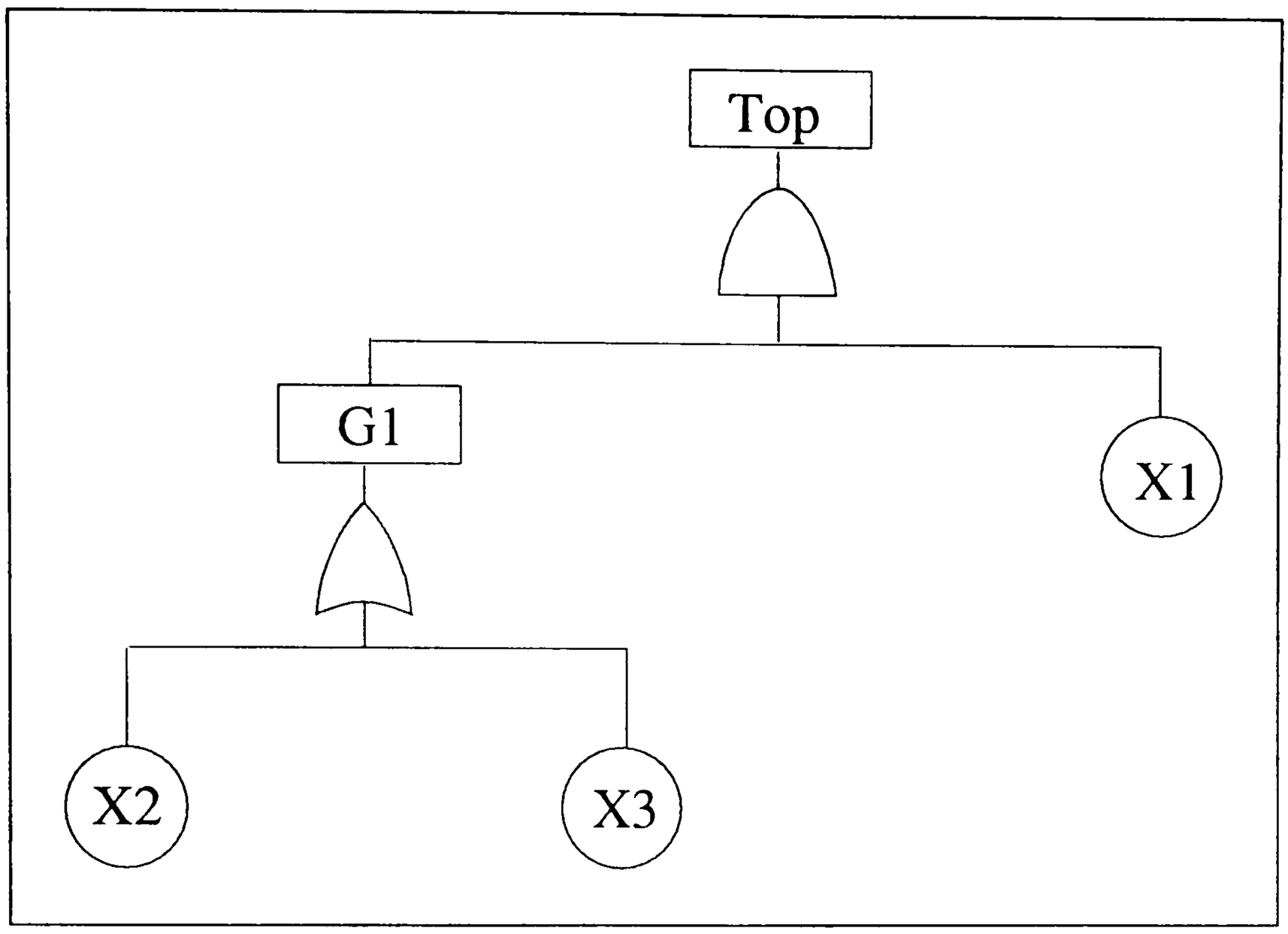


Figure 4.5 Example Fault Tree

Using an ordering of X1<X2<X3 for the fault tree in figure 4.5 the **ite** structure for the top event is obtained as follows:

Working from the bottom to the top of the tree and applying the procedure defined above we get:

G1=**ite**(X2, 1, 0)+**ite**(X3, 1, 0)
=**ite**(X2, 1+**ite**(X3, 1, 0), 0+**ite**(X3, 1, 0))
=**ite**(X2, 1, **ite**(X3, 1, 0))

Top=G1.X1
=**ite**(X2, 1, **ite**(X3, 1, 0)).**ite**(X1, 1, 0)
=**ite**(X1, 1.**ite**(X2, 1, **ite**(X3, 1, 0)), 0.**ite**(X2, 1, **ite**(X3, 1, 0)))
Top=**ite**(X1, **ite**(X2, 1, **ite**(X3, 1, 0)), 0)

To construct the BDD each **ite** structure in Top is successively broken down into its left and right branches. The root node of Top is the variable X1, the structure **ite**(X2, 1, **ite**(X3, 1, 0)) will lie below the 1 branch of X1 and the 0 branch of X1 will

terminate in a 0 end vertex. Next the **ite** structure **ite**(X2, 1, **ite**(X3, 1, 0)) is broken down into its left and right branches. The resulting BDD is the one shown in figure 4.6.

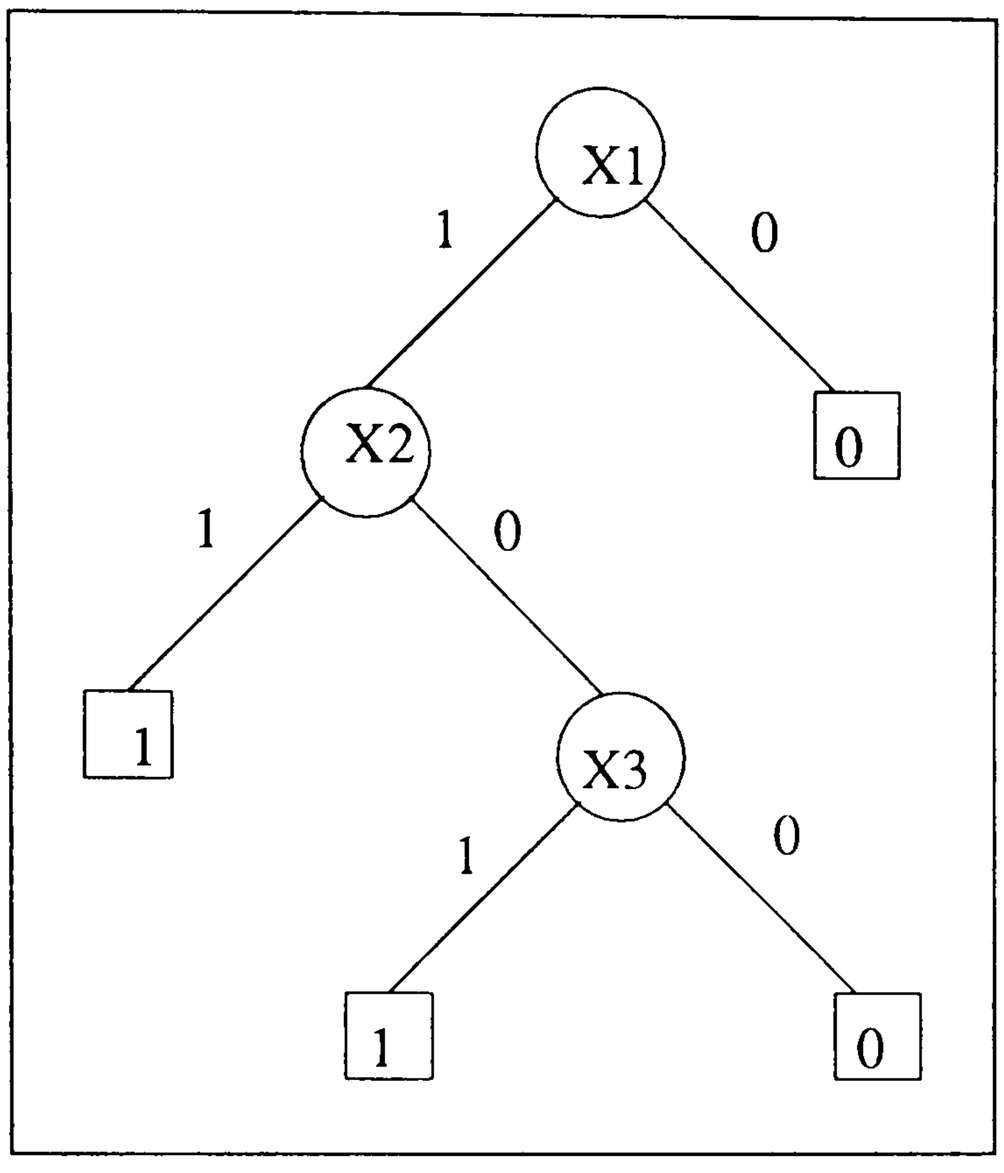


Figure 4.6 BDD for Example Fault Tree

Next, the paths through the BDD are obtained these being:

(1) X1.X2
(2) X1.X3

which correspond to the minimal cut sets of the fault tree.

To illustrate the **ite** calculation of a vote gate refer to the fault tree shown in figure 4.7.

Calculating the **ite** structure for G1, using the ordering X1<X2<X3:

G1=*at-least*(2, X1, X2, X3)
=(X1∩*at-least*(1, X2, X3))∪*at-least*(2, X2, X3)
=(X1∩[(X2∩*at-least*(0,X3))∪*at-least*(1, X3))]∪*at-least*(2, X2, X3)
=(X1∩[X2∪X3])∪(X2∩X3)

changing ∩→·, ∪→+ and giving the **ite** property to the basic events:

G1=**ite**(X1, 1, 0).[**ite**(X2, 1, 0)+**ite**(X3, 1, 0)]+[**ite**(X2, 1, 0).ite(X3, 1, 0)]

dealing with square brackets first:

=ite(X1, 1, 0).**ite**(X2, 1, **ite**(X3, 1, 0))+**ite**(X2, **ite**(X3, 1, 0), 0)
=ite(X1, **ite**(X2, 1, **ite**(X3, 1, 0)), 0)+**ite**(X2, **ite**(X3, 1, 0), 0)
=ite(X1, **ite**(X2, 1, **ite**(X3, 1, 0))+**ite**(X2, **ite**(X3, 1, 0), 0), **ite**(X2, **ite**(X3, 1, 0), 0))

G1=ite(X1, **ite**(X2, 1, **ite**(X3, 1, 0)), **ite**(X2, **ite**(X3, 1, 0), 0))

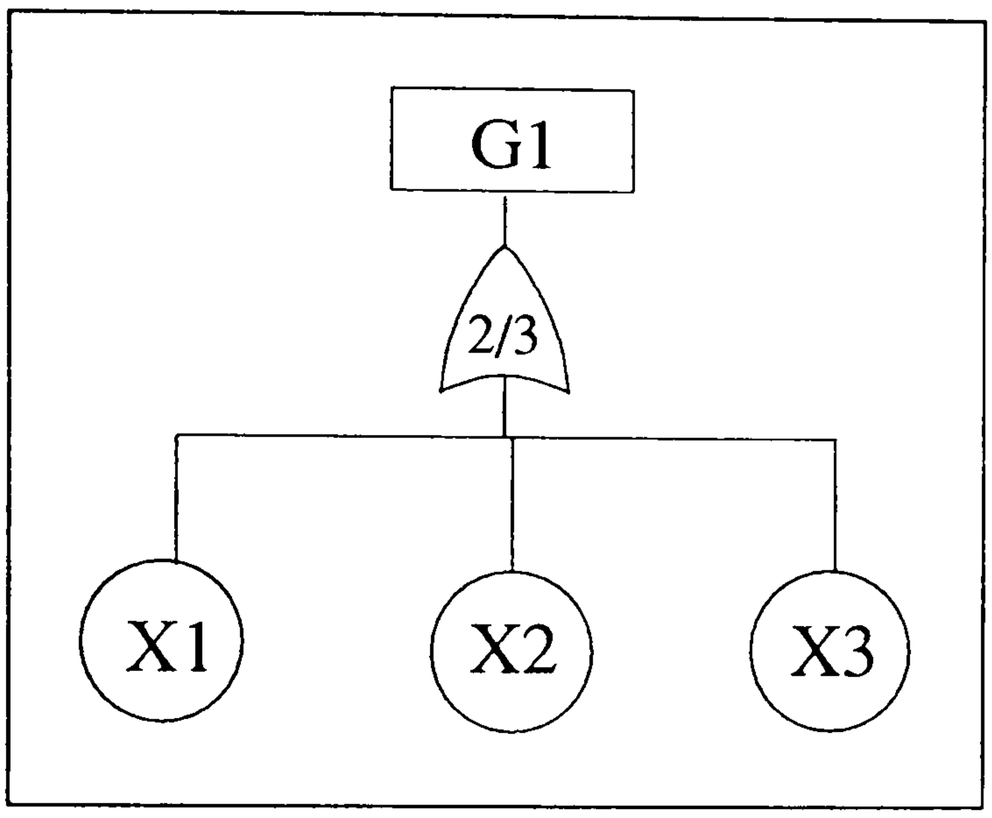


Figure 4.7 2/3 Vote Gate

And the corresponding BDD is given in figure 4.8.

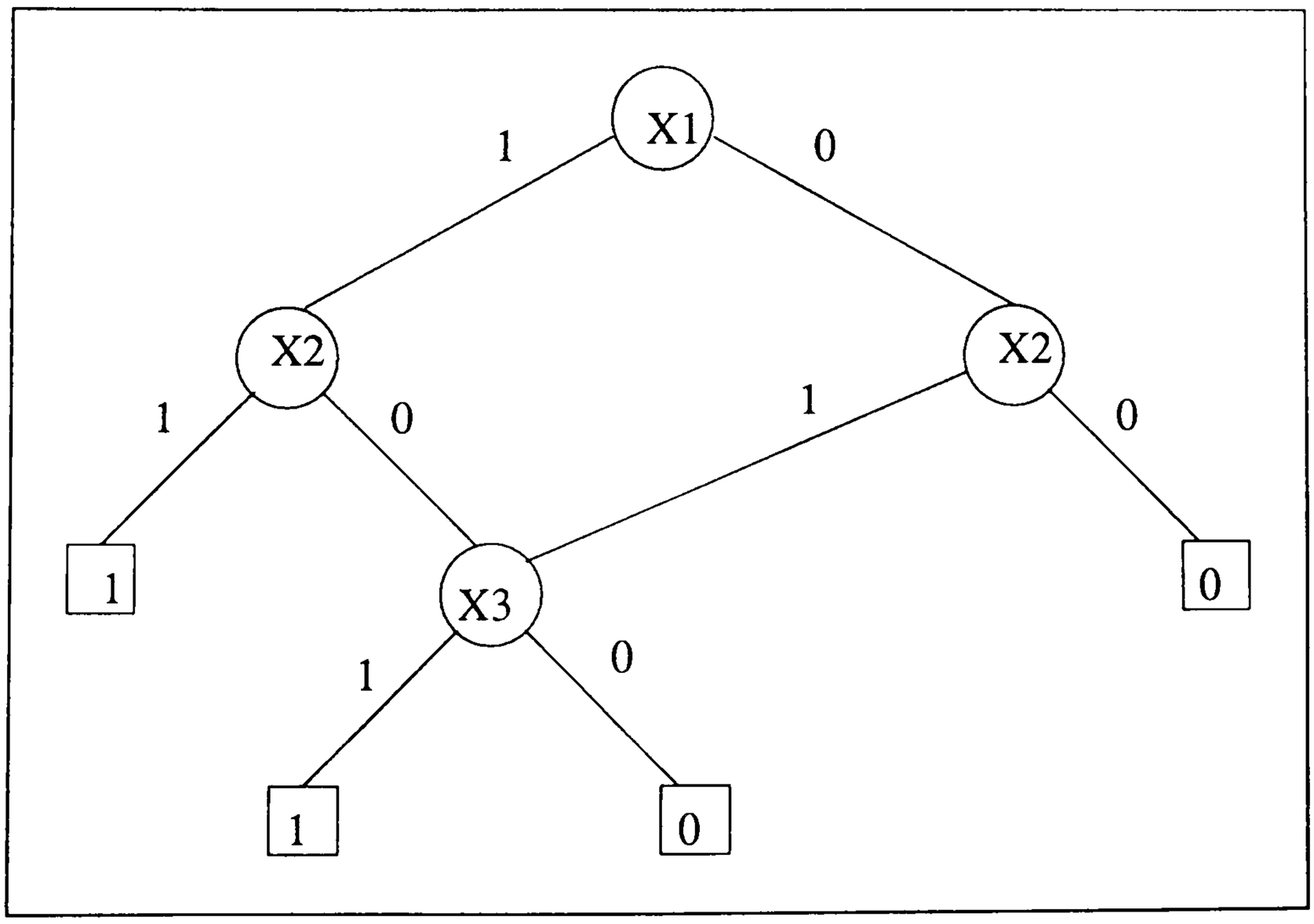


Figure 4.8 BDD for 2/3 Vote Gate

Therefore the paths in the BDD in figure 4.8 for the 2/3 vote gate are:

(1) X1.X2
(2) X1.X3
(3) X2.X3

which correspond to the minimal cut sets of the vote gate.

The most interesting property of BDD's is that, given a Boolean function f (and a total order < on variables), there exists one and only one BDD associated with f, this is proved by Bryant (34). The representation is therefore canonical.

4.6 Minimising Procedure

A problem occurs with some fault trees which is that the BDD produces cut sets which are not minimal. In such a situation there is little advantage to be gained in converting to a BDD structure unless some minimal form of the BDD can be derived which encodes only the minimal cut sets. To encode only the minimal cut sets the BDD needs to undergo a minimising procedure. The BDD shown in figure 4.3b is constructed using the structure function for the fault tree given in figure 4.2, this BDD is non-minimal because of the redundant cut set {X1, X3, X4}. The BDD for this fault tree shall be constructed again, using the **ite** procedure, and then used to illustrate the minimisation algorithm. The BDD resulting from the **ite** development will be the same as that produced using a structure function. However it is needed in its **ite** format to demonstrate the minimisation process.

A top-down ordering of basic events would give:

$$X1<X2<X3<X4 \tag{4.4}$$

Gates G2 and G1 can be resolved in a bottom-up procedure since they have only basic events as inputs.

$$G2=\textbf{ite}(X3, 1, 0).\textbf{ite}(X4, 1, 0)$$
$$=\textbf{ite}(X3, 1.\textbf{ite}(X4, 1, 0), 0.\textbf{ite}(X4, 1, 0))$$
$$=\textbf{ite}(X3, \textbf{ite}(X4, 1, 0), 0)$$

$$G1=\textbf{ite}(X1, 1, 0).\textbf{ite}(X2, 1, 0).\textbf{ite}(X3, 1, 0)$$

=**ite**(X1, 1.**ite**(X2, 1, 0), 0.**ite**(X2, 1, 0)).**ite**(X3, 1, 0)
=**ite**(X1, **ite**(X2, 1, 0), 0).**ite**(X3, 1, 0)
=**ite**(X1, **ite**(X2, 1, 0).**ite**(X3, 1, 0), 0.**ite**(X3, 1, 0))
=**ite**(X1, **ite**(X2, 1.**ite**(X3, 1, 0), 0.**ite**(X3, 1, 0)), 0)
=**ite**(X1, **ite**(X2, **ite**(X3, 1, 0), 0), 0)

With both inputs to the top event gate defined Top can now be evaluated.

Top=G1+G2
=**ite**(X1, **ite**(X2, **ite**(X3, 1, 0), 0), 0)+**ite**(X3, **ite**(X4, 1, 0), 0)
=**ite**(X1, **ite**(X2, **ite**(X3, 1, 0), 0)+**ite**(X3, **ite**(X4, 1, 0), 0),
0+**ite**(X3, **ite**(X4, 1, 0), 0))
=**ite**(X1, **ite**(X2, **ite**(X3, 1, 0)+**ite**(X3, **ite**(X4, 1, 0), 0),
0+**ite**(X3, **ite**(X4, 1, 0), 0)), **ite**(X3, **ite**(X4, 1, 0), 0)
=**ite**(X1, **ite**(X2, **ite**(X3, 1+**ite**(X4, 1, 0), 0+0), **ite**(X3, **ite**(X4, 1,
0), 0)), **ite**(X3, **ite**(X4, 1, 0), 0)
Top=**ite**(X1, **ite**(X2, **ite**(X3, 1, 0), **ite**(X3, **ite**(X4, 1, 0), 0)), **ite**(X3,
ite(X4, 1, 0), 0))

This top event **ite** structure represents the BDD shown in figure 4.9, which is the same as figure 4.3b. Notice that the node or vertex labelled F4, is shared by the right branch of node F2 and also the right branch of node F1. This sub-node sharing reduces memory requirements when implementing this analysis procedure on a computer. It also improves efficiency by eliminating the need to evaluate **ite** structures that have been previously calculated.

The cut sets obtained from this BDD are:

(1) X1.X2.X3
(2) X1.X3.X4
(3) X3.X4

Boolean Reduction Laws, as described in Chapter 1, could then be applied to these cut sets to produce the minimal cut sets, however this process increases computation time and memory requirements which destroys the aim of the BDD technique.

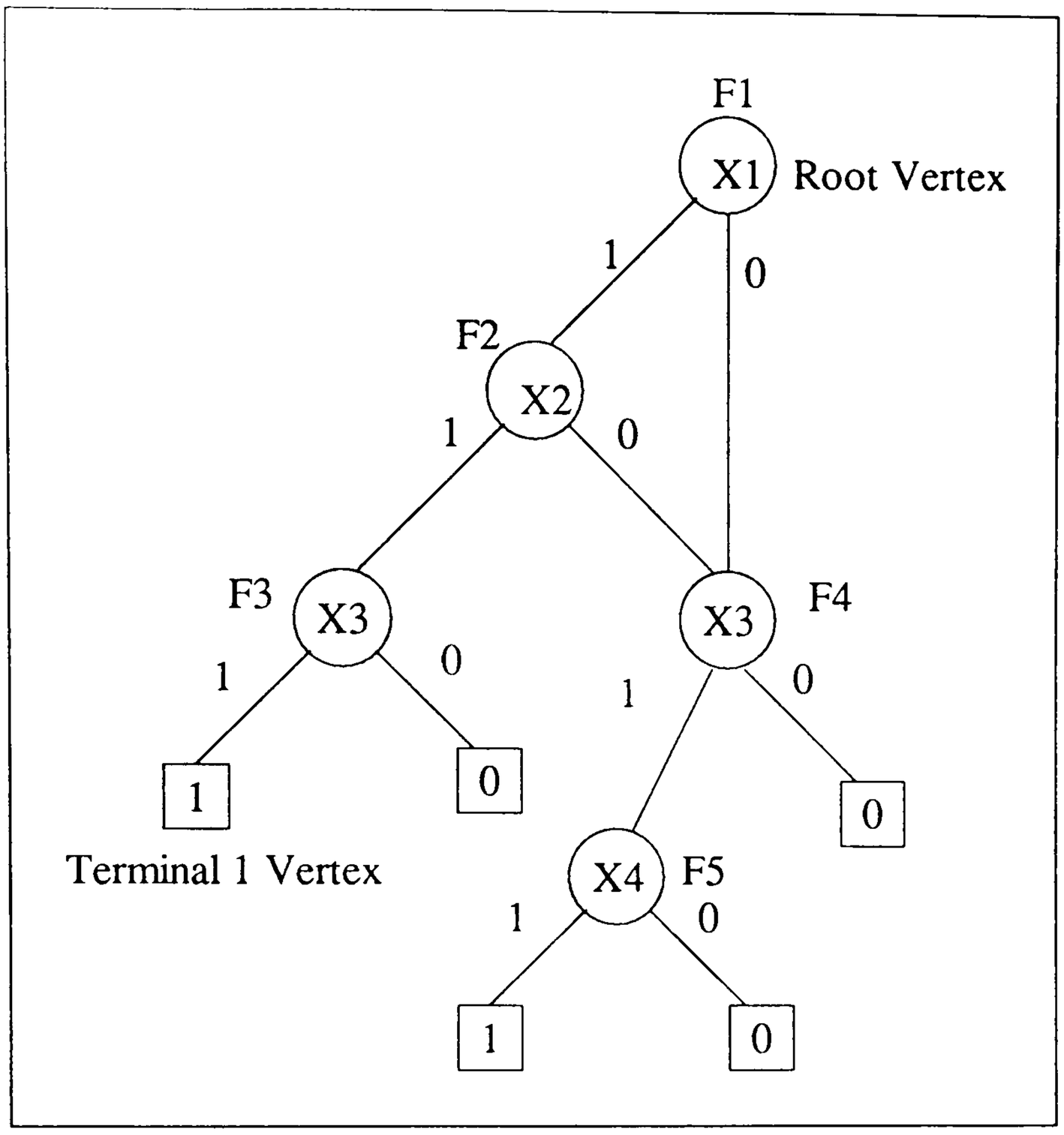


Figure 4.9 BDD for Fault Tree Shown in Figure 4.2

From the unminimised BDD obtained directly from the application of the **ite** operation, the minimisation algorithm of Rauzy (35) creates a new BDD that defines exactly the **minimal** cut sets of the fault tree. For example, consider a general node in a BDD. If the output of this node represents the function F where F=**ite**(x, G, H), let δ be a minimal solution of G which is **not** a minimal solution of H, then clearly the intersection of δ and x will be a minimal solution of F. Also, the set of all the minimal solutions of F, $sol_{\min}(F)$, will include the minimal solutions of H so:

$$sol_{\min}(F)=\{\sigma\} \tag{4.5}$$

where:

$$\sigma=[\{\delta\}\cap x]\cup[sol_{\min}(H)] \tag{4.6}$$

Rauzy has defined a 'without' operator, **without**($G_{\min}$, H) which removes from $G_{\min}$ all the paths included in a path of H. To demonstrate this algorithm it is applied to the BDD shown in figure 4.9. Consider the nodes in a top-down order. For the root vertex node F1, tracing the path on the 0 branch leads to node F4, which corresponds to H. This F4 node is also included in a path from the 1 branch of F1 which passes

through the 0 branch of F2. To establish the minimal solutions of F1 we need to formulate G_{min}, i.e. the minimal solutions of F2. In this case the solutions of F2 are minimal, therefore we remove from F2 all the paths that are included in a path of F4. This is performed by removing F4 from the 0 branch of F2 and replacing it by a terminal 0 vertex (refer to figure 4.10). The application of this minimising procedure to all other nodes in the BDD produces no other alterations. The BDD shown in figure 4.10 therefore is in its minimised form. Rauzy failed to recognise in his paper that to increase the efficiency when applying the minimisation procedure the following two results can be applied:

(1) **without**(F, F)={ }
(2) **ite**(x, F, F)=F

Details of the coded version of this algorithm with the slight modification suggested are given in Chapter 5.

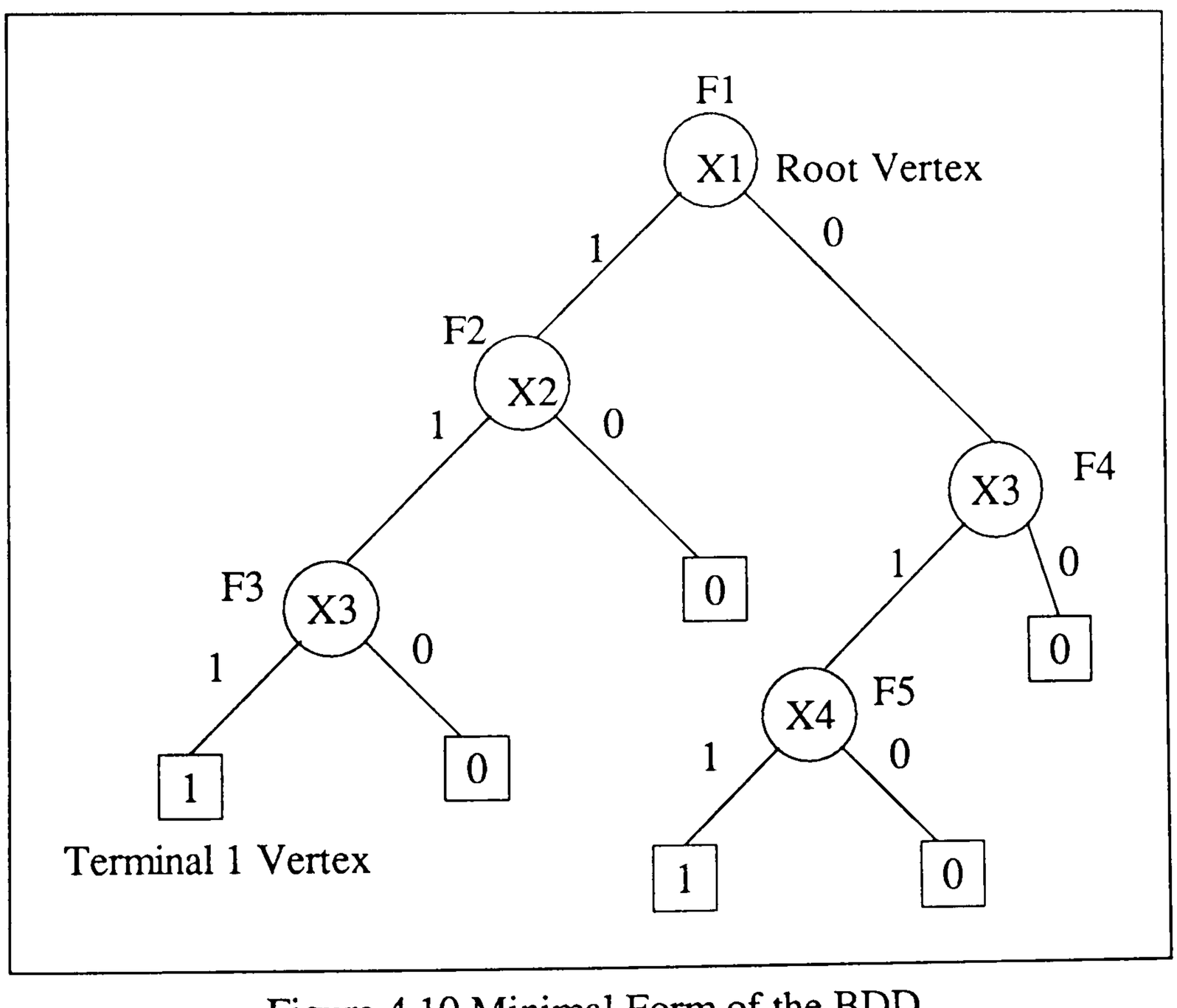


Figure 4.10 Minimal Form of the BDD

Tracing the paths through the minimised BDD we obtain the minimal cut sets:

(1) X1.X2.X3
(2) X3.X4

4.7 Influence of Ordering Schemes for Basic Events

The **ite** structure can only be obtained once the basic event inputs have been given an ordering. Usually a 'top-down' ordering procedure is employed where the basic events occurring higher up in the tree structure are 'less than', i.e. considered prior to those occurring lower down the tree.

Ordering of basic events in the fault tree is very important as it determines the size of the resulting BDD (refs. 33-35) and hence the number of cut sets.

To illustrate the importance of component ordering consider the fault tree in figure 4.11. The BDD for this fault tree with the conventional ordering X1<X2<X3<X4 is given in figure 4.12 and as a comparison the reverse ordering X4<X3<X2<X1, for the same fault tree, gives the BDD in figure 4.13.

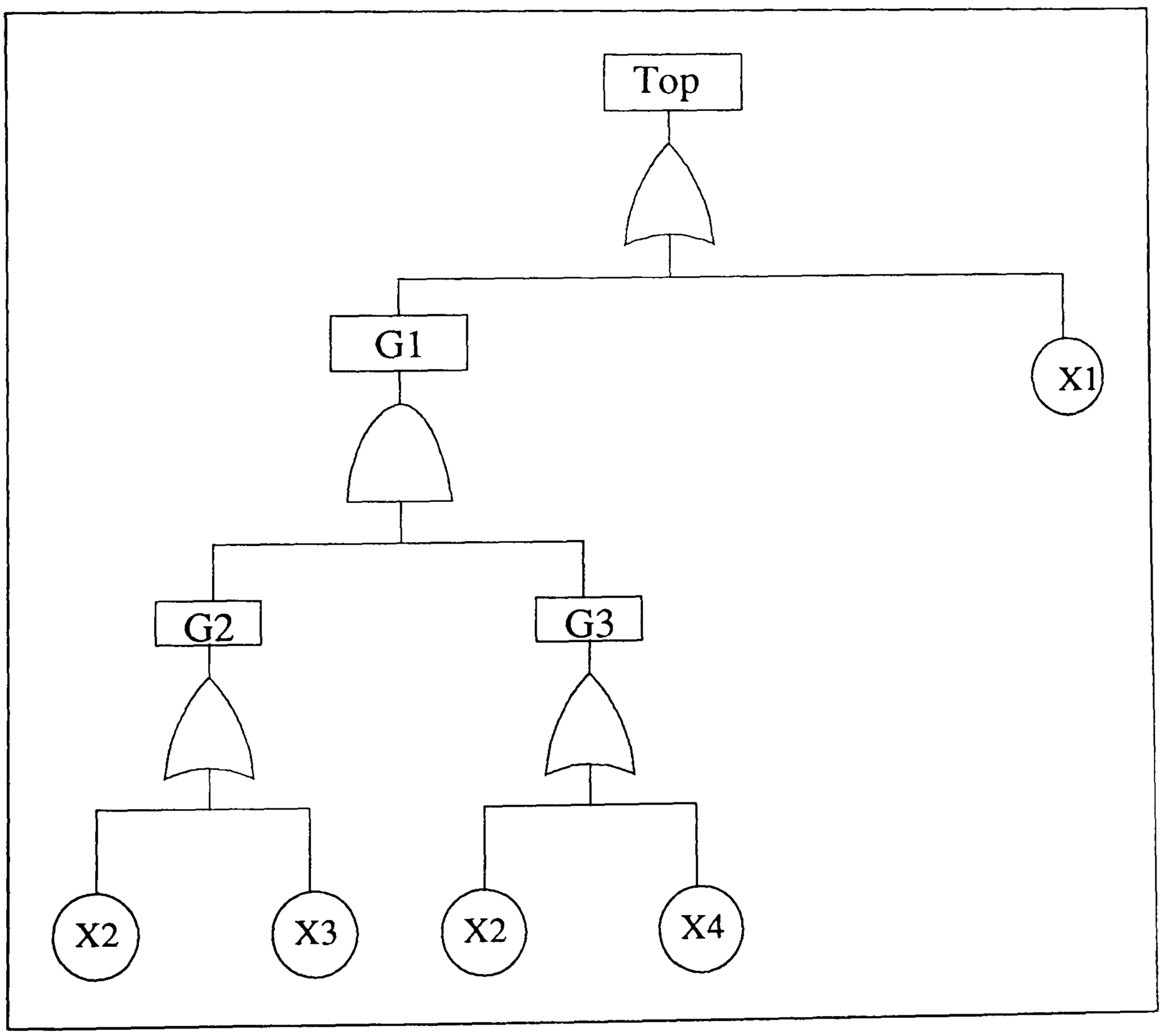


Figure 4.11 Example Fault Tree with Repeated Event X2

Clearly the first ordering creates a much smaller BDD, it has 4 nodes compared to the 7 nodes of figure 4.13 (here node X2 is shared by both X3 nodes). Figure 4.13 gives 3 non-minimal cut sets whereas figure 4.12 creates only minimal cut sets.

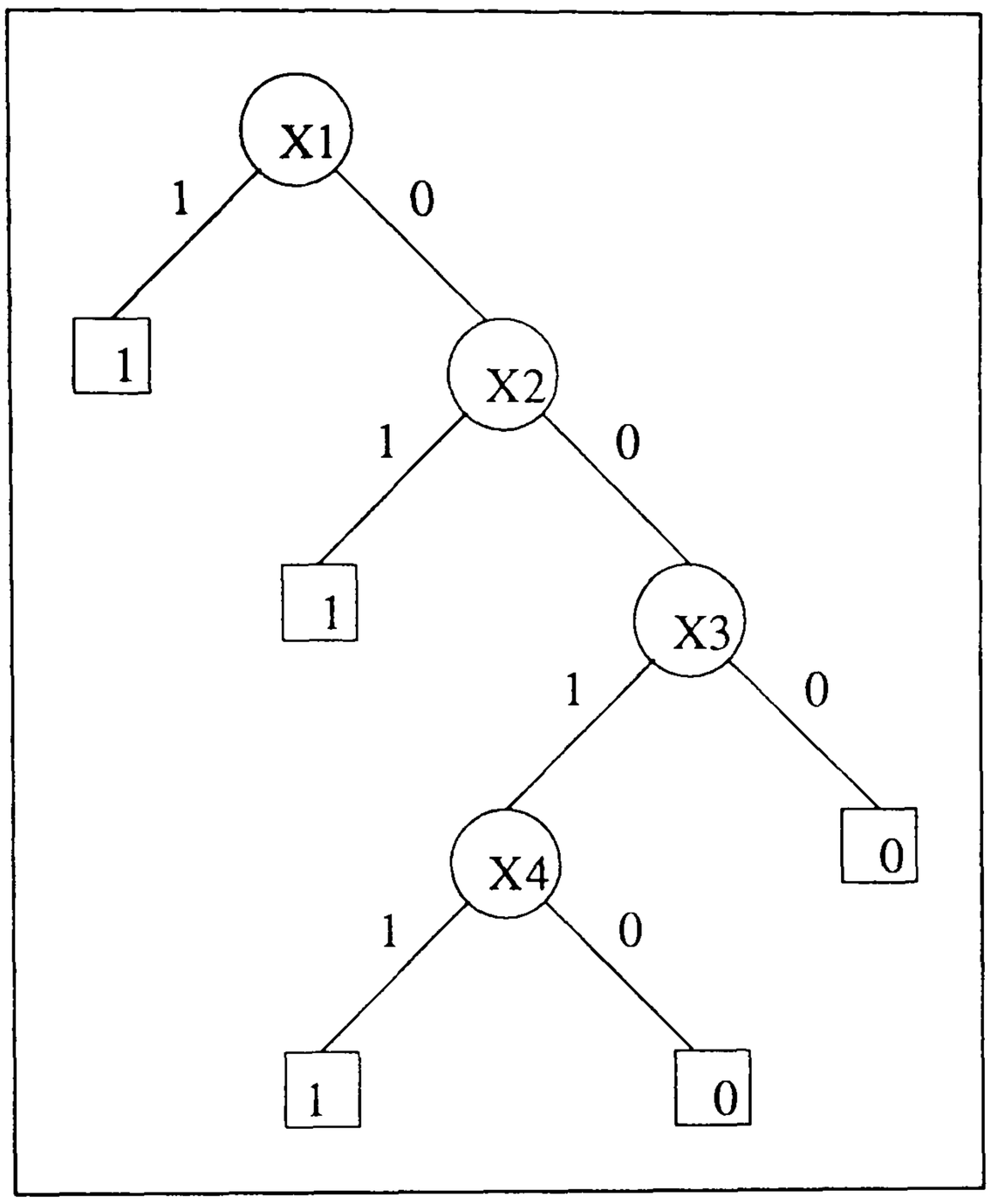


Figure 4.12 BDD for Ordering X1<X2<X3<X4

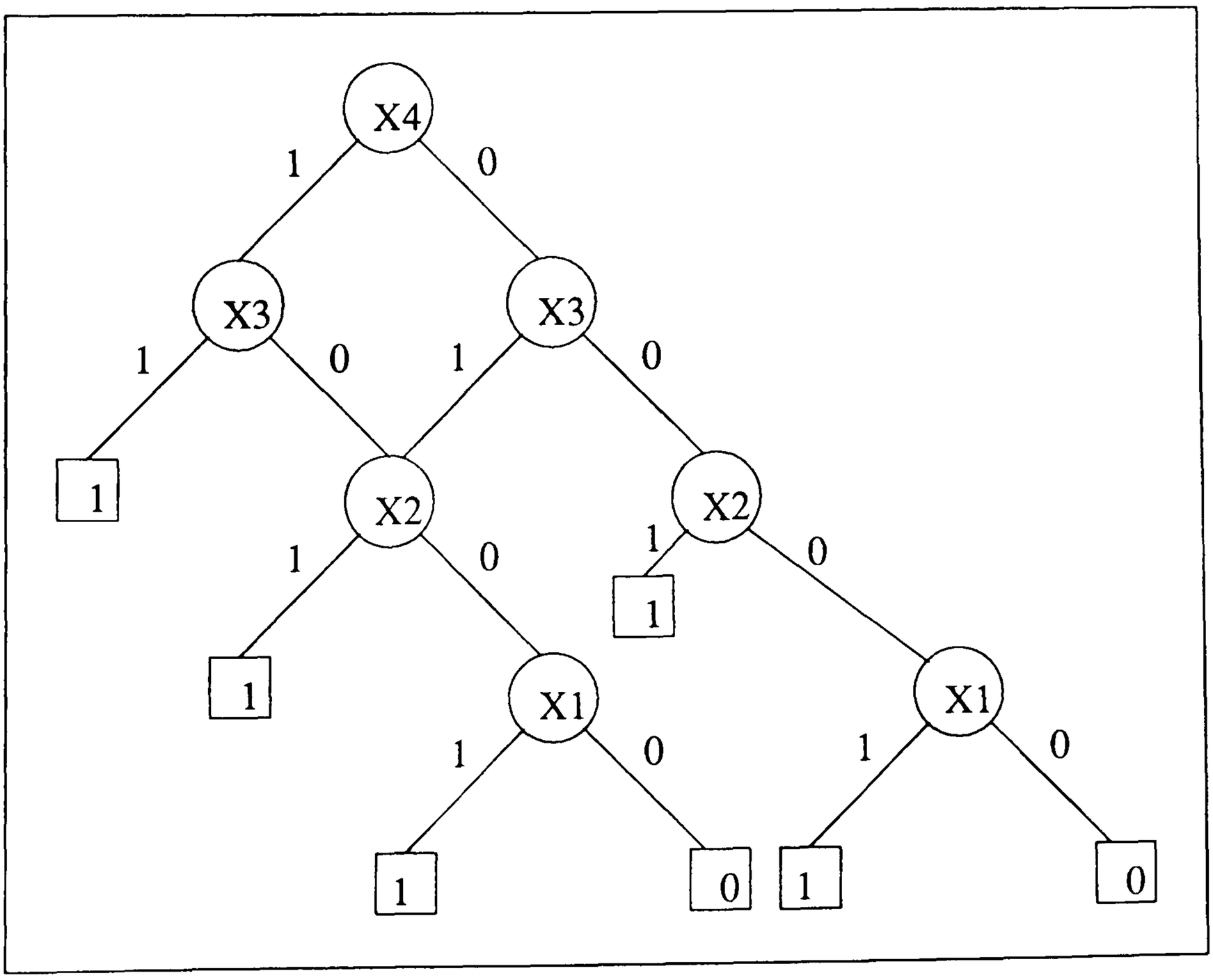


Figure 4.13 BDD for Ordering X4<X3<X2<X1

Although the top down ordering produces a minimal BDD for this example (in figure 4.11) which yields only minimal cut sets, this is not always the case. Obviously if a redundant BDD is obtained then it is necessary to perform some kind of minimising procedure to obtain only the minimal cut sets. This results in greater computation time. Therefore it would be more beneficial to produce a minimal or at least a 'near' minimal BDD from the fault tree thus reducing the computation time spent minimising the resulting cut sets.

Friedman and Supowit (37) researched the size of the BDD for a given Boolean function and identified that the size of the BDD for a given function is extremely sensitive to the choice of an ordering on the variables. They present an algorithm for finding the optimal ordering with time complexity $O(n^2 3^n)$ where n is the given number of variables in the BDD. Unfortunately this algorithm could not be applied to a fault tree directly, as initially the Boolean function for the fault tree needs to be determined. This in itself may pose a formidable problem since generally the Boolean function is established in terms of the minimal cut sets. Butler et al. (38) also addressed the problem of finding a 'good' variable ordering for large electronic circuits. They looked at a depth-first and a breadth-first traversal of the circuit to order the variables. The procedure is to order outputs from the logic node with the most inputs reaching it to the one with the least and select outputs from which to traverse in descending order from this list. The findings showed that neither traversal is dependably superior, although the depth-first based heuristics required less total BDD nodes for more circuits than breadth-first. However breadth-first is advisable as an alternative ordering heuristic.

Chevalier et al. (39) stated that there does not seem to exist a heuristic capable of ordering the variables systematically to lead to a minimal BDD. Of all the ordering schemes investigated (these ordering schemes are not illustrated in reference (39)) they considered that a 'depth-left-first' ordering of the basic events in the fault tree was the best.

4.8 Modularising

Improvements in terms of computational efficiency can be made for the more complex fault trees by modularising the fault tree before the analysis takes place. Increased efficiency is achieved by modularisation of a fault tree whatever the analysis technique

which is employed. Khoda et al. (40) define a module of a fault tree as having no inputs which appear elsewhere in the tree and no outputs to the rest of the tree except from its output event. For example consider the fault tree in figure 4.14. Modules which have the required properties are gates G2, G3 and Top. Here gates G2 and G3 do not have inputs which appear as inputs to any other gates in the fault tree and Top is called module Top as it too fulfils the definition of a module. G1 is not a module because it has the repeated gate G3 as an input.

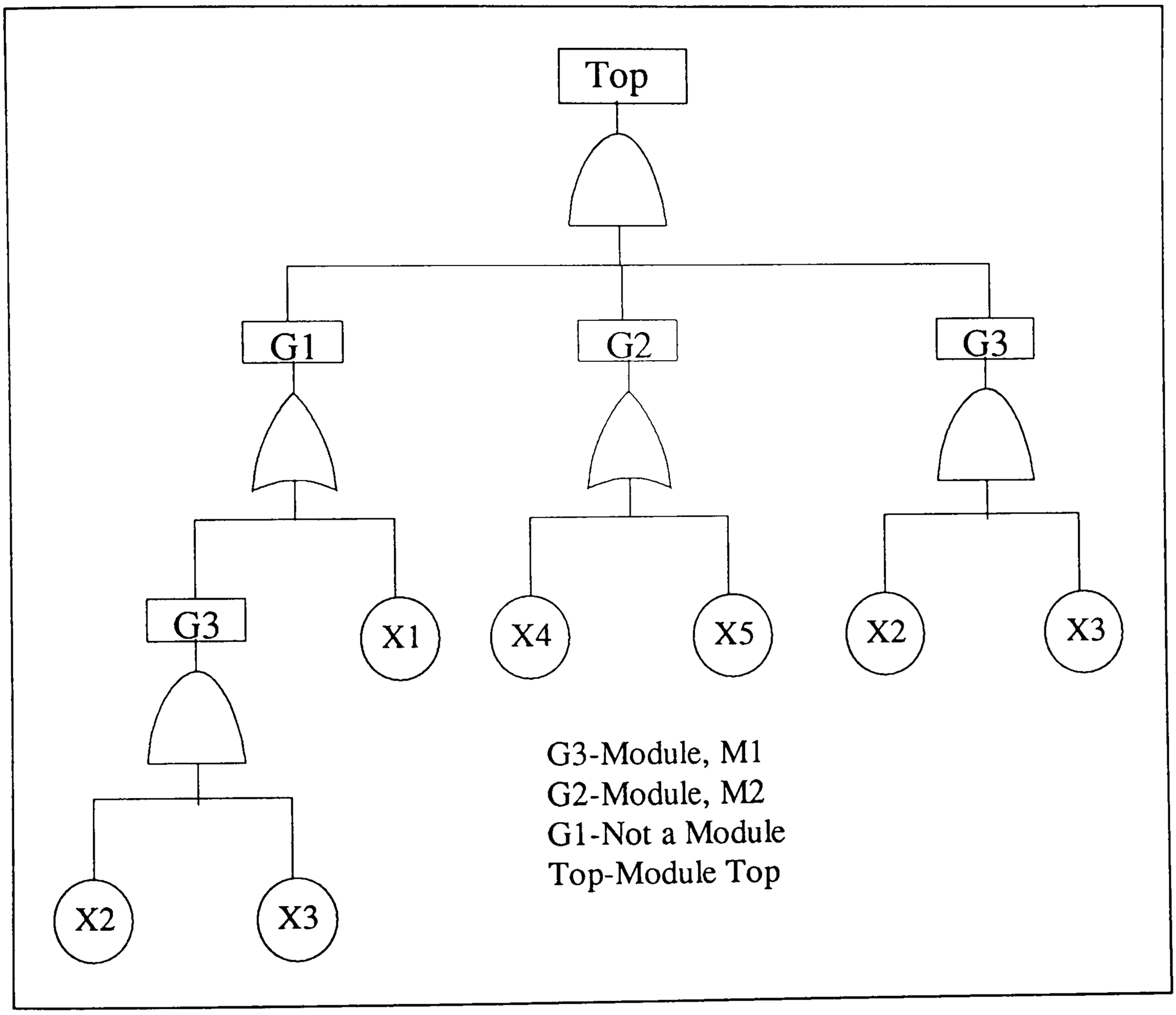


Figure 4.14 A Fault Tree which can be Modularised

The modularised fault tree is shown in figure 4.15. By then using the BDD method to analyse this tree in terms of the modules, the modularised BDD in figure 4.16 is obtained (using the top-down, left-right ordering M2<M1<X1).

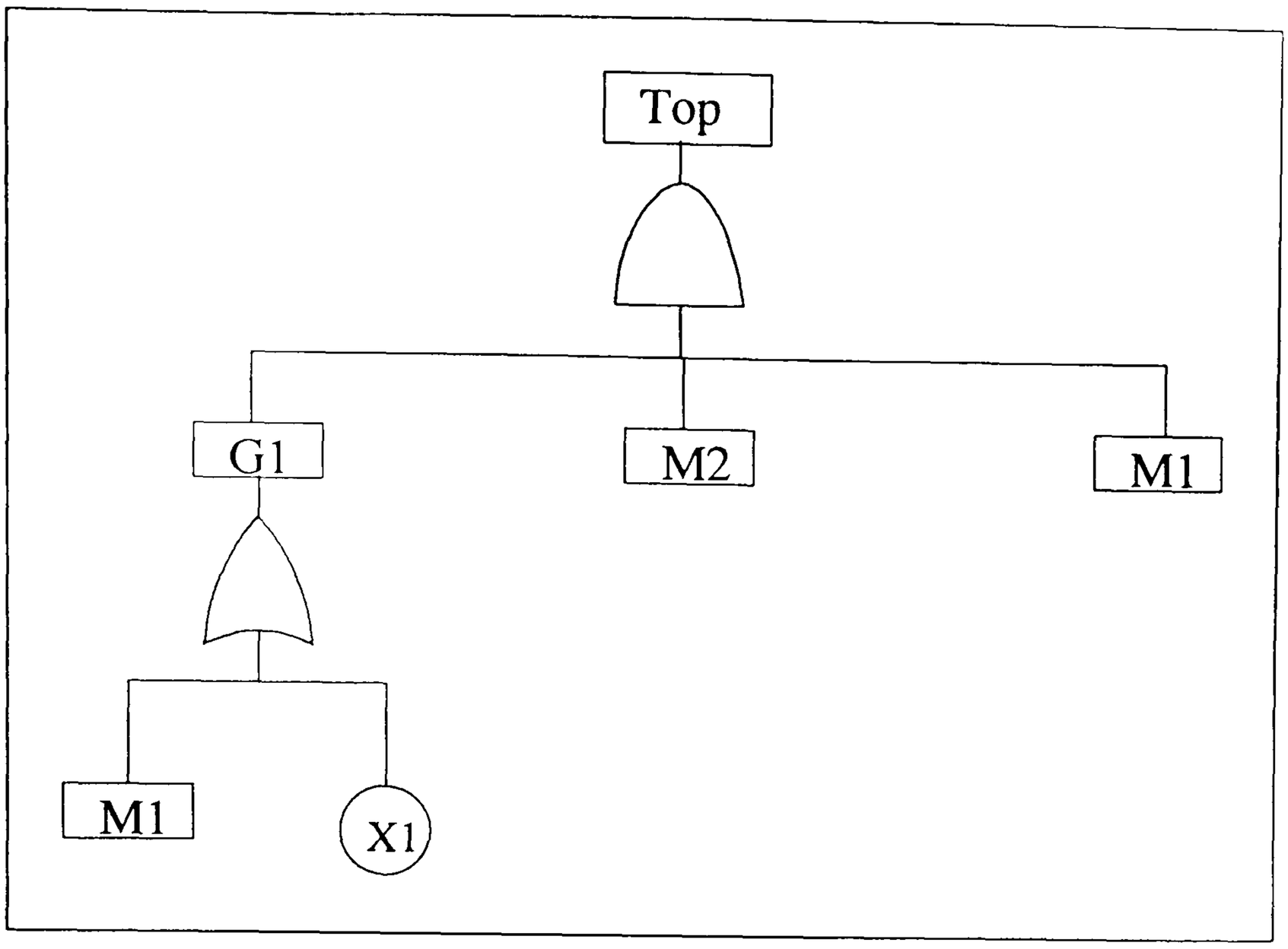


Figure 4.15 Modularised Fault Tree

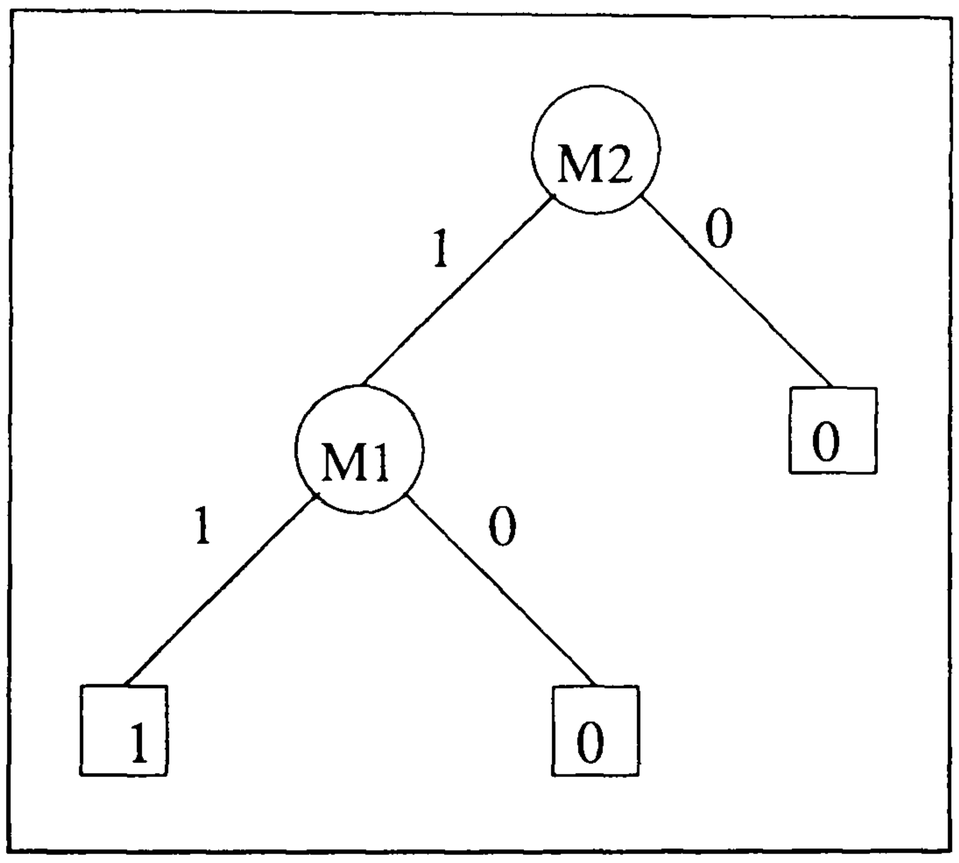


Figure 4.16 Modularised BDD

The modularised BDD results in the path:

(1) M2.M1

Notice that the basic event X1 has been omitted by this process.

The resulting BDD for the **ite** computation M2.M1 is given in Figure 4.17, from which the following minimal cut sets are obtained (using the top-down, left-right ordering X4<X5<X2<X3):

(1) X4.X2.X3
(2) X5.X2.X3

Without modularisation this fault tree would have resulted in a redundant BDD with 13 nodes, as opposed to the 4 nodes of figure 4.17. Therefore modularising can provide an efficient means of analysing the whole fault tree.

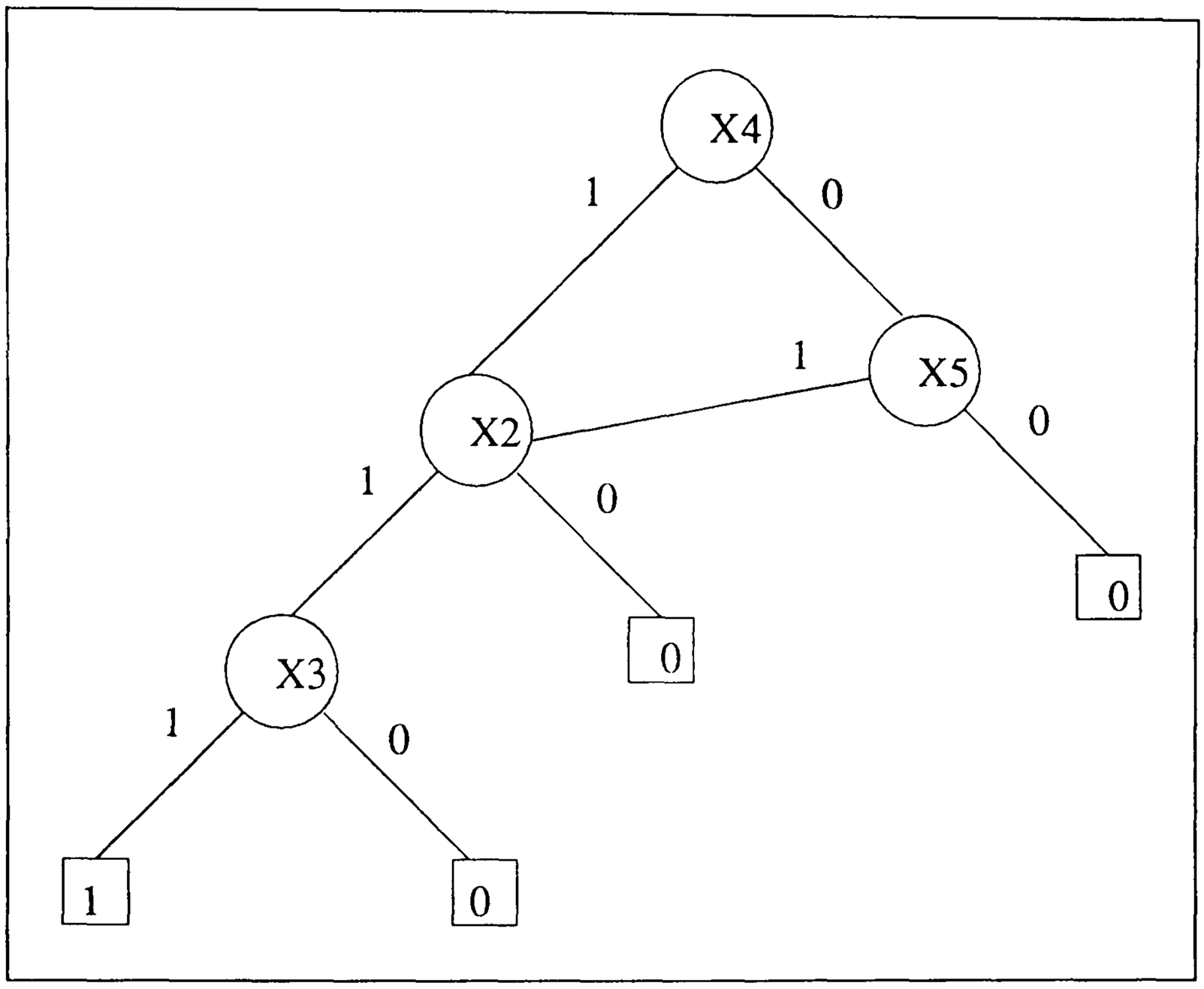


Figure 4.17 BDD for Computation M2.M1

4.9 Binary Decision Diagrams and Non-Coherent Fault Trees

The 'Group Aralia' (41), (a group composed of researchers from two laboratories of Bordeaux University and a research team called ISdF (French Institute for System Dependability) whose members are from several major French companies) have undertaken extensive research into the construction of a BDD which encodes the prime implicants of a non-coherent fault tree. The algorithm they use is based on the one proposed by Madre and Coudert (42) in 1992. In the methodology proposed by Madre and Coudert fault trees are essentially considered as Boolean formulae.

For example consider the non-coherent fault tree illustrated in figure 4.18. This fault tree has the prime implicants:

(1) {a, b}
(2) {$\overline{a}$, c}
(3) {b, c}

These prime implicants may be obtained using the techniques described in Chapter 2, section 2.3. As discussed in this section all prime implicants are often not obtained. The reason being that classical approaches fail to handle large size non-decomposable fault trees (i.e. fault trees which cannot be modularised) because they cannot avoid combinatorial explosions in both execution time and memory requirements. In these circumstances only the most important prime implicants are evaluated. The application of the BDD technique to non-coherent fault trees overcomes these problems.

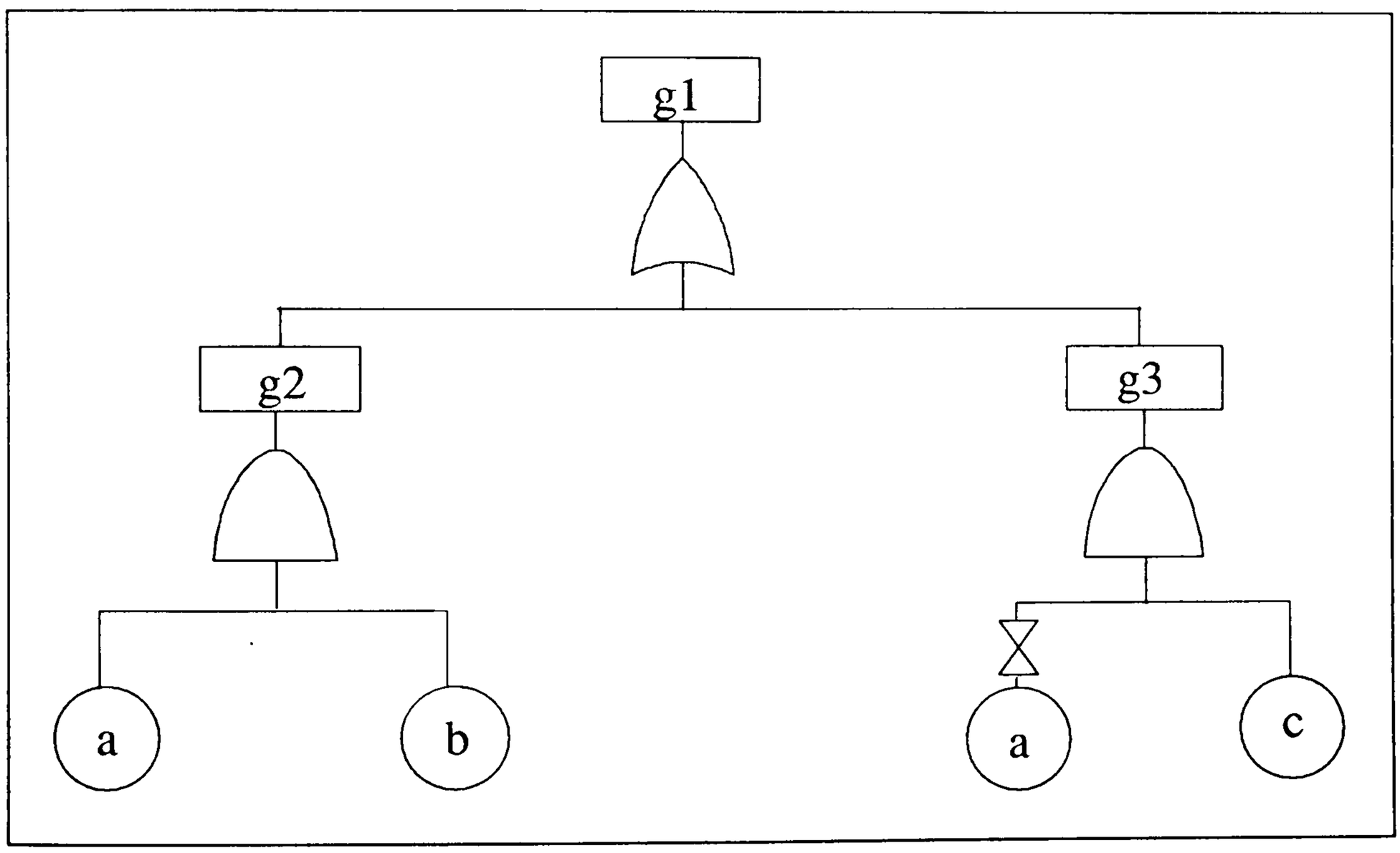


Figure 4.18 Example Non-Coherent Fault Tree

It has been shown in section 4.3 that by applying Shannon's decomposition recursively, one can rewrite any Boolean function into an equivalent one that has an **I**f-**T**hen-**E**lse (**ite**) structure. For example:

$$f(x1,\ldots\ldots,xn) \equiv (x1 \cap f(1,x2,\ldots\ldots xn)) \cup (\overline{x1} \cap f(0,x2,\ldots\ldots xn))$$
$$\equiv ite(x1, f(1,x2,\ldots\ldots,xn), f(0,x2,\ldots\ldots,xn))$$

For the fault tree presented in figure 4.18:

$$g1 = ((a \cap b) \cup (\overline{a} \cap c))$$
$$= ite(a,b,c)$$

This representation of a logic function is employed in the 'Decomposition Theorem' which forms the basis of the algorithm by Group Aralia to compute the prime implicants.

Decomposition Theorem

Let $f(x1,.....,xn)$ be a Boolean function. Then, the set of prime implicants of $f(x1,.....,xn)$ denoted $\Pr ime(f(x1,.....,xn))$ is the union of the three following sets.

(1) The set $\Pr ime(f(1,.....,xn) \cap f(0,.....,xn))$

(2) The set $\{x1\} \otimes \sigma$ where $\otimes$ is such that x1 is intersected with each member of σ, and σ is a product in $\Pr ime(f(1,.....,xn))$ that does not belong to $\Pr ime(f(1,.....,xn) \cap f(0,.....,xn))$, i.e. x1 and prime implicants of $f(1,.....,xn)$ that are not already contained in $f(x1,x2,.....,xn)$.

(3) The set of products $\{\overline{x1}\} \otimes \sigma$ where σ is a product in $\Pr ime(f(0,.....,xn))$ that does not belong to $\Pr ime(f(1,.....,xn) \cap f(0,.....,xn))$, i.e. $\overline{x1}$ and prime implicants of $f(0,.....,xn)$ that are not already contained in $f(x1,x2,.....,xn)$.

The decomposition theorem gives an inductive principle to compute prime implicants. For the given example:

$$g1(a,b,c) = (a \cap b) \cup (\overline{a} \cap c) = ite(a,ite(b,1,0),ite(c,1,0))$$
$$g1(1,b,c) = ite(b,1,0) = b$$
$$g1(0,b,c) = ite(c,1,0) = c$$
$$g1(1,b,c) \cap g1(0,b,c) = ite(b,1,0) \cap ite(c,1,0)$$
$$= ite(b,ite(c,1,0),0)$$
$$= b \cap c$$

It is clear that $\Pr ime(b) = \{\{b\}\}$, $\Pr ime(c) = \{\{c\}\}$ and $\Pr ime(b \cap c) = \{\{b,c\}\}$. Thus $\Pr ime(ite(a,ite(b,1,0),ite(c,1,0))) = U \cup V \cup W$ where

U - is the set of products $\{a\} \otimes \sigma$ where σ is in $\{\{b\}\}$ and not in $\{\{b,c\}\}$. Thus, $U = \{\{a,b\}\}$

V - is the set of products $\{\bar{a}\} \otimes \sigma$ where σ is in $\{\{c\}\}$ and not in $\{\{b,c\}\}$. Thus, $V = \{\{\bar{a},c\}\}$

W - is $\mathrm{Prime}(b \cap c) = \{\{b,c\}\}$

Finally, $\mathrm{Prime}(g1) = \{\{a,b\},\{\bar{a},c\},\{b,c\}\}$.

The remainder of the paper by Group Aralia proceeds to describe the computation of the set of prime implicants of a given formula by means of a BDD. The resulting BDD encodes implicitly this set without representing each of its elements. The principle of the algorithm is to use the decomposition theorem in order to compute inductively the Boolean function encoding the set of prime implicants of the formula under study. The primary concern of this thesis is the application of the BDD method to coherent fault trees therefore a full description of the prime implicant algorithm will be omitted here. However it is important to stress that the BDD method is applicable to both coherent and non-coherent fault trees.

4.10 Summary

(1) Initial work indicates the great potential of the BDD method to overcome some of the disadvantages of conventional fault tree analysis to produce the minimal cut sets.

(2) For it to fulfil its potential the BDD method must be capable of evaluating the full range of top event parameters and component importance measures available in the alternative conventional fault tree analysis method. Little work has been carried out on the use of the BDD for fault tree quantification.

(3) It appears the efficiency of the BDD method is totally dependent upon the ordering of the basic events. Work is required to try to give guidelines on how the ordering should be performed for efficiency.

(4) The claims of other workers for improved efficiency needs to be investigated over the whole fault tree qualitative and quantitative analysis process. In order to carry out this task, benchmark test cases need to be identified and tested against conventional analysis techniques for both accuracy and computational effort required.

CHAPTER 5

IMPLEMENTATION OF THE BINARY DECISION DIAGRAM METHOD FOR MINIMAL CUT SET EVALUATION

5.1 Introduction

The computational method for constructing the BDD for a particular fault tree and producing the minimal cut sets has been implemented in a program called BADD (Binary Algorithm for Decision Diagrams). BADD is composed of several subroutines which shall be discussed in detail in the sections which follow.

This chapter deals with the step-by-step computation that is required to convert the fault tree to a BDD structure. In addition it describes the minimisation algorithm of this BDD structure if it is to encode the minimal cut sets of the original fault tree. The minimal cut sets are then evaluated by tracing the paths through the minimal BDD.

Further, the BDD technique is compared and contrasted to a conventional fault tree analysis technique (top-down approach) and several benchmark fault trees are evaluated.

The last section of this chapter discusses a new ordering scheme for the basic events in the fault tree to enable a more efficient construction of the BDD.

5.2 Computational Method for Binary Decision Diagram Analysis - BADD

The following sections describe in detail the computational method of the program called BADD.

5.2.1 Input for BADD

When BADD is executed it prompts the user for the name of the fault tree structure input file, ***.ats**, where * is a name of up to eight characters in length. This data file contains the logic equations which relate gates to their inputs and the gate logic type. The data file may be constructed by the user (the format is given in Appendix I) or

alternatively the graphics on the software package called FAULTREE+ (43) can be used. BADD has been developed in such a way so as to allow both pieces of software to be used in conjunction which each other. The fault tree can be drawn using the conventional symbols for the logic gates and the basic events in FAULTREE+. Once the fault tree has been drawn, FAULTREE+ automatically creates the ***.ats** file for this tree which can be read directly into the BADD code. To illustrate the nature of the data file refer to the fault tree called **fatram** taken from ref. (30) and reproduced here in figure 5.1. The data file for this tree is given in figure 5.2.

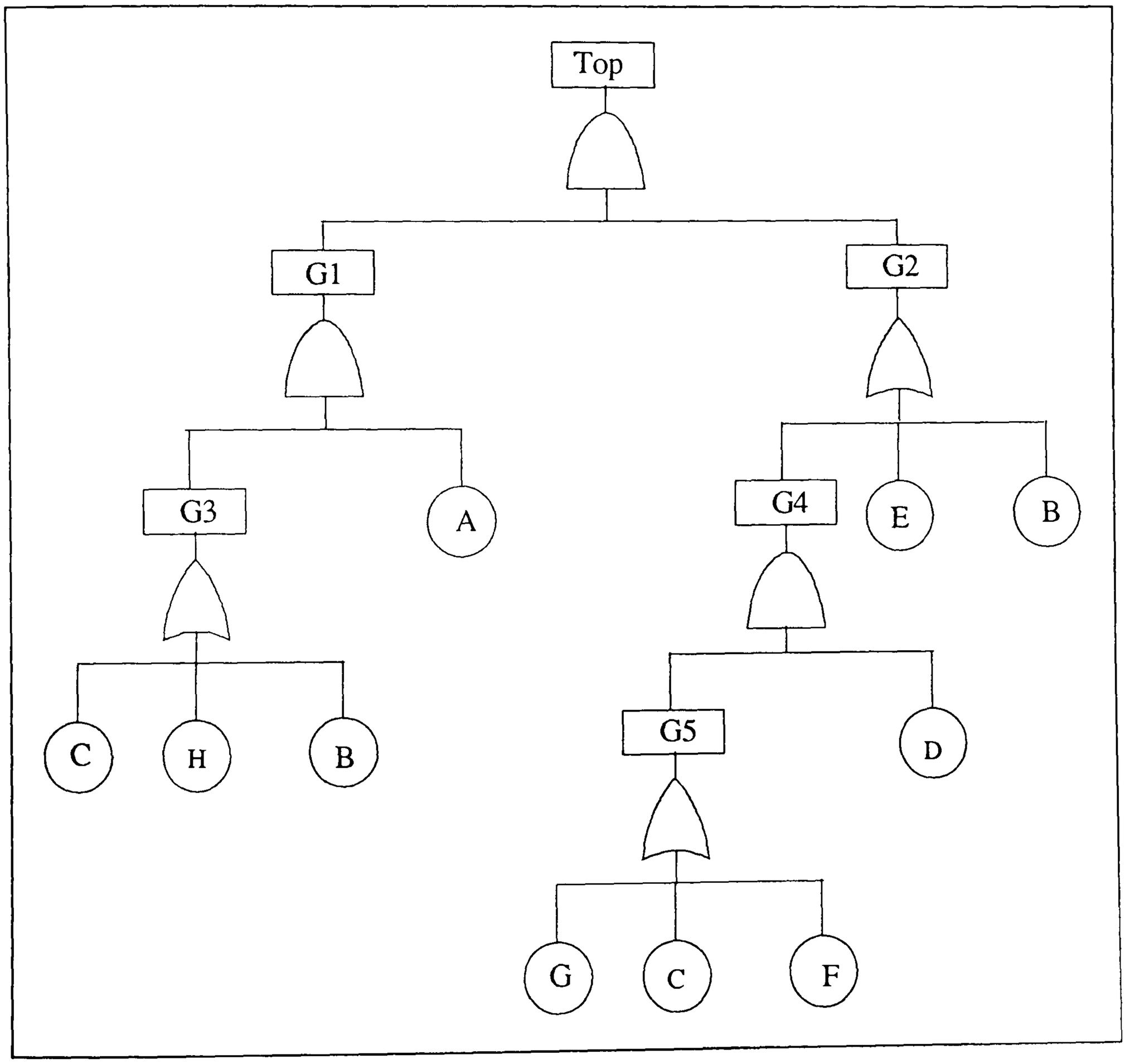


Figure 5.1 **fatram** Fault Tree

Top	AND	2	0	G1	G2	
G1	AND	1	1	G3	A	
G2	OR	1	2	G4	E	B
G3	OR	0	3	C	H	B
G4	AND	1	1	G5	D	
G5	OR	0	3	G	C	F

Figure 5.2 Data File Called **fatram.ats**

The contents of each column in figure 5.2 are described below:

column 1 - gate name
column 2 - gate type, i.e. AND, OR, VOTE
column 3 - number of gate inputs to gate in column 1
column 4 - number of basic event inputs to gate in column 1
column 5,....,13 - names of the gate inputs and basic event inputs (up to nine inputs allowed)

A subroutine called *Input* reads in the ***.ats** file and converts all the gate and event names to unique integers. Additionally it stores the information for later use. It also assigns the following integers to each gate type:

OR - 1
AND - 2
VOTE - 3

This provides an efficient means of storing the fault tree structure within the code.

5.2.2 Information about the Fault Tree

Next a subroutine called *Repevent* obtains some important information about the fault tree. It identifies which gate is the top event (as the top gate may not necessarily lie in row 1 column 1 of the ***.ats** file). It checks for circular logic, i.e. an event cannot provide a cause to itself. Additionally it checks that the fault tree is not disjoint (has more than one top event). It also assigns an integer to each event according to the rule that 1 represents the identified top gate, the integer 2 represents an intermediate event (gate) and lastly 0 represents a basic event. This integer is called the events '**ecode**'. Lastly the number of occurrences of each event in the fault tree is calculated, so the number of repeated events is known. The difficulty of calculating repeated events occurs when there are many repeated gates within the fault tree. In this case consideration needs to be given to the inputs of these repeated gates, as these inputs are now repeated structures themselves and so on.

The problem of repeated events is tackled in *Repevent* by a one dimensional array, whose initial elements are the first sons or gate inputs of the top event. Next, the

inputs to the first son are added to the end of the array and the same procedure is applied to the second son. A pointer moves downwards through the array and deals with each gate whose inputs need to be added to the end of the array. This process terminates when all the gate events have had their inputs added to the array.

To illustrate this method refer to the array development in figure 5.3 for **fatram**. The next gate whose inputs have to be added to the end of the array is indicated by *. For each event in this one dimensional array a counter is incremented each time it occurs, this will give the total number of occurrences of each event in the fault tree.

$$\begin{pmatrix} G1* \\ G2 \end{pmatrix} \rightarrow \begin{pmatrix} G1 \\ G2* \\ G3 \\ A \end{pmatrix} \rightarrow \begin{pmatrix} G1 \\ G2 \\ G3* \\ A \\ G4 \\ E \\ B \end{pmatrix} \rightarrow .....etc..... \rightarrow \begin{pmatrix} G1 \\ G2 \\ G3 \\ A \\ G4 \\ E \\ B \\ C \\ H \\ B \\ G5 \\ D \\ G \\ C \\ F \end{pmatrix}$$

Figure 5.3 Array Development for Repeated Events in **fatram**

From the final array in figure 5.3 it can be seen that basic events B and C occur twice and all other events occur once.

5.2.3 Ordering the Basic Events in the Fault Tree

A subroutine called *Ordering* deals with the ordering of the basic events in the fault tree. The ordering of the basic events is performed by a top-down, left-right manner previously discussed in Chapter 4, section 4.7. If other orderings are required for the basic events a program called ORDER which is described in Chapter 8 may be

employed. The top-down, left-right ordering of the basic events in the fault tree shown in figure 5.1 will give:

A<E<B<C<H<D<G<F

Note that if repeated events have been previously encountered and placed in the ordering then they are ignored when subsequently encountered. Each basic event is assigned a number for the purposes of ordering. This number is called the basic events '**index**', with the indexes starting at 1 (for the basic event to be considered first) and increasing to represent the position of the event within the ordering, therefore for the basic events in the ordering above:

A-1, E-2, B-3, C-4, H-5, D-6, G-7, F-8

Additionally this subroutine allocates each basic event Xi with **index** j the **ite** structure **ite**(j, -1, 0), i.e. the basic event can either fail or work (for computational purposes -1 indicates a terminal 1 vertex to differentiate from the **index** 1). The **ite** structure for each basic event is placed in an **ite** table at a position corresponding to the **index** of the basic event. Therefore basic event C will be stored in row 4 of the **ite** table with structure **ite**(4, -1, 0).

5.2.4 Re-configuring the Fault Tree

The computation of the **ite** structure for intermediate gate events involves the operation on <u>two</u> **ite** structures (this is illustrated later in the algorithm called *Compute*). Therefore for ease of computation the fault tree is converted to a 'binary' fault tree, i.e. each gate has only two inputs. The subroutine that accomplishes this operation is called *Binary*. *Binary* simply searches through the fault tree for the gates that have more than two inputs (i.e. 1,.....,n where n>2). When such a gate is found a new gate NGi (with the same logical operation as the original gate) is created with original gate inputs 2,.....,n provided as inputs to this new gate. The original gate is now defined as having input event 1 together with the newly defined gate NGi. This search is performed recursively until no more gates exist with more than two inputs. The only gates whose inputs remain unchanged when *Binary* is executed are the VOTE gates. When a vote gate is encountered its inputs are left untouched and the computation of its **ite** structure is dealt with by a subroutine called *Votegate* which is

described later. To illustrate this re-configuring of the fault tree refer to figure 5.4 which shows the 'binary' tree of **fatram**.

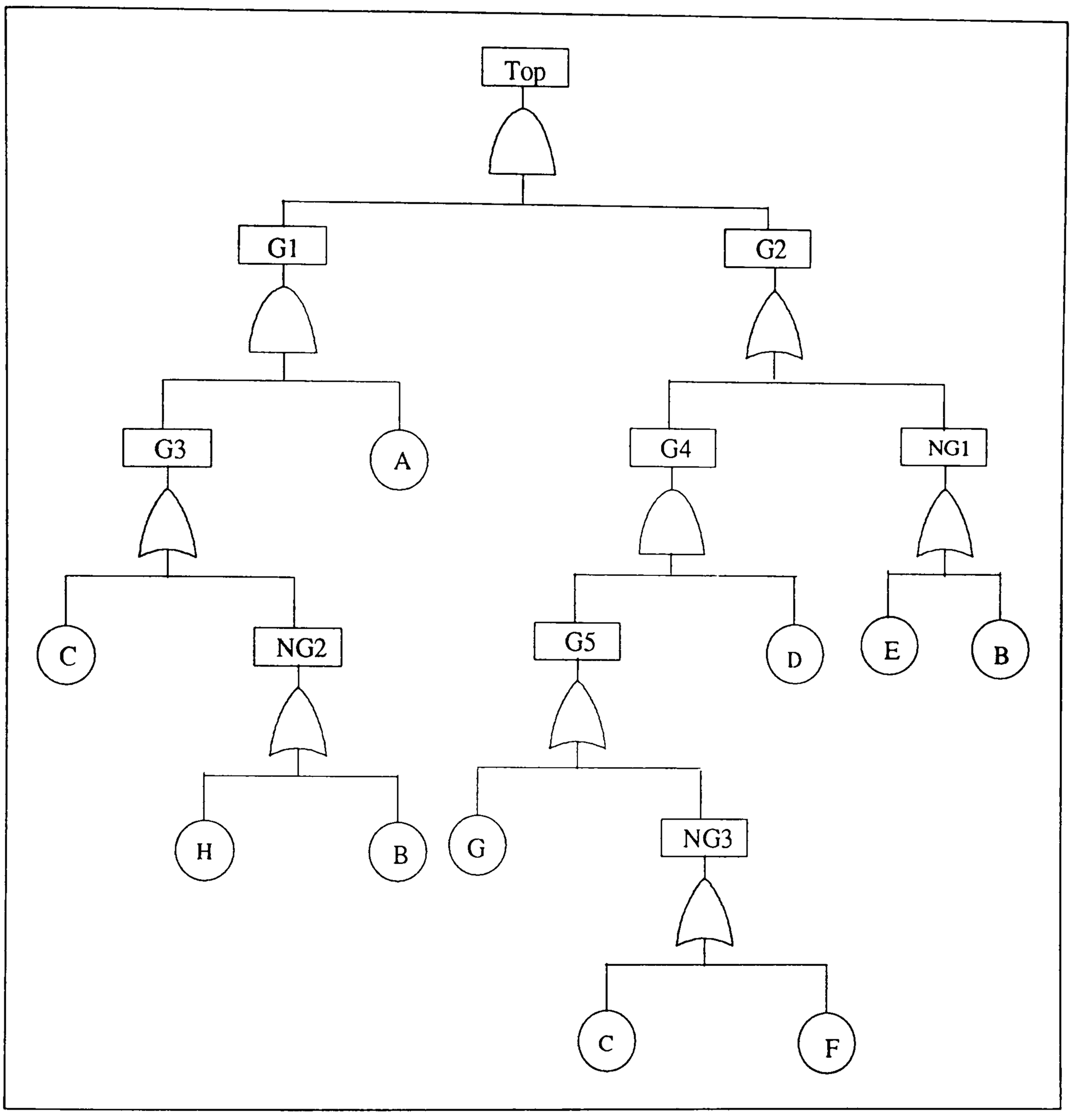


Figure 5.4 Re-configured 'Binary' **fatram** Tree

5.2.5 Computing the ite Structure of Each Gate

Essentially BADD employs the **ite** (**I**f-**T**hen-**E**lse) procedure for producing the BDD. It performs a step-by-step process through the fault tree, calculating the **ite** structure for each gate event. This process proceeds from the bottom gates (those fully defined by basic event inputs only) up through the fault tree, evaluating the **ite** structure of each gate until the top gate is reached. The **ite** structure of each gate is such that it is composed only of the basic events in the fault tree.

```
Computation(<op>, F, G)
        if (F=1)
                if <op>='OR'
                        return 1
                else if <op>='AND'
                        return G
        else if (G=1)
                if <op>='OR'
                        return 1
                else if <op>='AND'
                        return F
        else if (F=0)
                if <op>='OR'
                        return G
                else if <op>='AND'
                        return 0
        else if (G=0)
                if <op>='OR'
                        return F
                else if <op>='AND'
                        return 0
        else if computation-table has entry {(<op>, F, G), R}
                return R
        else if F=ite(x, F1, F2) and G=ite(y, G1, G2)
                if x<y
                        U← Computation(<op>, F1, G)
                        V← Computation(<op>, F2, G)
                        if U=V return U
                        else R← find-or-add to ite table ite(x, U, V)
                                return R
                if x=y
                        U← Computation(<op>, F1, G1)
                        V← Computation(<op>, F2, G2)
                        if U=V return U
                        else R← find-or-add to ite table ite(x, U, V)
                                return R
```

Figure 5.5 BADD's Main Algorithm Called *Compute*

The main algorithm, for evaluating the resulting **ite** structure of two **ite** structures F and G together with the operation <op>, is shown in pseudo-code form in figure 5.5. This algorithm corresponds to the subroutine *Compute* in the code BADD which can be seen in reference (48).

The results of any computation are stored in a computation table and this table is consulted for each subsequent computation. In this way a result is readily available if the computation has been previously completed and no computation is repeated. Before a result is placed in the **ite** table another subroutine is executed called *Remove*. Each time an **ite** structure has been completed, *Remove* is called to check whether **ite**(x, U, U) has occurred in which case the node U is returned.

The subroutine which enables the computation of the **ite** structure of each gate in the fault tree is called *Search*. This subroutine searches the binary fault tree for gate events whose **ite** structure has yet to be calculated. The calculation of the gate **ite** expression is then completed by the main subroutine *Compute*. Once the **ite** structure has been evaluated for a particular gate, the gate is given an **index**, in much the same way as the basic events are given an **index**. The gates **index** corresponds to the position in the **ite** table where the gate **ite** structure is stored. The flow chart in figure 5.6 exhibits the actions carried out by the subroutine *Search*. *Search* is executed until all of the gates in the fault tree have been indexed. If a VOTE gate is found subroutine *Votegate* is called which is based on **procedure (4)** in Chapter 4, section 4.5. *Votegate* includes subroutine *Compute* in the calculation of the **ite** structure of a VOTE gate.

The **ite** table which holds the nodes of the BDD provides an ordered triple. This means that each node is represented by three values, the first value indicates the basic event for this node, the second value gives the position in the **ite** table for the 1 branch node and the third value indicates the position of the 0 branch node in the **ite** table. If a fault tree has n basic events and m gates (after it has been converted to the 'binary' tree) then the first n rows in the **ite** table represent these basic events and the next m rows represent the gates. As a result the $n+m^{th}$ row will represent the node of the top gate, all rows after the $n+m^{th}$ row correspond to intermediate calculations. Note that only a proportion of the addresses in the **ite** table will be used in the construction of the BDD. To illustrate the nature of the resulting **ite** table for a particular fault tree the **ite** table for the **fatram** fault tree is shown in table 5.1. To then draw the BDD from the **ite** table a pointer is used to indicate the position of the top gate **ite** structure in the table, this is the root node, the 1 and 0 branches are then drawn using their respective positions in the table. The position of the top node for the **fatram** fault tree in the **ite**

table is 17 and the BDD for this fault tree is pictured in figure 5.7. The **index** for each basic event of a node in the BDD has been converted back to the name (letter) for that basic event. Further, the position in the **ite** table of each node is included in figure 5.7.

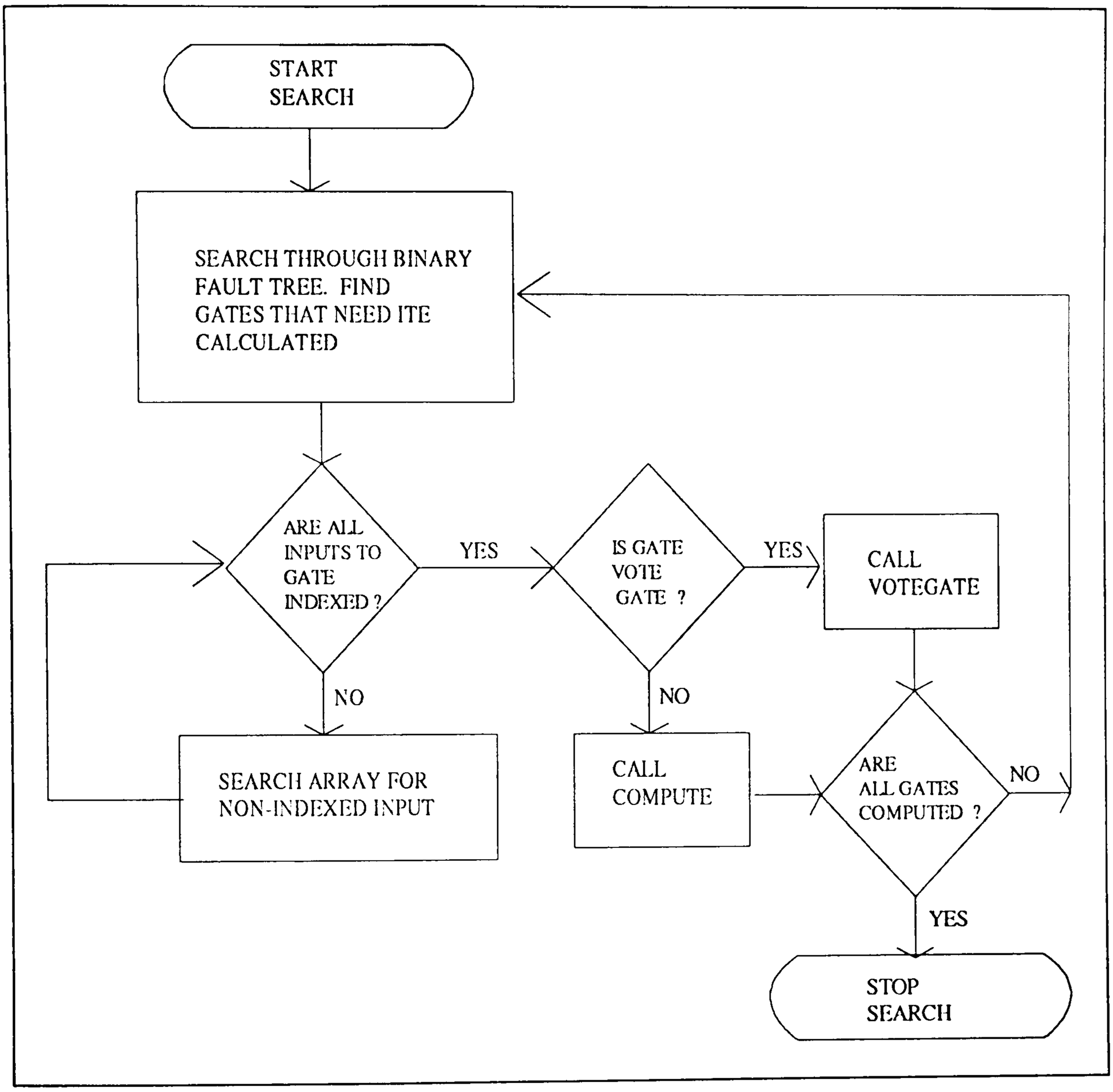


Figure 5.6 Flow Chart Illustrating Subroutine *Search*

5.2.6 Minimising the Top Event ite Structure

Once the top event **ite** structure has been evaluated, there is no guarantee that the resulting BDD will be minimum i.e. will produce the minimal cut sets. The form of the BDD generated at this stage needs to be retained for the fault tree quantification as described in Chapters 6 and 7. To produce the qualitative results a minimising procedure needs to be executed. The programmed version of the minimising algorithm is based on the algorithm by Rauzy (35). The subroutine *Minsol* and *Without* are

employed in this minimising process. The algorithms for these two subroutines can be found in figures 5.8 and 5.9 respectively.

Row Number	Basic Event Value	1 Branch Position	0 Branch Position
1	1	-1	0
2	2	-1	0
3	3	-1	0
4	4	-1	0
5	5	-1	0
6	6	-1	0
7	7	-1	0
8	8	-1	0
9	4	-1	8
10	4	-1	18
11	4	6	19
12	3	-1	5
13	3	-1	20
14	2	-1	3
15	2	-1	21
16	1	13	0
17	1	22	0
18	7	-1	8
19	6	18	0
20	4	-1	5
21	3	-1	11
22	2	13	23
23	3	-1	24
24	4	6	25
25	5	19	0

Table 5.1 **Ite** Table for **fatram**

Next, *Minsol* is executed with the starting point being the position of the top node in the **ite** table. A tracking pointer is used to indicate which branch of each node is being dealt with. Referring back to Chapter 4 equation (4.6) it is shown that a subset of the total set of solutions of the BDD is $[\{\delta\} \cap x]$ where the set $\{\delta\}$ constitutes the minimal solutions of the 1 branch of the top node (with basic event x) which are not included

on the 0 branch of the top node. Therefore the minimal solutions of the 1 branches are dealt with first, followed by the 0 branches.

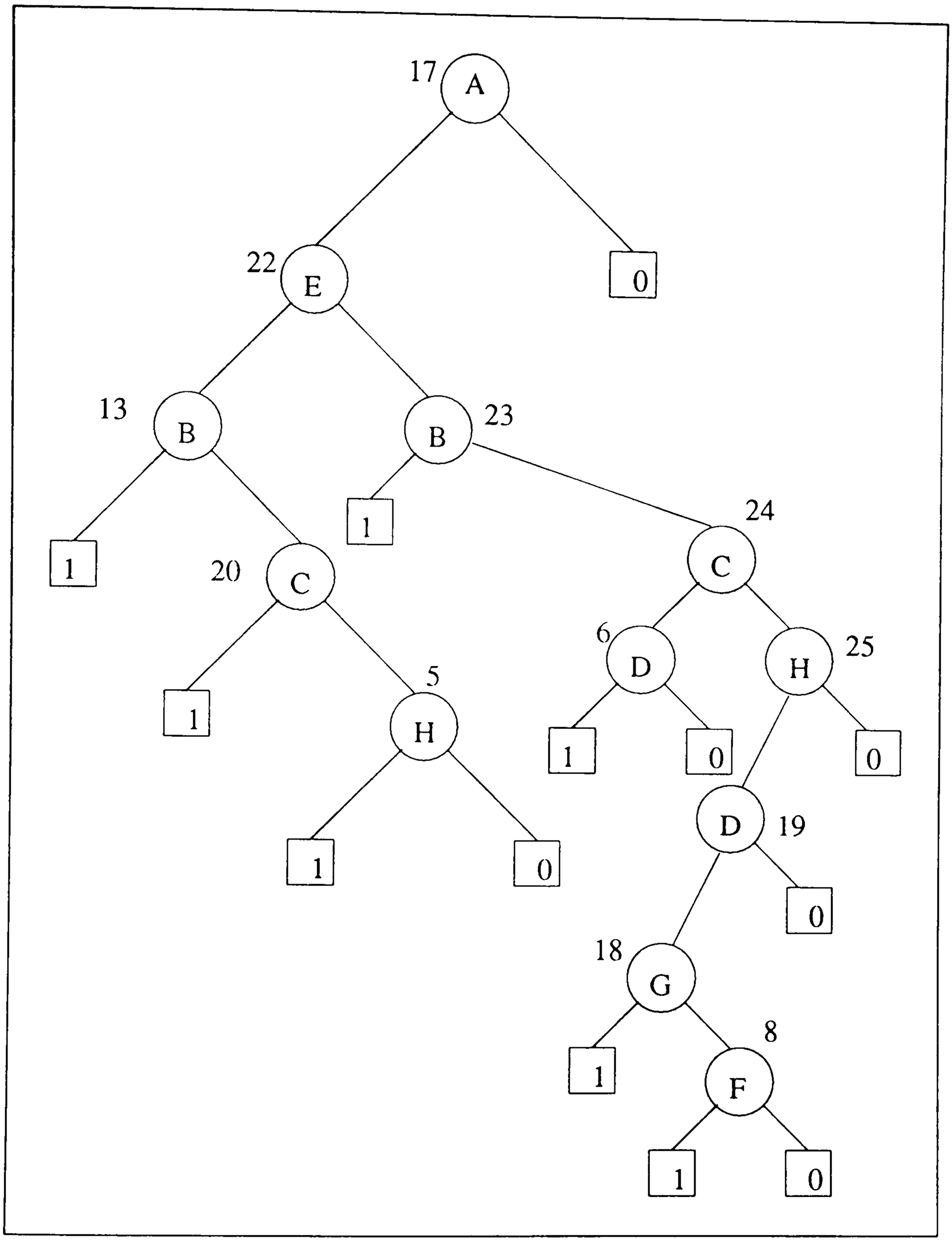


Figure 5.7 BDD for **fatram**

Looking at the *Minsol* algorithm in figure 5.8, it is shown that if a node is a terminal node then it is automatically minimal. However if the node is non-terminal and the result has not been found in the computation table we need to continue to track through the BDD, i.e. increase the level of computation (level 1 corresponds to the top node). For each node at a particular level, information about this node must be stored in an array, here called **inform**, and available for later use. Seven pieces of information

for each node F, where F=**ite**(x, G, H), are stored. This information is illustrated in figure 5.10.

```
Minsol(F)
        if (F=0 or F=1) return F
        else if computation table has entry {(Minsol, F), R}
                return R
        else F=ite(x, G, H)
                K← Minsol(G)
                U← Without(K, H)
                V← Minsol(H)
                if (U=V) return U
                R← find-or-add to ite-table (x, U, V)
                insert-in-computation-table {(Minsol, F), R)
                return R
```

Figure 5.8 Algorithm for Computing Minimal Solutions - Subroutine *Minsol*

```
Without(F, G)≡
        if (F=0) return 0
        else if (G=1) return 0
        else if (G=0) return F
        else if (F=1) return 1
        else if computation table has entry {(Without, F, G), R}
                return R
else F=ite(x, F1, F2) G=ite(y, G1, G2)
        if (x<y) U← Without(F1, G)
                V← Without(F2, G)
                R← find-or-add to ite-table (x, U, V)
                insert-in-computation-table {(Without, F, G), R}
                return R
else if (x>y)
        return Without (F, G2)
else x=y
        U← Without(F1, G1)
        V← Without(F2, G2)
        R← find-or-add to ite-table (x, U, V)
        insert-in-computation-table {(Without, F, G), R}
        return R
```

Figure 5.9 Algorithm for Computing *Without*(F, G)

inform(level, 1)=position of F in **ite** table
inform(level, 2)=**index** of basic event of node F, i.e. index of x
inform(level, 3)=position of 1 branch of F in **ite** table, i.e. G
inform(level, 4)=position of 0 branch of F in **ite** table, i.e. H
inform(level, 5)=*Minsol*(G)
inform(level, 6)=*Minsol*(H)
inform(level, 7)=*Without*(*Minsol*(G), H)

Figure 5.10 Information Stored for Each Level of Computation in Subroutine *Minsol*

The nature of the algorithm is such that if all the seven pieces of information are not known for a particular level, then the algorithm increments a level to deal with the next node. When a level can be fully completed the algorithm backtracks i.e. moves back up the branch it has just come from to the previous level and fills in the missing information for that node. If both branches of the node at this level have been dealt with then the code backtracks 1 level again. However if both branches have not been considered the tracking pointer moves to the 0 branch and the algorithm continues.

The algorithm terminates when the level reaches 1 representing the top node and both branches of the top node have been dealt with. The minimum structure of F will be:

ite(x, **inform**(level, 7), **inform**(level, 6))

Subroutine *Without* deals with the **ite** structures passed through from subroutine *Minsol*, which are *Minsol*(G) and H (i.e. **inform**(level, 5) and **inform**(level, 4) respectively). *Without* then calculates the **ite** structure for *Without*(*Minsol*(G), H) and passes the result back to subroutine *Minsol* to be placed into **inform**(level, 7).

Referring to figure 5.9 which deals with the computation of *Without*(F, G), we see that if F and G are terminal vertices then the result can be readily returned to *Minsol* without further computation. However if F and G are both non-terminal vertices and the result has not been found in the computation table then further processing is required. Three different cases have to be dealt with which depend on the ordering of the basic events for the nodes F and G as shown in figure 5.9.

Similar to subroutine *Minsol* we have to deal with levels of computation for each operation, each level in subroutine *Without* is called wlevel. Again seven pieces of

information need to be stored in an array called **winfo** and available for later use. This information is illustrated in figure 5.11.

winfo(wlevel, 1)=F
winfo(wlevel, 2)=G
winfo(wlevel, 3)=basic event of node F
winfo(wlevel, 4)=basic event of node G
winfo(wlevel, 5)=position of resulting 1 branch of *Without*(F, G) in **ite** table
winfo(wlevel, 6)=position of resulting 0 branch of *Without*(F, G) in **ite** table
winfo(wlevel, 7)=position in **ite** table of the resulting structure *Without*(F, G)

Figure 5.11 Information Stored for Each Level of Computation in Subroutine *Without*

Once all the information for a level of computation has been obtained the algorithm backtracks to the previous level of computation and fills the missing results into the **winfo** array. The algorithm returns to *Minsol* when the level of computation equals 1 and all the information for that level has been completed.

The **ite** table after the minimising procedure for **fatram** is illustrated in table 5.2. Notice the extra **ite** calculation in row 26 which eliminates the redundant path {A, E, B} from the original non-minimal BDD shown in figure 5.7.

This **ite** table can now be used to draw the minimal BDD for **fatram** which is displayed in figure 5.12. Again the top event **ite** is found in row 17 of table 5.2.

5.2.7 Finding the Solutions or Minimal Cut Sets of the Binary Decision Diagram

After the BDD has been minimised a path tracing algorithm is needed to obtain the minimal cut sets of the fault tree. Remember paths commence at the root node or vertex of the BDD and finish at a terminal 1 vertex. The *Solutions* algorithm is given in figure 5.13, again this is based on the algorithm by Rauzy. The solutions of the BDD will be referred to as σ and the algorithm will be executed with $\sigma=\varnothing$ (empty set).

Solutions is executed with the starting point being the position of the top node in the **ite** table. A tracking pointer deals with the left and right branch of each node. The left branches (1 branches) are dealt with first.

The nature of the algorithm is such that the BDD is initially traced from the root node and travels along the 1 branches of each node encountered until it reaches a terminal 1 vertex. As each node has been passed through on the 1 branch the basic events of these nodes are all included in this particular solution of the BDD.

Row Number	Basic Event Value	1 Branch Position	0 Branch Position
1	1	-1	0
2	2	-1	0
3	3	-1	0
4	4	-1	0
5	5	-1	0
6	6	-1	0
7	7	-1	0
8	8	-1	0
9	4	-1	8
10	4	-1	18
11	4	6	19
12	3	-1	5
13	3	-1	20
14	2	-1	3
15	2	-1	21
16	1	13	0
17	1	22	0
18	7	-1	8
19	6	18	0
20	4	-1	5
21	3	-1	11
22	2	13	23
23	3	-1	24
24	4	6	25
25	5	19	0
26	3	0	20

Table 5.2 Minimal **ite** Table for **fatram**

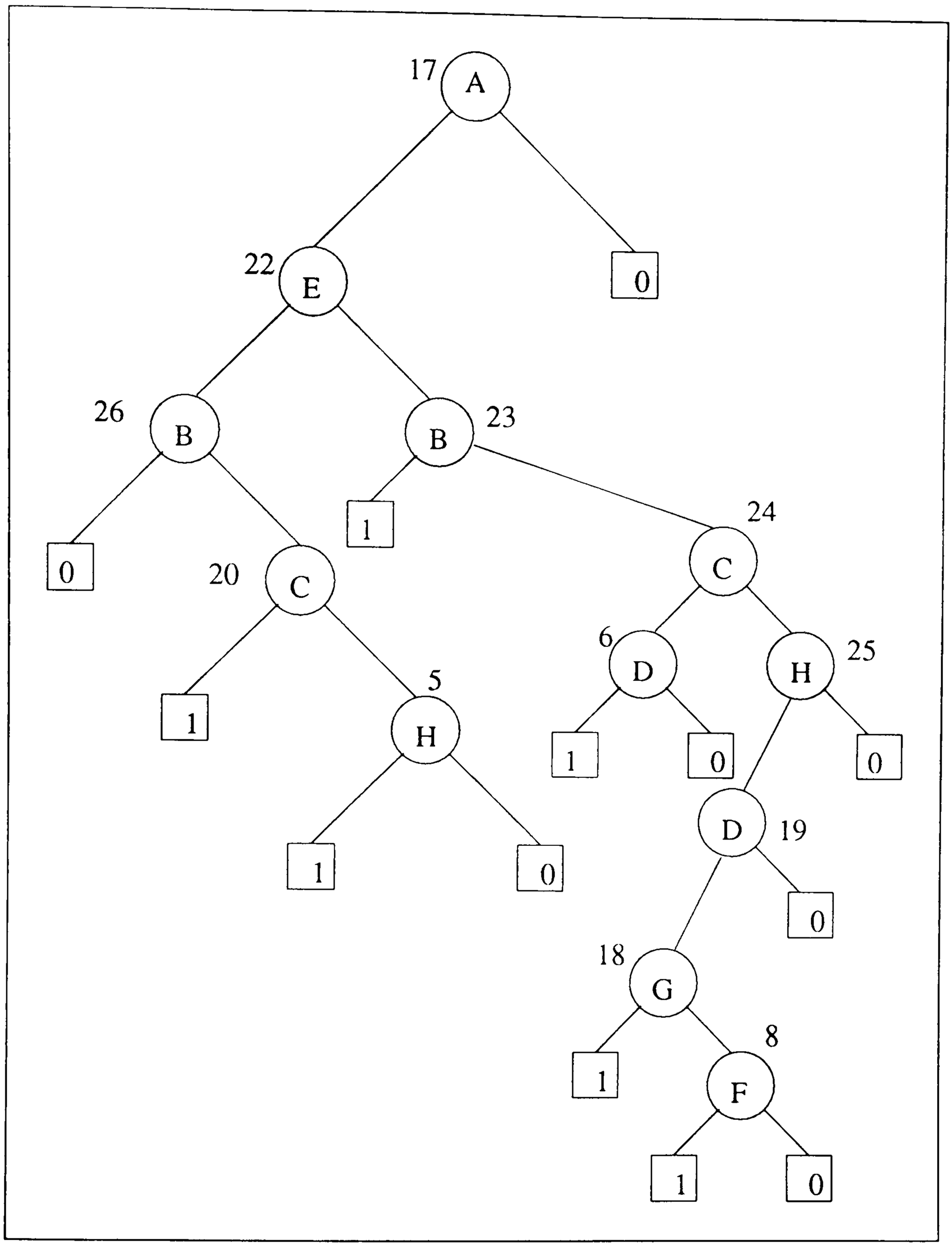


Figure 5.12 Minimal BDD for **fatram** Fault Tree

Solutions$(F, \sigma)\equiv$
    if $(F=0)$ *return* $\varnothing$
    else if $(F=1)$ *return* $\{\sigma\}$
    else $F=$***ite***$(x, F1, F2)$
        $S\leftarrow$ *Solutions*$(F1, \sigma\cup\{x\})$
        $T\leftarrow$ *Solutions*$(F2, \sigma)$
        return $S\cup T$

Figure 5.13 *Solutions* Algorithm which Obtains the Minimal Cut Sets

Once a terminal 1 vertex has been encountered the algorithm writes this path off as a minimal cut set and develops the next minimal cut set which starts off as being identical to the original. The algorithm then backtracks to the previous node and the tracking pointer deals with the 0 branch of this node. As the path is now going through on the 0 branch the basic event for this node is removed from the minimal cut set. The algorithm deals with the next node encountered in the same way as the root node at the start and the process continues.

Solutions terminates when both branches of the top node have been dealt with. The minimal cut sets obtained from the minimal BDD of the **fatram** fault tree are:

(1) {A, E, C}
(2) (A, E, H}
(3) {A, B}
(4) {A, C, D}
(5) {A, H, D, G}
(6) {A, H, D, F}

These are in complete agreement with an assessment of the original fault tree using FAULTREE+.

5.2.8 Output of BADD

BADD writes the results of the computation to an output file called ***.out** where * is the same name as used to specify the ***.ats** file. This output file provides the following information:

1. Name of the top event of the fault tree.
2. Number of basic events.
3. Number of gate events.
4. Number of occurrences of each event in the fault tree.
5. Ordering of the basic events.
6. Number of **ite** calculations required before minimising, after minimising and the difference between the two. This difference gives the number of extra **ite** calculations required to make the BDD minimum. These **ite** calculations can be taken directly from the computation table if the result exists.
7. The size of the computation table.

8. Both the non-minimal and minimal **ite** tables are provided, if requested by the user when BADD is executed, and the position of the top event in the minimal table is given, therefore the BDD can be drawn if desired.
9. The minimal cut sets are listed for the fault tree.
10. Lastly the number of non-repeated nodes and total number of nodes in the BDD are both given.

5.3 Comparing the Binary Decision Diagram Technique with a Conventional Approach

To test the efficiency of the BDD method, ten example fault trees were analysed using the BADD program and the results compared to the analysis using a conventional fault tree analysis package (FAULTREE+ (43)). A top-down, left-right, ordering of the basic events was used for the BDD analysis. The results are given in table 5.3 along with a summary of each fault tree. Both of the codes run on a Sun workstation and the execution time is given in seconds. Note that the times given for the BDD analysis include the conversion of the fault tree to the BDD.

Tree	No. of gates	No. of basic events	No. of minimal cut sets	BDD Time (s)	FAULT REE+ Time (s)	% Improve ment
1	19	19	27	0.5	1.0	50
2	17	11	43	0.5	1.0	50
3	29	61	7,471	0.2	1.0	80
4	60	57	11,934	0.6	-	-
5	19	19	63	0.6	0.9	33
6	21	21	75	0.5	0.8	38
7	58	57	36,990	0.6	1.5	60
8	70	68	4,892	0.8	1.1	27
9	21	40	416	0.2	0.9	78
10	26	16	20	0.5	1.0	50

Table 5.3 Ten Example Fault Trees BDD v FAULTREE+

It is evident from the results in table 5.3 that the BDD method is very efficient in terms of computation time, even for trees that have a large number of minimal cut sets as in tree 7. The analysis of fault tree 4 could not be executed in a reasonable time using the conventional approach, it took in excess of 4hrs to obtain all the minimal cut sets.

5.4 Variable Ordering Scheme

BDD's produced using a simple "top-down, left-right" ordering of the variables are frequently inefficient since they produce a large number of non-minimal cut sets and therefore require the minimisation algorithm. An alternative ordering scheme is presented here which focuses on those basic events which are repeated in the fault tree structure. It is the repeated events which cause the problem of non-minimal cut sets (10), and it has been found that considering these events first simplifies the resulting BDD structure and therefore makes it more optimal (Sinnamon and Andrews (56)).

A study of fifteen example fault trees indicates that using more restrictive event ordering produces a minimal BDD for thirteen out of the fifteen fault trees, whereas the top-down, left-right ordering results in a redundant BDD. This restrictive ordering has been called 'new' ordering, and it is employed when the fault tree contains repeated events. A summary of the fifteen example fault trees is given in table 5.4 below. The results were compared to the cut sets obtained using a manual conventional fault tree analysis technique (bottom-up approach) without the application of the Boolean Reduction Laws.

Fault Tree	Gates	Basic Events	Repeated Events	Cut Sets using bottom-up	Minimal Cut Sets from BDD
1	6	8	1	5	2
2	3	7	1	3	1
3	3	7	2	5	3
4	4	8	2	4	2
5	3	4	1	2	2
6	5	6	1	4	3
7	4	8	2	4	3
8	6	10	2	14	6
9	5	9	1	19	15
10	4	5	1	4	2
11	3	4	1	4	2
12	6	9	2	6	3
13	3	4	1	2	1
*14	4	7	1	3	3
*15	9	17	3	17	8

Table 5.4 Summary of Fifteen Example Fault Trees

Relatively small fault trees were chosen to emphasise that even for small fault trees savings can be made using the new ordering. Trees 14 and 15 did not produce the absolute minimal BDD, however from the investigation of these trees the new ordering appears to produce the most minimal BDD when compared to any other ordering.

The new ordering technique considers the gate events in a top down ordering similar to the top-down, left-right approach. However at each gate the basic event inputs are listed with the repeated events first (i.e., they are 'less than' the other inputs to that gate). If the gate has more than one repeated event as an input then the most repeated event is placed first, if they occur the same number of times then the events are taken in gate list order to break the tie. Also if an event has been ordered due to its occurrence higher up the tree then it is ignored for the ordering as an input to the lower gates.

To illustrate the advantages of new ordering, consider the simple fault tree in figure 5.14.

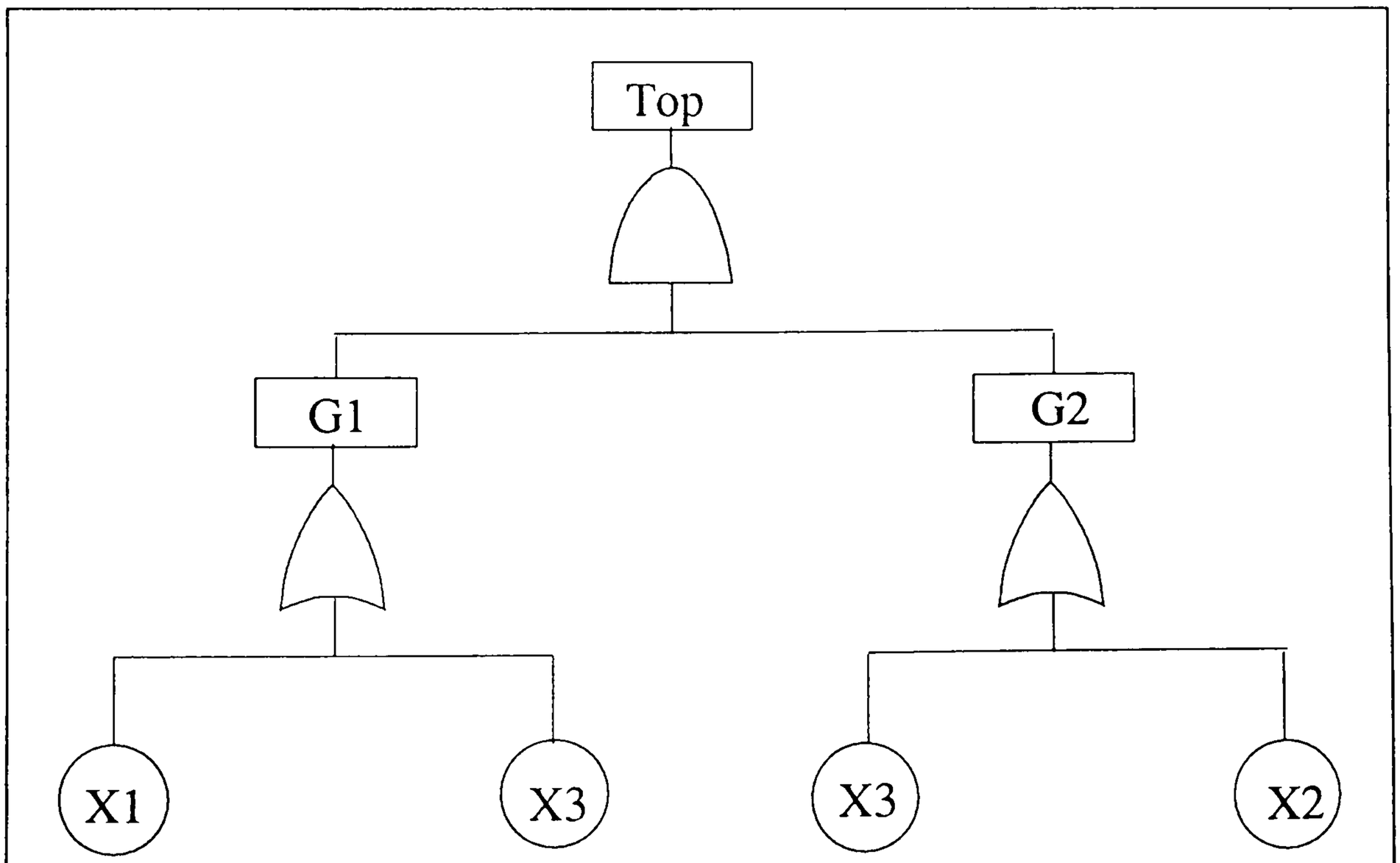


Figure 5.14 Example Fault Tree with Repeated Event X3

Top-down, left-right ordering would give X1<X3<X2. The **ite** structure calculations are then:

G2=X3+X2

=**ite**(X3, 1, 0)+**ite**(X2, 1, 0)

=**ite**(X3, 1, **ite**(X2, 1, 0))

G1=X1+X3

=**ite**(X1, 1, 0)+**ite**(X3, 1, 0)

=**ite**(X1, 1, **ite**(X3, 1, 0))

Top=G1.G2

=**ite**(X1, 1, **ite**(X3, 1, 0)).**ite**(X3, 1, **ite**(X2, 1, 0))

=**ite**(X1, **ite**(X3, 1, **ite**(X2, 1, 0)), **ite**(X3, 1, 0).**ite**(X3, 1, **ite**(X2, 1, 0)))

Top=**ite**(X1, **ite**(X3, 1, **ite**(X2, 1, 0)), **ite**(X3, 1, 0))

The BDD for Top resulting from this ordering is given in figure 5.15.

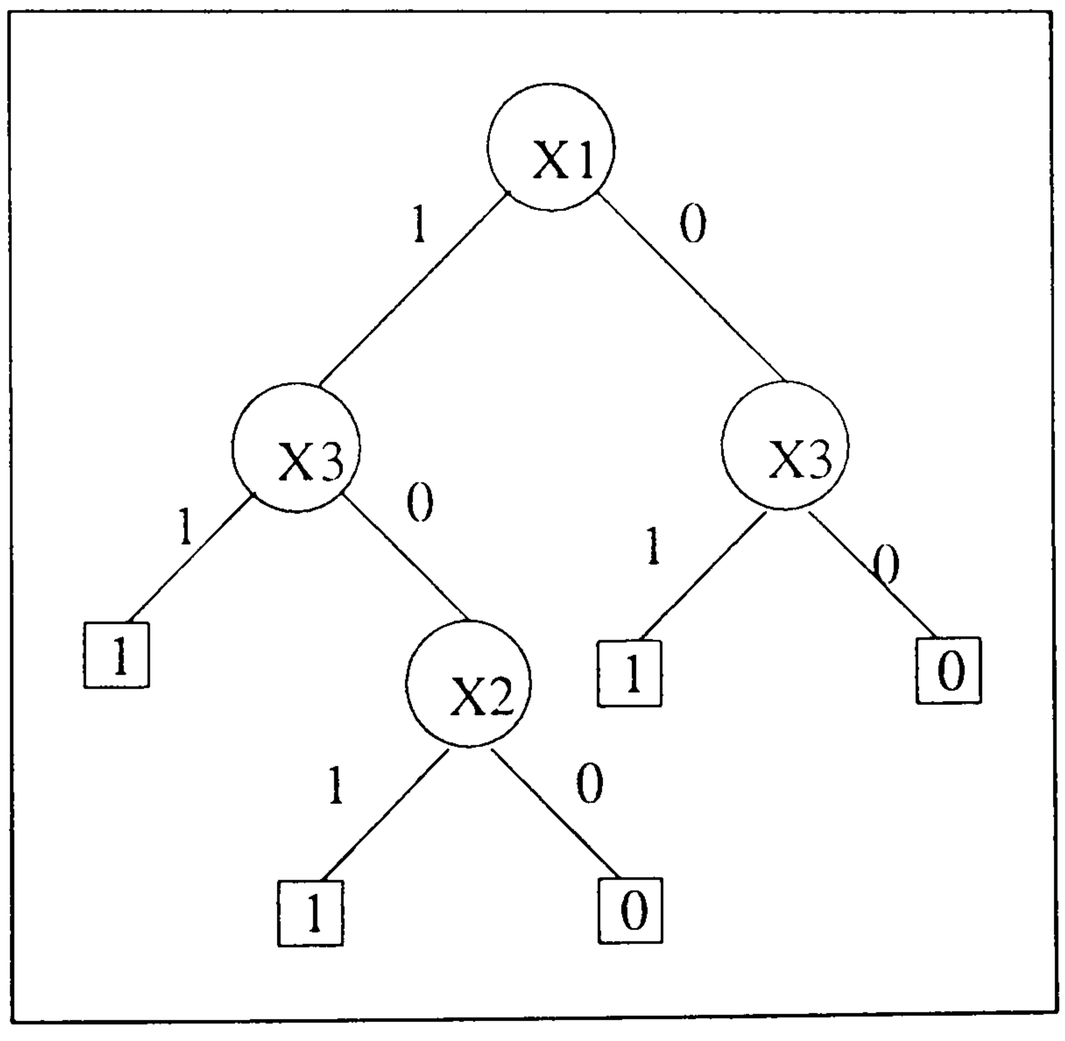


Figure 5.15 BDD for the Fault Tree in Figure 5.14

The paths through the BDD in figure 5.15 and hence cut sets are:

(1) X1.X3

(2) X1.X2

(3) X3

Therefore cut set {X1, X3} is redundant which leaves the minimal cut sets:

(1) {X1, X2}
(2) {X3}

Now using the new ordering one obtains X3<X1<X2. Note that G1 has the repeated event input X3 so it is placed before X1 in the ordering.

The **ite** structure for this ordering is calculated below:

G2=X3+X2
=**ite**(X3, 1, 0)+**ite**(X2, 1, 0)
=**ite**(X3, 1, **ite**(X2, 1, 0))

G1=X1+X3
=**ite**(X1, 1, 0)+**ite**(X3, 1, 0)
=**ite**(X3, 1, **ite**(X1, 1, 0))

Top=G1.G2
=**ite**(X3, 1, **ite**(X1, 1, 0)).**ite**(X3, 1, **ite**(X2, 1, 0))
=**ite**(X3, 1, **ite**(X1, 1, 0).**ite**(X2, 1, 0))
Top=**ite**(X3, 1, **ite**(X1, **ite**(X2, 1, 0), 0))

The BDD for this new ordering is given in figure 5.16.

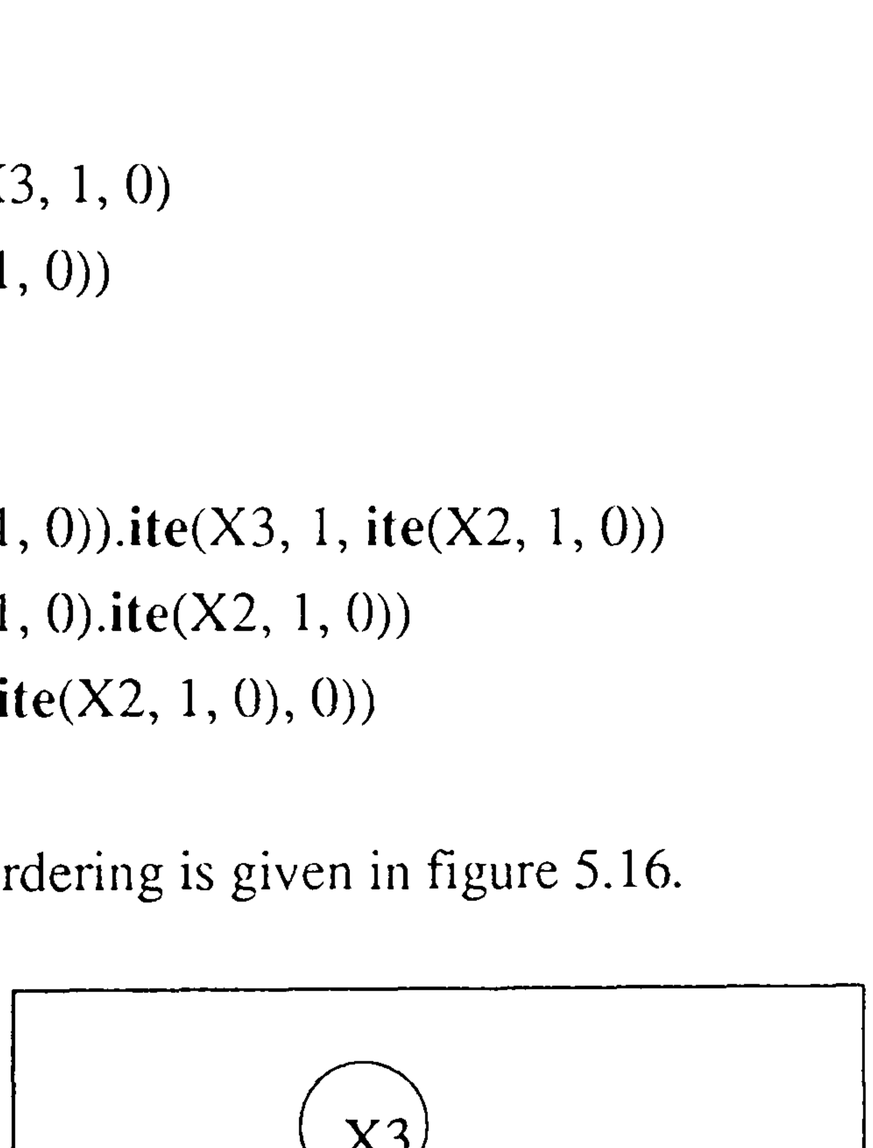


Figure 5.16 BDD for 'new' Ordering
X3<X1<X2

The paths through the BDD in figure 5.16 are:

(1) X3
(2) X1.X2

Therefore the new ordering gives only the minimal cut sets {X3} and {X1, X2}.

As the trees increase in size the new ordering becomes more beneficial in terms of eliminating redundant cut sets. This is demonstrated for the case of the slightly larger fault tree presented in figure 5.17.

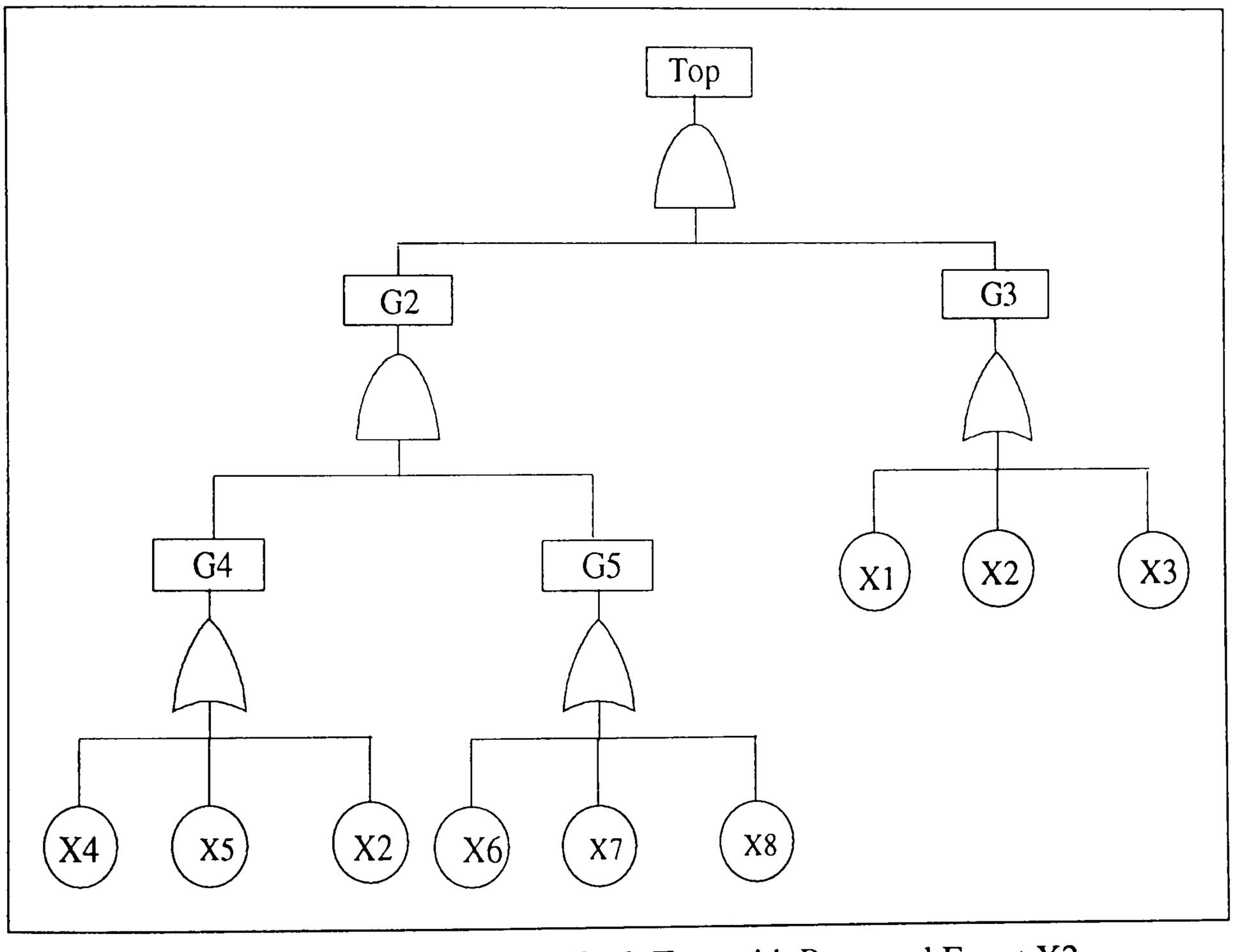


Figure 5.17 Slightly Larger Fault Tree with Repeated Event X2

The BDD for the ordering X1<X2<X3<X4<X5<X6<X7<X8 produces eighteen cut sets of which fifteen are minimal. However using the new ordering X2<X1<X3<X4<X5<X6<X7<X8 directly gives the fifteen minimal cut sets.

Bryant (34) recognised the problem of computing an ordering that minimises the size of the BDD and stated that for some trees it may not be possible to produce a minimal BDD whatever the ordering. Although it is shown here that using the new ordering

scheme, computation time can be reduced by creating a near minimal BDD. Further ordering schemes are discussed in depth in Chapter 8.

The advantage of using this simple restriction on ordering becomes more obvious when comparing the BDD analysis using this ordering to a bottom-up fault tree analysis approach (56). A bottom-up method is used to obtain the minimal cut sets of the fault tree shown in figure 5.18.

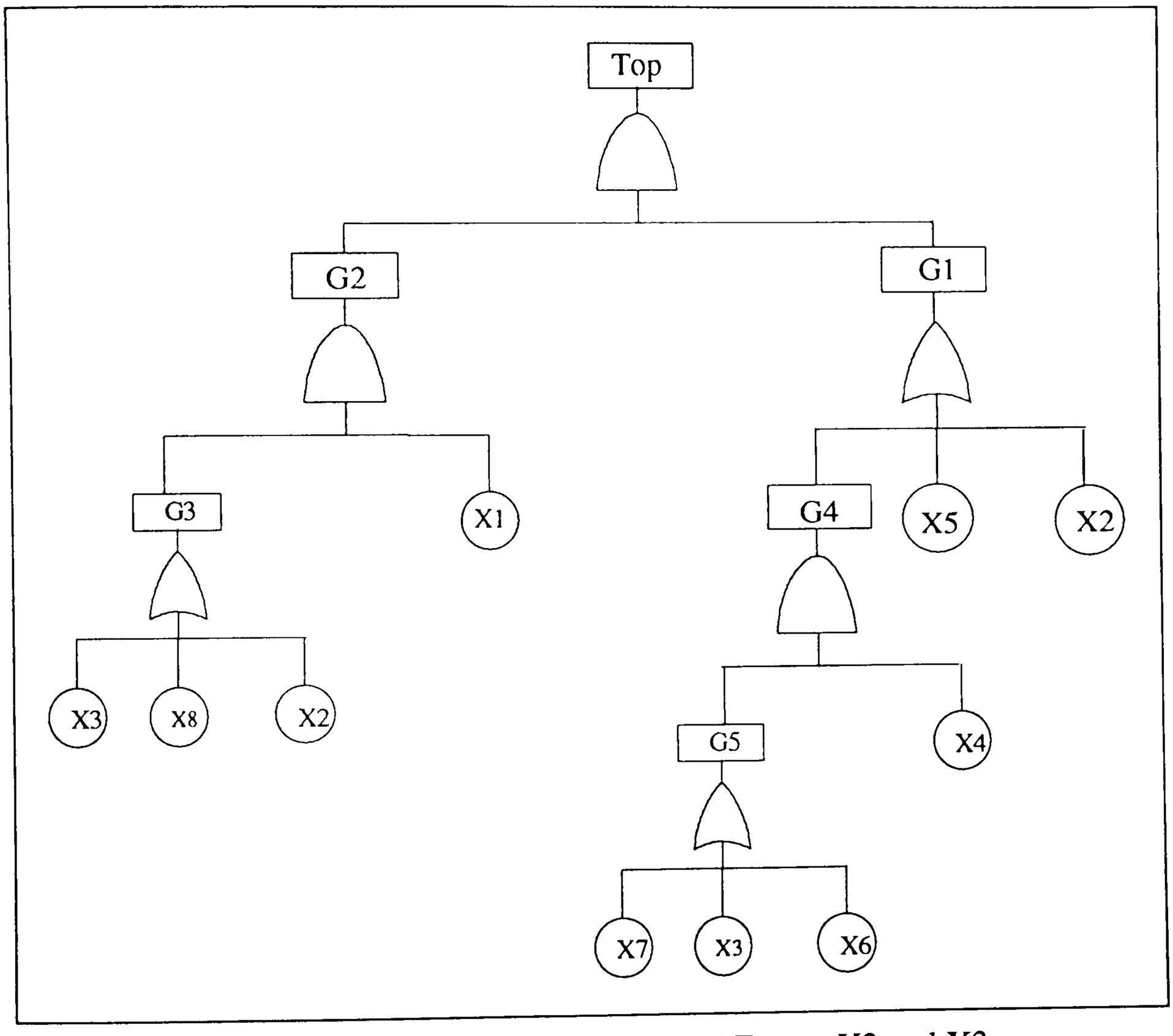


Figure 5.18 Fault Tree with Repeated Events X2 and X3

G5=X7+X3+X6

G4=G5.X4

=(X7+X3+X6).X4

=X7.X4+X3.X4+X6.X4

G1=G4+X5+X2

=X7.X4+X3.X4+X6.X4+X5+X2

G3=X3+X8+X2

G2=G3.X1
=(X3+X8+X2).X1
=X3.X1+X8.X1+X2.X1

Top=G2.G1
=(X3.X1+X8.X1+X2.X1).(X7.X4+X3.X4+X6.X4+X5+X2)

Top=X3.X1.X7.X4+X3.X1.X3.X4+X3.X1.X6.X4+X3.X1.X5+X3.X1.X2+
X8 X1.X7.X4+X8.X1.X3.X4+X8.X1.X6.X4+X8.X1.X5+X8.X1.X2+
X2.X1.X7.X4+X2.X1.X3.X4+X2.X1.X6.X4+X2.X1.X5+X2.X1.X2

Eliminating redundancies from the top event expression leaves the following six minimal cut sets:

(1) {X1, X3, X4}
(2) {X1, X3, X5}
(3) {X1, X8, X4, X7}
(4) {X1, X4, X6, X8}
(5) {X1, X8, X5}
(6) {X1, X2}

Comparing this approach with the BDD method, the ordering X1<X2<X5<X3<X8<X4<X7<X6 on variables directly produces the six minimal cut sets. Therefore in this case time is not wasted in computing and eliminating the seven redundant cut sets that were obtained with the bottom-up method.

5.5 Summary

1. The efficiency of the BDD technique to analyse the fault tree has shown promising results. It can analyse a fault tree with more than thirty thousand minimal cut sets in 0.6 seconds.

2. The BDD technique has been shown to obtain results in a faster computation time than a state of the art commercial fault tree analysis package.

3. Converting the fault tree to the BDD in the first instance causes no increase in computation time provided a good ordering of basic events is chosen. This demonstrates that further work on ordering schemes needs to be undertaken to select a good ordering for the basic events each time, Chapter 8 deals with this further work.

4. The nature of the BDD structure is such that it lends itself better to Boolean manipulation, i.e. minimising the BDD structure to obtain the minimal cut sets is more efficient than applying Boolean Reductions Laws to cut sets obtained using conventional fault tree analysis methods.

5. A BDD may be constructed which encodes the minimal cut sets directly without the need to apply the minimisation algorithm. This may be achieved by applying a 'new' ordering scheme which gives repeated events priority.

CHAPTER 6

TOP EVENT QUANTIFICATION USING THE BINARY DECISION DIAGRAM

6.1 Introduction

The fault tree diagram defines the causes of the system failure mode or "top event" in terms of the component failures and human errors, represented by basic events. By providing information which enables the probability of each basic event to be calculated the fault tree can then be quantified to yield reliability parameters for the system.

This chapter deals with calculating the top event probability, its failure rate and expected number of occurrences through the use of the binary decision diagram. In the quantification of the top event failure characteristics it is necessary to use approximations when implementing Kinetic Tree Theory (discussed in Chapter 3). Even for moderate sized problems it is not possible to evaluate all terms in the series expansions which yield the top event probability and failure intensity. The approximations that can be applied usually rely on the basic events having a small likelihood of occurrence. When this condition is not met it can result in large inaccuracies. These difficulties can be overcome by employing the Binary Decision Diagram (BDD) approach. Since the BDD method converts the fault tree diagram into a format which encodes Shannon's decomposition it allows the exact failure probability to be determined in a very efficient calculation procedure.

6.2 Top Event Probability

Quantification of the fault tree top event probability is generally calculated using the component failure/basic event existence probabilities and the minimal cut sets. Since the top event exists when at least one of the minimal cut sets exist, the top event probability is given by:

$$P(Top) = \sum_{i=1}^{nc} P(C_i) - \sum_{i=2}^{nc}\sum_{j=1}^{i-1} P(C_i \cap C_j) + \ldots + \ldots (-1)^{nc-1} P(C_1 \cap C_2 \cap \ldots \cap C_{nc})$$

where C_i, i=1,....nc are the minimal cut sets of the top event.

For all but the most simple fault tree structure the evaluation of each term in the expansion is not a practical proposition. For example, if a fault tree has 100, 000 minimal cut sets, the first term in the series expansion will have 100, 000 individual elements, the second term will consist of about 5×10^9 elements, the third term 1.67×10^{14} elements etc.., therefore resulting in a tedious and formidable calculation. Upper bound approximations are therefore frequently used to obtain the system failure probability. However these approximations can, under certain circumstances, lead to large errors in the calculation of the top event probability. Errors can result if the failure events are not rare, which is often the case when human error events are included in the fault tree. In this situation the neglected terms in the approximation may cause a significant truncation error. In addition, errors may occur when the fault tree contains 'conditional' events which are used with an inhibit gate (6). The inhibit gate is probabilistic and is determined by the conditional event. This conditional event may have a high probability of occurrence.

A feature of the binary decision diagram structure is that the exact top event probability can be calculated and approximations are not necessary.

A binary decision diagram encodes an **ite** structure, as shown in section 4.5, derived from Shannon's formula (22), such that if $f(\mathbf{x})$ is the Boolean function for the top event of a fault tree then the Shannon formula can be written as:

$$\begin{aligned} f(\mathbf{x}) = x_i . F_1(x_1, x_2, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n) \\ + \overline{x_i} . F_2(x_1, x_2, \ldots x_{i-1}, 0, x_{i+1}, \ldots x_n) \end{aligned} \tag{6.1}$$

and the corresponding **ite** structure for equation (6.1) is $ite(x_i, F_1, F_2)$. When a Boolean function is expressed in this form, the probability of the top event is obtained by taking the expectation of each term which results in:

$$\begin{aligned} E[f(\mathbf{x})] &= Q_{sys}(\mathbf{q}) \\ E[F_1(\mathbf{x})] &= Q_{sys}(q_1, \ldots q_{i-1}, 1, q_{i+1}, \ldots q_n) \\ E[F_2(\mathbf{x})] &= Q_{sys}(q_1, \ldots q_{i-1}, 0, q_{i+1}, \ldots q_n) \\ E[f(\mathbf{x})] &= q_i . E[F_1(\mathbf{x})] + (1 - q_i) . E[F_2(\mathbf{x})] \end{aligned}$$

where $q_i = E[x_i]$, the probability that event i has occurred.

This means that each path through the BDD to a terminal 1 vertex is mutually exclusive or disjoint, therefore to obtain the probability of occurrence of the top event ($Q_{sys}(t)$) the sum of the probabilities of the disjoint paths through the BDD is calculated (Sinnamon and Andrews (51)). For probability calculations the unminimised BDD is used to find the disjoint paths. The probability of the disjoint paths are obtained from the BDD by taking the probability of all events including both the 0 branches and the 1 branches in the paths ending in a terminal 1 state. Schneeweiss (52) uses a similar approach to the BDD method to obtain an expression for the top event which was is in the form of disjoint products. Calculation of the top event probability was then obtained by summing the probabilities of these events. Schneeweiss called this approach the Decision Tree Method.

The reason that the unminimised BDD is used for the probability calculation of the top event is that the minimisation algorithm alters the structure function and therefore its expected value, this is further discussed in Chapter 7.

To illustrate the calculation of the top event probability refer to the fault tree in figure 6.1. The BDD for this fault tree, using a top-down, left-right ordering of basic events, X1<X2<X3<X4, is the one shown in figure 6.2. The detailed **ite** calculation for this example can be seen in Ref. (53).

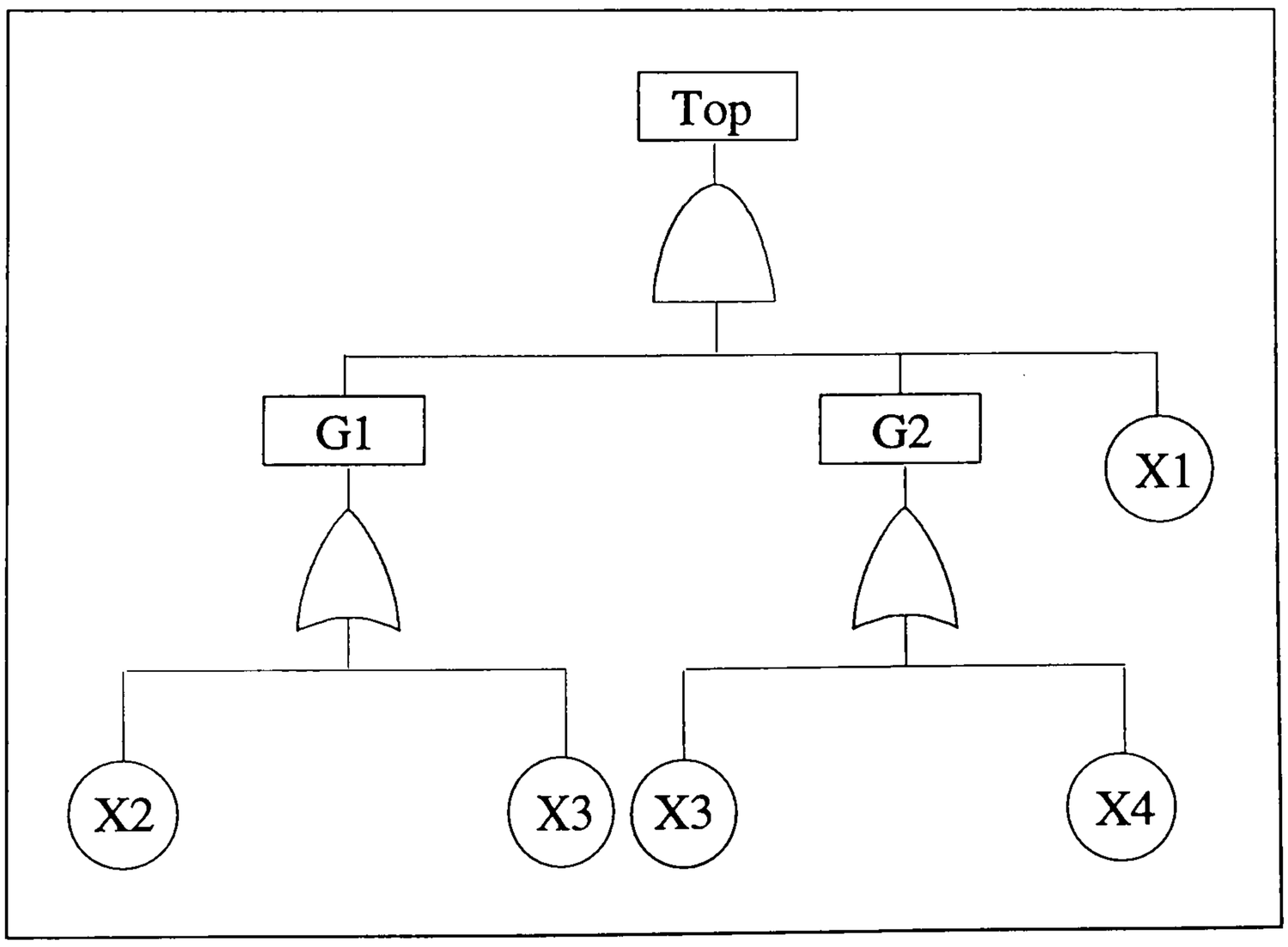


Figure 6.1 Example Fault Tree

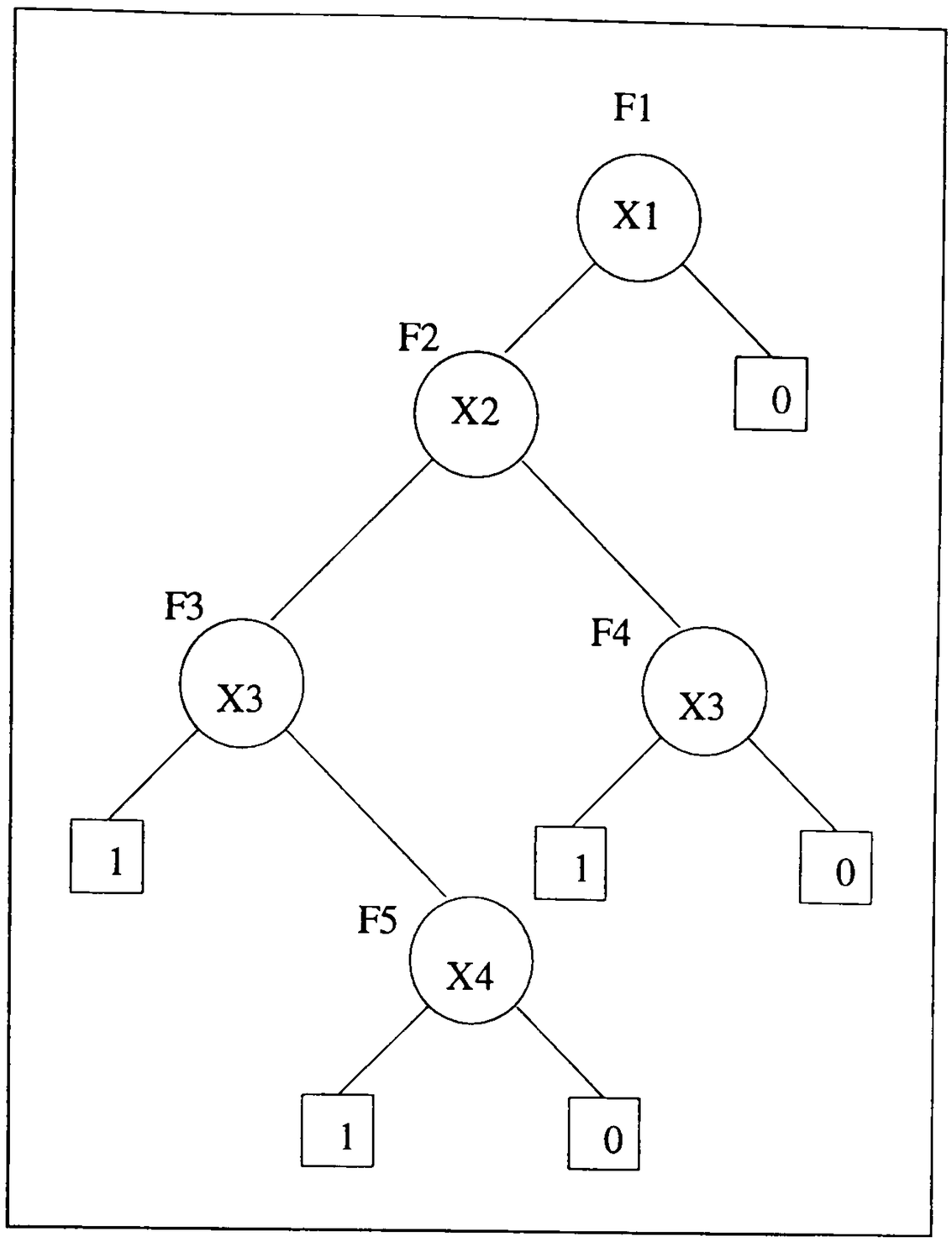


Figure 6.2 BDD for **ite**(X1, **ite**(X2, **ite**(X3, 1, **ite**(X4, 1, 0)), **ite**(X3, 1, 0)), 0)

The disjoint paths through the BDD in figure 6.2 are:

(1) X1.X2.X3
(2) X1.X2.$\overline{X3}$.X4
(3) X1.$\overline{X2}$.X3

Before continuing with the calculation of $Q_{sys}(t)$ the basic events need to be assigned probabilities. Component failure data for this example are given in table 6.1. Probabilities contained in the table are steady-state probabilities (see section 6.3). The values in table 6.1 are again used later for further calculations.

Basic Event	q_i	λ_j	$w_j=\lambda_j(1-q_j)$
X1	0.01	1.0×10^{-6}	9.9×10^{-7}
X2	0.02	4.0×10^{-6}	3.92×10^{-6}
X3	0.03	2.0×10^{-4}	1.94×10^{-4}
X4	0.04	$3.0E\times10^{-5}$	2.88×10^{-5}

Table 6.1 Basic Event Data

where:

$\mathbf{q_i}$ - Unavailability of component i.
$\mathbf{\lambda_i}$ - Conditional failure intensity of component i.
$\mathbf{w_i}$ - Unconditional failure intensity of component i.

Q_{sys} is obtained by summing the probabilities of the disjoint paths through the BDD giving:

$$Q_{sys} = P(X1.X2.X3 + X1.X2.\overline{X3}.X4 + X1.\overline{X2}.X3)$$
$$= q_{X1} \cdot q_{X2} \cdot q_{X3} + q_{X1} \cdot q_{X2} \cdot (1 - q_{X3}) \cdot q_{X4} + q_{X1} \cdot (1 - q_{X2}) \cdot q_{X3}$$
$$=(0.01)(0.02)(0.03)+(0.01)(0.02)(1\text{-}0.03)(0.04)+(0.01)(1\text{-}0.02)(0.03)$$
$$Q_{sys}=3.0776\times10^{-4}$$

A computational method to calculate the probability of the top event using the BDD has been implemented in a computer program called QUANT. QUANT employs the algorithm used by Rauzy (35) which can be seen in Figure 6.3.

The subroutine which performs this probability calculation is called *Qsolutions*. *Qsolutions* is very similar to the subroutine called *Solutions* discussed in section 5.2.7, with the exception that *Qsolutions* is applied to the original unminimised BDD and the 0 branches are also included in each path to a terminal 1 vertex.

```
probability(F)≡
        If (F=0) return 0
        else if (F=1) return 1
        else if (computation-table has entry {<probability, F, - >, R})
                return R
        else if (F=ite(x, G, H))
                R← p(x).probability(G)+(1-p(x)).probability(H)
                insert-in-computation-table ({<probability, F, - >, R})
                return R
```

Figure 6.3 Algorithm for computing *probability*(F)

6.3 Basic Event Model Types

The various mathematical expressions that are available to calculate the component unavailability constitute the components 'Model Type'. The quantification program, QUANT allows four different model types which are discussed below. These model types are those commonly available in commercial fault tree packages. The particular model type and its failure and repair parameters for each basic event are stored in an input file called ***.aqd**. The ***.aqd** file must be compatible with the ***.ats** file for the fault tree which is to be analysed. This means that the events for which the data is specified in the ***.aqd** file must be consistent with the basic events appearing in the fault tree structure defined in the ***.ats** file. When QUANT is executed both the ***.ats** and ***.aqd** files are read into the program.

Each basic event in the fault tree has two lines of data within the ***.aqd** file to specify its failure and repair characteristics, the first line gives the name of the basic event and the model type. The second line gives a list of the failure parameters provided for that basic event. The simple ***.aqd** text file may be constructed by the user (the format is given in Appendix II) or it can be developed using the software package FAULTREE+ (43).

6.3.1 Fixed Unavailability and Unconditional Failure Intensity

This model specifies a constant unavailability (q_i) for basic event i and a constant unconditional failure intensity (w_i). This is the simplest model type as q_i and w_i are specified there is no calculation necessary.

6.3.2 Constant Failure and Repair Rate Model

This model is relevant for components which experience a constant failure rate (λ) and a constant repair rate (μ) and where the failure event is revealed and repair is instigated immediately. When the program QUANT encounters this type of model in the ***.aqd** file it prompts the user to give a specified time point for the analysis. For example if λ and μ are given per hour then the user would input a time in consistent units.

The unavailability and unconditional failure intensity for the basic event are calculated using the following equations:

$$q(t) = \frac{\lambda}{\lambda + \mu}\{1 - \exp[-(\lambda + \mu)t]\} \tag{6.2}$$

[If a component is non-repairable with constant failure rate specify μ=0 and then $q(t) = 1 - \exp(-\lambda t)$]

$$w(t) = \lambda\{1 - q(t)\} \tag{6.3}$$

where:

$q(t)$ = unavailability at time t.
$w(t)$ = unconditional failure intensity at time t.

Steady - State Constant Failure and Repair Rate Model

When QUANT is executed it prompts the user to specify whether or not a steady-state analysis is desired. If this is the case then as $t \rightarrow \infty$.

$$q(t) \rightarrow \frac{\lambda}{\lambda + \mu} \tag{6.4}$$

However if a time-dependent, transient analysis is desired then a different program needs to be executed. This program is called QUANTIME and it deals with the time dependent analysis by initially dividing the time point specified for the analysis into 10 equal intervals. It then calculates $q(t)$ and $w(t)$ for each component at each time t where $t = t_0, t_1, \ldots\ldots t_{10}$. QUANTIME proceeds to calculate the unavailability of the system and also the unconditional system failure intensity at each of these times. The reason for dividing the time duration into 10 equal intervals, and doing the aforementioned calculations, lies with the evaluation of the expected number of top event occurrences using numerical integration which is discussed in section 6.4.

6.3.3 Mean Time to Failure and Repair Model

For this model the mathematical expression for component unavailability is also that given in equation 6.2. However the parameters specified are the Mean Time to Failure (MTTF - r) and the Mean Time to Repair (MTTR - τ). As shown in Chapter 3 for

constant failure rate $r = \frac{1}{\lambda}$ (equation 3.5) and for constant repair rate $\tau = \frac{1}{\mu}$ (equation 3.7), therefore by substituting r and τ into equations (6.2) and (6.3) the following equations are used to calculate $q(t)$ and $w(t)$ respectively:

$$q(t) = \frac{\tau}{\tau + r}\{1 - \exp[-(\frac{1}{r} + \frac{1}{\tau})t]\} \tag{6.5}$$

$$w(t) = \frac{1}{r}\{1 - q(t)\} \tag{6.6}$$

Steady-State Mean Time To Failure and Repair Model

If a steady-state analysis is desired then q_i will be calculated using:

$$q_i = \frac{\tau}{\tau + r} \tag{6.7}$$

and w_i will be calculated using equation (6.6).

6.3.4 Dormant Failure and Periodic Inspection Model

This model is appropriate for dormant components whose failure remains unrevealed until periodic inspection is performed. It produces a mean unavailability and unconditional failure intensity from the constant failure rate (λ), MTTR (Mean Time To Repair) (τ) and inspection interval (θ).

The unavailability and unconditional failure intensities for each basic event are given by:

$$q_{AV} = \lambda(\tau + \frac{\theta}{2}) \tag{6.8}$$

$$w = \lambda(1 - q) \tag{6.9}$$

6.4 Unconditional System Failure Intensity

For some systems it is the unreliability, $F(t)$ which is required for the top event i.e., the probability it will not work continuously over a given time period. An upper bound for this is the expected number of top event occurrences W(0, t):

i.e. $Q(t) \leq F(t) \leq W(0,t)$

Availability≤Unreliability≤Expected number of top event occurrences

Let $P_i(t) = P$(expected i system failures in [0, t])

then $$F(t) = \sum_{i=1}^{\infty} P_i(t) \leq \sum_{i=1}^{\infty} i.P_i(t) = W(0,t)$$

and $$W(0,t) = \int_0^t w_{sys}(u)du \qquad (6.10)$$

where $w_{sys}(t)$ is the system unconditional failure intensity:

$$w_{sys}(t) = \sum_i G_i(\mathbf{q}).w_i(t) \qquad (6.11)$$

where $G_i(\mathbf{q})$ is the criticality function (see Chapter 3, section 3.5) and the summation is over each component i.

The criticality function $G_i(\mathbf{q})$ is defined as the probability that the system is in a critical state with respect to component i and that the failure of component i will then cause the system to go from the working to the failed state, i.e., the probability that the system fails only if component i fails. Therefore:

$$G_i(\mathbf{q}) = Q(1_i, \mathbf{q}) - Q(0_i, \mathbf{q}) \qquad (6.12)$$

where:

$Q(1_i, \mathbf{q})$ - is the probability of system failure with $q_i(t) = 1$

$Q(0_i, \mathbf{q})$ - is the probability of system failure with $q_i(t) = 0$

Since $Q_{sys}(t)$ is a linear function in each $q_i(t)$ then $G_i(\mathbf{q})$, for each basic event can also be given by:

$$G_i(\mathbf{q}) = \frac{\partial Q_{sys}(t)}{\partial q_i(t)} \tag{6.13}$$

Evaluating each of the two terms $Q(1_i, \mathbf{q})$ and $Q(0_i, \mathbf{q})$ for each component could be achieved by first substituting $q_i(t) = 1$ and then $q_i(t) = 0$, i.e., the probability that component i equals 1 and 0 respectively, and re-running the system failure probability calculations. This would require the equivalent of 2n evaluations of the top event probability where n is the number of components in the system to deduce all terms required in the expression for $w_{sys}(t)$ in equation (6.11).

However a more efficient calculation method can be produced which requires only one pass of the BDD. Consider the variable Xi which occurs at least once in the BDD (refer to figure 6.4).

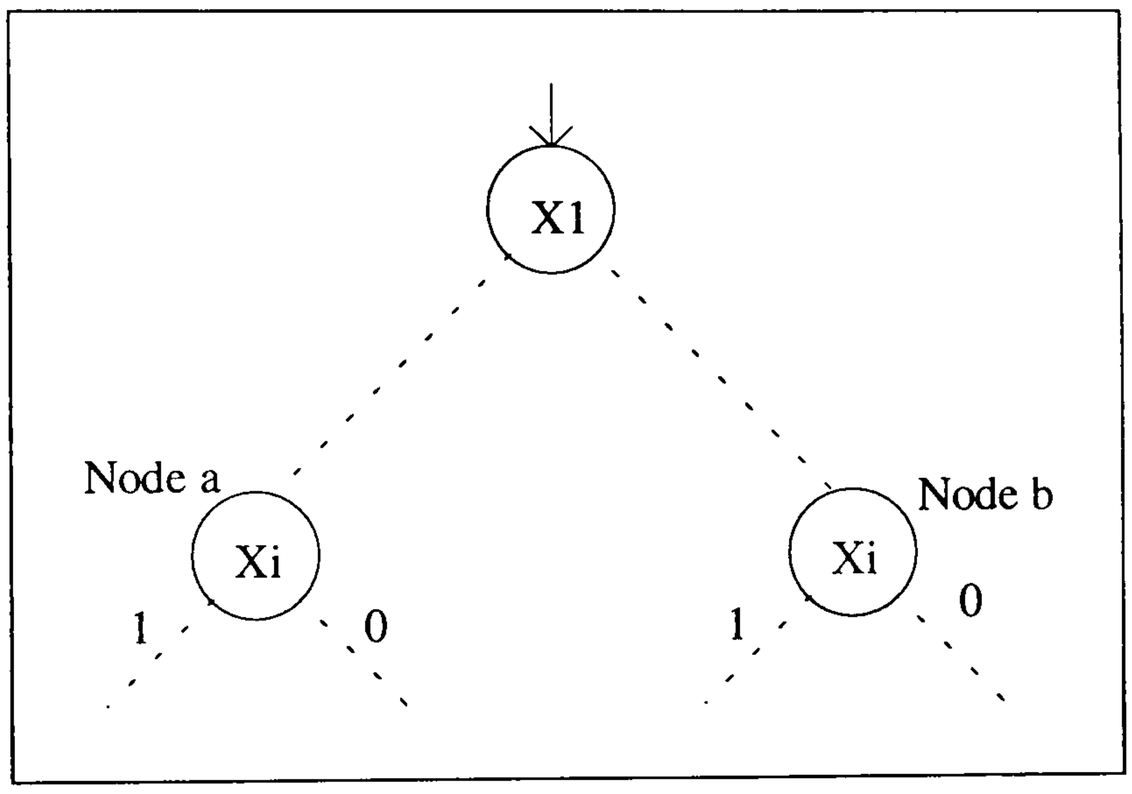


Figure 6.4 Considering Variable Xi

The following equations can then be used:

$$Q(1_i, \mathbf{q}) = \sum_n (pr_{xi}(\mathbf{q}).po^1_{xi}(\mathbf{q})) + Z(\mathbf{q}) \tag{6.14}$$

$$Q(0_i, \mathbf{q}) = \sum_n (pr_{xi}(\mathbf{q}).po^0_{xi}(\mathbf{q})) + Z(\mathbf{q}) \tag{6.15}$$

where:

$pr_{xi}(\mathbf{q})$ - is the probability of the path section from the root node to node xi (**Probprev**).

$po^1_{xi}(\mathbf{q})$ - is the probability of the path section from the 1 branch of node xi to a terminal 1 node (**Probpost 1 branch**).

$po^0_{xi}(\mathbf{q})$ - is the probability of the path section from the 0 branch of node xi to a terminal 1 node (**Probpost 0 branch**).

$Z(\mathbf{q})$ - is the probability of the paths from the root node to the terminal 1 nodes which do not go through a node for variable xi.

n - All nodes for variable xi in the BDD.

Therefore:

$$G_i(\mathbf{q}) = \sum_n pr_{xi}(\mathbf{q})[po^1_{xi}(\mathbf{q}) - po^0_{xi}(\mathbf{q})] \tag{6.16}$$

A more efficient way to calculate $w_{sys}(t)$ is to make one pass of the BDD to calculate $pr_{xi}(\mathbf{q})$, $po^1_{xi}(\mathbf{q})$ and $po^0_{xi}(\mathbf{q})$ for each node. With this information each $G_i(\mathbf{q})$ can be evaluated from equation (6.16) and $w_{sys}(t)$ formed using equation (6.11).

The algorithm *Probpost* to calculate $po^1_{xi}(\mathbf{q})$ and $po^0_{xi}(\mathbf{q})$ is given in figure 6.5. For each node x_i in the BDD, *Probpost* calculates the sum of the probabilities of all the paths ending in a terminal 1 vertex leading from the 1 branch of the node x_i ($po^1_{xi}(\mathbf{q})$). Then the algorithm calculates the same value for all paths leading from the 0 branch of node x_i ($po^0_{xi}(\mathbf{q})$). The calculation of $pr_{xi}(\mathbf{q})$ can be achieved by the algorithm *Probprev* given in figure 6.6. For each node x_i *Probprev* calculates the probability of the path section leading from the root vertex to node x_i. The criticality function $G_i(\mathbf{q})$ for each basic event is calculated as shown in figure 6.7. The calculation of $G_i(\mathbf{q})$ simply requires the values $po^1_{xi}(\mathbf{q})$, $po^0_{xi}(\mathbf{q})$ and $pr_{xi}(\mathbf{q})$. These algorithms have been incorporated into the program QUANT.

```
Probpost(F)≡
        Do for all F
        F=ite(xi, G, H)
        po^1_xi(q)=prob(G)
        po^0_xi(q)=prob(H)
        insert in Probtable, R← Probtable(xi, po^1_xi(q), po^0_xi(q))
        Q← p(xi).prob(G)+(1-p(xi)).prob(H)
        insert-in-computation-table ({<prob, F, - >, Q})
        return R
        return Q
        next F
```

Figure 6.5 *Probpost* Algorithm

The implementation of these algorithms can be demonstrated by their application to the example BDD given in figure 6.2. The **ite** table indicating how the BDD is stored within the computer program is given in table 6.2.

Node Label	**Variable**	**1 branch pointer**	**0 branch pointer**
F1	X1	F2	0
F2	X2	F3	F4
F3	X3	1	F5
F4	X3	1	0
F5	X4	1	0

Table 6.2 **ite** table for the BDD in figure 6.2

Set Probprev(Fi)=0 for all i
Probprev(F)≡
start at root vertex, F
Probprev(F)=1
Add Probprev(F) to Probtable, i.e., Probtable(xi, $po_{xi}^{1}(\mathbf{q})$, $po_{xi}^{0}(\mathbf{q})$, $pr_{xi}(\mathbf{q})$*)*
Do for all F, root vertex to end vertices
*F=**ite**(xi, H1, H2)*
if H1=0 or 1 Goto [A]
Probprev(H1)=Probprev(H1)+p(xi).Probprev(F)
Add Probprev(H1) to Probtable

[A] *if H2=0 or 1 next F*
Probprev(H2)=Probprev(H2)+(1-p(xi)).Probprev(F)
Add Probprev(H2) to Probtable
next F

Figure 6.6 *Probprev* Algorithm

Set G(xi)=0 for all i
Do for all F
if F=Probtable(xi, $po_{xi}^{1}(\mathbf{q})$, $po_{xi}^{0}(\mathbf{q})$, $pr_{xi}(\mathbf{q})$*)*
$G(xi)=G(xi)+pr_{xi}(\mathbf{q})(po_{xi}^{1}(\mathbf{q})-po_{xi}^{0}(\mathbf{q}))$
insert-in-criticality table G(xi)
next F

Figure 6.7 Algorithm for calculating the Criticality Function $G_i(\mathbf{q})$

Performing one pass of the BDD to evaluate $po_{xi}^{1}(\mathbf{q})$ *and* $po_{xi}^{0}(\mathbf{q})$ for each node using *Probpost* and referring to the probability table, PROBTABLE in figure 6.8, gives:

Probpost(F5)

F5=**ite**(X4, 1, 0)

$po_{x4}^{1}(\mathbf{q})$=prob(1)=1

$po_{x4}^{0}(\mathbf{q})$=prob(0)=0

R←Probtable(X4, 1, 0)

Q←p(X4).p(1)+(1-p(X4)).p(0)=0.04

Probpost(F4)

F4=**ite**(X3, 1, 0)

$po_{x3}^{1}(\mathbf{q})$=prob(1)=1

$po_{x3}^{0}(\mathbf{q})$=prob(0)=0

R←Probtable(X3, 1, 0)

Q←p(X3).p(1)+(1-p(X3)).p(0)=0.03

Probpost(F3)

F3=**ite**(X3, 1, F5)

$po_{x3}^{1}(\mathbf{q})$=prob(1)=1

$po_{x3}^{0}(\mathbf{q})$=prob(F5)=0.04

R←Probtable(X3, 1, 0.04)

Q=p(X3).p(1)+(1-p(X3)).(0.04)=0.0688

Probpost(F2)

F2=**ite**(X2, F3, F4)

$po_{x2}^{1}(\mathbf{q})$=prob(F3)=0.0688

$po_{x2}^{0}(\mathbf{q})$=prob(F4)=0.03

R←Probtable(X2, 0.0688, 0.03)

Q=p(X2).(0.0688)+(1-p(X2)).(0.03)=0.030776

Probpost(F1)

F1=**ite**(X1, F2, 0)

$po_{x2}^{1}(\mathbf{q})$=prob(F2)=0.030776

$po_{x2}^{0}(\mathbf{q})$=prob(0)=0

R←Probtable(X1, 0.030776, 0)

Q=p(X1).(0.030776)+(1-p(X1)).p(0)=3.0776×10^{-4}

As can be seen the probability of the top event Q (calculated for F1 above) agrees with the probability calculated previously using the disjoint paths of the BDD.

The values of *Probpost* 1 branch and *Probpost* 0 branch for each node are entered into the node probability table called Probtable shown in figure 6.8.

Next *Probprev* (figure 6.6) is calculated and entered into the 5th column of the Probtable.

Probprev Algorithm:

Probprev(F1)=*Probprev*(F2)=*Probprev*(F3)=*Probprev*(F4)=*Probprev*(F5)=0

Probprev(F1)=1

F1=**ite**(X1, F2, 0)

Probprev(F2)=p(X1).*Probprev*(F1)

=(0.01).(1)=0.01

H2=0

F2=**ite**(X2, F3, F4)

Probprev(F3)=p(X2).*Probprev*(F2)

$=(0.02).(0.01)=2.0\times10^{-4}$

Probprev(F4)=(1-p(X2)).*Probprev*(F2)

$=(1-0.02).(0.01)=9.8\times10^{-3}$

F3=**ite**(X3, 1, F5)

H1=1

Probprev(F5)=(1-p(X3)).*Probprev*(F3)

$=(1-0.03).(2.0\times10^{-4})=1.94\times10^{-4}$

F4=**ite**(X3, 1, 0)

H1=1

H2=0

F5=**ite**(X4, 1, 0)

H1=1

H2=0

Calculation of the criticality function is then straight forward using the algorithm provided in figure 6.7.

Criticality Algorithm:

G(X1)=G(X2)=G(X3)=G(X4)=0

F1=Probtable(X1, 0.030776, 0, 1)

G(X1)=0+1(0.030776-0)

=0.030776

F2=Probtable(X2, 0.0688, 0.03, 0.01)

G(X2)=0+0.01(0.0688-0.03)

$=3.88\times10^{-4}$

F3=Probtable(X3, 1, 0.04, 2.0×10^{-4})

G(X3)=0+2.0×10^{-4}(1-0.04)

$=1.92\times10^{-4}$

F4=Probtable(X3, 1, 0, 9.8×10^{-3})

G(X3)=1.92E-4+9.8×10^{-3}(1-0)

$=9.992\times10^{-3}$

F5=Probtable(X4, 1, 0, 1.94×10^{-4})

G(X4)=1.94×10^{-4}(1-0)

$=1.94\times10^{-4}$

Since we have calculated the criticality function for each component, the steady-state unconditional failure intensity w_{sys} for the example fault tree shown in figure 6.1, can now be evaluated using the frequency data from table 6.1 and equation (6.11).

$$w_{sys} = G(X1).w_{X1} + G(X2).w_{X2} + G(X3).w_{X3} + G(X4).w_{X4}$$

$=(0.030776)(9.9\times10^{-7})+(3.88\times10^{-4})(3.92\times10^{-6})+(9.992\times10^{-3})(1.94\times10^{-4})+$

$(1.94\times10^{-4})(2.88\times10^{-5})$

$=1.97602\times10^{-6}$

Probtable

Node Label	Variable	post 1	post 0	*Probprev*
F1	X1	0.030776	0	1
F2	X2	0.0688	0.03	0.01
F3	X3	1	0.04	2.0E-4
F4	X3	1	0	9.8E-3
F5	X4	1	0	1.94E-4

where:

Probtable(i, 1)=Node Label
Probtable(i, 2)=Basic event of node Fi
Probtable(i, 3)=Probability of post 1 branch
Probtable(i, 4)=Probability of post 0 branch
Probtable(i, 5)=Probability of previous

Figure 6.8 Probtable Array

Using equation (6.10) the expected number of top event occurrences in time, t, can be obtained. For example if the expected number of failures over a 10 year operating period (i.e. 87600 hours) was required then:

$$W(0,87600) = \int_0^{87600} 1.97602 \times 10^{-6}\, dt$$
$$= 1.97602 \times 10^{-6} \times 87600$$
$$= 0.173$$

The calculation above for the expected number of top event occurrences was evaluated for system steady-state condition. If the system under study has components which have a time dependent model type then numerical integration is required to calculate W(0, t).

As previously mentioned in section 6.3.2 and 6.3.3, if the components unavailability and unconditional failure intensity are time dependent then these values are calculated for the times $t_0, t_1, \ldots\ldots t_{10}$. In addition $G_i(\mathbf{q})$ is evaluated for each of these times, which enables the calculation of $w_{sys}(t)$ using equation (6.11) for times $t_0, t_1, \ldots\ldots t_{10}$.

The program QUANTIME tackles this integration by utilising the trapezium rule, with 10 equal step lengths and ordinates $t_0, t_1, \ldots\ldots t_{10}$, to approximate the expected number of top event occurrences.

6.5 Applications

The BDD quantification method was benchmarked against a test example fault tree called 'Dresden-3' used by Platz and Olsen (44). The structure file (Dresden-3.ats) and data file (Dresden-3.aqd) for this tree are both given in Appendix III. For this particular fault tree a specially created model type was used for each component, where the failure parameters λ and τ (λ must be multiplied by 1.0×10^{-6}) are provided for each component. λ represents the component constant failure rate and τ its Mean Time to Repair (MTTR).

The following 'steady-state' calculations are then used for the model type, to obtain q_i and w_i for each basic event:

$$q_i = \frac{\lambda.\tau}{\lambda.\tau + 1} \tag{6.17}$$

$$w_i = \lambda(1 - q_i) \tag{6.18}$$

A summary of the quantification results is given in table 6.3. The code QUANT runs on a Sun workstation, the execution time is given in seconds.

Name	Dresden-3
No. of Gates	60
No. of Basic Events	57
No. of Minimal Cut Sets	11,934
Time (s)	0.6
$\mathbf{Q_{sys}}$	4.70499×10^{-7}
$\mathbf{w_{sys}}$	2.88139×10^{-8}

Table 6.3 BDD Quantification Results for Dresden-3 Fault Tree

As a comparison, Dresden-3 was analysed using a state-of-the art conventional fault tree analysis package (FAULTREE+ (43)), whose results can be seen in table 6.4.

No. of Minimal Cut Sets	11,934
Time	4hrs 10min 28s
Q_{sys}	4.81119E-7
W_{sys}	2.97304E-8

Table 6.4 FAULTREE+ Quantification of Dresden-3 Fault Tree

Hence, for this example the BDD method is significantly faster than conventional quantification techniques. Also, along with great savings in computation time the BDD technique gives exact probability values for Q_{sys} and w_{sys}, whereas the FAULTREE+ method results in a loss in accuracy of 2.21% and 3.08% respectively for these parameters.

6.6 Accuracy - Comparison with a Conventional Approach

To compare the accuracy of the BDD technique with the conventional Kinetic Tree Theory approach of FAULTREE+, 10 example fault trees were analysed (57), the results of which are reproduced here in table 6.5. Some of these benchmark fault trees are taken from industry (Railway Industry) and the others are produced as simple structures to test different aspects of the analysis code.

Tree	No. of Gates	No. of Basic Events	No. of Minimal Cut Sets	BDD Q_{sys}	FAULT-REE+ Q_{sys}	BDD w_{sys}	FAULT-REE+ w_{sys}
1	17	11	43	2.08587×10^{-2}	2.09883×10^{-2}	7.52376×10^{-6}	8.03221×10^{-6}
2	63	32	8,716	4.27226×10^{-7}	4.27248×10^{-7}	2.77792×10^{-4}	2.77835×10^{-4}
3	21	40	416	1.31777×10^{-6}	1.31778×10^{-6}	8.99077×10^{-4}	8.99100×10^{-4}
4	10	10	13	6.80022×10^{-2}	7.0067×10^{-2}	1.97366×10^{-4}	2.11151×10^{-4}
5	4	6	3	3.39397×10^{-8}	3.4×10^{-8}	2.7419×10^{-10}	2.7474×10^{-10}
6	4	6	6	7.06927×10^{-5}	7.10911×10^{-5}	3.09498×10^{-6}	3.12839×10^{-6}
7	3	4	2	3.07760×10^{-4}	3.08×10^{-4}	6.07692×10^{-7}	6.08360×10^{-7}
8	10	8	10	1.23233×10^{-5}	1.23710×10^{-5}	2.27332×10^{-7}	2.28347×10^{-7}
9	3	4	2	2.02398×10^{-6}	2.024×10^{-6}	1.1392×10^{-11}	1.1392×10^{-11}
10	30	60	7,056	4.37185×10^{-7}	4.38002×10^{-7}	3.09211×10^{-4}	3.15744×10^{-4}

Table 6.5 Quantification Results of 10 Example Fault Trees

It is evident from the results in table 6.5 that the FAULTREE+ approach results in an over estimate for both Q_{sys} and for w_{sys}. This is an average over estimate of 0.5% and 1.7% respectively.

6.7 Conclusion

This chapter develops algorithms which extend the use of the BDD method to calculate top event parameters, such as system failure probability, failure intensity and expected number of top event occurrences. The added advantage of obtaining these parameters directly from the BDD, when compared to the traditional Kinetic Tree Theory approach (1), is that the resulting values are exact. Approximations used in conventional fault tree analysis are shown to be inadequate for some fault trees. Further, the BDD method has proven to be extremely efficient as a means of quantification. Only one pass of the BDD structure is required to calculate all parameters.

An additional feature of the BDD method is that it does not require the minimal cut sets to be determined prior to the quantification. This reduces computation time and memory requirements that would otherwise be needed.

CHAPTER 7

IMPORTANCE MEASURES

7.1 Introduction

Component importance measures where introduced in section 3.5 along with the equations used for their calculation. The importance of a component indicates its relative contribution to the top event probability, $Q_{sys}(t)$ or to the expected number of top event occurrences, $W(0,t)$ depending on the type of analysis being performed. Applying the kinetic tree theory to quantify the fault tree top event to obtain $Q_{sys}(t)$ or $W(0,t)$, requires the use of approximations. Since these quantities are needed to calculate the importance measures, as a result these too will be approximations.

This chapter describes the use of the binary decision diagram to calculate the importance measures for each basic event. The method presented overcomes the need to use approximations. Additionally the computer implementation of this method is discussed.

7.2 Importance Measures Concerned with Top Event Probability

7.2.1 Birnbaum Measure of Component Importance

As shown in section 3.5 Birnbaums measure of importance (3), also known as the criticality function, $G_i(\mathbf{q})$ is given by:

$$G_i(\mathbf{q}) = Q(1_i, \mathbf{q}) - Q(0_i, \mathbf{q}) \tag{7.1}$$

The algorithms which allow the calculation of the criticality function, i.e. *Probpost* and *Probprev* are discussed in detail in section 6.4. These algorithms are needed for the computation of the system unconditional failure intensity, $w_{sys}(t)$. Thus this importance measure is already calculated as a result of evaluating $w_{sys}(t)$ and only the initial pass of the BDD structure is required.

7.2.2 Criticality Measure of Component Importance

The criticality measure of importance has been previously defined in Chapter 3 as:

$$I_{C_i} = \frac{G_i(\mathbf{q})q_i(t)}{Q_{sys}(t)} \tag{7.2}$$

Therefore the calculation of I_{C_i}, using the BDD, is straightforward as all the parameters in equation (7.2) are known from the system evaluation.

7.2.3 Fussell-Vesely Measure of Component Importance

This measure of importance is usually close in numerical value to the criticality measure, therefore its calculation is not essential if the criticality measure is available. However it is worth mentioning the difficulties encountered when trying to employ the BDD structure to calculate this importance measure.

The Fussell-Vesely importance measure for component i, (I_{FV_i}) is calculated as the probability of the union of the minimal cut sets which contain event i divided by the top event occurrence probability, i.e.

$$I_{FV_i} = \frac{P(\bigcup_{k/i \in C_k} C_k)}{Q_{sys}(t)} \tag{7.3}$$

Therefore I_{FV_i} requires the use of the minimal BDD to trace the minimal cut sets that contain i. However the minimal BDD cannot be used to evaluate the probability of each minimal cut set. Since in forming the minimal BDD the original structure function of the fault tree has been altered by the 'without' procedure. The construction of the BDD using the structure function and the 'without' procedure has been discussed in-depth in Chapter 4, section 4.3 and 4.6 respectively. The altered structure function represented by the minimal BDD is no longer in the form of Shannon's decomposition. As a result the sum of the probabilities of the disjoint paths through the diagram can not be used to form a probability relationship.

To illustrate this 'altering' of the structure function refer to the fault tree shown in figure 7.1, whose unminimised BDD can be seen in figure 7.2 The program BADD

discussed in Chapter 5 was used for the construction of the BDD. The following top-down, left-right ordering was given to the basic events:

a<e<b<c<h<d<g<f.

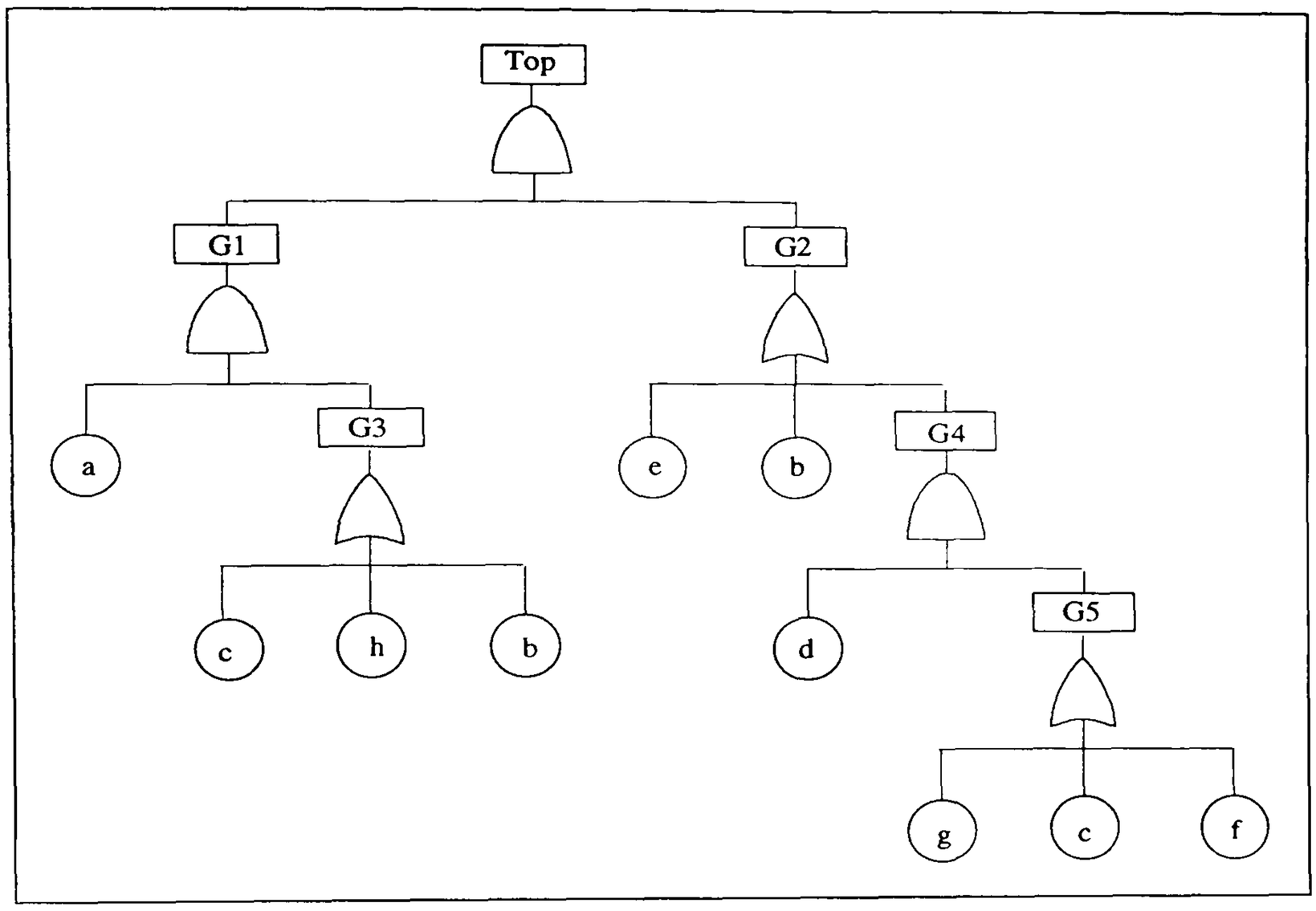


Figure 7.1 Example Fault Tree

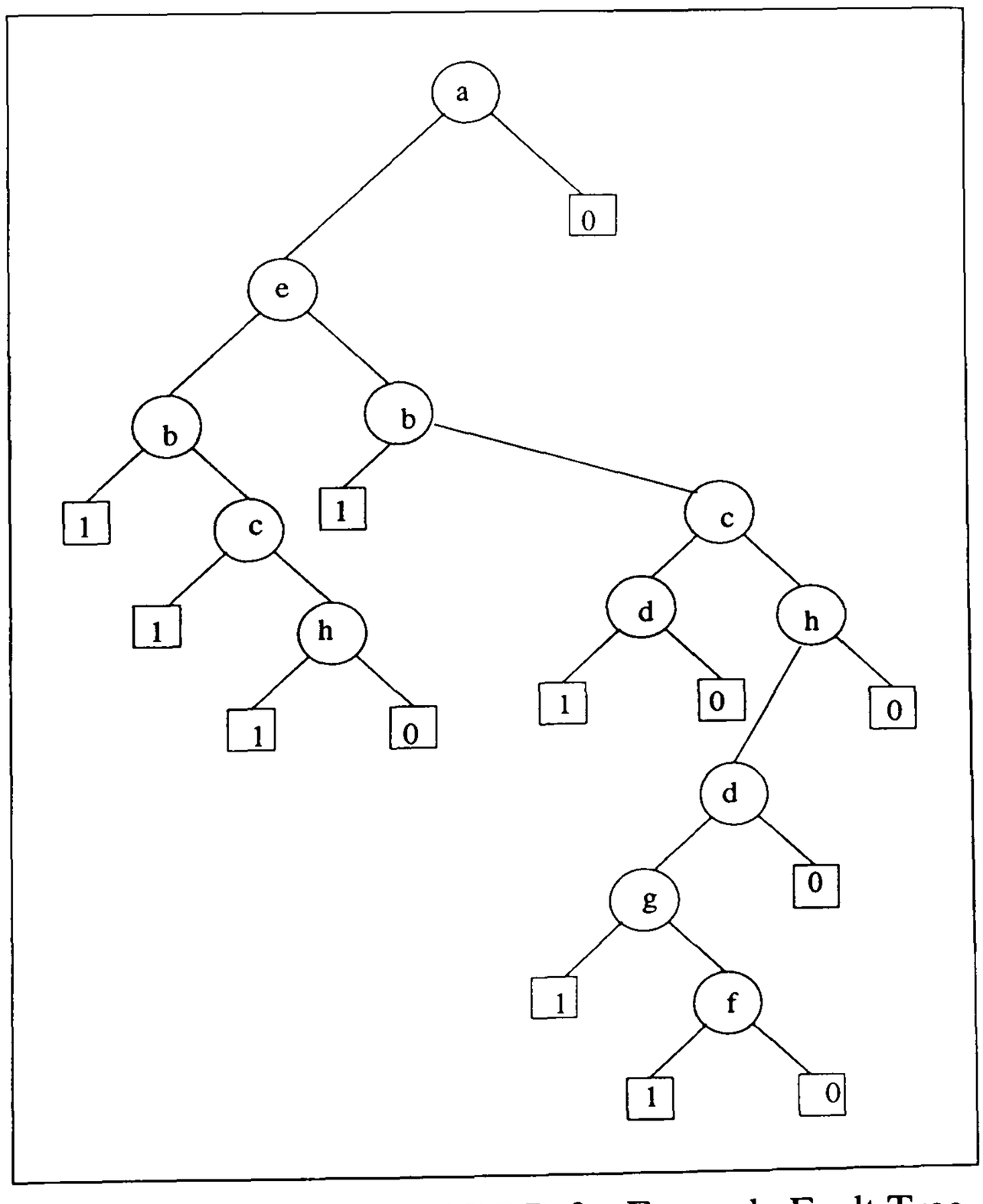


Figure 7.2 Unminimised BDD for Example Fault Tree

The BDD in figure 7.2 could also be constructed using the structure function for the fault tree and applying Shannon's decomposition, pivoting about the variables in the same order.

The minimal cut sets obtained using the program BADD for the fault tree shown in figure 7.1 are:

(1) {a, e, c}
(2) {a, e, h}
(3) {a, b}
(4) {a, c, d}
(5) {a, h, d, g}
(6) {a, h, d, f}

When implementing the BDD method for fault tree analysis it is known that the sum of the probabilities of the disjoint paths through the BDD equals the probability of the union of the minimal cut sets (51), i.e.

$$\sum_{i=1}^{n} P(d_i) = P(\bigcup_{i=1}^{nc} C_i) \tag{7.4}$$

where:

$P(d_i)$ - is the probability of a disjoint path, d_i in the BDD.
C_i - is a minimal cut set of the fault tree.
n - total number of disjoint paths in the BDD.
nc - total number of minimal cut sets.

Let us now investigate basic event e in the fault tree and the task of obtaining the probability of the union of the minimal cut sets containing e from the BDD.

It can be shown that:

$$\sum_{k/e \in d_k} P(d_k) \neq P(\bigcup_{m/e \in C_m} C_m) \tag{7.5}$$

i.e., the sum of the probabilities of disjoint paths containing e in the BDD **does not** equal the probability of the union of the minimal cut sets containing e.

To illustrate, the left-hand-side (L.H.S) of equation (7.5) is developed by the use of the BDD in figure 7.2, the sum of the probabilities of the disjoint paths containing e are:

$$\sum_{k/e\in d_k} P(d_k) = P(a.e.b) + P(a.e.\bar{b}.c) + P(a.e.\bar{b}.\bar{c}.h)$$

$$= P(a).P(e).P(b) + P(a).P(e).(1-P(b)).P(c) + P(a).P(e).(1-P(b)).(1-P(c)).P(h)$$

$$\sum_{k/e\in d_k} P(d_k) = P(a).P(e)[P(b)+P(c)-P(b).P(c) + P(h)-P(c).P(h)-P(b).P(h)+P(b).P(c).P(h)]$$

Next developing the right-hand-side (R.H.S) of equation (7.5), the probability of the union of the minimal cut sets containing e:

$$P(\bigcup_{m/e\in C_m} C_m) = P(a.e.c \cup a.e.h)$$

$$= P(a.e.c) + P(a.e.h) - P(a.e.c.h)$$

$$P(\bigcup_{m/e\in C_m} C_m) = P(a).P(e)[P(c)+P(h)-P(c).P(h)]$$

Thus it is clear that equality does not hold for equation (7.5). The reason for this becomes clear when the BDD is constructed using the structure function, $\phi(\mathbf{x})$ from the minimal cut sets containing e. The structure function for the minimal cut sets {a, e, c} and {a, e, h} is $\phi(\mathbf{x}) = 1-(1-a.e.c)(1-a.e.h)$ and the BDD constructed using this structure function can be seen in figure 7.3. The disjoint paths in this BDD do not correspond to the disjoint paths in the full BDD shown in figure 7.2. Thus the probability of the disjoint paths in the full BDD cannot be used to obtain $P(\bigcup_{m/e\in C_m} C_m)$ (the probability of the union of the minimal cut sets containing e).

There are two possible ways to overcome the problem of computing the numerator in equation (7.3):

(1) Use the upper bound estimation for $P(\bigcup_{k/i\in C_k} C_k)$ as $\sum_{\substack{k=1 \\ i\in C_k}}^{n} P(C_k)$, i.e. the sum of the minimal cut sets containing i.

(2) If an **exact** calculation of I_{FV_i} is desired; then for each basic event i (which is contained in minimal cut sets $C_1, C_2, \ldots\ldots, C_j$) a fault tree could be drawn in a form whose top gate is an OR gate which has the AND gate inputs $G_1, G_2, \ldots\ldots G_j$ and

each of these AND gates have the basic event inputs that constitute minimal cut sets $C_1, C_2, \ldots\ldots, C_j$ respectively. In this way the sum of the probabilities of the disjoint paths of the resulting BDD for this fault tree will equal the probability of the union of the minimal cut sets containing component i, i.e. $\sum_{k/i \in d_k} P(d_k) = P(\bigcup_{m/i \in C_m} C_m)$ and equation (7.3) can be solved.

To illustrate, basic event e above is contained in minimal cut sets {a, e, c} and {a, e, h}. The fault tree whose BDD will give the probability of the union of these minimal cut sets is shown in figure 7.4.

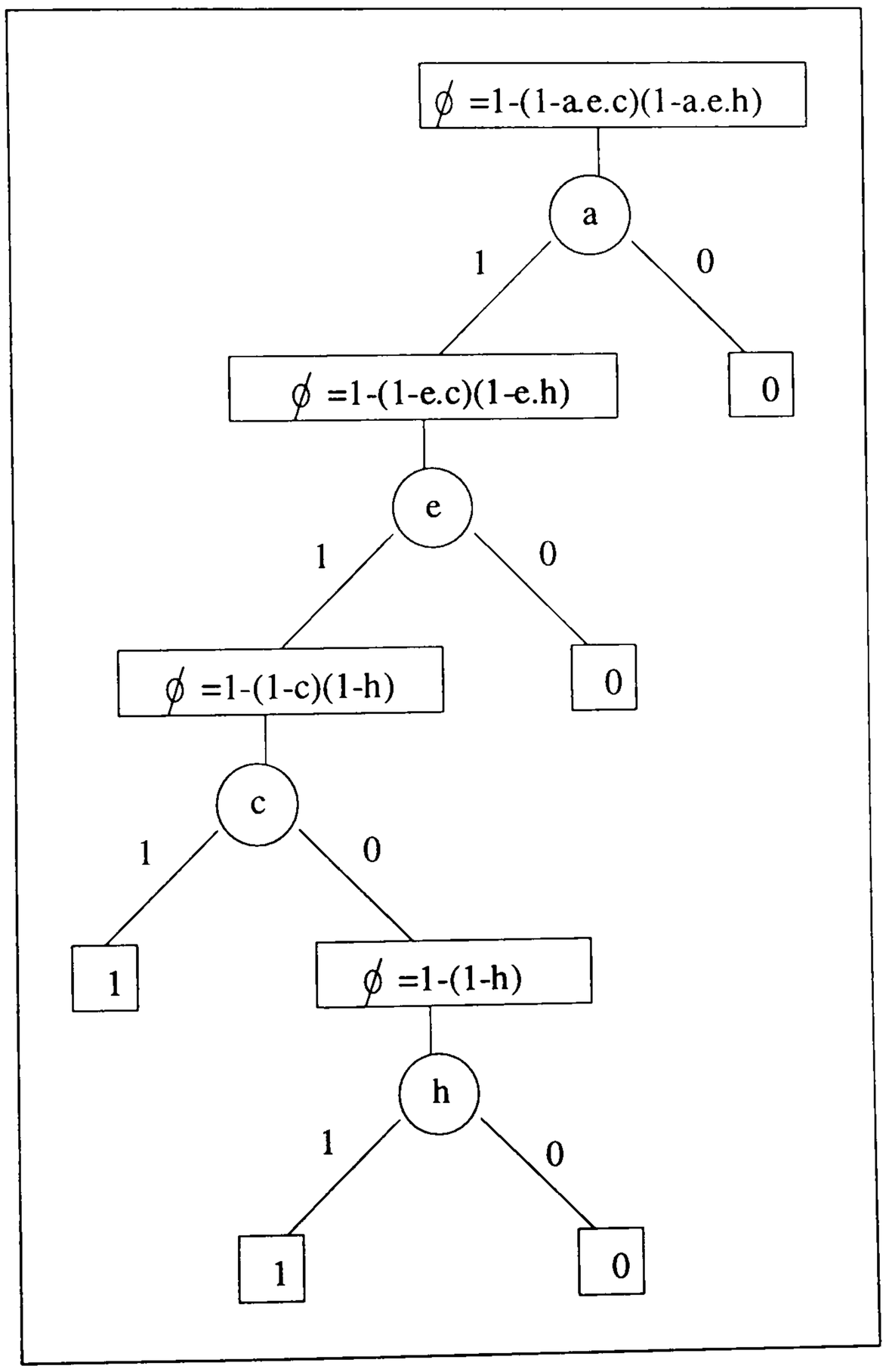


Figure 7.3 BDD Constructed from Structure Function $\phi(\mathbf{x}) = 1-(1-a.e.c)(1-a.e.h)$

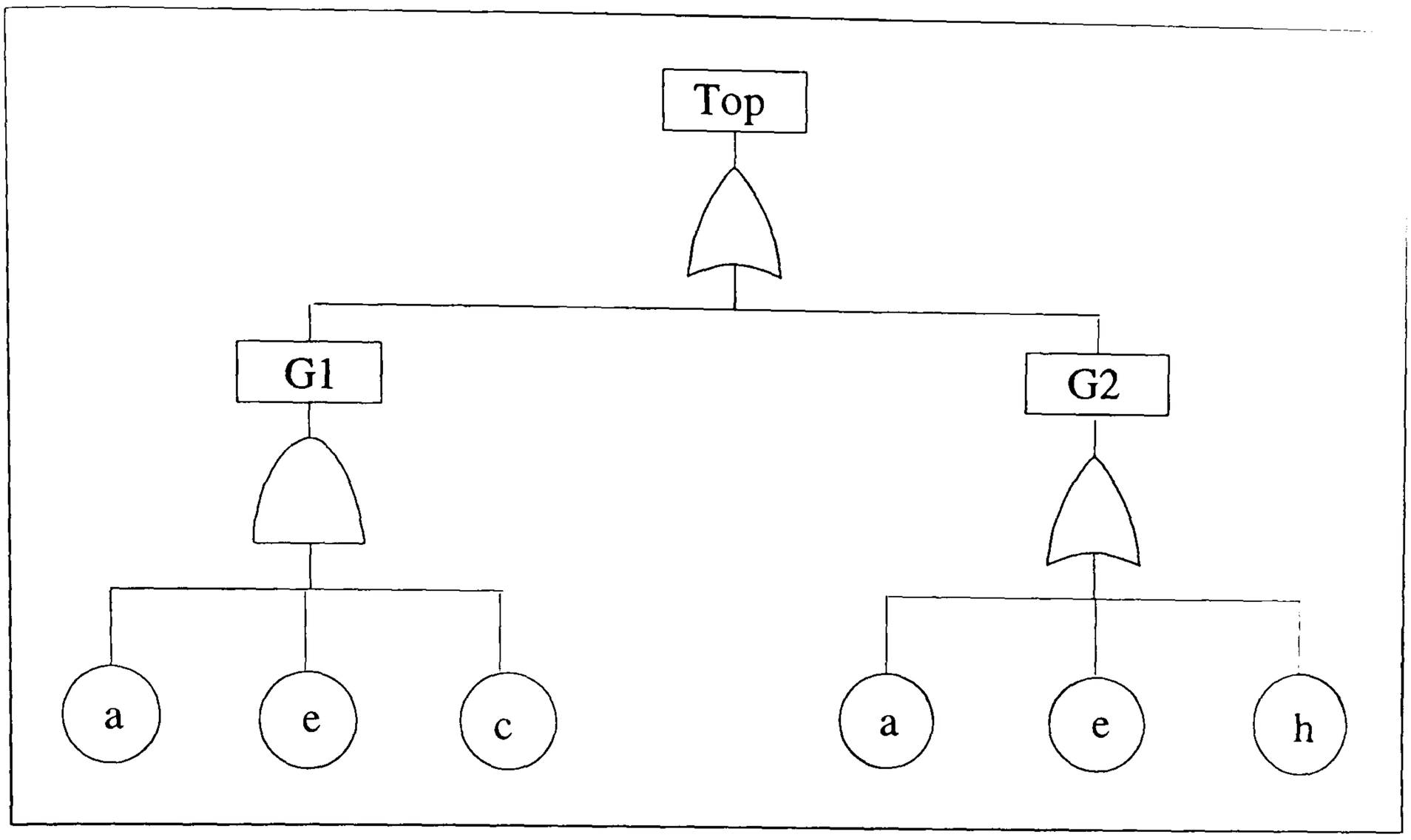


Figure 7.4 Fault Tree for Minimal Cut Sets {a, e, c} and {a, e, h}

The BDD constructed for the fault tree in figure 7.4 with the ordering of basic events a<e<c<h is identical to the BDD in figure 7.3 which was constructed using the structure function for the minimal cut sets containing e.

The disjoint paths through the BDD in figure 7.3 are:

(1) a.e.c

(2) a.e.$\overline{c}$.h

Therefore the sum of the probabilities of the disjoint paths will be P(a).P(e).P(c)+P(a).P(e).P(h)-P(a).P(e).P(c).P(h) which equals the probability of the union of the minimal cut sets containing e.

Obviously if a fault tree has many basic events and many minimal cut sets, this method would prove inefficient both in terms of speed and efficiency of calculation, therefore the approximation in step (1) may be desirable.

If a variable has just one occurrence then calculating I_{FV_i} is not a problem, for these basic events the following equation can be used.

$$I_{FV_i} = \frac{P(C_i)}{Q_{sys}(t)} \tag{7.6}$$

7.2.4 Fussell-Vesely Measure of Minimal Cut Set Importance

The previously defined importance measures ranked component failures in the order of their contribution to the top event. This measure provides a similar function except that the minimal cut sets are themselves ranked. The importance measure is defined simply as the probability of occurrence of cut set j given that the system has failed:

$$I_j = \frac{P(C_j)}{Q_{sys}(t)} \tag{7.7}$$

7.3 Initiators and Enablers

The treatment given in Chapter 6 for the system unconditional failure intensity assumes that the order in which the component failure events occur in any minimal cut set is unimportant. In some analyses the order of basic event failures is vital to the occurrence of the fault tree top event. This is particularly true when the analysis of a safety protection system is being carried out. For example, if a hazardous event occurs and the protection systems have already failed the outcome will be a dangerous system failure. However, if failures occur in a sequence where the hazardous event occurs prior to the protection systems failing then a safe shutdown will have resulted. This type of situation can be modelled by considering the failures as either **initiating** or **enabling** events (6, 8).

In the diagram shown in figure 7.5 the order of events are considered. The safety systems are shown to be inactive between times t_0 and t_1 (safety system fails at t_0 and is repaired at t_1). During this time period the safety systems are unable to respond to a hazardous condition (the initiating event) and the system is in a critical state due to the occurrence of enabling events. If the initiating event occurs between t_0 and t_1 the hazardous system failure will happen. However, if the initiating event occurs prior to t_0 or after t_1 the safety systems respond as designed. Therefore the order of component failures needs to be considered for an accurate system assessment. Initiating and enabling events are formally defined as follows:

Initiating events perturb system variables and place a demand on control/protection systems to respond.

Enabling events are inactive control/protection systems which permit initiating events to cause the top event.

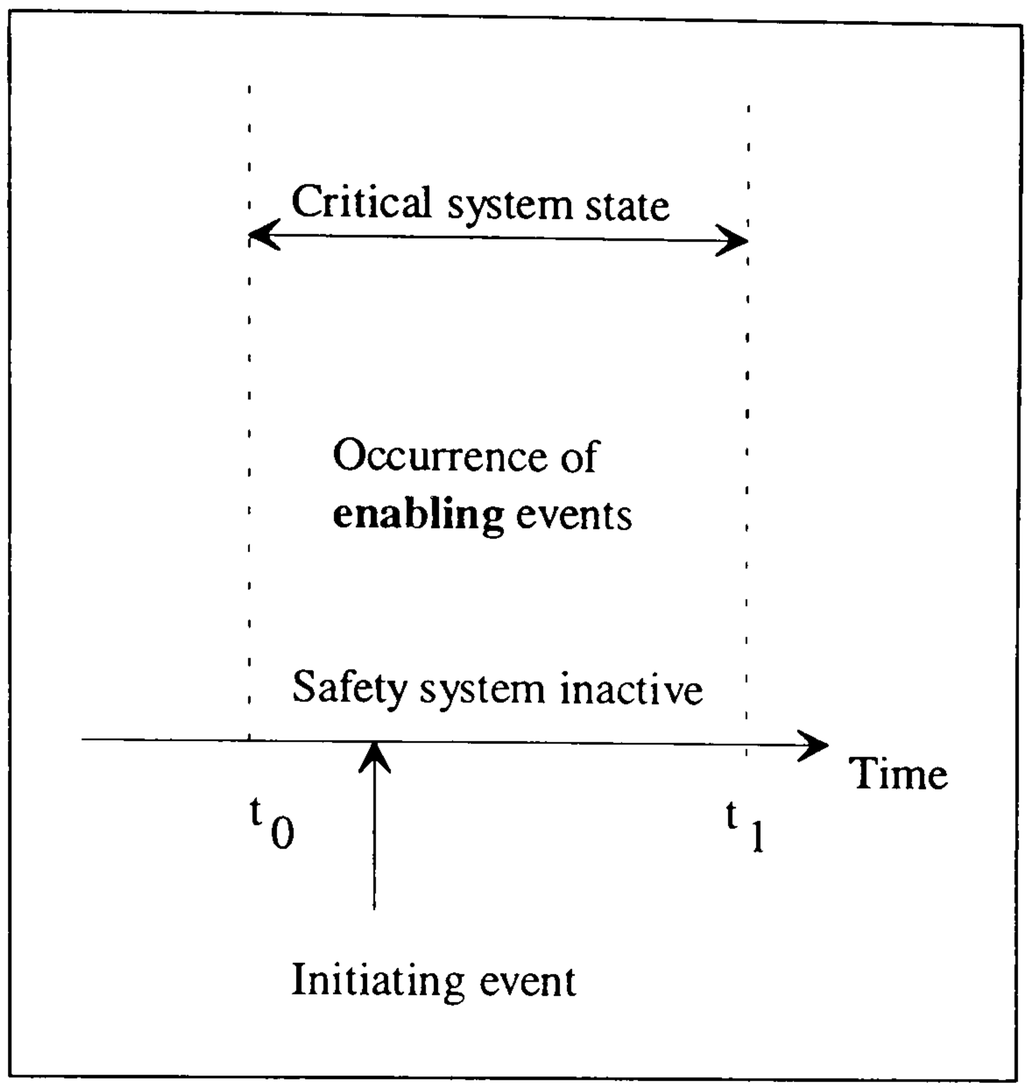


Figure 7.5 Initiating Event Window

The system unconditional failure intensity, $w_{sys}(t)$ is evaluated for interval reliability by restricting the summation in equation (6.11) to be over the initiating events only i.e.:

$$w_{sys}(t) = \sum_{initiators} G_i(\mathbf{q}) w_i(t) \tag{7.8}$$

7.4 Importance Measures Concerned with Top Event Reliability

It has previously been discussed in Chapter 6, section 6.4 that an upper bound for the unreliability of a system is the expected number of top event occurrences W(0,t). To investigate the contribution of component failures to this quantity it is appropriate to use the following importance measures.

7.4.1 Barlow-Proschan Measure of Initiator Importance

The Barlow-Proschan Initiator importance, BPI_i is the conditional probability that initiating event i caused the failure, given that the system fails prior to time t.

$$BPI_i = \frac{\int_0^t G_i(\mathbf{q}) w_i(t) dt}{W(0,t)} \tag{7.9}$$

Therefore the Barlow-Proschan measure of initiator importance requires the criticality function, component unconditional failure intensity and the expected number of top event occurrences for its evaluation. Since all elements in equation (7.9) can be calculated using the BDD, determining this importance measure for each initiator is again easily performed.

7.4.2 Barlow-Proschan Measure of Enabler Importance

This importance measure (BPE_i) assesses the contribution of an enabling component i when initiating event j causes the system failure. The failure of enabler i is only then a factor when enabler i and initiator j both occur in the same minimal cut set (C).

$$BPE_i = \frac{\sum_{\substack{j \\ i \neq j \\ i,j \in C}} \int_0^t \{Q(1_i,1_j,\mathbf{q}) - Q(1_i,0_j,\mathbf{q})\} q_i(t) w_j(t) dt}{W(0,t)} \tag{7.10}$$

The calculation of this importance measure using a BDD is more involved than the evaluation of other measures. Essentially its difficulty is due to the evaluation of the term:

$$Q(1_i,1_j,\mathbf{q}) - Q(1_i,0_j,\mathbf{q}) \tag{7.11}$$

from the BDD for each combination of events i and j (where i is the enabler and j the initiator) which occur in the same minimal cut set.

In evaluating this term the minimal BDD must first be scanned for the enabler event i and a list formed of every other event which appears in a minimal cut set along with i. Any events which can only act as enablers and not initiators are then removed from this list to leave only the initiating events j.

The contribution to BPE_i from each initiating event j is then determined and summed to obtain the numerator in equation (7.10). The difficulty comes in evaluating equation (7.11) from the BDD and two cases must be considered for the ordering of initiating event j and enabling event i in the BDD structure.

7.4.2.1 Case A: i<j

In this situation the nodes for the enabling event appear at higher levels than the initiating event nodes in the BDD. $Q_{sys}(t)$ is obtained as described in Chapter 6 by the summation of the probability of each path leading to a terminal 1 vertex on the BDD. These paths can be placed into one of four categories with respect to the initiator and enabler events.

i) paths go through a node x_i for which at least one node x_j appears directly below it at some point in the BDD.
ii) paths go through a node x_i for which x_j does not appear directly below it.
iii) paths go through a node x_j for which x_i does not appear directly above it.
iv) paths go through neither a x_i node nor a x_j node.

To evaluate equation (7.11) in both terms $q_i(t)$ is set to one and for the first term $q_j(t)$ is set to one and in the second term $q_j(t)$ is set to zero. The paths in categories (ii) and (iv) will make no contribution to the difference between these two terms.

Consider the contribution of the sections of the BDD which are in categories (i) and (iii). First of all category (i) (see figure 7.6).

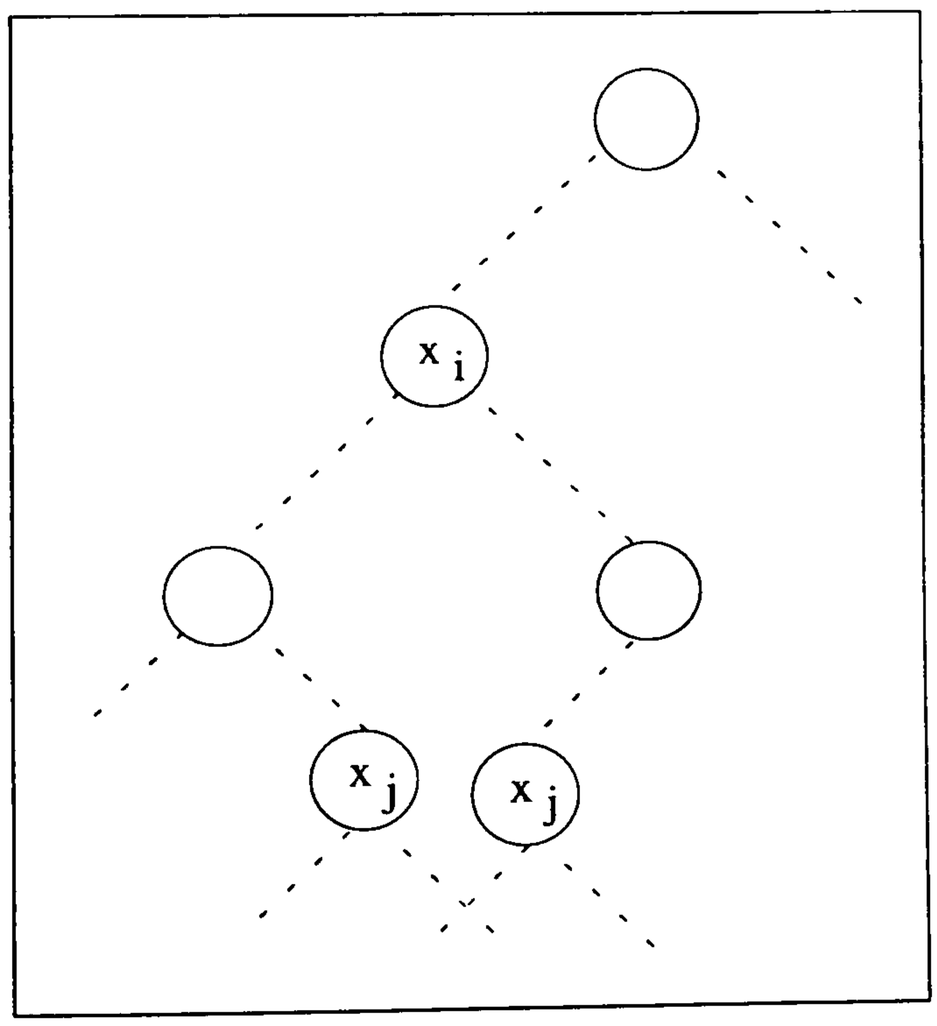


Figure 7.6 Section of BDD with i<j

$Q_{sys}(t)$ is the probability of system failure calculated with $P(x_i) = q_i(t)$ and $P(x_j) = q_j(t)$. If we first subtract the contribution from $Q_{sys}(t)$ of each section in category (i) (a section being all the paths that go through the x_i node and at least one x_j node down to a 1 end vertex). This can then be replaced with the contribution that

this section will make with $q_i(t)$=1. This will give $Q(1_i, \mathbf{q})$. If we then remove from this the probability of each path through x_i on the 1 branch that passes through x_j with $p(x_j) = q_j(t)$ and replace it with the contribution given when $q_j(t) = 1$ we will have $Q(1_i, 1_j, \mathbf{q})$ for this one section in category (i). This process can then be repeated for each such category (i) section. This gives:

$$Q(1_i, 1_j, \mathbf{q}) = Q_{sys}(t) - \sum_{\substack{\text{all } x_i \\ \text{for category (i)}}} \{pr(x_i)[q_{x_i} po^1(x_i) + (1 - q_{x_i}) po^o(x_i)]$$

$$+ pr(x_i) po^1(x_i)$$

$$- \sum_{\substack{\text{all } x_j \text{ below} \\ \text{each } x_i \text{ on a} \\ \text{1 branch}}} \{\frac{pr(x_j)}{q_{x_i}}[q_{x_j} po^1(x_j) + (1 - q_{x_j}) po^0(x_j)] \qquad (7.12)$$

$$+ \frac{pr(x_j)}{q_{x_i}} po^1(x_j)\}\}$$

where:

$pr(x_i)$=*Probprev* of x_i, $po^1(x_i)$=*Probpost* 1 branch of x_i and $po^0(x_i)$=*Probpost* 0 branch of x_i. The algorithms for *Probprev* and *Probpost* have already been given in Chapter 6. A similar approach to the second term gives:

$$Q(1_i, 0_j, \mathbf{q}) = Q_{sys}(t) - \sum_{\substack{\text{all } x_i \\ \text{for category (i)}}} \{pr(x_i)[q_{x_i} po^1(x_i) + (1 - q_{x_i}) po^o(x_i)]$$

$$+ pr(x_i) po^1(x_i)$$

$$- \sum_{\substack{\text{all } x_j \text{ below} \\ \text{each } x_i \text{ on a} \\ \text{1 branch}}} \{\frac{pr(x_j)}{q_{x_i}}[q_{x_j} po^1(x_j) + (1 - q_{x_j}) po^0(x_j)] \qquad (7.13)$$

$$+ \frac{pr(x_j)}{q_{x_i}} po^0(x_j)\}\}$$

Taking the difference between equations (7.12) and (7.13) gives us the contribution of each term in category (i).

$$[Q(1_i,1_j,\mathbf{q})-Q(1_i,0_j,\mathbf{q})]_{cat(i)}=\sum_{\substack{\text{each } x_j\\ \text{below each } x_i\\ \text{on a 1 branch}}}\frac{pr(x_j)}{q_{x_i}}[po^1(x_j)-po^0(x_j)] \qquad (7.14)$$

Considering the nodes in category (iii). Since they do not go through x_i we get:

$$[Q(1_i,1_j,\mathbf{q})-Q(1_i,0_j,\mathbf{q})]_{cat(iii)}=Q(1_j,\mathbf{q})-Q(0_j,\mathbf{q})$$
$$=\sum_{\substack{\text{each } x_j\\ \text{in cat (iii)}}}pr(x_j)[po^1(x_j)-po^0(x_j)] \qquad (7.15)$$

and equation (7.11) can be formed by summing equations (7.14) and (7.15).

7.4.2.2 Case B: j<i

When the initiating event j appears at a higher level in the BDD than enabling event i, it becomes a little more complex to evaluate the contributions to equation (7.11) of the four categories for the paths which lead to a terminal 1 node:

i) paths go through a node x_j for which node x_i appears directly below it at some point in the BDD.
ii) paths go through a node x_j for which x_i does not appear directly below it.
iii) paths go through a node x_i for which x_j does not appear directly above it.
iv) paths go through neither a x_i node nor a x_j node.

This time it is only the sections of the BDD identified for categories (i) and (ii) which contribute to $Q(1_i,1_j,\mathbf{q})-Q(1_i,0_j,\mathbf{q})$.

Considering first the BDD sections in category (i). To evaluate each of the two terms, the approach is similar to that for Case A. Initially the contribution to $Q_{sys}(t)$ of the paths in category (i) are removed. Setting $q_j(t)=1$ the contribution of this section is then added back to give $Q(1_j,\mathbf{q})$ for this section. This means the section which is added back from the BDD is only that which connects to the 1 branch of the x_j node. Tracing the paths on the BDD from the 1 branch of x_j to an x_i node are then removed which extracts the contribution to $Q_{sys}(t)$ with $P(x_j)=1$ and $P(x_i)=q_i(t)$. These paths are then replaced with the contribution made by values $P(x_j)=P(x_i)=1$. This gives the first term in equation (7.11).

Replacing the relevant terms with $P(x_j)=0$ gives the second term. So:

$$Q(1_i, 1_j, \mathbf{q}) = Q_{sys}(t) - \sum_{\substack{\text{each } x_j \\ \text{for category (i)}}} \{pr(x_j)[q_{x_j} po^1(x_j) + (1 - q_{x_j}) po^o(x_j)]$$

$$+ pr(x_j) po^1(x_j)$$

$$- \sum_{\substack{\text{each } x_i \text{ below} \\ \text{each } x_j \text{ on a} \\ 1 \text{ branch}}} \{\frac{pr(x_i)}{q_{x_j}}[q_{x_i} po^1(x_i) + (1 - q_{x_i}) po^0(x_i)] \qquad (7.16)$$

$$+ \frac{pr(x_i)}{q_{x_j}} po^1(x_i)\}\}$$

and

$$Q(1_i, 0_j, \mathbf{q}) = Q_{sys}(t) - \sum_{\substack{\text{each } x_j \\ \text{for category (i)}}} \{pr(x_j)[q_{x_j} po^1(x_j) + (1 - q_{x_j}) po^o(x_j)]$$

$$+ pr(x_j) po^0(x_j)$$

$$- \sum_{\substack{\text{each } x_i \text{ below} \\ \text{each } x_j \text{ on a} \\ 0 \text{ branch}}} \{\frac{pr(x_i)}{(1 - q_{x_j})}[q_{x_i} po^1(x_i) + (1 - q_{x_i}) po^0(x_i)] \qquad (7.17)$$

$$+ \frac{pr(x_i)}{(1 - q_{x_j})} po^1(x_i)\}\}$$

Taking the difference between equations (7.16) and (7.17) gives:

$$Q(1_i, 1_j, \mathbf{q}) - Q(1_i, 0_j, \mathbf{q}) = \sum_{\substack{\text{each node } x_j \\ \text{in category (i)}}} \{pr(x_j)[po^1(x_j) - po^0(x_j)]$$

$$+ \sum_{\substack{\text{each } x_i \text{ below} \\ x_j \text{ on a 1 branch}}} \{\frac{pr(x_i)(1 - q_{x_i})}{q_{x_j}}[po^1(x_i) - po^0(x_i)] \qquad (7.18)$$

$$- \sum_{\substack{\text{each } x_i \text{ below} \\ x_j \text{ on a } 0 \text{ branch}}} \{\frac{pr(x_i)(1 - q_{x_i})}{(1 - q_{x_j})}[po^1(x_i) - po^0(x_i)]\}\}$$

Thus the contribution provided to equation (7.11) for the nodes on the BDD in category (i) can be established.

For those nodes x_j which are in category (ii) and do not occur with any x_i nodes on a path to a terminal 1 node the contribution to equation (7.11) is given by:

$$Q(1_i,1_j,\mathbf{q}) - Q(1_i,0_j,\mathbf{q}) = \sum_{\substack{\text{all } x_j \\ \text{category (ii)}}} pr(x_j)[po^1(x_j) - po^0(x_j)] \qquad (7.19)$$

The equations presented for the Barlow-Proschan enabler importance are based on the assumption that the BDD form is such that every node has only one branch leading down to it (other than the root vertex). BDD's with repeated structures have been encoded to give the most efficient storage representation in which case any node can be linked to more than one other higher node in the BDD structure. The repeated sections would need to be extracted and explicitly represented in the BDD structure when evaluating the Barlow-Proschan enabler importance measure. Otherwise the values stored for $pr(x_i)$ may not be the ones required for the algorithm.

To illustrate this problem consider the BDD shown in figure 7.7 which has the repeated sub-node F3. The values of *Probpost* and *Probprev* for each node is given in table 7.1, where q_{xi} is the unavailability of component xi.

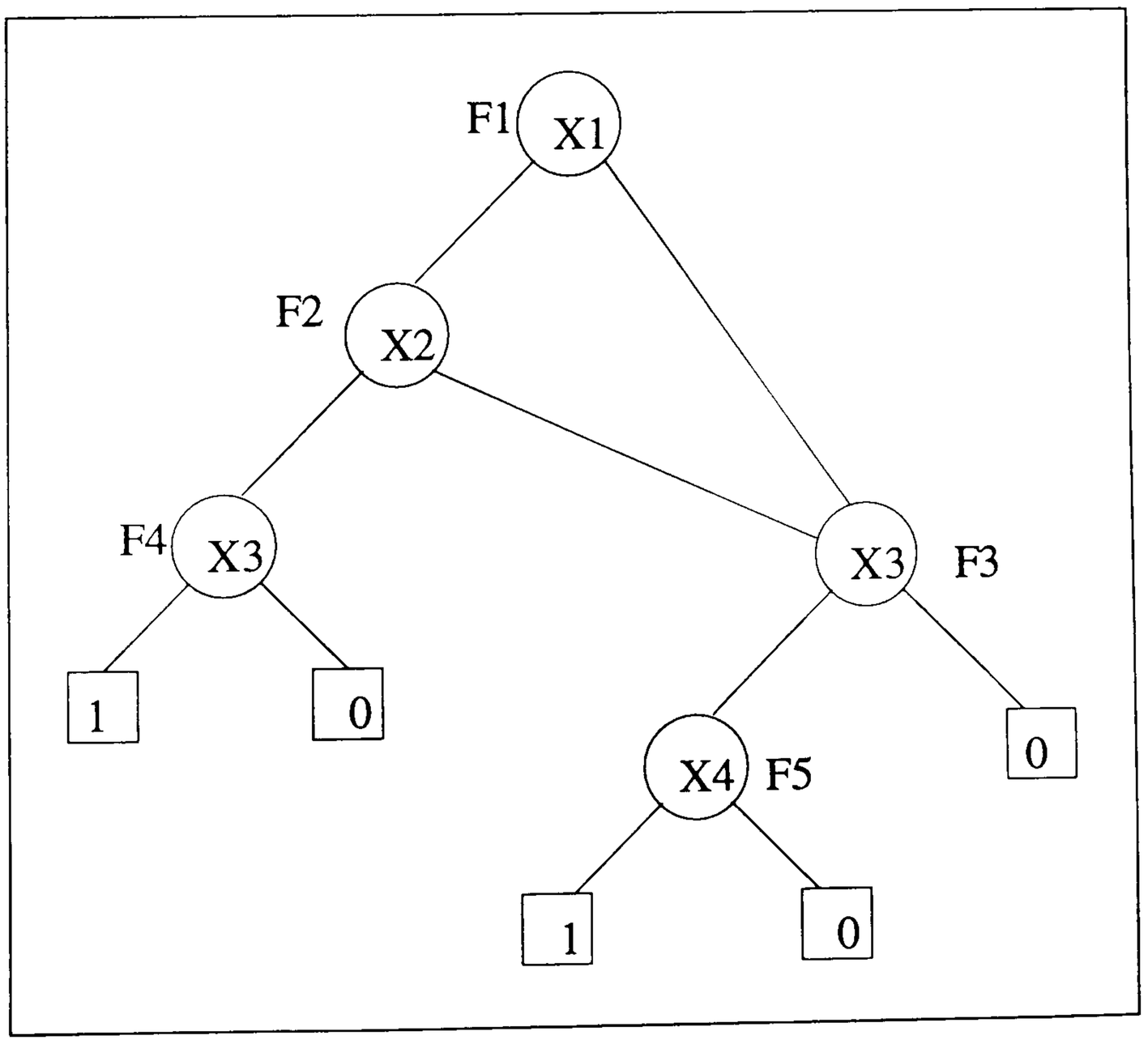


Figure 7.7 BDD with Repeated Sub-node F3

Node	Basic Event of Node	$po^1(x_i)$	$po^0(x_i)$	$pr(x_i)$
F1	X1	$q_{X2}.q_{X3}+$ $(1-q_{X2}).q_{X3}.q_{X4}$	$q_{X3}.q_{X4}$	1
F2	X2	q_{X3}	$q_{X3}.q_{X4}$	$1-q_{X1}$
F3	X3	q_{X4}	0	$(1-q_{X1})+$ $q_{X1}.(1-q_{X2})$
F4	X3	1	0	$q_{X1}.q_{X2}$
F5	X4	1	0	$q_{X3}[(1-q_{X1})$ $+q_{X1}.(1-q_{X2})]$

Table 7.1 *Probpost* and *Probprev* Values for Nodes in the BDD

Let X2, X3 and X4 be enablers and X1 be an initiator and say we wish to find BPE_{X3}. The paths of the BDD in figure 7.7 which contain both basic events X1 and X3 are:

(1) X1.X2.X3
(2) X1.(1-X2).X3.X4

The X3 in path (1) belongs to the node F4 shown in table 7.1 and the X3 in path (2) belongs to the repeated sub-node F3. The problem of the repeated sub-node occurs when calculating $pr(x_i)$ for the second term of equation (7.18).

From table 7.1 we see that the value of $pr(X3)$ for the node F3 is equal to $(1-q_{X1})+q_{X1}.(1-q_{X2})$. This expression has the extra term $(1-q_{X1})$ which is not required for path (2) and it occurs because of the shared sub-node F3. In the algorithm for *Probprev* the value of *Probprev* is summed for each occurrence of a node, therefore the value of $pr(X3)$ in this case is not the one that is required, as a result of F3 occurring twice. The desired value of $pr(X3)$ for path (2) is $q_{X1}.(1-q_{X2})$.

To enable the correct calculation of $pr(x_i)$ for this example, the BDD in figure 7.7 must be converted to the BDD presented in figure 7.8, where the repeated structure has been explicitly represented using the extra nodes F6 and F7.

When repeated structures occur in the BDD then it may provide a faster solution time to change the probabilities of the initiator and enabler first to $q_j(t)=1$, $q_i(t)=1$ and then to $q_j(t)=0$ and $q_i(t)=1$ (given that i and j are in a minimal cut set together) and re-evaluate the system failure probability for these conditions to give $Q(1_i,1_j,\mathbf{q})$ and $Q(1_i,0_j,\mathbf{q})$.

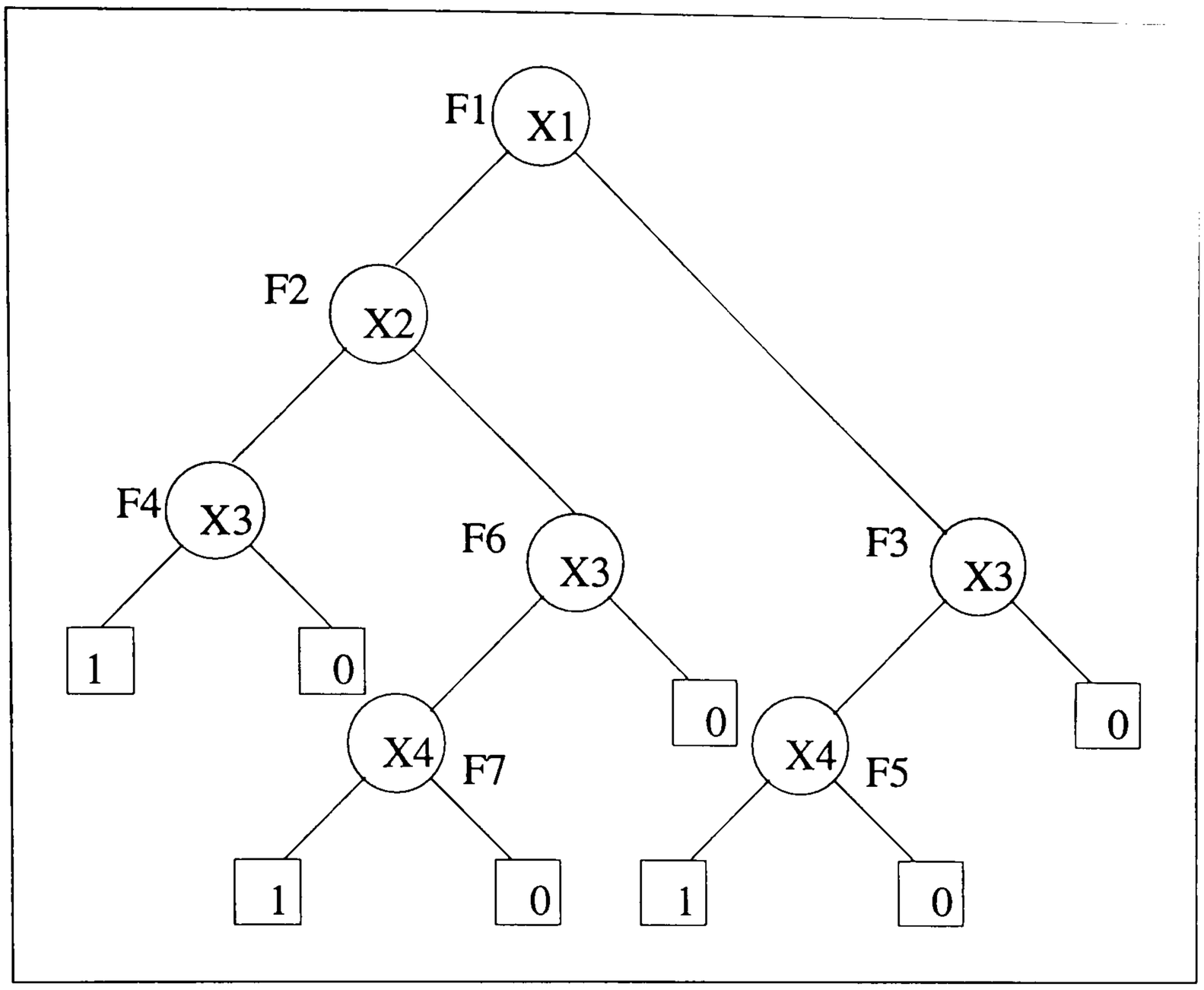


Figure 7.8 Equivalent BDD where Shared Sub-node has been Explicitly Represented

7.5 Computer Implementation of Importance Measures

The computer implementation of the above importance measures has been achieved by the addition of subroutines to the program QUANT (for steady-state analysis) and the resulting program has been called IMPORTANCE. For time dependent analysis the subroutines have been added to the program QUANTIME and the resulting program is named IMPORTIME.

When a steady-state analysis is undertaken equation (7.9) (Barlow-Proschan Initiator Importance) can be replaced by:

$$BPI_i = \frac{G_i(\mathbf{q})w_i}{w_{sys}} \tag{7.20}$$

and equation (7.10) (Barlow-Proschan Enabler Importance) can be replaced by:

$$BPE_i = \frac{\sum_{\substack{j \\ i \neq j \\ i,j \in C}} \{Q(1_i,1_j,\mathbf{q}) - Q(1_i,0_j,\mathbf{q})\}q_i w_j}{w_{sys}} \tag{7.21}$$

If a time dependent analysis is required then the integration in equation (7.9) and (7.10) must be evaluated numerically. The program IMPORTIME uses the trapezium rule for these integrations and the calculation procedure is dealt with in a similar way to that for W(0, t) in section 6.4.

When the importance codes are executed the user is asked which importance measures they require. The Fussell-Vesely measure of component importance is calculated using the upper bound approximation described in section 7.2.3. Additionally the Barlow-Proschan Measure of Enabler importance is calculated by changing the probabilities of the initiator and enabler first to $q_j(t)=1$ and $q_i(t)=1$ and then $q_j(t)=0$ and $q_i(t)=1$ and re-evaluating the system failure probability for these conditions to give $Q(1_i,1_j,\mathbf{q})$ and $Q(1_i,0_j,\mathbf{q})$

7.6 Applications

The program called IMPORTANCE has been utilised to obtain top event quantification and all the importance measures just described for the example fault tree shown in figure 7.9. The steady-state condition is considered for this example fault tree which has two **initiating** events TM and TC and four **enabling** events R, PG, OP, SW.

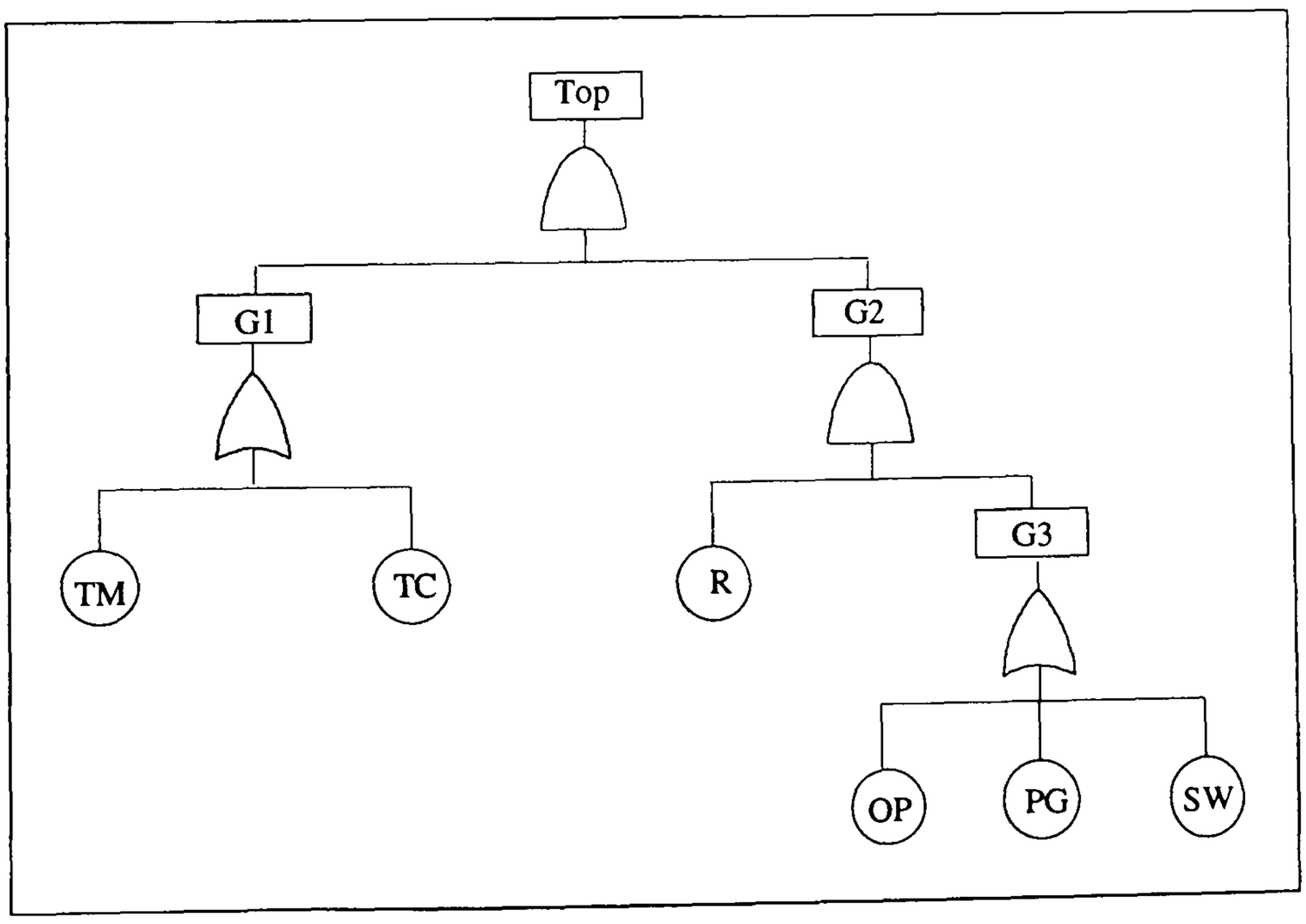


Figure 7.9 Example Fault Tree

The summary information for the fault tree can be seen in table 7.2. The computation time given includes conversion to the BDD and calculating all the importance measures.

Name	Example
No. of Gates	4
No. of Basic Events	6
No. of Minimal Cut Sets	6
Time (s)	1.69
Q_{sys}	7.069E-05
w_{sys}	6.365E-05

Table 7.2 Summary Information for Example Fault Tree

The failure parameters used for the components in the fault tree are shown in table 7.3.

Component	Failure rate per hour λ	Unavailability q_i	Unconditional Failure Intensity w_i
TM	5.0E-05	1.2E-03	4.994E-05
TC	1.0E-04	2.4E-03	9.976E-05
PG	2.0E-04	4.8E-03	1.9904E-04
R	1.0E-04	0.188	8.120E-05
OP	0.1	0.1	0.09
SW	1.0E-05	2.4E-04	9.998E-06

Table 7.3 Component Failure Parameters for Example Fault Tree

Each of the importance measures calculated using the BDD can be seen in table 7.4.

Component	Birnbaums	Criticality	Fussell-Vesely	Barlow-Proschan Initiator	Barlow-Proschan Enabler
TM	1.961E-02	0.333	0.335	1.538E-02	-
TC	1.963E-02	0.666	0.670	3.076E-02	-
PG	6.085E-04	4.132E-02	4.595E-02	-	2.119E-03
R	3.760E-04	1.0	1.0	-	4.615E-02
OP	6.729E-04	0.952	0.957	-	4.414E-02
SW	6.057E-04	2.056E-03	2.298E-03	-	1.059E-04

Table 7.4 Importance Measures for Components in Example Fault Tree

Lastly the minimal cut sets are given in table 7.5 along with their respective Fussell-Vesely minimal cut set importance measure.

Minimal Cut Set	Fussell-Vesely MCS Importance
{TM, R, OP}	0.319
{TM, R, PG}	1.532E-02
{TM, R, SW}	7.659E-04
{TC, R, OP}	0.638
{TC, R, PG}	3.064E-02
{TC, R, SW}	1.532E-03

Table 7.5 Minimal Cut Set Importance Measures

Using the criticality measure of importance we can rank the components in the following order:

Component	Rank
R	1
OP	2
TC	3
TM	4
PG	5
SW	6

Therefore basic event R is the most critical component for the system and SW is the least. The figures above agree with the implementation of these approaches in the commercial package FAULTREE+, when accounting for the approximate calculation procedures of FAULTREE+.

7.7 Conclusion

In addition to the methods available to calculate the top event unavailability and unconditional failure intensity this chapter has shown that the BDD method for fault tree analysis can provide exact calculations for the importance measures of the basic events. Thus a full quantitative analysis of the fault tree can be achieved using the BDD structure.

CHAPTER 8

VARIABLE ORDERING SCHEMES

8.1 Introduction

Chapter 5 discussed the results of analysing BDD's obtained using a top-down, left-right ordering of basic events and also the 'new' ordering scheme. The results showed that savings can be made on the BDD computation by simply considering the repeated events in the fault tree first in the ordering. However it is evident that the conventional top-down, left-right ordering or even the new ordering scheme can be totally inadequate for certain fault trees. Indeed this is the case for the fault tree referred to as HPIS in the paper by Platz and Olsen (44), a summary of the fault tree features can be seen in table 8.1. The BDD for this fault tree exploded in size with the conventional ordering and could not be analysed in a reasonable time, it took 3hrs 36min CPU time on a Sun workstation to determine the full set of minimal cut sets.

Name	HPIS
No. of gates	81
No. of basic events	199
No. of repeated basic events	68
No. of minimal cut sets	8, 179

Table 8.1 Summary of HPIS Fault Tree

Table 8.1 clearly shows that the fault tree is not a very large fault tree structure with 280 events (gates and basic events) and producing just over eight thousand minimal cut sets. The problem would appear to occur due to the number of repeated basic events. This and other fault tree examples created unacceptable BDD sizes, when using a top-down, left-right ordering of basic events. As a result of this further research was undertaken into alternative ordering methods. It would be advantageous to try to find a universally beneficial ordering effective for all fault trees. However since this is unlikely it was hoped to identify features of the fault tree structures which would produce efficient BDD's with different ordering schemes.

8.2 Depth-First Ordering

One approach for investigating the basic event ordering was to break the whole problem (i.e. the fault tree) into smaller problems (i.e. subtrees) and look at the optimum ordering for these subtrees. A subtree of a fault tree is simply a gate event other than the top gate. It was found that giving each subtree a top-down, left-right ordering, working from the first gate inputs of the top event, gives an improved ordering scheme. This type of ordering constitutes a 'depth-first' ordering of basic events (Sinnamon and Andrews (45)). To illustrate the depth-first ordering scheme refer to the fault tree shown in figure 8.1.

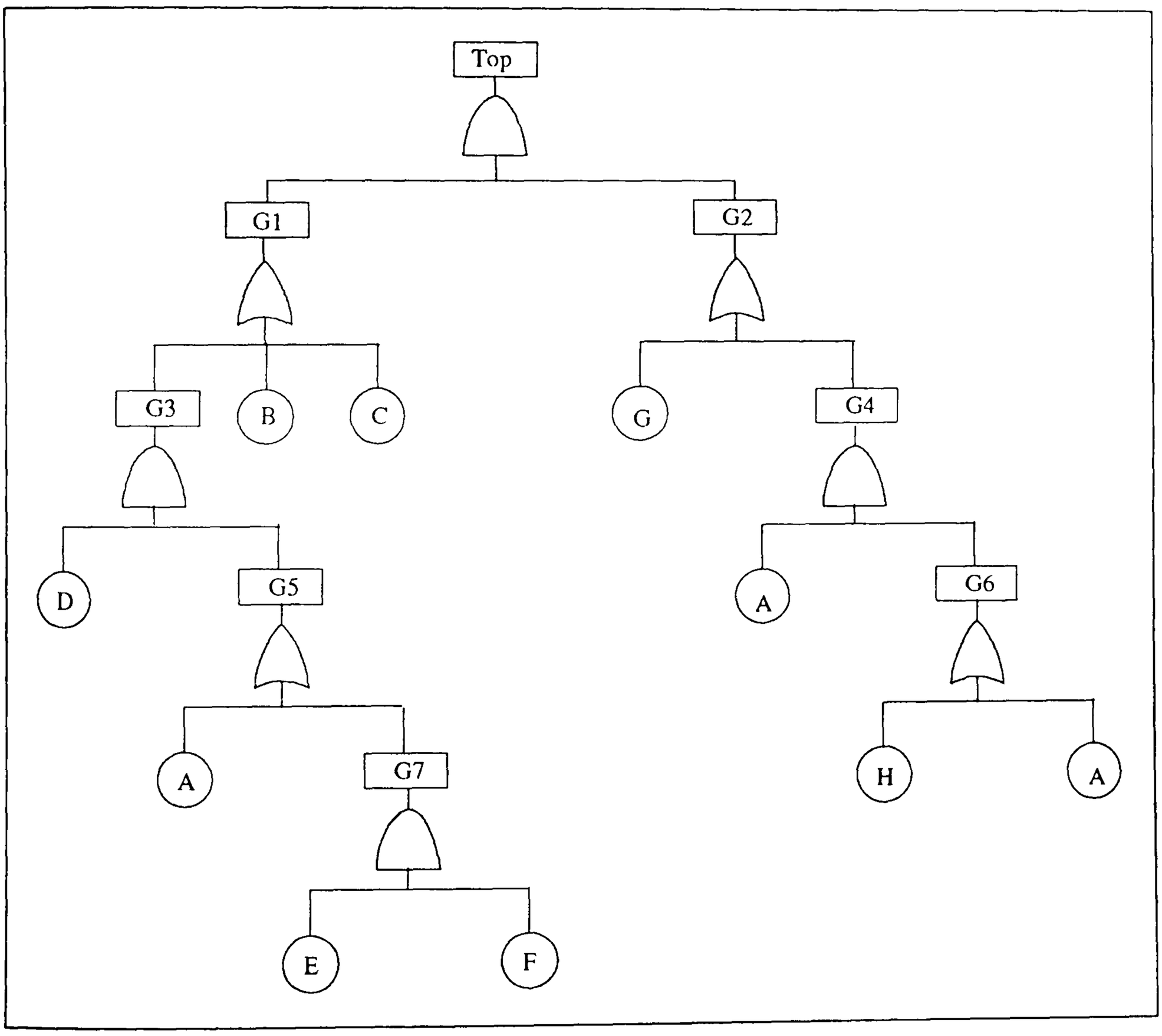


Figure 8.1 Example Fault Tree

Subtrees are identified for the fault tree in figure 8.1 which are focused on the top event output branches G1 and G2. Subtrees are dealt with in a left-right manner as before. The depth-first ordering of the fault tree will first order the subtree G1 i.e. inputs to the gate event G1 which are B and C, followed by the inputs to gate G3 (D)

then G5 (A) and G7 (E, F). Next comes the ordering of the second top event branch subtree G2, followed by G4 and lastly G6. Considering G2 adds basic event G to the ordering list. G4 does not add any events since A has been previously ordered. For the same reason consideration of G6 only adds basic event H to the list. The resulting ordering of basic events will therefore be:

$$B<C<D<A<E<F<G<H \tag{8.1}$$

This depth-first ordering directly produces the minimal form of the BDD without the need to minimise its structure. As a comparison the conventional top-down, left-right ordering for this fault tree is:

$$B<C<G<D<A<H<E<F \tag{8.2}$$

This ordering produces a redundant BDD to which the minimisation algorithm has to be applied to derive the minimal cut sets. This depth-first ordering scheme (8.1) proved fruitful for the HPIS fault tree which now produced the minimal cut sets in only 0.94 secs CPU time, rather than over three hours when previously analysed using the top-down ordering, the results are shown in table 8.2.

Name	HPIS
No. of minimal cut sets	8, 179
No. of nodes in BDD	605
Time (s)	0.94

Table 8.2 BDD Results for HPIS Fault Tree

Bouissou (54) considered a similar type of ordering heuristic in his paper and provided five fault tree examples. Bouissou stated that this type of ordering gave a good average performance. However he stated that this ordering is very sensitive to the writing of the fault tree (i.e. it is an arbitrary choice as to the ordering of the top event inputs) and can lead to BDD sizes (and CPU times) which differ by orders of magnitude. In addition he stated that the major problem with conventional ordering heuristics is the lack of theoretically proved properties, which calls for huge test and programming efforts.

8.3 Priority-Depth-First Ordering

To take this depth-first approach one step further in order to investigate any additional potential benefits which can be gained, one needs to refer back to the construction of the BDD using the structure function (Chapter 4, section 4.3). To obtain the smallest, most efficient BDD from the structure function those basic events which have the greatest influence over the structure function were considered first. Experience shows that frequently these basic events lie higher up the fault tree, close to the top event, hence the reason why a top-down approach is effective. Therefore when applying the depth-first ordering it may have some advantage in the numbering order to give priority to those subtrees that have basic event inputs only. In this way the basic events that occur at higher levels in the tree will be ordered prior to any others. This ordering will be referred to as 'priority-depth-first' ordering. To illustrate the application of this ordering technique refer to the fault tree in figure 8.2.

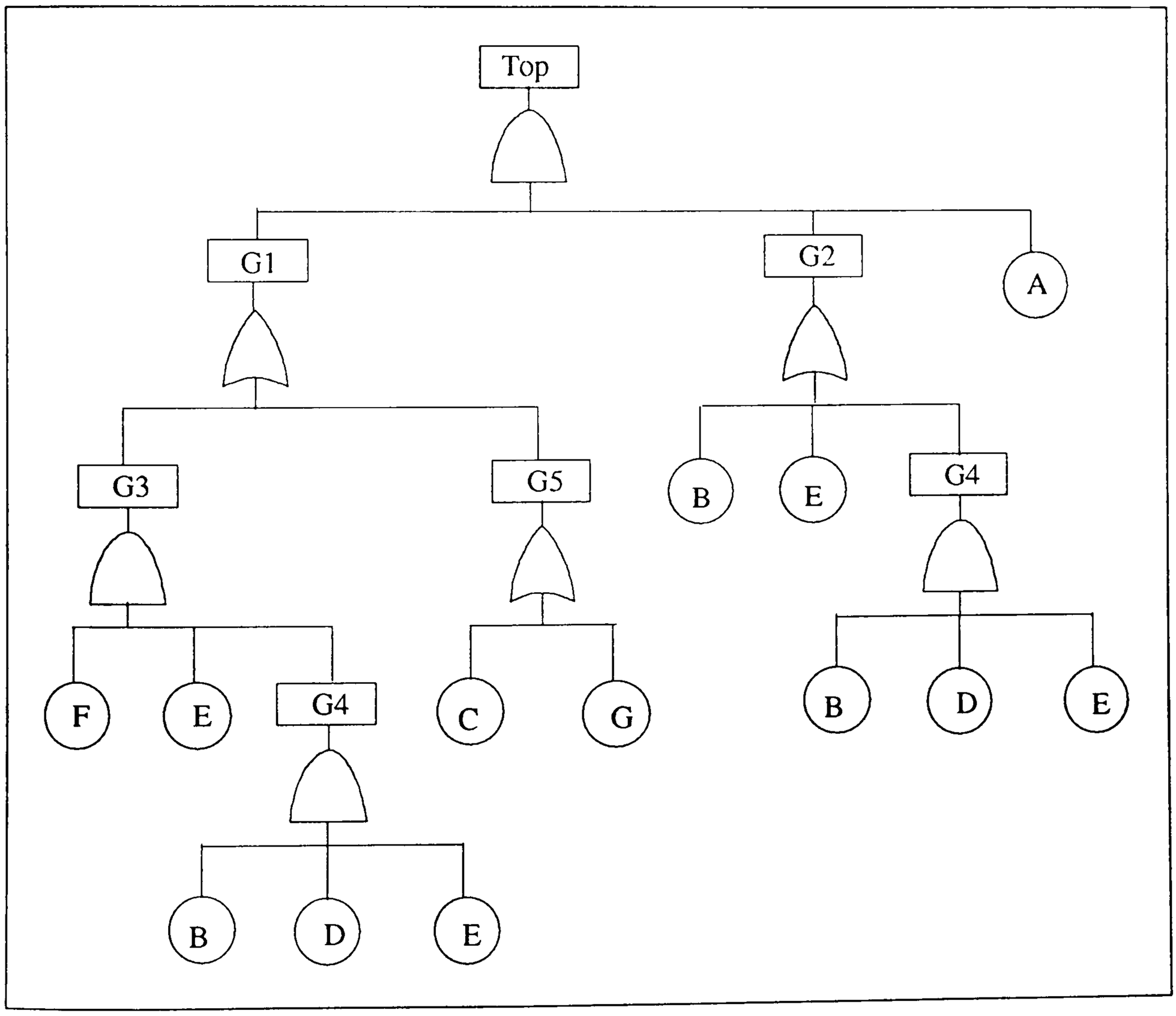


Figure 8.2 Example Fault Tree for 'depth' Orderings

The gates in this fault tree should be considered in the following order to give a priority-depth-first ordering G1, G5, G3, G4, G2, Top, note G5 has basic event inputs

only and therefore should be ordered before G3. Subtrees which do not have priority are again dealt with in a left-right manner. Therefore the basic event 'priority-depth-first' ordering will be:

C<G<F<E<B<D<A

This ordering directly produces a minimal BDD with the following minimal cut sets:

(1) {C, E, A}
(2) {C, B, A}
(3) {G, E, A}
(5) {G, B, A}
(6) {F. E, B, D, A}

whereas the depth-first ordering:

A<F<E<B<D<C<G

produces a redundant BDD that needs to be minimised.

Five example fault trees were analysed using the three different ordering options where:

ordering 1 - top-down, left-right ordering
ordering 2 - depth-first ordering
ordering 3 - priority-depth-first ordering

Table 8.3 provides a summary of each fault tree structure and the results can then be seen in table 8.4. Tree1 and Tree2 were obtained from papers reviewed in the literature survey in Chapter 2. Tree3 was created as a simple test case fault tree. Tree4 and Tree5 are fault trees from the transport industry. The number of **ite** calculations to construct the original BDD and also those required to produce a minimal form for the BDD have been entered in table 8.4. Also entered in this table is the difference in these values which is the number of extra **ite** calculations that are needed to make the BDD minimal. The **ite** calculations correspond to the number of AND and OR operations that are needed to create the **ite** structure for each gate in the fault tree.

Fault Tree	Number of Gates	Number of Basic Events	Number of Repeated Basic Events	Number of Minimal Cut Sets
Tree1	60	57	41	11, 934
Tree2	30	34	28	35
Tree3	3	4	1	2
Tree4	32	63	0	8, 716
Tree5	30	60	0	7, 056

Table 8.3 Summary of Five Example Fault Trees

Trees and orderings	No. of ite calculations before minimising	No. of ite calculations after minimising	Difference	Time (s)
Tree1				
1	1162	1734	572	2.23
2*	1439	1802	363	2.34
3	1644	2081	437	1.51
Tree2				
1	136	181	45	0.41
2	284	375	91	0.24
3*	336	378	42	0.23
Tree3				
1*	25	26	1	0.12
2	25	27	2	0.14
3	25	27	2	0.14
Tree4				
1	248	382	134	1.12
2*	315	399	84	1.21
3	331	458	127	1.16
Tree5				
1	234	354	120	1.00
2	299	377	78	0.59
3*	259	334	75	0.56

Table 8.4 Results of Different Ordering Schemes on Five Example Fault Trees

The results show that there are large differences in the number of **ite** computations when different ordering schemes are used for the basic events. Hence great savings can be made in terms of computation time and memory requirements when an efficient ordering of the basic events can be established. However it is also clear from these examples that each tree has an individual variable ordering that will optimise the size of its BDD. Here, there is no unique ordering scheme which is 'best' for all five fault trees (the 'best' ordering for each fault tree has been marked with a *).

8.4 BDD Size Dependence on Ordering Schemes

The number of **ite** calculations for each basic event ordering scheme has been discussed in the previous section. The number of computations is however not the only important consideration. The size of the resulting BDD, i.e. the number of nodes it contains is also an extremely important consideration as far as memory requirements are concerned. The **ite** table may become very large due to the number of **ite** computations and the dimensions of the arrays used for storage cannot be increased any further due to memory restrictions within the Sun workstation. Therefore extensive investigations have been performed on fifty-one different benchmark fault trees and the number of non-repeated nodes produced in the BDD by the three different ordering schemes has been compared (ordering 1 - top-down left-right, ordering 2 - depth-first, ordering 3 - priority-depth-first). The results can be seen in table 8.5.

Appendix IV provides the summary and information characterising these fifty-one fault trees. Many of the trees were obtained from the papers reviewed in Chapter 2 and Chapter 3 where they had been used to test other algorithms. Additionally some have been obtained directly from industry (nuclear, transport and offshore) because they have features which make analysis difficult. The remainder have been constructed to have certain features required to test methods as part of the research for this thesis. The smallest number of BDD nodes, created by each ordering, is given in bold text in table 8.5.

Fault Tree Number	ordering 1		ordering 2		ordering 3	
	No. of Nodes	Time (s)	No. of Nodes	Time (s)	No. of Nodes	Time (s)
1	333	0.45	**307**	1.05	308	0.65
2	10	0.14	**8**	0.67	**8**	0.58
3	5	0.14	5	0.53	**4**	0.56
4	28	0.12	**26**	0.61	**26**	0.57
5	**17**	0.15	20	0.59	20	0.55
6	43	0.16	**41**	0.60	**41**	0.57
7	106	1.12	**63**	1.58	**63**	0.53
8	106	0.90	**61**	1.26	**61**	0.58
9	98	1.04	**60**	1.43	**60**	0.52
10	**40**	0.16	**40**	0.52	**40**	0.60
11	**4**	0.13	**4**	0.59	5	1.54
12	155	11.40	62	11.00	**61**	1.34
13	31	0.58	**20**	0.61	**20**	1.38
14	30	0.45	**22**	0.50	**22**	1.45
15	52	0.64	**33**	0.77	**33**	1.28
16	**180**	1.49	275	1.65	335	1.37
17	**272**	3.71	598	3.55	647	1.37
18	**175**	1.73	268	1.67	394	1.75
19	16	0.64	**11**	0.62	**11**	0.62
20	**12**	0.47	**12**	0.66	**12**	1.52
21	216	0.61	**103**	0.51	104	1.40
22	**43**	4.90	49	0.67	59	6.42
23	481	5.15	214	0.87	**162**	5.65
24	52	4.86	68	0.68	**42**	5.19
25	-	-	605	0.94	**475**	5.05
26	10	0.65	**7**	0.55	**7**	1.31
27	10	0.47	**7**	0.58	**7**	1.42
28	37	0.58	**21**	0.48	**21**	1.30
29	31	0.65	**19**	0.57	**19**	1.40
30	21	0.52	**20**	0.60	21	1.45
31	-	-	**361**	0.64	366	1.58
32	**39**	0.61	60	0.57	**39**	1.57
33	64	0.57	**38**	0.67	**38**	1.59
34	**7**	0.61	**7**	0.54	**7**	1.55
35	**4**	0.52	**4**	0.55	**4**	1.28
36	-	-	**7290**	57 mins	-	-
37	**6**	0.66	**6**	0.59	**6**	1.43
38	1472	1.31	450	1.21	**413**	1.04
39	**4**	0.61	**4**	0.56	**4**	1.49
40	**4**	0.48	**4**	0.65	**4**	1.54
41	11	0.48	**8**	0.56	**8**	1.66
42	**5**	0.47	**5**	0.57	**5**	1.59
43	**2**	0.51	**2**	0.56	**2**	1.39
44	**4**	0.60	**4**	0.59	**4**	1.58
45	19	0.52	16	0.50	**14**	1.26
46	**333**	5.31	372	0.63	390	1.41
47	**5**	0.62	**5**	0.56	6	1.47
48	8	0.64	**7**	0.60	8	1.71
49	**6**	0.57	**6**	0.66	**6**	1.77
50	**7**	0.57	**7**	0.63	**7**	1.50
51	14	0.55	13	0.55	**12**	1.51
Total & average time	21	n=48 $\bar{x}$=0.948	36	n=48 $\bar{x}$=0.787	**37**	n=48 $\bar{x}$=1.487

Table 8.5 The Number of Nodes the BDD has for Three Different Orderings

The results in table 8.5 show that ordering 3 (priority-depth-first) created the largest number of 'optimum' BDD's, i.e. BDD's that have the smallest number of nodes when compared to the other orderings. Since smaller BDD structures require less processing to derive the minimal cut sets they can be thought of as being optimum. The totals in the columns headed 'No. of Nodes' shows the number of 'optimum' BDD's for that particular ordering. However ordering 2 (depth-first) was the only ordering that could solve fault tree number 36, the other orderings could not analyse this fault tree because of memory restrictions. Taking the forty-eight fault trees that could be analysed by all three orderings (i.e. omit trees 25, 31 and 36) the average execution times are 0.948s for ordering 1, 0.787s for ordering 2 and 1.487s for ordering 3. Although the best BDD ordering came from ordering 3 the best execution time came from ordering 2. This implies that the extra time required for ordering 3 was that needed to search for the subtrees which had basic event inputs only and then ordering them accordingly. Nevertheless as the average execution times are very small the extra 'searching' time can be considered to be unimportant.

8.5 The 'New' Ordering

'New' ordering was briefly discussed in Chapter 5 and, from the results produced in that chapter, was worth further consideration. New ordering identifies the repeated basic events in the fault tree which are given priority in the ordering process. In Chapter 5 the new ordering was applied within the basic top-down, left-right ordering scheme. Here the new ordering is used in conjunction with all three ordering schemes discussed, i.e. top-down, left-right ordering, depth-first ordering and priority-depth-first ordering. As a result a total of six different ordering schemes are investigated.

The number of nodes produced for each BDD using these six ordering schemes is given in table 8.6. The table shows clearly that for each of the three different orderings, applying the new technique results in more of the BDD's being optimum. Again the optimum BDD for each ordering is displayed in bold text. As described previously in Chapter 5, gains in computation can be made by simply considering the repeated events first in the fault tree for all three orderings. The average execution times for the new orderings 1, 2 and 3, are 1.615s, 1.569s and 1.6775s respectively (again using the forty-nine fault trees that could be analysed by all of the ordering schemes). The very small extra CPU time for each new ordering can be attributed to the sorting that is required to select the repeated events when ordering the inputs to each gate.

Fault Tree Number	ordering 1	new ordering 1	ordering 2	new ordering 2	ordering 3	new ordering 3
1	333	333	**307**	**307**	308	308
2	10	9	8	**7**	8	8
3	5	5	5	**4**	**4**	**4**
4	28	27	26	26	26	**25**
5	**17**	**17**	20	20	20	20
6	43	42	41	**38**	41	**38**
7	106	106	**63**	**63**	**63**	**63**
8	106	106	**61**	**61**	**61**	**61**
9	98	98	**60**	**60**	**60**	**60**
10	**40**	**40**	**40**	**40**	**40**	**40**
11	**4**	**4**	**4**	**4**	5	5
12	155	151	62	**61**	**61**	**61**
13	31	31	**20**	**20**	**20**	**20**
14	30	30	**22**	**22**	**22**	22
15	52	52	**33**	**33**	**33**	**33**
16	**180**	**180**	275	275	335	335
17	**272**	**272**	598	598	647	647
18	**175**	176	268	268	394	394
19	16	16	**11**	**11**	**11**	**11**
20	12	11	12	**10**	12	11
21	216	215	**103**	**103**	104	104
22	43	**42**	49	48	59	59
23	481	479	214	212	**162**	**162**
24	52	50	68	66	**42**	**42**
25	-	-	605	600	475	**473**
26	10	10	**7**	**7**	**7**	**7**
27	10	10	**7**	**7**	**7**	**7**
28	37	37	**21**	**21**	**21**	**21**
29	31	31	**19**	**19**	**19**	**19**
30	21	21	**20**	22	21	21
31	-	-	361	**355**	366	358
32	**39**	**39**	60	60	**39**	**39**
33	64	63	38	**37**	38	**37**
34	7	**6**	7	**6**	7	7
35	**4**	**4**	**4**	**4**	**4**	**4**
36	-	-	7290	-	-	-
37	**6**	**6**	**6**	**6**	**6**	**6**
38	1472	1412	450	428	**413**	**413**
39	**4**	**4**	**4**	**4**	**4**	**4**
40	**4**	**4**	**4**	**4**	**4**	**4**
41	11	11	**8**	**8**	**8**	**8**
42	**5**	**5**	**5**	**5**	**5**	**5**
43	**2**	**2**	**2**	**2**	**2**	**2**
44	**4**	**4**	**4**	**4**	**4**	**4**
45	19	19	16	16	**14**	**14**
46	**333**	**333**	372	372	390	390
47	5	**4**	5	**4**	6	5
48	8	8	**7**	**7**	8	8
49	**6**	**6**	**6**	**6**	**6**	**6**
50	**7**	**7**	**7**	**7**	**7**	**7**
51	14	14	13	13	**12**	**12**
Total	17	19	27	**35**	30	34

Table 8.6 The Number of Nodes the BDD has for Six Different Orderings

Number	ite calculations before min	ite calculations after min	differ-ence
1	1479	2194	715
2	**22**	**22**	**0**
3	11	13	2
4	75	102	27
5	66	123	57
6	69	87	18
7	366	450	84
8	356	436	80
9	350	428	78
10	237	276	39
11	**10**	**10**	**0**
12	**175**	**175**	**0**
13	59	60	1
14	81	82	1
15	136	148	12
16	446	709	263
17	1237	2140	903
18	1440	1803	363
19	25	26	1
20	**26**	**26**	**0**
21	385	529	144
22	**173**	215	42
23	649	818	169
24	196	283	87
25	2763	4385	1622
26	**16**	**16**	**0**
27	**19**	**19**	**0**
28	**59**	**59**	**0**
29	**66**	**66**	**0**
30	61	80	19
31	1299	2359	1060
32	289	380	91
33	117	172	55
34	19	23	4
35	11	14	3
36	-	-	-
37	**13**	**13**	**0**
38	1238	2065	827
39	**14**	**14**	**0**
40	**17**	**17**	**0**
41	**23**	**23**	**0**
42	**15**	**15**	**0**
43	**12**	**12**	**0**
44	**19**	**19**	**0**
45	41	53	12
46	6494	8674	2180
47	**10**	**10**	**0**
48	**18**	**18**	**0**
49	**17**	**17**	**0**
50	**21**	**21**	**0**
51	34	37	3

Table 8.7 Number of **ite** Calculations needed for New Ordering 2

All the different ordering schemes have been implemented in a program called ORDERING. When the program is executed the user is prompted by several questions asking which ordering is to be used for the fault tree.

Although the nodes of the BDD tell us what size it is, it does not give us any indication as to whether or not it is minimal. For this purpose we need to know the number of **ite** computations before and after minimisation. On looking at table 8.6 it can be seen that new ordering 2 creates the largest number of optimal BDD's when compared to the other orderings. The last row in table 8.6 provides the totals for the number of optimum BDD's that each ordering creates. The number of **ite** computations to derive the original and minimal BDD's for new ordering 2 is provided in table 8.7, (the fault trees whose BDD was directly minimal for this particular ordering can be seen in bold text).

The results in table 8.7 show that new ordering 2 created minimal BDD's in nineteen out of the fifty-one fault trees, i.e. 37.25% of the cases. For the thirty-two other fault trees it produced the best ordering of the six schemes investigated and it is of course not always possible to produce a minimal BDD. Fault tree number 12 is interesting, even though the fault tree had 84,424 minimal cut sets, a minimal BDD was constructed for this ordering. It is also worth mentioning that fault tree number 36 could not be analysed using new ordering 2 (ran out of memory on the Sun workstation) even though this fault tree could be analysed using ordering 2. Therefore in this instance ordering the repeated events first in the fault tree proved a disadvantage.

8.6 Variable Ordering Using Repeated Basic Events and Subtree Levels

From the results of the research into the ordering of the basic events in the fault tree, it is clear that great gains in computation can be made if a 'good' ordering scheme can be found. Even if a minimal BDD cannot be obtained it is advantageous to produce a 'near-minimal' diagram. The results show that an ordering scheme that employs a depth-first approach is generally a good one. Further, it has been discovered that the repeated basic events in the fault tree have a significant influence on the size of the BDD. In addition to this, the priority-depth-first ordering scheme shows that by first ordering those subtrees which have basic event inputs only, even further gains in computational efficiency can be made. Therefore an improved method of ordering the

basic events in the fault tree could result from a scheme which incorporates all these aspects within a depth-first approach. To investigate this an ordering scheme has been developed to deal with fault trees which contain repeated basic events. It is a variable ordering technique which considers **RE**peated **B**asic **E**vents and **SU**btree **L**evels called REBESUL. To illustrate the notion used to identify subtree levels refer to subtree G1 in figure 8.3. The algorithm for REBESUL is based on the following six steps. Each step of REBESUL is justified and then explained by the use of an example.

REBESUL

(1) *Create a list of the repeated events in the fault tree, those with the highest number of occurrences are listed first. Repeated events that have an equal number of occurrences are placed in the rows between the next highest and next lowest.*

(2) *For each repeated event in step (1) create a list of the subtrees (first sons of the top gate) that contain this repeated event in the order of the highest number of* ***different*** *repeated event occurrences within each subtree to the lowest.*

(i) *If two or more subtrees share the same number of repetitions for an event, the subtree with the greatest number of levels takes precedence over how many repetitions there are in a subtree.*

(3) *Create a list of the levels in the subtree at which the repeated event in step (2) occurs.*

(4) *Order the gates (depth-first) starting with the gate that 'contains' the lowest level occurrence (obtained in step (3)) of the repeated event, followed by the other gates which 'contain' the next level of occurrence of the repeated event. Note that the term 'contains' does not necessarily mean that the repeated event is a direct input to the gate, it may be an input a few levels down. List the repeated events first when ordering the inputs of each gate.*

(5) *If all the repeated events have been dealt with in this subtree order any remaining events to gates in the subtree depth-first and goto (6). Otherwise goto (3) for the next repeated event obtained in (1).*

(6) *If all subtrees containing repeated events have been dealt with order any remaining subtrees depth-first. Otherwise order the next subtree containing repeated events, i.e. goto (2).*

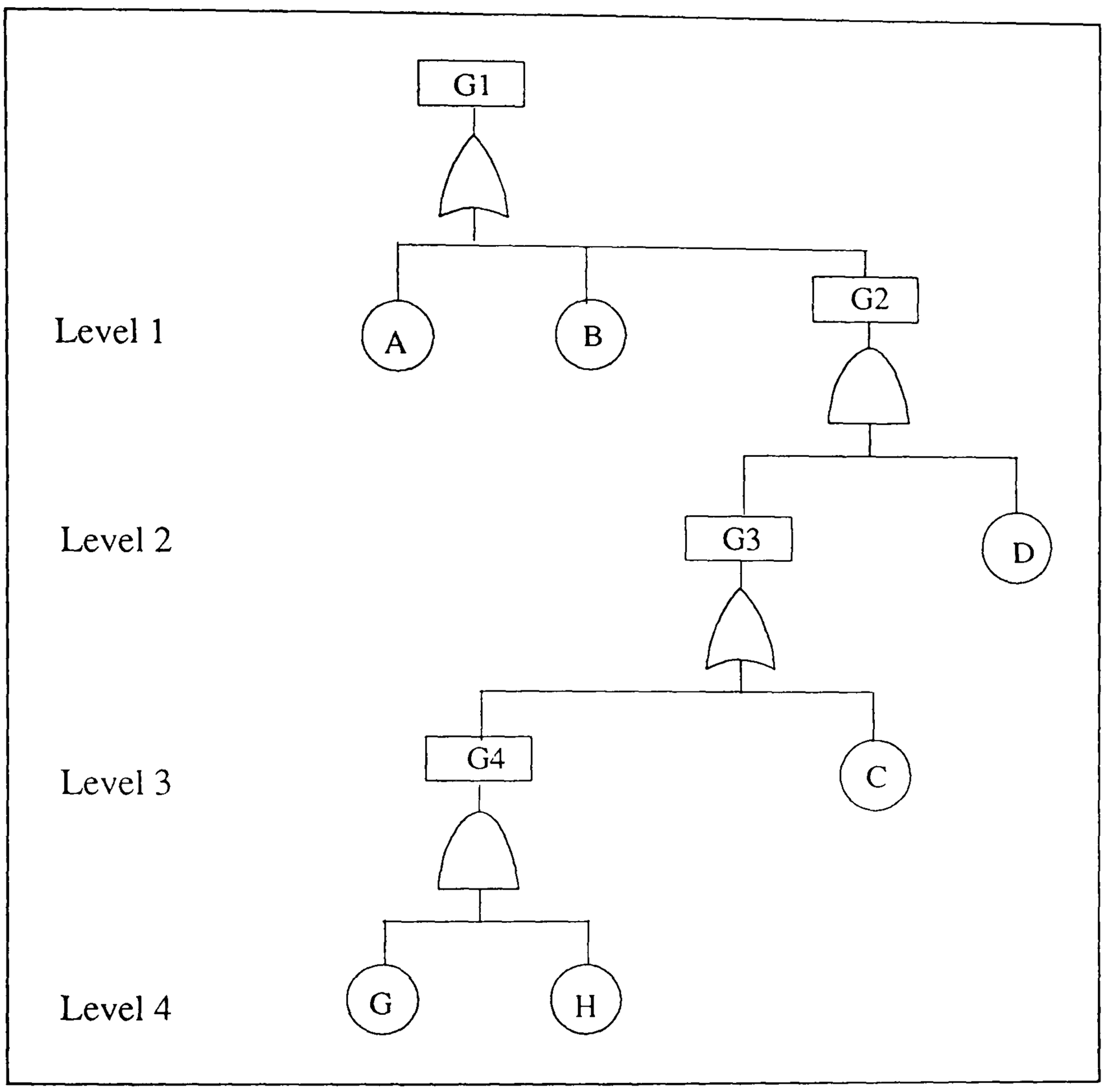


Figure 8.3 Subtree Levels

8.6.1 Justification for the REBESUL Ordering

The steps in the ordering have each been given a justification by the use of a heuristic. The heuristics produced are based on observations and knowledge gained while undertaking the ordering research.

Step (1) *"Create a list of the repeated events in the fault tree, those with the highest number of occurrences are listed first. Repeated events that have an equal number of occurrences are placed in the rows between the next highest and next lowest."*

Heuristic (1)

Repeated basic events cause redundancy in the Boolean logic expression for the top event of a fault tree. Since without this type of event the cuts sets produced are already minimal, they need special consideration for an efficient fault tree analysis process. It is has been shown in Chapter 4 that the BDD is in the form of Shannon's decomposition. An efficient decomposition of a structure function is obtained by pivoting about the most repeated variables through to the least repeated within each residue. Therefore to follow the rules for efficiency based on Shannon's decomposition to construct the BDD, the repeated basic events in the fault tree are listed by the highest number of occurrences through to the lowest. This list will form the basis of the ordering.

Step (2) *"For each repeated event in step (1) create a list of the subtrees (first sons of the top gate) that contain this repeated event in the order of the highest number of* ***different*** *repeated event occurrences within each subtree to the lowest."*

Heuristic (2)

The subtree with the largest number of different repeated events (including the most occurring repeated event) holds more influence over the rest of the tree.

Step (2) (i) *"If two or more subtrees share the same number of repetitions for an event, the subtree with the greatest number of levels takes precedence over how many repetitions there are in a subtree."*

Heuristic (2) (i)

Let one subtree of a top event be the basic event r, which has the BDD structure **ite**(r, 1, 0), this is obviously a minimal structure. Let a second subtree be F which also contains r and many more levels. The most efficient ordering for this subtree is depth-first and listing r first when ordering the inputs to the gate where it occurs.

To demonstrate, let F have the BDD structure **ite**(x, F1, F2) and let r lie below F1. Suppose it is necessary to compute **ite**(x, F1, F2) <op> **ite**(r, 1, 0) (see figure 8.4 for the BDD representation of this operation).

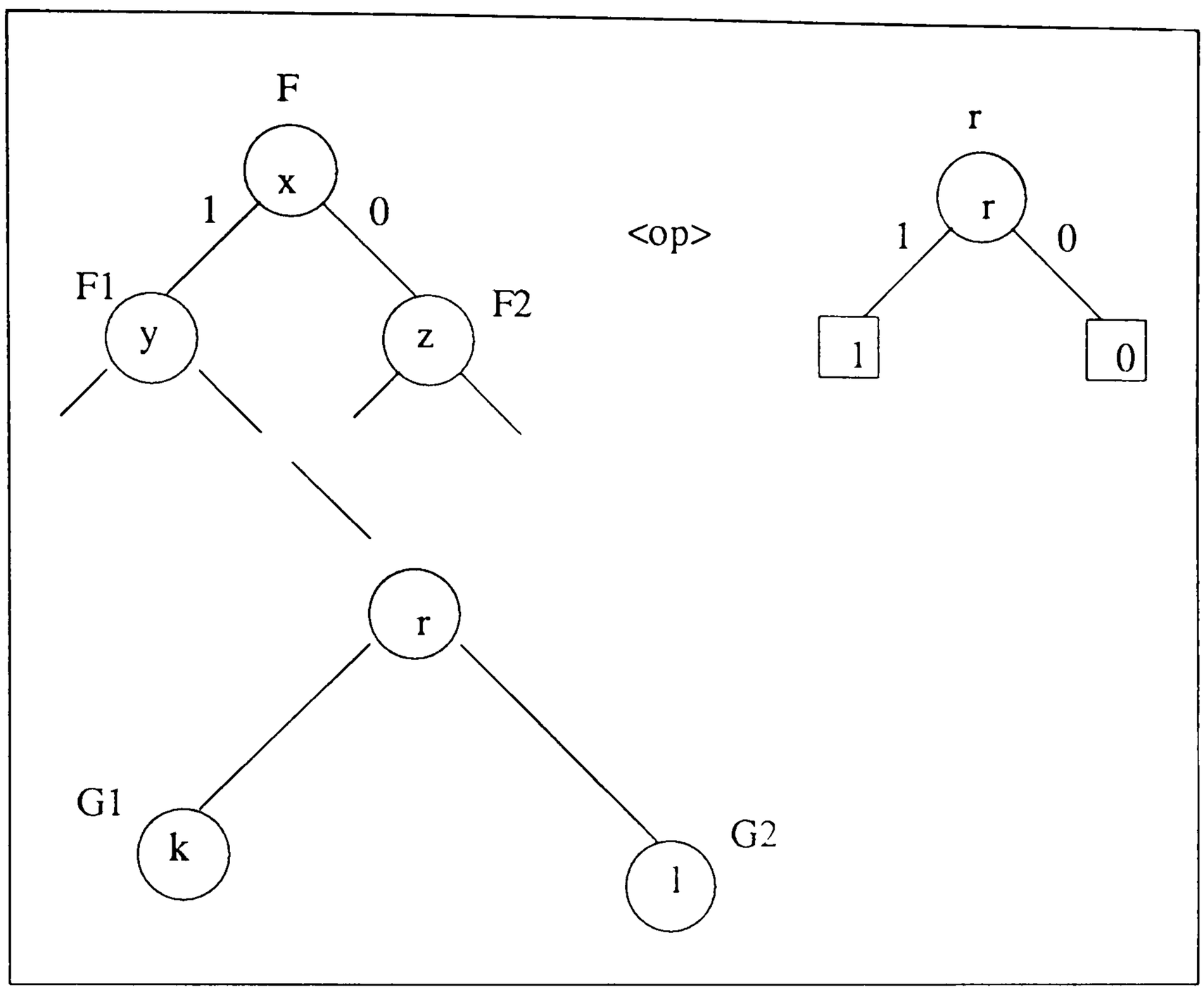


Figure 8.4. **ite**(x, F1, F2) <op> **ite**(r, 1, 0)

As F has more levels then x<r in the ordering and **ite**(x, F1, F2) <op> **ite**(r, 1, 0)=**ite**(x, F1<op>r, F2<op>r). The computation of F2<op>r would lead to a minimal structure as all the basic events of F2 have been ordered prior to r, and F2 does not contain r. Therefore we must consider whether the development of F1<op>r will lead to an efficient structure. The operation F1<op>r would eventually lead to the computation of:

ite(r, G1, G2) <op> **ite**(r, 1, 0)
=**ite**(r, 1<op>G1, 0<op>G2)

It is evident that **ite**(r, 1<op>G1, 0<op>G2) will be a minimal structure as G1 and G2 contain no repeated events. Also all other computations of the nodes below F1 and r would be minimal, as again no other nodes in F1 contain r.

It follows that the ordering x<r would be more efficient than having r ordered first in the F structure, i.e. let F=**ite**(r, K1, K2). In this case F, which has the most influence over the tree as a whole, has not been ordered in the most efficient way because a depth-first approach has not been used (this ordering would not be compatible with the computations of the gates in the subtree F, unless r occurred on the first level of F). Therefore K1 and K2 would be two inefficient structures.

(3)+(4) "*Create a list of the levels in the subtree at which the repeated event in step (2) occurs.*" "*Order the gates (depth-first) starting with the gate that 'contains' the lowest level occurrence (obtained in step (3)) of the repeated event, followed by the other gates which 'contain' the next level of occurrence of the repeated event. Note that the term 'contains' does not necessarily mean that the repeated event is a direct input to the gate, it may be an input a few levels down. List the repeated events first when ordering the inputs of each gate.*"

Heuristic (3)

This will provide the choice of gates for the depth-first ordering, which has proven to be the most desirable ordering.

Steps (5) and (6) simply deal with any remaining repeated events that require ordering and any remaining subtrees that have not been ordered, by redirecting the algorithm to previous steps.

8.6.2 Computation and Examples of REBESUL Event Ordering

The computation of steps 1-6 of the REBESUL ordering has been achieved through the use of eighteen different subroutines. Table 8.8 provides a summary of the action of each of these subroutines and the diagram shown in figure 8.5 illustrates the relationship between these subroutines, each ellipse represents a subroutine and each branch represents the calling of another subroutine within a subroutine.

Subroutine Name	Summary of Operation.
1. Repevent	Finds number of occurrences of each basic event in the fault tree.
2. Hilorepevent	Orders the basic events highest to the lowest number of occurrences.
3. Subtree	Obtains the subtrees of the top gate.
4. Create	Provides a list of all the elements below a gate.
5. Repeatcheck	Deals with the subtrees of the top gate. Finds out which of the subtrees contains the most repeated basic event. If any two contain the most repeated event then subroutine *Different* is called.
6. Different	Counts the number of different repeated basic events within each subtree.
7. Hilosub	Orders the subtrees depending on the number of different repeated events.
8. Levelcount	Counts the number of levels in each subtree.
9. Sortlevel	Orders the subtrees depending on the number of levels.
10. Sort	Deals with the subtrees in the order depending on different repeated events or subtree levels and calls subroutine *Levelsearch* to search for the repeated event in question.
11. Levelsearch	Finds the level of occurrence of the repeated event and deals with the gates that need to be ordered above or below it.
12. Checkbe	Subroutine that checks if all basic events have been ordered.
13. Iordbelow	Orders the gates that lie above the repeated event.
14. Abovebelow	The level of occurrence of the repeated event is not a level 1 input to the subtree, therefore the gates above it need to be ordered first, this subroutine provides the list of such gates.
15. Rauzrepevent	Provides a 'depth-first array' of inputs to a gate.
16. Altordering	Orders the basic event inputs in 'depth-first array'.
17. Rauzspecord	Orders a gate depth-first and gives repeated events priority in the ordering.
18. Abovegate	Orders remaining gates in a subtree.

Table 8.8 Summary of each of the Subroutines in the REBESUL Ordering

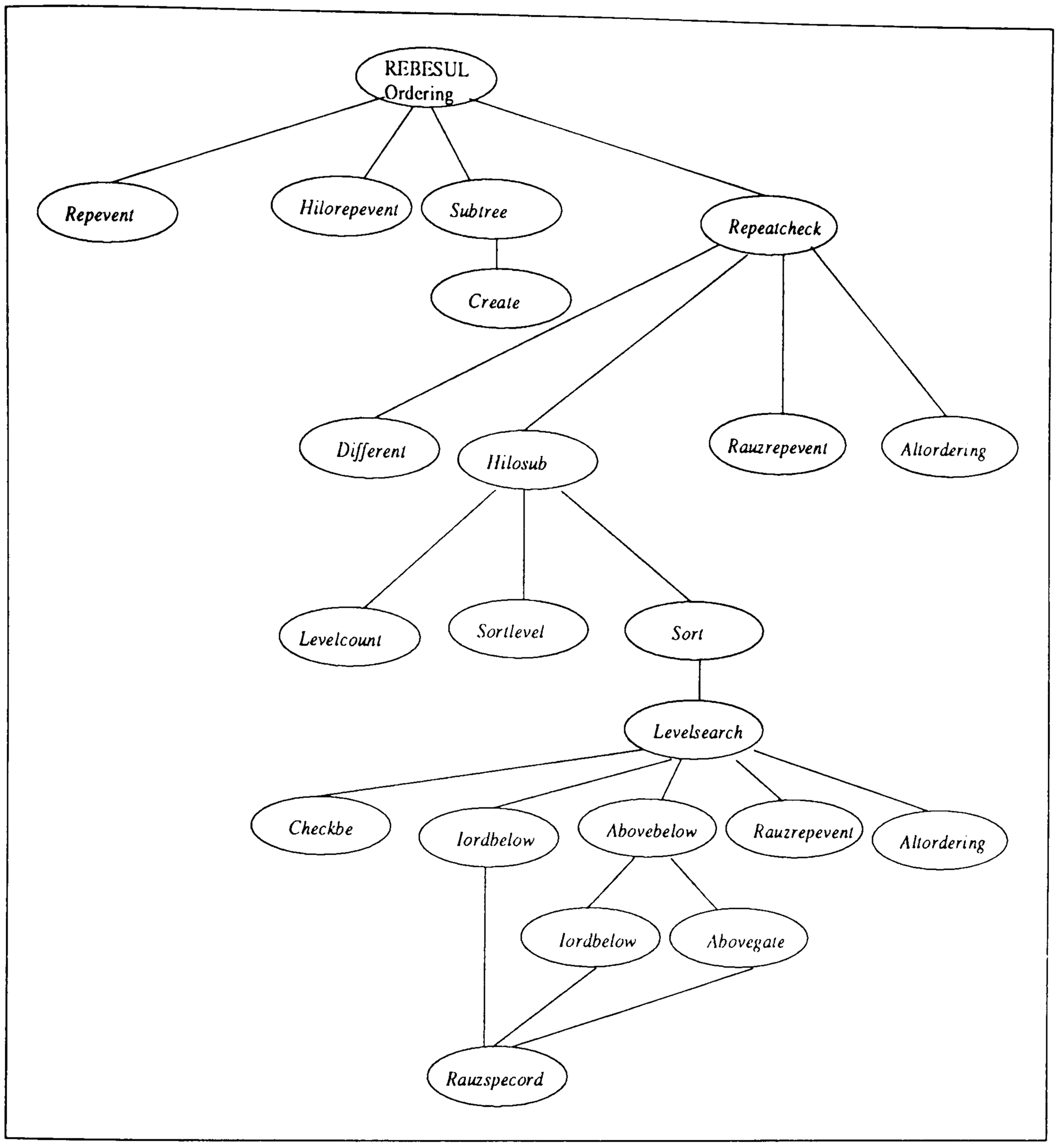


Figure 8.5 Diagram Illustrating the Relationship Between the Subroutines of the REBESUL Ordering

The REBESUL ordering has been applied to five fault trees to illustrate the ordering technique.

Example 1 - Two occurrences of one event

Consider the fault tree structure shown in figure 8.6. It is expressed as an alternating gate sequence, has one repeated basic event labelled 'a' which appears as an input to the top gate and also as a third level input to subtree G1. The ordering algorithm REBESUL proceeds as follows:

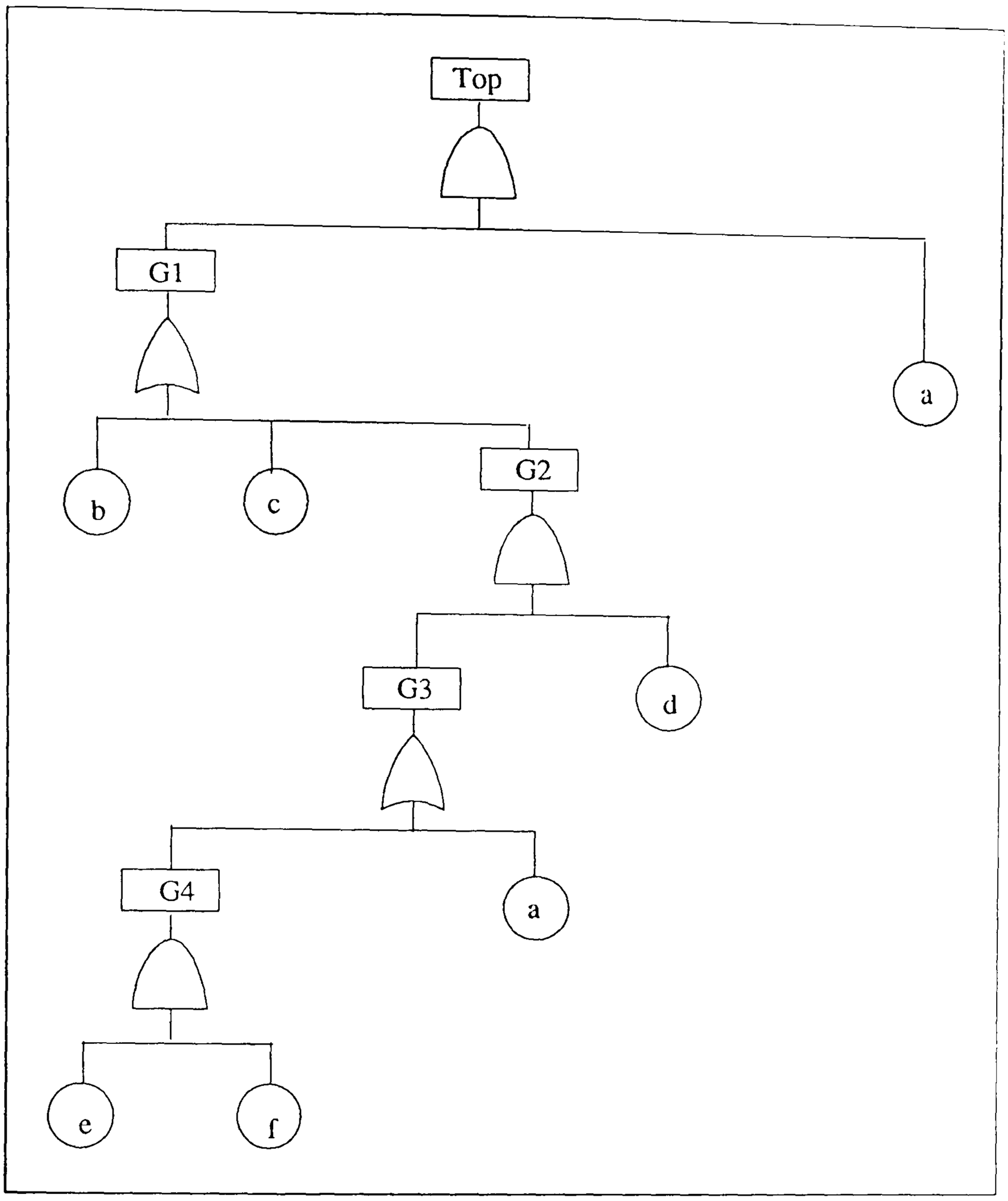


Figure 8.6. Example Fault Tree 1

Ordering Steps

(1) The only repeated event is **a**. It occurs twice.

(2) Both subtree 1 (G1) and subtree 2 (basic event **a**) contain **a**.

(i) Subtree 1 takes precedence as it has a greater number of levels than subtree 2.

(3) Event **a** occurs at level three of subtree 1.

(4) Only G1 contains the level of occurrence of event **a**, via G3. Therefore ordering gate G1 depth-first gives:

b<c<d<a<e<f

(5) All basic events have been ordered.

The BDD for this ordering can be seen in figure 8.7.

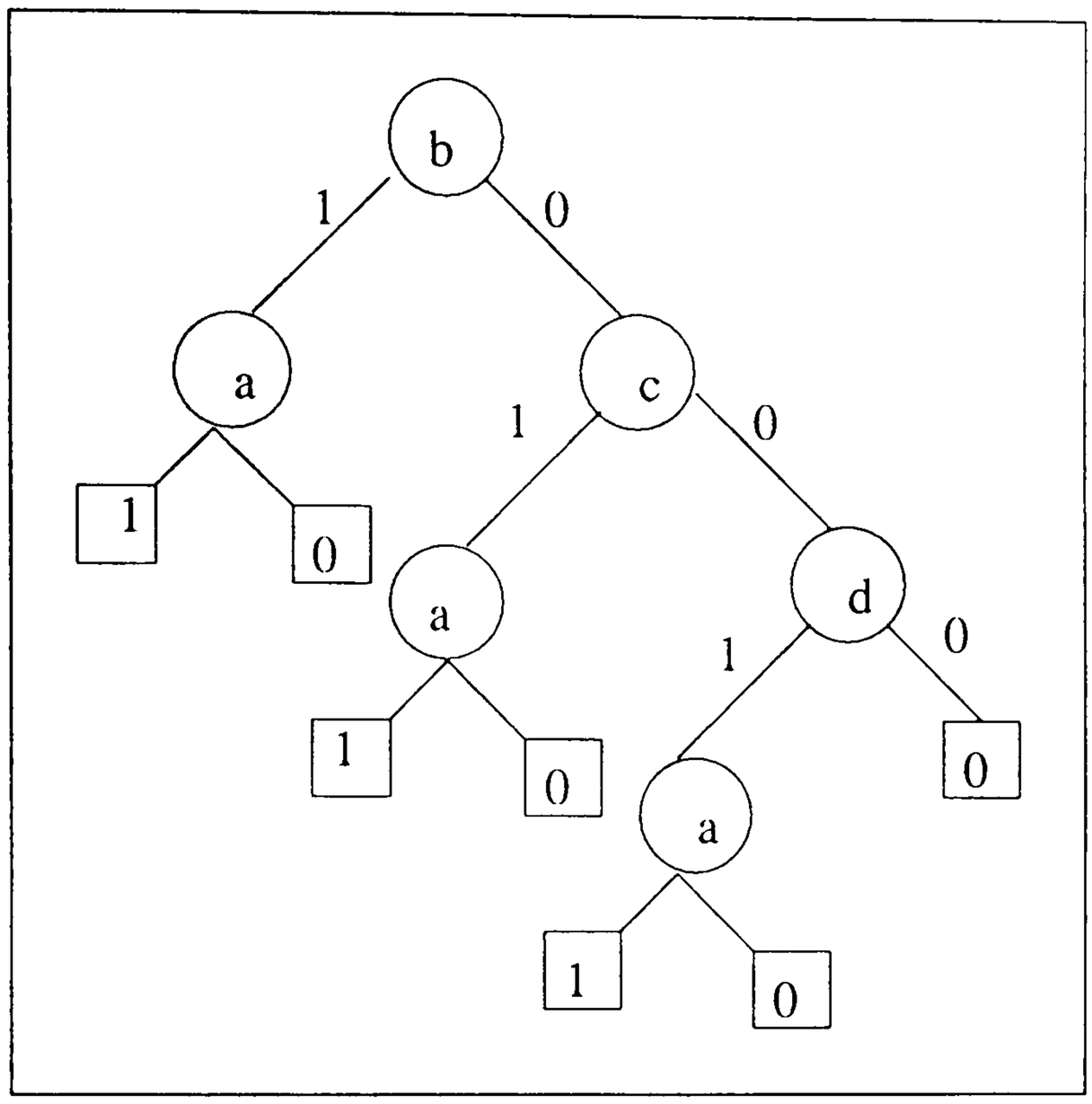


Figure 8.7 BDD for Fault Tree 1

This is a minimal BDD giving the minimal cut sets:

(1) {b, a}
(2) {c, a}
(3) {d, a}

Example 2 - Three occurrences of one event

The fault tree structure for the second example is illustrated in figure 8.8. It consists of eight basic events one of which (**a**) is repeated three times.

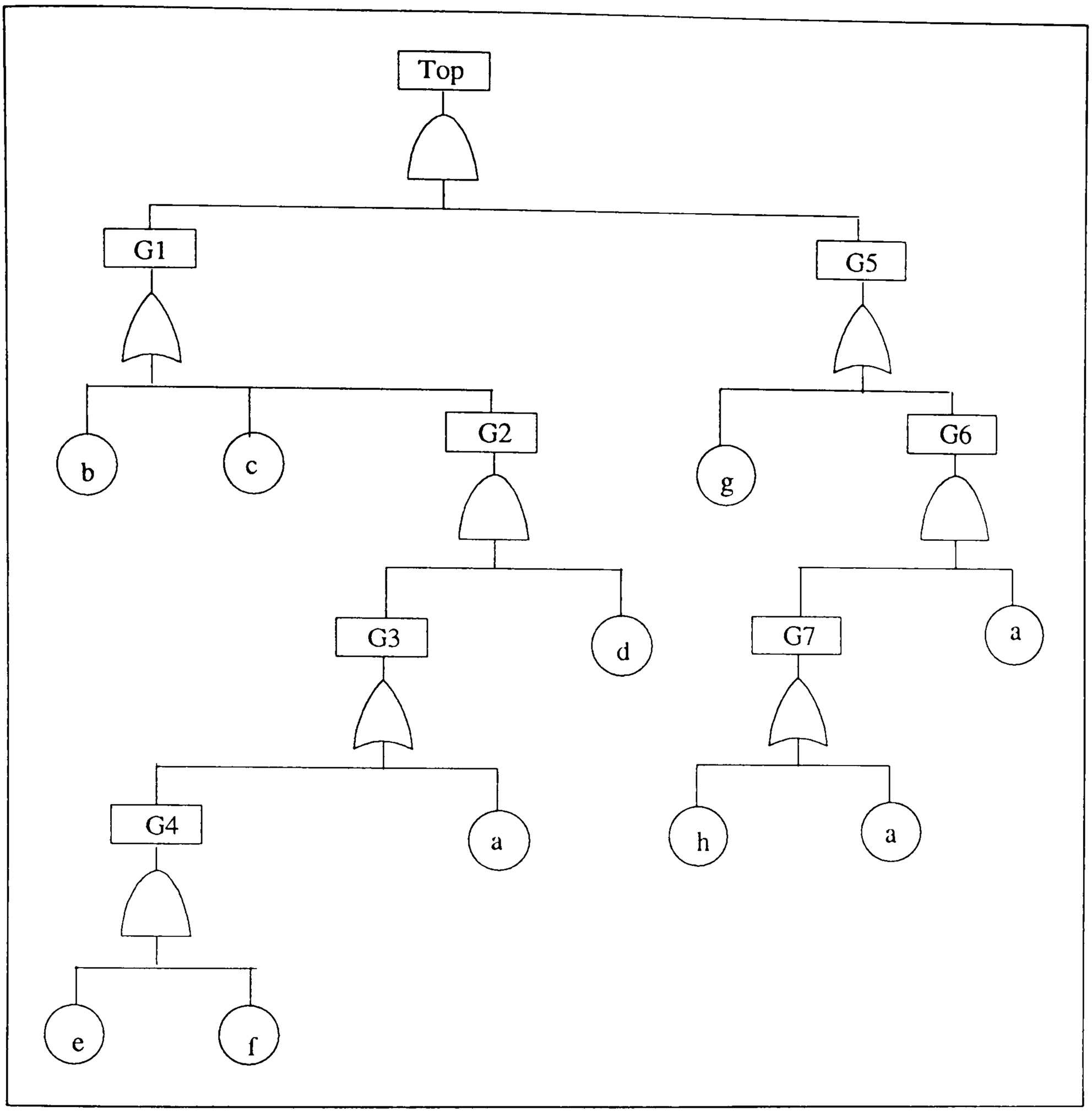


Figure 8.8 Example Fault Tree 2

Ordering Steps

(1) The only repeated event is **a**. It occurs three times.

(2) Both subtree 1 (G1) and subtree 2 (G5) have the same number of different repeated events.

(i) Subtree 1 has four levels and subtree 2 has three levels, therefore subtree 1 takes precedence.

(3) Event **a** occurs at level three of G1.

(4) Only G1 contains the level of occurrence of **a**, therefore the depth-first ordering for this subtree is:

b<c<d<a<e<f

(5) All events have been ordered in this subtree, goto (6).

(6) The depth-first ordering of the remaining subtree 2 is:

g<h

Finally the combined ordering of this fault tree is b<c<d<a<e<f<g<h and it results in the BDD in figure 8.9.

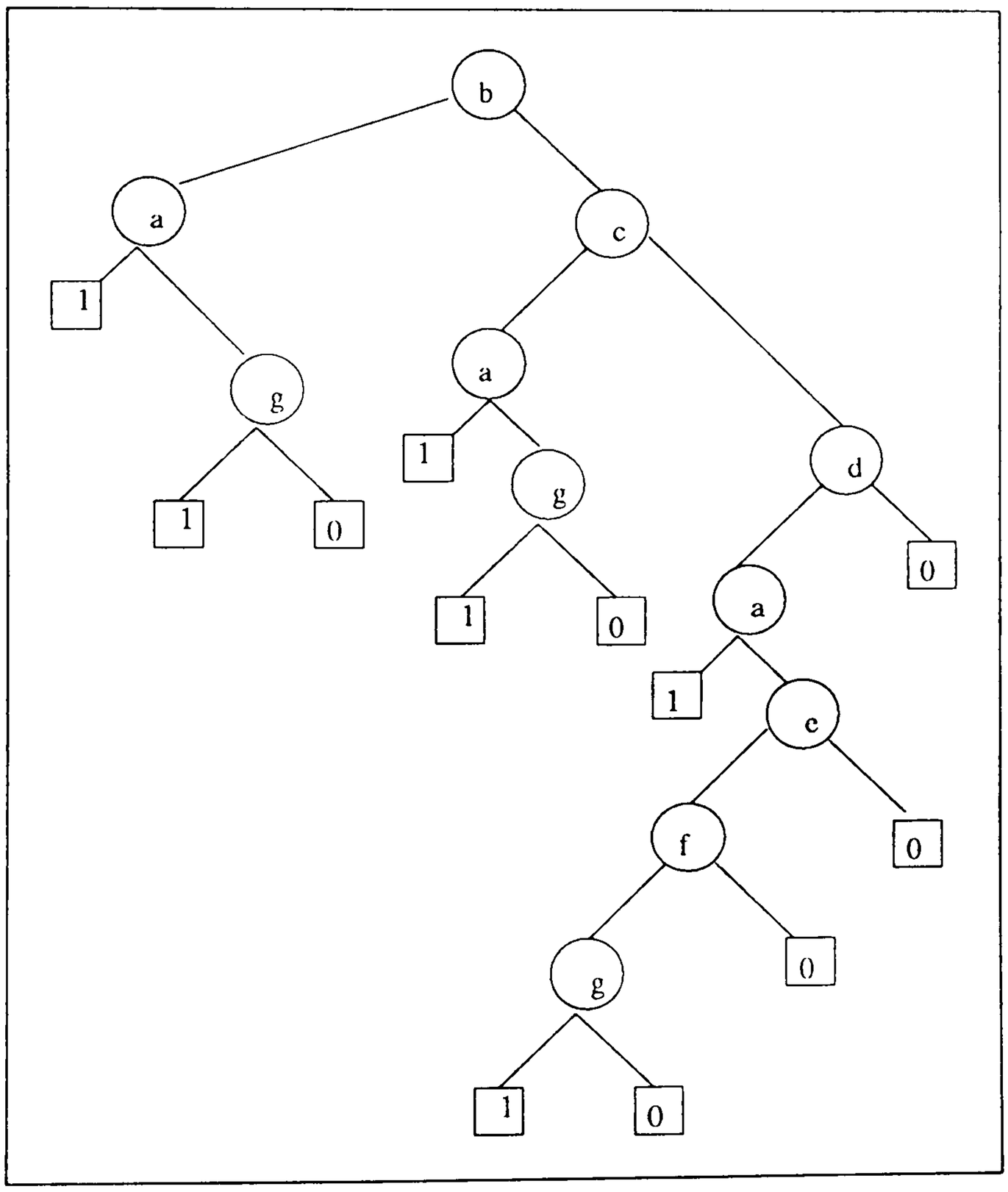


Figure 8.9 BDD for Fault Tree 2

This BDD is minimal and it directly provides the minimal cut sets:

(1) {b, a}
(2) {b, g}

(3) {c, a}

(4) {c, g}

(5) {d, a}

(6) {d, e, f, g}

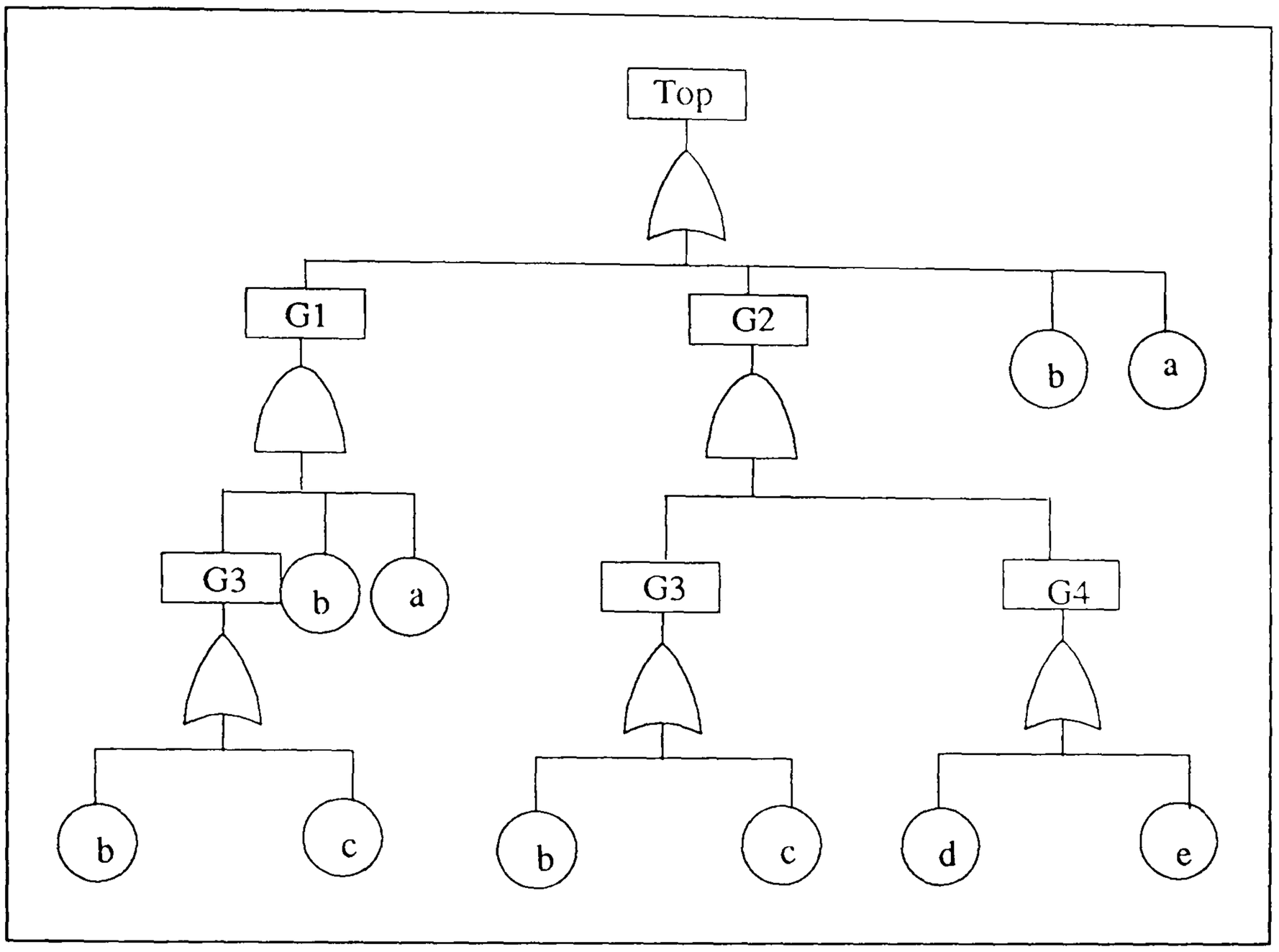


Figure 8.10 Example Fault Tree 3

Example 3 - Three repeated events

The alternating sequence of AND and OR gates shown in figure 8.10 provides the fault tree structure for example 3.

Ordering Steps

(1) Event **b** occurs four times (note G3 occurs twice), event **a** occurs twice and event **c** occurs twice.

(2) Subtree 1 (G1) has the most repeated events (three repeated events), G2 has only two.

(3) Event **b** occurs at level one and at level two of G1.

(4) Order the basic events of G1 followed by G3:

b<a<c

(5) Goto (6).

(6) The ordering for subtree 2 (G2) is:

d<e

All basic events have been ordered, giving the combined ordering for fault tree 3 as b<a<c<d<e and the resulting BDD is minimal with the minimal cut sets:

(1) {b}
(2) {a}
(3) {c, d}
(4) {c, e}

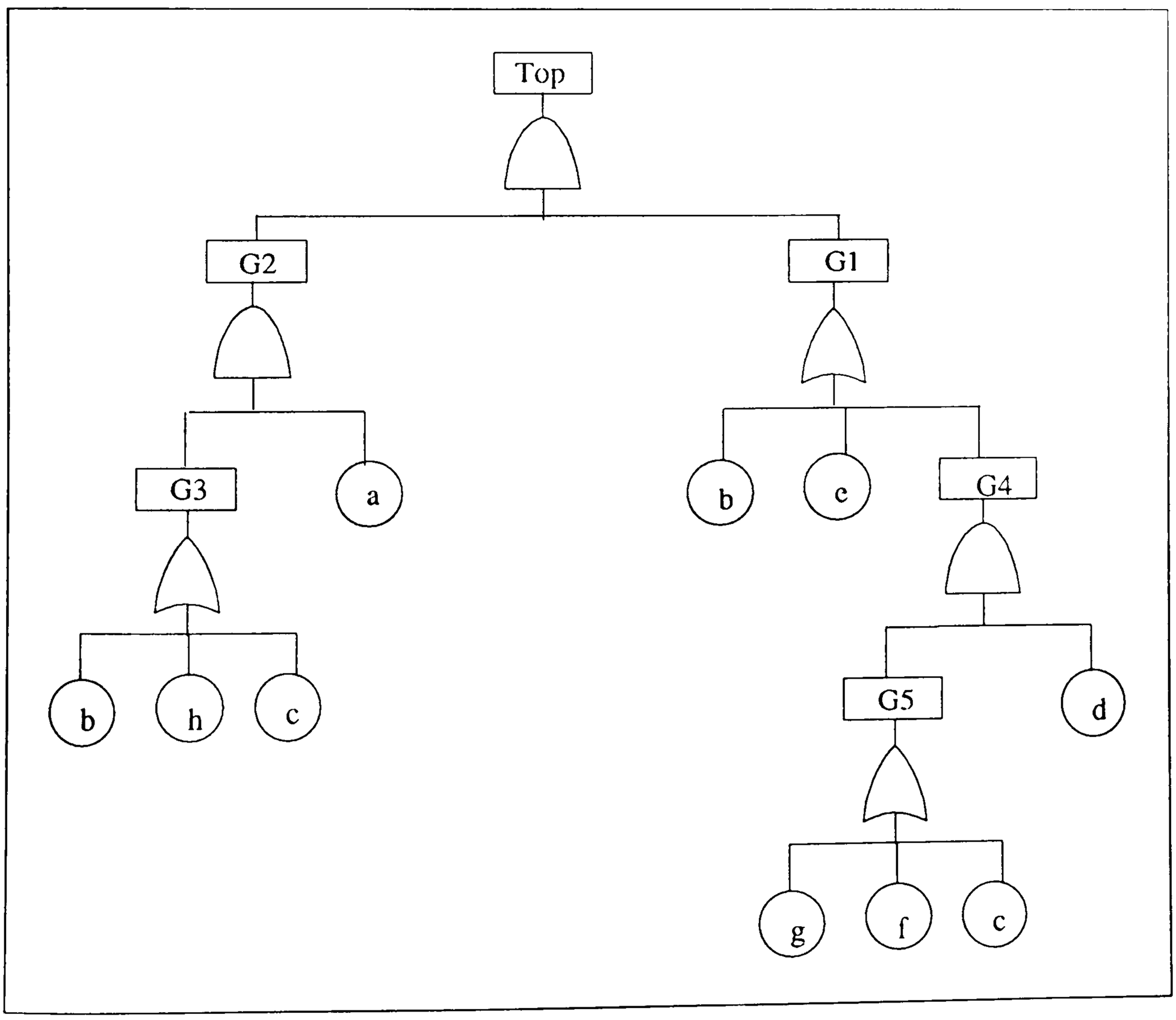


Figure 8.11 Example Fault Tree 4

Example 4 - Two repeated events with the same number of occurrences

The fault tree shown in figure 8.11 contains eight different basic events with two of these basic events repeated the same number of times.

Ordering Steps

(1) Event **b** occurs twice and event **c** occurs twice.

(2) Both subtree 1 (G2) and subtree 2 (G1) have the same number of different repeated events.

(i) Subtree 1 has two levels and subtree 2 has three levels, therefore subtree 2 takes precedence.

(3) Event **b** occurs at level one of G1.

(4) The ordering for subtree 2 is:

b<e<d<c<g<f

(5) Goto (6).

(6) The ordering for subtree 1 is:

a<h

All the basic events have been ordered and the overall ordering for this fault tree is b<e<d<c<g<f<a<h. The BDD for this ordering is minimal and produces the minimal cut sets:

(1) {b, a}
(2) {e, c, a}
(3) {e, a, h}
(4) {d, c, a}
(5) {d, g, a, h}
(6) {d, f, a, h}

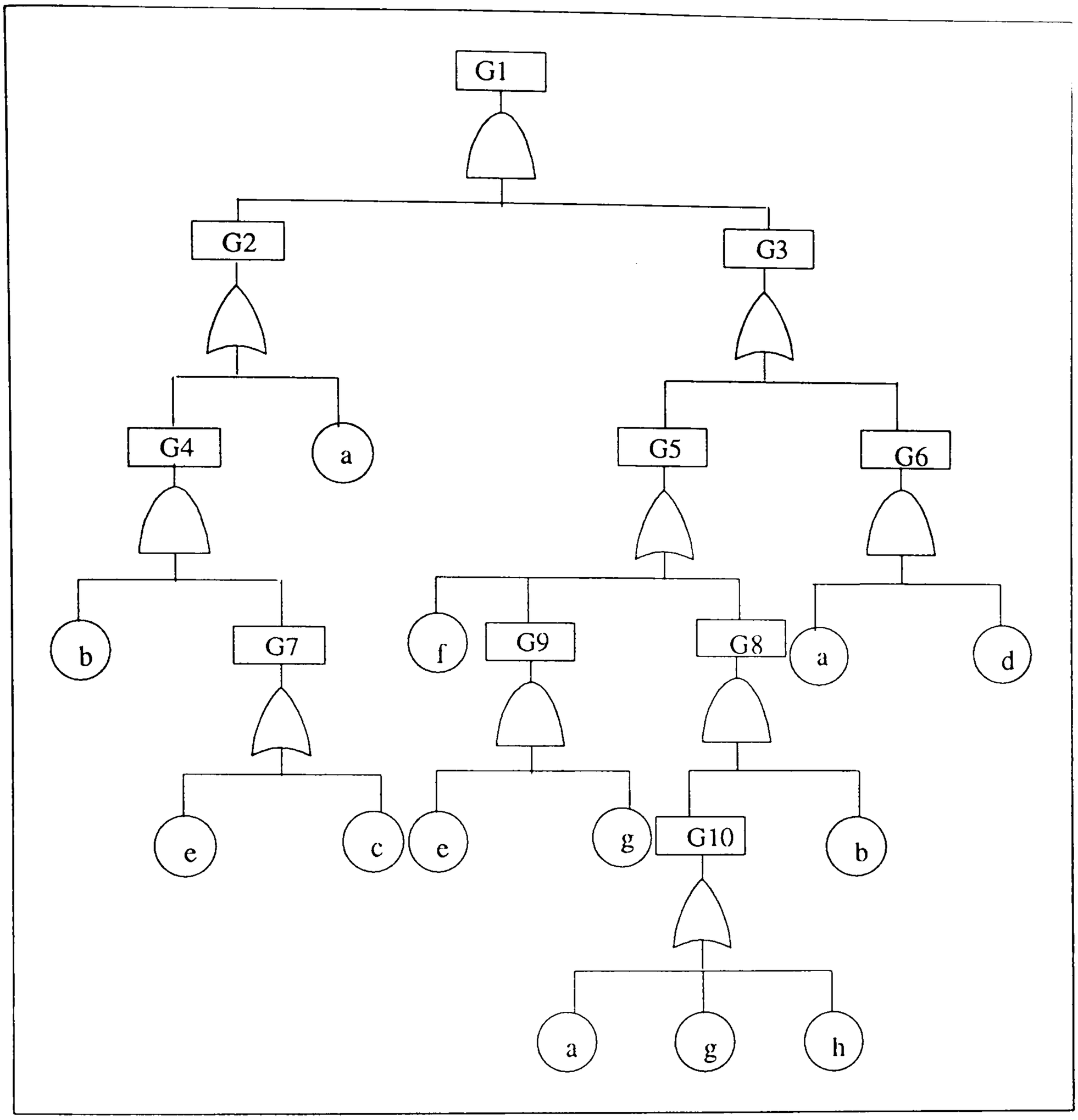


Figure 8.12 Example Fault Tree 5

Example 5 - Multiple repeated events

The fault tree shown in figure 8.12 contains eight basic events with four of these events being repeated.

Ordering Steps

(1) **a** - occurs three times

b - occurs two times

e - occurs two times

g - occurs two times

(2) Subtree 2 (G3) has the highest number of different repeated events (four), therefore it is ordered first. Subtree 1 (G2) has three different repeated events.

(3) Event **a** occurs at level two and level four of G3.

(4) G6 contains the lowest level occurrence of **a** and G5 contains the next level of **a** (here **a** is an input to G10 which in turn is an input to G8 which in turn is an input to G5), therefore take the order of gates, G6, G5, G8, G10, G9 which provides the basic event ordering:

a<d<f<b<g<h<e

(5) Goto (6).

(6) The ordering for subtree 1 provides the last basic event c.

All basic events have been dealt with and the ordering for the fault tree is a<d<f<b<g<h<e<c. Again this ordering produces a minimal BDD and the following minimal cut sets.

(1) {a, d}
(2) {a, f}
(3) {a, b}
(4) {a, g, e}
(5) {b, g, e}
(6) {f, b, c}
(7) {b, g, e}
(8) {b, h, e}
(9) {b, h, e}
(10) {b, h, c}

8.6.3 Efficiency of REBESUL Ordering

The five previous example fault trees where chosen to illustrate the ordering technique as they encountered all the options within the ordering REBESUL, therefore enabling clarification of the procedure.

Fault Tree Number	ordering 3	REBESUL ordering
1	308	**281**
2	8	**7**
3	**4**	**4**
4	**26**	**26**
5	**20**	**20**
6	41	**38**
7	**63**	**63**
8	**61**	**61**
9	**60**	**60**
10	**40**	**40**
11	5	**4**
12	**61**	62
13	**20**	**20**
14	22	**22**
15	**33**	**33**
16	335	**201**
17	647	**506**
18	394	**252**
19	**11**	**11**
20	12	**10**
21	**104**	122
22	59	**52**
23	**162**	179
24	**42**	**42**
25	**475**	550
26	**7**	**7**
27	**7**	**7**
28	**21**	**21**
29	**19**	**19**
30	**21**	**21**
31	**366**	491
32	**39**	60
33	**38**	46
34	7	**6**
35	**4**	**4**
36	-	**8762**
37	**6**	**6**
38	**413**	501
39	**4**	**4**
40	**4**	**4**
41	**8**	**8**
42	**5**	**5**
43	**2**	**2**
44	**4**	**4**
45	**14**	16
46	390	**382**
47	6	**4**
48	8	**7**
49	**6**	**6**
50	**7**	**7**
51	**12**	13
	37	**41**

Table 8.9 Comparing the Number of Nodes for Ordering REBESUL Ordering

Number	ite calculations before min	ite calculations after min	differ-ence
1	1452	2151	699
2	**26**	**26**	**0**
3	**10**	**10**	**0**
4	75	102	27
5	66	123	57
6	75	93	18
7	315	399	84
8	305	385	80
9	299	377	78
10	204	243	39
11	**11**	**14**	**3**
12	178	180	2
13	75	88	13
14	95	110	15
15	147	173	26
16	441	611	170
17	1268	2051	783
18	1546	1813	267
19	34	39	5
20	**27**	**27**	**0**
21	562	764	202
22	193	239	46
23	859	1058	199
24	154	197	43
25	3393	4668	1275
26	**17**	**17**	**0**
27	**23**	**23**	**0**
28	**67**	**67**	**0**
29	**55**	**55**	**0**
30	57	77	20
31	1456	2997	1541
32	338	430	92
33	129	207	78
34	20	24	4
35	11	14	3
36	17665	47683	30018
37	**12**	**12**	**0**
38	1400	2469	1069
39	**13**	**13**	**0**
40	**15**	**15**	**0**
41	**23**	**23**	**0**
42	**15**	**15**	**0**
43	**12**	**12**	**0**
44	**18**	**18**	**0**
45	41	53	12
46	4047	5063	1016
47	**11**	**11**	**0**
48	**17**	**17**	**0**
49	**17**	**17**	**0**
50	**21**	**21**	**0**
51	**30**	**30**	**0**

Table 8.10 The Number of **ite** Calculations Before and After Minimising the BDD for BEBESUL Ordering

Moreover they illustrate the efficiency of the ordering scheme as the REBESUL ordering directly produced a minimal BDD for each fault tree without the need to apply the minimising algorithm.

Referring back to section 8.3 we saw in table 8.5 that when comparing the orderings of the fifty-one benchmark fault trees, ordering 3 (priority-depth-first) provided the largest number of optimum BDD's.

Therefore the nodes of the fifty-one BDD's for ordering 3 have been compared to the REBESUL ordering, to illustrate which ordering is more efficient and the results are given in table 8.9.

It is important to note that fault tree 36 could not be analysed with ordering 3 but with the REBESUL ordering this fault tree could be successfully analysed obtaining 46,188 minimal cut sets in 2.49s (CPU time). Additionally the number of **ite** calculations before and after minimising the BDD have been entered in table 8.10 to give an indication of how many BDD's are directly minimal. The results show that nineteen out of the fifty-one fault trees are minimal.

8.6.4 Optimum Ordering for Fault Trees with No Repeated Events

Occurring less frequently are fault trees which have no repeated events. For such fault trees the research has shown that a depth-first ordering of the basic events will enable an efficient computation of the **ite** structures for each gate. The reason for this is that the gates are dealt with in a bottom-up level to level manner and at each level the basic events are then compatible with the level of computation. Refer to figure 8.13 for an example fault tree.

The lowest to the highest level, depth-first ordering gives, a<e<f<b<c<d. This ordering results in a minimal BDD with the minimal cut sets:

(1) {a, e, b}
(2) {a, e, c, d}
(3) {a, f, b}
(4) {a, f, c, d}

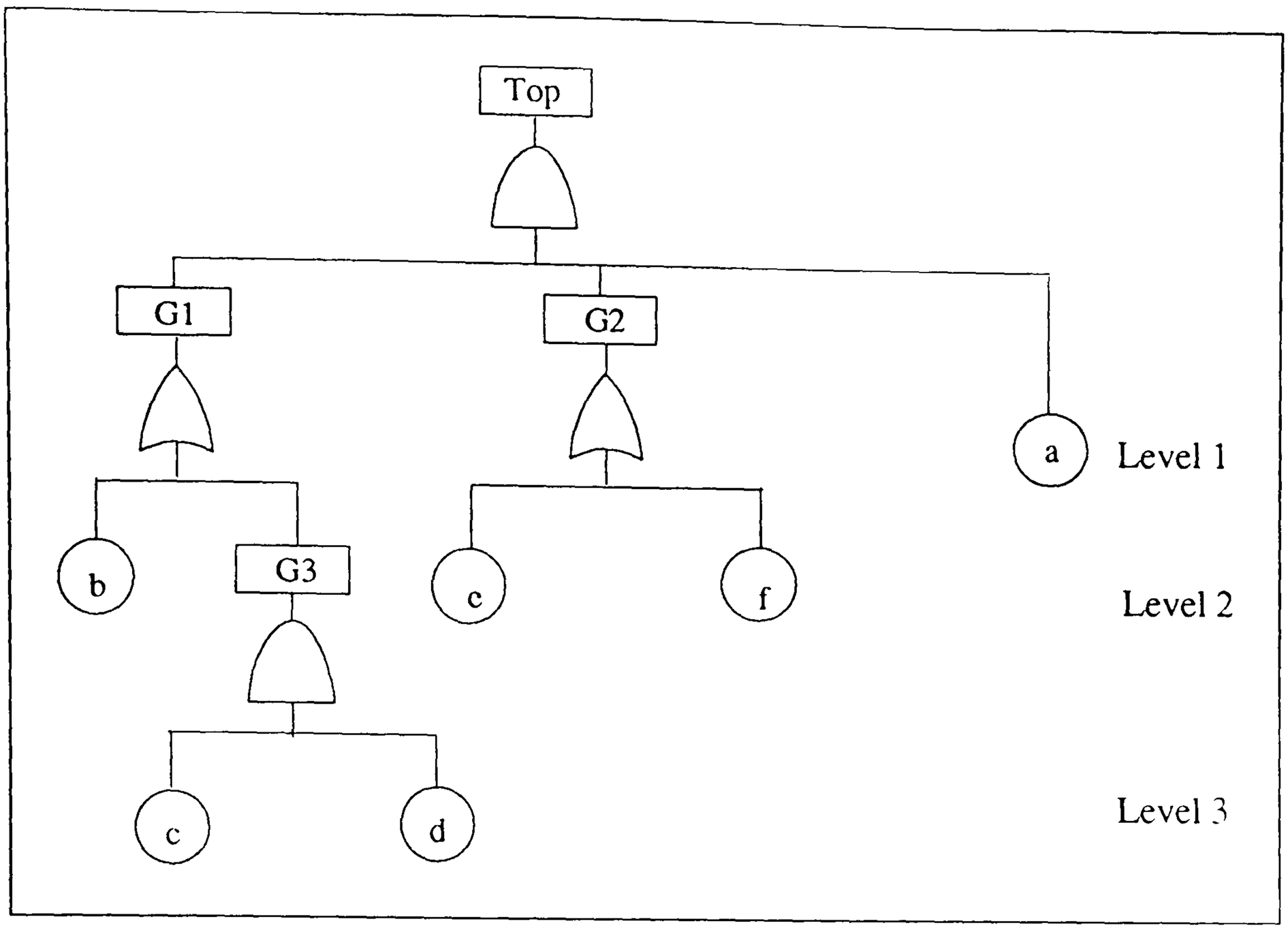


Figure 8.13 Example Fault Tree with No Repeated Events

8.7 Summary

From the investigation on the fifty-one benchmark fault trees there does not appear to be a general ordering scheme that will be 'best' for all trees. Bryant (34) recognised the problem of computing an ordering that minimises the size of the BDD and stated that for some trees it may not be possible to produce a minimal BDD whatever the ordering. In this case a "near-minimal" ordering would be required. Out of all the orderings considered the REBESUL ordering gave the most promising results in terms of efficiency of the BDD analysis.

To make even further improvements concerning the ordering of the basic events a more sophisticated ordering scheme maybe required. One approach could possibly involve the use of neural networks, where a 'learning' process could be evoked due to pattern recognition between fault trees and their optimal ordering schemes. As a result a suitable ordering scheme for each fault tree could be automatically selected based on past experiences.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

9.1 Summary of Work

After an extensive critical literature review of techniques for fault tree analysis the qualitative binary decision diagram method of Rauzy (35) to analyse fault trees was considered worthy of further investigation. The successful computer implementation of both top-down and bottom-up approaches by the author of this thesis together with criticisms in the literature highlighted the limitations of these conventional techniques which were therefore not further considered.

The binary decision diagram (BDD) approach to produce the minimal cut sets of the fault tree has been successfully demonstrated by its manual application to example test case fault trees. A computer program, BADD has been developed which implements this procedure. The program requires an input data file, containing the fault tree structure, in the form of the connectivity of the fault tree. If both a qualitative and quantitative assessment of the fault tree is required then the program needs an additional input file which provides the reliability data for the components in the fault tree.

The computational BDD method was tested against fifty-one benchmark fault trees and the results compared to the qualitative analysis using a state of the art commercial fault tree analysis package. This comparison illustrated the benefits to be gained in terms of speed of computation and memory requirements when the BDD method is used. These test results indicated that the BDD method did fulfil its potential improvements in qualitative fault tree analysis. However if it is to be considered a serious rival to the traditional fault tree analysis techniques it must be capable of performing the quantification of all parameters which can be determined by Kinetic Tree Theory (1).

The BDD approach was then extended to perform the quantification of the top event occurrence parameters. This included the exact calculation of the probability of the top event occurrence, $Q_{sys}(t)$ and the system unconditional failure intensity, $w_{sys}(t)$. The calculation of these parameters using the BDD proved superior to the approximations obtained using the kinetic tree theory approach. The results of test

case fault trees showed that the BDD method allowed faster and more accurate results to be obtained when compared to the kinetic tree theory procedure.

The use of the BDD structure for quantification purposes was further developed to obtain the importance measures concerning the basic events in the fault tree. The computational method to calculate these importance measures proved successful and illustrated the efficiency of the technique as only one pass of the BDD structure is necessary.

The technique was then further extended to incorporate the initiating and enabling event concepts in the top event quantification along with relevant importance measures.

A feature of the BDD method which required yet further investigation was that of the ordering used for the basic events in the fault tree. It was shown during the manual application of the BDD method that the choice of basic event ordering greatly influenced the size of the BDD. Thus it is advantageous to use an ordering scheme which reduced the BDD size, as a smaller BDD would increase the speed of the analysis and reduce the memory requirements. Therefore six different ordering schemes were investigated and successfully programmed with the aim to reduce the size of the BDD. These ordering schemes were then applied to the fifty-one benchmark fault trees. From the results of the comparison of these ordering schemes promising features of the 'best' ordering schemes were retained and these were incorporated into the development of a more sophisticated ordering scheme called REBESUL which proved more effective than any previous ordering scheme considered.

9.2 Conclusions

1. The BDD method has been shown to overcome some of the disadvantages of conventional fault tree analysis procedures in determining the minimal cut sets. The nature of the BDD structure is such that it lends itself to efficient Boolean manipulation i.e. minimising the BDD structure to obtain the minimal cut sets avoids the vast number of event comparisons required when applying the Boolean reduction laws to obtain minimal cut sets using conventional fault tree analysis methods.

2. The BDD method is capable of evaluating the full range of top event parameters and component importance measures. For top event quantification the BDD approach does not require the prior determination of the minimal cut sets unlike the Kinetic Tree Theory method. This reduces computation time and memory requirements.

3. The application of the BDD approach to benchmark test cases has shown that this method improves efficiency over the whole fault tree qualitative and quantitative analysis process.

4. In addition to improved efficiency the BDD method also improves the accuracy of quantitative analysis as it enables the calculation of exact values.

5. Guidelines on how to order the basic events in the fault tree have been established and justified to provide a good level of efficiency over a wide range of fault tree structures.

9.3 Other Applications of the Binary Decision Diagram

Optimal safety system performance can be obtained using the fault tree analysis method to determine the availability of each feasible system design. The use of the binary decision diagram has been applied to determine the optimal performance of a safety system (58) in conjunction with a genetic algorithm.

During the optimisation phase of the design scheme it is required to derive the system failure probability and unconditional failure intensity for a large number of potential design variations. To ensure that all designs can be analysed in the most efficient manner the fault tree structure representing the High Integrity Protection System (HIPS) in the paper by Andrews and Pattison (58) was converted to a binary decision diagram using the program BADD described in this thesis. All potential designs were incorporated into the fault tree structure using house events. Analysis of the BDD has been shown to be much faster than the quantification of the fault tree structure itself.

9.4 Future Work

9.4.1 A Different Minimising Process

The minimising process employed by Rauzy (35) is applied to the BDD once the top event **ite** structure has been computed. However it is evident that redundancies are developed as the **ite** structures are constructed for each gate in the fault tree. This is best illustrated by an example, refer to figure 9.1.

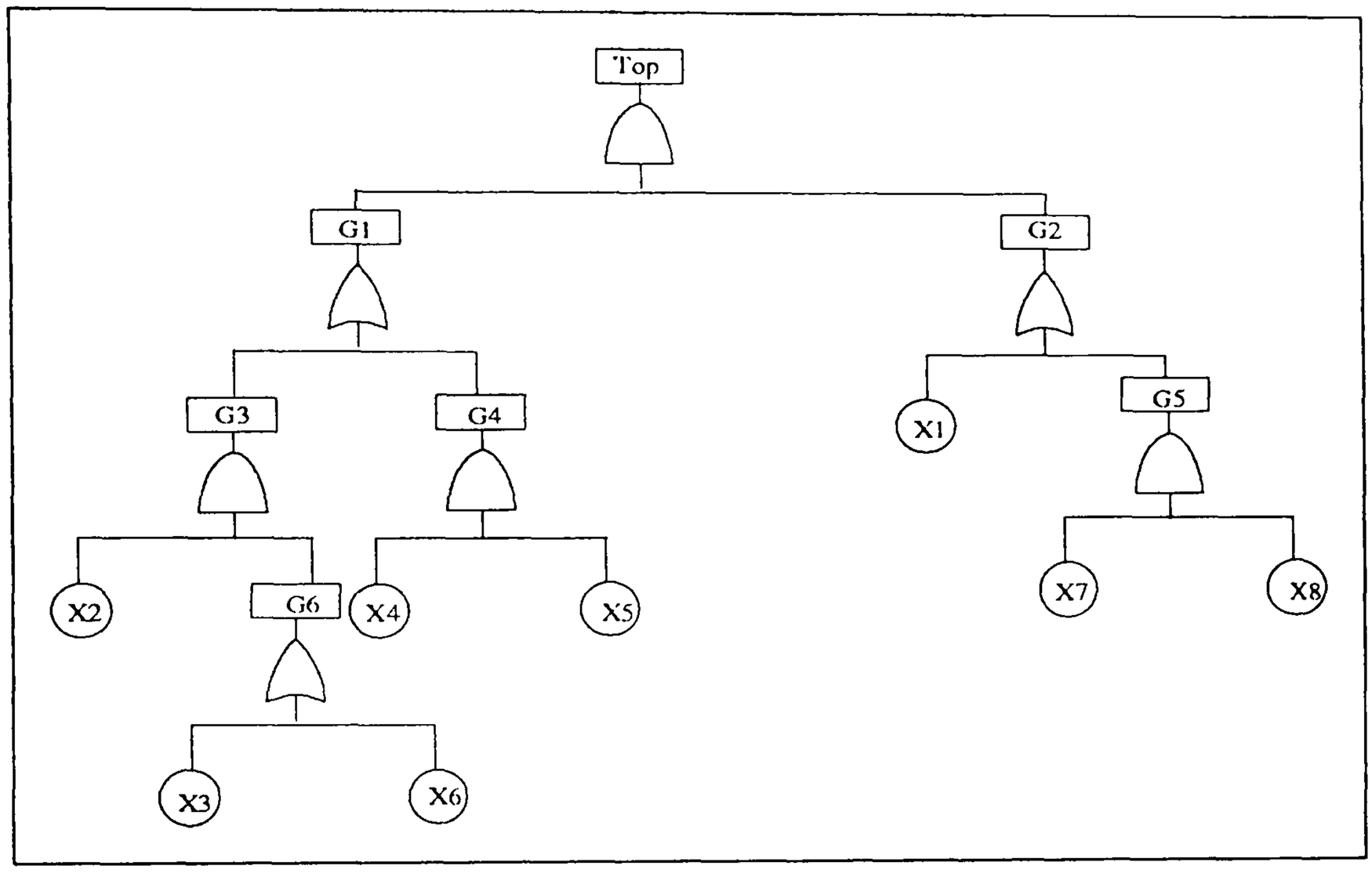


Figure 9.1 Example Fault Tree to Illustrate an Alternative Minimising Procedure

To formulate the **ite** structure of the fault tree shown in figure 9.1 the following depth-first, left-right ordering is used:

X2<X3<X6<X4<X5<X1<X7<X8

G6=**ite**(X3, 1, 0)+**ite**(X6, 1, 0)
=**ite**(X3, 1, **ite**(X6,1,0))

G4=**ite**(X4, 1, 0).**ite**(X5, 1, 0)
=**ite**(X4, **ite**(X5, 1, 0), 0)

G3=X2.G6
=**ite**(X2, 1, 0).**ite**(X3, 1, **ite**(X6,1,0))
=**ite**(X2, **ite**(X3, 1, **ite**(X6, 1, 0)), 0)

G1=G3+G4

=**ite**(X2, **ite**(X3, 1, **ite**(X6, 1, 0)), 0)+**ite**(X4, **ite**(X5, 1, 0), 0)

Following the usual procedure one would obtain the **ite** structure for G1 as:

ite(X2, **ite**(X3, 1, **ite**(X6, 1, **ite**(X4, **ite**(X5, 1, 0), 0))), **ite**(X4, **ite**(X5, 1, 0), 0))

However the BDD for this gate is non-minimal, refer to figure 9.2.

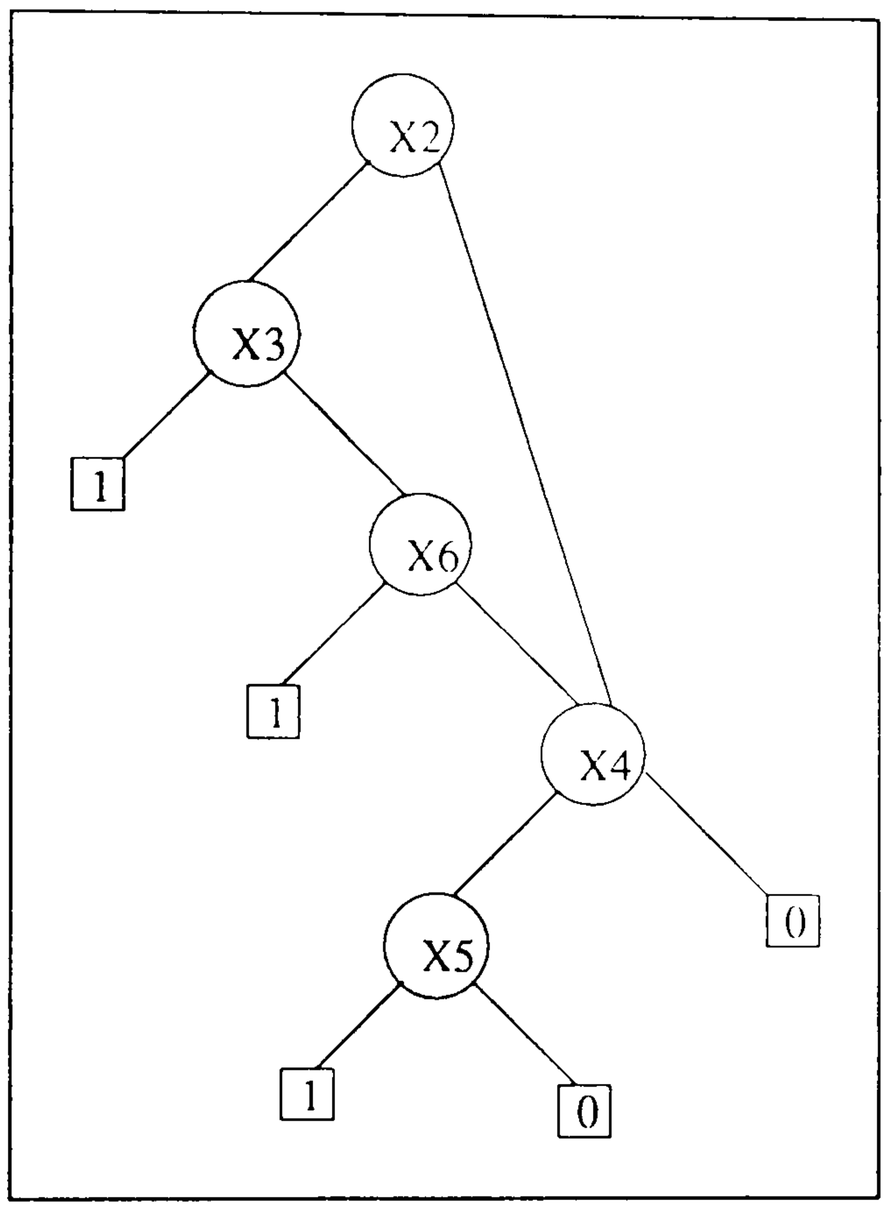


Figure 9.2 BDD for Gate G1

The cut sets obtained from figure 9.2 are:

(1) {X2, X3}
(2) {X2, X6}
(3) {X2, X4, X5}
(4) {X4, X5}

Here cut set (3) is redundant. This redundancy can be avoided by employing an alternative **ite** construction which involves a different minimising procedure. Reconsider the **ite** formulation of G1.

G1=G3+G4

G1=**ite**(X2, **ite**(X3, 1, **ite**(X6, 1, 0)),**0**)+**ite**(X4, **ite**(X5, 1, 0), 0)

Here it is obvious that the resulting right branch (0 branch) of this 'OR' operation will be minimal due to the **0** in the **ite** structure for G3, therefore the left branch (1 branch) which results from, **ite**(X3, 1, **ite**(X6, 1, 0))+**ite**(X4, **ite**(X5, 1, 0), 0) will clearly be non-minimal as **ite**(X4, **ite**(X5, 1, 0), 0) is a solution of the 0 branch. Hence the minimal **ite** for G1 is:

ite(X2, **ite**(X3, 1, **ite**(X6, 1, 0)), **ite**(X4, **ite**(X5, 1, 0), 0))

This leads to the following theorem.

Theorem 1

If F=**ite**(x, F1, 0) and G=**ite**(y, G1, G2) where x<y then
F<op>G=**ite**(x, F1, G) where <op> is the OR operation represented by +.

Continuing with the **ite** construction of the right hand side of the fault tree in figure 9.1:

G5=**ite**(X7, 1, 0).**ite**(X8, 1, 0)
=**ite**(X7, **ite**(X8, 1, 0), 0)

G2=X1+G5
=**ite**(X1, 1, 0)+**ite**(X7, **ite**(X8, 1, 0), 0)
=**ite**(X1, 1, **ite**(X7, **ite**(X8, 1, 0), 0))

Lastly:

Top=G1.G2
=**ite**(X2, **ite**(X3, 1, **ite**(X6, 1, 0)), **ite**(X4, **ite**(X5, 1, 0), 0)).**ite**(X1, 1, **ite**(X7, **ite**(X8, 1, 0), 0))
=**ite**(X2, B, C)

where:

B=**ite**(X3, **ite**(X1, 1, **ite**(X7, **ite**(X8, 1, 0), 0)),
ite(X6, **ite**(X1, 1, **ite**(X7, **ite**(X8, 1, 0), 0)), 0))

and

C=**ite**(X4, **ite**(X5, **ite**(X1, 1, **ite**(X7, **ite**(X8, 1, 0), 0)), 0), 0), 0).

The BDD for Top is illustrated in figure 9.3.

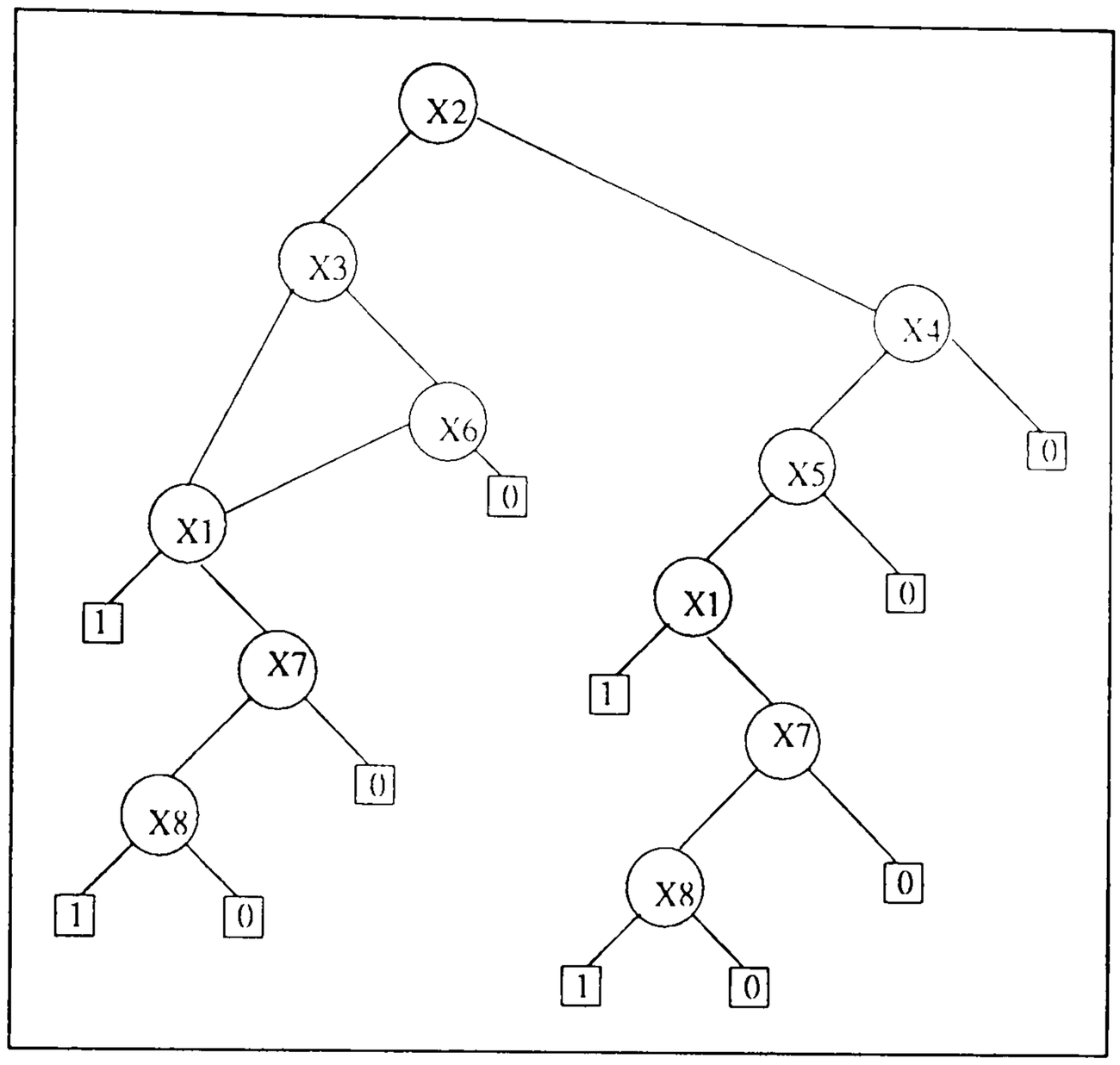


Figure 9.3 Minimal BDD for Fault Tree in Figure 9.1

The BDD in figure 9.3 results in the minimal cut sets:

(1) {X2, X3, X1}
(2) {X2, X3, X7, X8}
(3) {X2, X6, X1}
(4) {X2, X6, X7, X8}
(5) {X4, X5, X1}
(6) {X4, X5, X7, X8}

If the usual **ite** procedure had been undertaken the resulting BDD would have created two redundant cut sets, {X2, X4, X5, X1} and {X2, X4, X5, X7, X8}, therefore this alternative minimising technique has increased the efficiency of the **ite** procedure.

This alternative form of minimising when an OR gate is encountered caters for fault trees that have not got any repeated events. Applying this minimising procedure to

fault trees containing repeated events results in a loss of some of the minimal cut sets. Therefore further research into an alternative form of minimising is required for such fault trees. Additionally an alternative form of minimising when an AND gate is encountered could be developed.

For fault trees with repeated events it may prove more beneficial in terms of computational efficiency to apply the minimisation procedure of Rauzy to each gate structure, during the **ite** construction before reaching the top event, to simplify the overall process.

9.4.2 Computer Implementation of Modularising the Fault Tree

It has been demonstrated in section 4.8 that modularising the fault tree before applying the BDD technique can reduce the complexity of the problem. Therefore modularising the fault tree is a feature which could be incorporated into the BDD program to increase efficiency.

9.4.3 Ordering of the Basic Events using Neural Networks or Genetic Algorithms

It has already been mentioned in Chapter 8 that a feasible approach to creating an 'optimal' or at least a good ordering scheme for the basic events of each fault tree could result from the use of a neural network. One could 'teach' the neural network to choose the best ordering for each particular fault tree that needs to be analysed. Alternatively, the use of genetic algorithms may be the way forward to determine a basic event ordering scheme which will optimise the size of the BDD for a certain fault tree.

9.4.4 Quantification of Non-Coherent Fault Trees

Group Aralia (41) demonstrated that the BDD technique could be applied to analyse non-coherent fault trees. However work on quantifying the fault tree to find, in addition to the top event probability, such parameters as the unconditional failure intensity and all the component importance measures has yet to be undertaken. This

would prove that the BDD technique for analysing fault trees can provide a full qualitative and quantitative analysis for both coherent and non-coherent fault trees.

REFERENCES

(1) W. E. Vesely, "A Time-Dependent Methodology for Fault Tree Evaluation," *Nuclear Design and Engineering,* vol. 13, 1970, pp337-360.

(2) C. Dunglinson and H. Lambert, "Interval Reliability for Initiating and Enabling Events," *IEEE Trans. Reliability,* vol. R-32, No. 2, 1983 June.

(3) Z. W. Birnbaum, "On the Importance of Different Components in a Multicomponent System," in *Multivariate Analysis 11*, P. R. Krishnaiah (Ed), Academic Press, 1969.

(4) R. E. Barlow and F. Proschan, "Importance of System Components and Fault Tree Events," *Stochastic Processes and their Applications,* Vol. 3, 1975.

(5) D. F. Hassl, N. H. Roberts, W. E. Vesely, F. F. Goldberg, *"Fault Tree Handbook"*, US Nuclear Regulatory Commission, 1981, NUREG-0492.

(6) J. D. Andrews and T. R. Moss, *"Reliability and Risk Assessment,"* Longman Scientific and Technical, UK, 1993.

(7) E. J. Henley and H. Kumamoto, *"Reliability Engineering and Risk Assessment,"* Englewood Cliffs, 1981.

(8) J. I. Ansell and M. J. Phillips, *"Practical Methods for Reliability Data Analysis"*, Oxford Science Publications, Clarendon Press, Oxford 1994.

(9) M. Astolfi, S. Contini, C. L. Van den Muyzenberg, G. Volta, *"SALP-3, A Computer Program for Fault Tree Analysis Description and How-to-use"*, JRC Ispra Establishment, EUR 6185 En, 1978.

(10) N. Liminios, R. Ziani, "An algorithm for reducing the minimal cut sets in fault tree analysis," *IEEE Trans. Reliability,* vol R-35, No. 5, 1986 Dec, pp 559-561.

(11) R.J. Nelson, "Simplest normal truth functions", *J. Symbol. Logic,* vol. 20, pp 105-108, June 1954.

(12) R.G. Bennetts, "On the analysis of fault trees," *IEEE Trans. Reliability,* vol R-24, No. 3, 1975 Aug, pp 175-185.

(13) R.B. Worrell, "Using the set equation transformation system in fault tree analysis," *Reliability And Fault Tree Analysis. Theoretical and Applied Aspects Of System Reliability And Safety Assessment.* SIAM. Philadelphia. 1975, pp 165-185.

(14) B. L. Hulme and R. B. Worrell, "A prime implicant algorithm with factoring," *IEEE trans. Computers*, vol c-24, 1975 Nov, pp1129-1131.

(15) E. Mendelson, *"Boolean Algebra and Switching Circuits"*, McGraw-Hill, 1970.

(16) H. Kumamoto, E. J. Henley, "Top-down algorithm for obtaining prime implicant sets of non-coherent fault trees," *IEEE Trans. Reliability*, vol R-27, No. 4, 1978 Oct, pp242-249.

(17) K. Nakashima, Y. Hattori, "An efficient bottom-up algorithm for enumerating minimal cut sets of fault trees," *IEEE Trans. Reliability,* vol R-28, No. 5, 1979 Dec, pp 353-357.

(18) R. B. Worrell, D. W. Stack, B. L. Hulme, "Prime implicants of non coherent fault trees," *IEEE Trans. Reliability*, vol R-30, No. 2, 1981 Jun, pp98-100.

(19) S. N. Semanderes, "'ELRAFT,' A computer program for the efficient logic reduction analysis of fault trees," *IEEE Trans. Nuclear Science,* vol NS-18, 1971 Feb, pp 481-487.

(20) J. B. Fussell, W. E. Vesley, "A new methodology for obtaining cut sets for fault trees," *Trans. Am. Nucl. Soc.,* vol 15, 1972 Jun, pp 262-263.

(21) M. Astolfi, S. Contini, C. L. Van den Muyzenberg, G. Volta, "Fault tree analysis by list processing techniques," (1978). In book by Apostolakis, Garribba and Volta *'Synthesis and analysis methods for safety and reliability studies',* Plenum Press, New York and London (1980).

(22) W. G. Schneeweiss, *"Boolean Functions with Engineering Applications and Computer Programs,"* Springer-Verlag, Berlin, 1989.

(23) B. Harris, *"Theory of Probability,"* Addison-Wesley, 1966.

(24) T. Inagaki and E. J. Henley, "Probabilistic Evaluation of Prime Implicants and Top-Events for Non-Coherent Systems," *IEEE Trans. Reliability*, vol R-29, No. 5, 1980 Dec, pp361-367.

(25) T. L. Chu and G. Apostolakis, "Methods for Probabilistic Analysis of Non coherent Fault Trees", *IEEE Trans. Reliability*, vol R-29, No. 5, 1980 Dec, pp354-360.

(26) M. O. Locks, "Recursive Disjoint Products, Inclusion-Exclusion and Min-Cut Approximations," *IEEE Trans. Reliability*, vol R-29, No. 5, 1980 Dec, pp368-371.

(27) J. A. Abraham, "An improved algorithm for Network Reliability", *IEEE Trans. Reliability*, vol R-28, 1979 April, pp58-61.

(28) N. N. Bengiamin, B. A. Bowen, K. F. Schenk, "An efficient algorithm for reducing the complexity of computation in fault tree analysis," *IEEE Trans. Nuclear Science,* vol NS-23, No. 5, 1976 Oct, pp 1442-1446.

(29) D. B. Wheeler, J. S. Husan, R. R. Duersch, G. M. Roe, "Fault Tree Analysis Using Bit Manipulation", *IEEE Trans. Reliability,* vol R-26, No. 2, June 1977, pp 95-98.

(30) D. M. Rasmuson, N. H. Marshall, "FATRAM-A core efficient cut-set algorithm," *IEEE Trans. Reliability,* vol R-27, No. 4, 1978 Oct, pp 250-253.

(31) G. Zipf, "Computation of minimal cut sets of fault trees: Experiences with three different methods," *Reliability Engineering,* vol 7, 1984, pp 159-167.

(32) W. Güldner, H. Polke, H. Spindler, G. Zipf, "Programmsystem RALLY-Zur probabilistischen Sicherheitsbeurteilung groβer technischer Systeme", GRS-44, March 1982.

(33) S. B. Akers, "Binary decision diagrams," *IEEE Trans. Computers,* vol C-27, No. 6, 1978 Jun, pp509-516.

(34) R. E. Bryant, "Graph-Based algorithms for Boolean function manipulation," *IEEE Trans. Computers,* vol C-35, No. 8, 1986 Aug, pp677-691.

(35) A. Rauzy, "New algorithms for fault tree analysis," *Reliability Engineering and System Safety,* vol 40, 1993, pp203-211.

(36) C. Lee, "Representation of switching circuits by binary decision diagrams," *Bell Syst. Tech. J.,* No.38, pp. 985-999, July 1959.

(37) S. J. Friedman and K. J. Supowit, "Finding the Optimal Variable Ordering for Binary Decision Diagrams," *IEEE Trans. on Computers*, vol 39, No.5, 1990 May, pp710-713.

(38) K. M. Butler, D. E. Ross, R. Kapur, M. R. Mercer, "Heuristics to Compute Variable Orderings for Efficient Manipulation of Ordered Binary Decision Diagrams," *IEEE Design Automation Conf. 28th Proc.,* San Francisco, pp417-420.

(39) M. Chevalier, Y. Dutuit, A. Lapassat, A. Laviron, I. Morlaes, A. Rauzy, J. Signoret (Groupe Aralia), "Abres Des Défaillances Et Diagrammes De Décision Binaires," *Actes du 1er Congrès Interdisciplinaire sur la Qualité et la Sûreté de Fonctionnement,* pp47-56 Tome 2, 1994, Paris (Compiègne).

(40) T. Kohda, E. J. Henley, K. Inoue, "Finding modules in fault trees," *IEEE Trans. Reliability*, vol 38, No.2, 1989 Jun, pp165-176.

(41) Groupe Aralia (LaBRI-LADS), Université Bordeaux, "Computation of Prime Implicants of a Fault Tree Within Aralia," *Proceedings of ESREL'95 Conf.,* Bournemouth, June, pp190-202.

(42) O. Coudert and J. C. Madre, "A New Method to Compute Prime and Essential Prime Implicants of Boolean Functions," In T. Knight and J. Savage, editors, *Advanced Research In VLSI and Parallel Systems,* pp113-128, March 1992.

(43) *Fault Tree Analysis and Graphics Program (FAULTREE+),* by Isograph Ltd 1996, Manchester, UK

(44) O. Platz and J. V. Olsen, "FAUNET: A Program Package for Evaluation of Fault Trees and Networks," *Research Establishment, Risø Report No. 348,* DK-4000 Roskilde, Denmark, Sept. 1976.

(45) R. M. Sinnamon and J. D. Andrews, "Improved Efficiency in Qualitative Fault Tree Analysis", *Proceedings of the 12th Arts, Advances in Reliability Technology Symposium,* Manchester, April 1996.

(46) A. Rosenthal, "Decomposition methods for fault tree analysis," *IEEE Trans. Reliability,* vol R-29, No.2, 1980 Jun, pp136-138.

(47) R. R. Willie, *"Computer-Aided Fault Tree Analysis",* Operations Research Centre, University of California, Berkeley, Order No. 7800103, Aug. 1978.

(48) R. M. Sinnamon, "Fault Tree Analysis Codes", 1996 Supplement Report to PhD Thesis, Mathematical Sciences Department, Loughborough University, Leicestershire, UK

(49) M. O. Locks, "Modularizing, Minimizing, and Interpreting the K&H Fault-Tree", *IEEE Trans. Reliability,* vol R-30, No.5, 1981 Dec, pp411-415.

(50) J. M. Wilson, "Modularizing and Minimizing Fault Trees", *IEEE Trans. Reliability,* vol R-34, No.4, 1985 Oct, pp320-322.

(51) R. M. Sinnamon and J. D. Andrews, "Quantitative Fault Tree Analysis Using Binary Decision Diagrams", *European Journal of Diagnosis and Safety in Automation*, in press.

(52) W. G. Schneeweiss, "Fault-tree Analysis Using a Binary Decision Tree", *IEEE Trans. Reliability,* vol R-34, No.5, 1985 Dec, pp453-457.

(53) R. M. Sinnamon and J. D. Andrews, "Fault Tree Analysis and Binary Decision Diagrams", In *Proceedings of RAMS'96 Conference,* Las Vegas, Nevada, 22-25 January 1996, pp. 215-222.

(54) M. Bouissou, "An Ordering Heuristic for Building Binary Decision Diagrams from Fault-Trees", In *Proceedings of RAMS'96 Conference*, Las Vegas, Nevada, 22-25 January 1996, pp. 208-214.

(55) M. Astolfi, C. Clarotti, S. Contini, F.R. Picchia, *"SALP-MP, A Computer Program for Fault Tree Analysis of Complex Systems and Phased Missions"*, JRC Ispra Establishment, PER 389, 1980.

(56) R. M. Sinnamon and J. D. Andrews, "New Approaches to Evaluating Fault Trees", *Proceedings of ESREL'95,* Bournemouth, U.K., June, 1995, pp241-254 and *Reliability Engineering and System Safety*, in press.

(57) R. M. Sinnamon and J. D. Andrews, "Improved Accuracy in Quantitative Fault Tree Analysis", *Proceedings of the 12th Arts, Advances in Reliability Technology Symposium,* Manchester, April 1996.

(58) J. D. Andrews and R. L. Pattison, "Optimal Safety System Performance", *Proceedings of RAMS'97 Conference,* Philadelphia, January 1997.

(59) P. Chatterjee, "Modularization of Fault Trees: A Method to Reduce the Cost of Analysis", *Reliability and Fault Tree Analysis,* SIAM, Philadelphia, 1975, pp. 101-126.

APPENDIX I - *.ats File Format

The fault tree structure file with extension '*.ats' is an ASCII file with the following format.

Each line of the file contains a definition for each gate in the fault tree. The first 10 columns of a line must contain the name of a gate. The gate name must be left-justified and consist of no more than 10 alphanumeric characters. The next 7 columns represent the gate type. The type label must be left-justified and be one of the following options (can be upper or lower case).

OR
AND
VOTEm/n

VOTE gates are defined in terms of m out of n failures. For example VOTE2/4 indicates 2 out of 4 input failures will result in the gate failure.

Columns 18 and 20 are used to indicate the number of gate and event inputs. Up to 9 gate inputs and 9 events inputs are allowed. The remaining columns in each line are used to represent input names (gate and primary event). Names are left-justified in groups of 10 columns.

An example tree structure file is shown below.

```
Gate1     AND        1 1Gate2     X1
Gate2     VOTE2/4    1 4Gate3     X2        X3        X4
Gate3     OR         0 2X2        X4
```

Note that all gates named as inputs must themselves be defined in the file. Gates may be defined in any order.

APPENDIX II - *.aqd File Format

The failure data file with the extension '.aqd' is an ASCII file with the following format.

Each event and its associated data is defined on two lines. The first line specifies the event name and failure model type. The second line specifies the model parameters. There are 2 or 3 model parameters depending on the model type. The event name must be left-justified on the first 10 columns and the model type indicator in the 11th column. The model parameters are specified in free format.

Valid model type indicators are:

F: Fixed unavailability and unconditional failure intensity
R: Constant failure and repair rate
M: Mean time to failure and repair
P: Dormant failure with periodic inspection

The model parameters (in the order required) for each model type are given below.

F: Unavailability, unconditional failure intensity
R: Failure rate, repair rate
M: Mean time to failure, mean time to repair
P: Failure rate, inspection interval, mean time to repair

Consider the following example of the definition of an event model in a '.aqd' file:

```
X1      R
0.1,120
```

The first line consists of the basic event code 'X1' left-justified in the first 10 spaces of the line, followed by the model code 'R' in the 11th space. In this example the constant failure and repair rate has been chosen.

The second line indicates the quantitative parameters associated with the model type. In our example above the parameters represent failure rate (0.1) and repair rate (120). No time units need be specified. Time units are assumed to be consistent.

APPENDIX III - Dresden-3.ats File and Dresden-3.aqd File

```
1060    and    2 01058    1059
1058    and    2 01057    1055
1059    and    2 01053    1045
1057    and    3 01037    1034    1032
1055    and    3 01026    1022    1019
1053    or     2 01052    1051
1045    or     2 01044    1043
1044    or     0 249      48
1043    or     1 11042    47
1052    or     0 257      56
1051    or     1 11050    55
1042    or     2 01041    1040
1050    or     2 01049    1048
1049    or     0 254      53
1048    or     1 11047    52
1041    or     0 246      45
1040    or     1 11039    44
1047    or     2 01057    1046
1039    or     2 01055    1038
1046    or     0 250      51
1038    or     0 243      42
1037    or     2 01036    1025
1034    or     2 01033    1020
1032    or     2 01056    1031
1036    or     0 241      40
1025    or     2 11024    1035    29
1033    or     0 237      36
1020    or     0 224      23
1056    and    2 01030    1028
1031    or     0 235      34
1024    or     0 228      27
1035    or     0 239      38
1030    or     2 01029    1009
1028    or     2 01027    1017
1029    or     0 233      32
1009    or     1 11008    10
1027    or     0 231      30
1017    or     1 11015    18
1015    or     1 11014    16
1008    or     2 01007    1006
1014    or     2 01013    1012
1007    or     0 29       8
1006    or     0 27       2
1013    or     0 215      14
1012    or     1 11005    13
1005    or     2 01004    1003
1004    or     0 26       5
1003    or     2 01002    1001
1002    or     0 24       3
1001    or     0 22       1
1026    or     2 01025    1023
1022    or     2 01021    1020
1019    or     2 01054    1018
1023    or     0 226      25
1021    or     0 222      21
1054    and    2 01016    1011
1018    or     0 220      19
1016    or     1 11015    17
1011    or     1 11010    12
1010    or     1 11009    11
```

```
1               n
1,10
2               n
0.1,15
3               n
9.009,48
4               n
9.009,48
5               n
100,334
6               n
1,5
7               n
1,10
8               n
9.009,48
9               n
9.009,48
10              n
1,5
11              n
1,5
12              n
5,8
13              n
1,5
14              n
9.009,48
15              n
9.009,48
16              n
1,5
17              n
5,8
18              n
1,5
19              n
5,8
20              n
5,3
21              n
5,8
22              n
10,3
23              n
10,200
24              n
1,5
25              n
10,3
26              n
5,8
27              n
9.009,48
28              n
9.009,48
```

```
29          n
1,5
30          n
5,8
31          n
10,3
32          n
5,8
33          n
10,3
34          n
5,8
35          n
5,3
36          n
5,8
37          n
10,3
38          n
1,70
39          n
10,10
40          n
5,8
41          n
10,3
42          n
5,8
43          n
10,3
44          n
1,5
45          n
9.009,48
46          n
9.009,48
47          n
1,5
48          n
5,8
49          n
10,3
50          n
5,8
51          n
10,3
52          n
1,5
53          n
9.009,48
54          n
9.009,48
55          n
1,5
56          n
5,8
57          n
10,3
```

APPENDIX IV - Summary of Fifty-one Benchmark Fault Trees

Fault Tree Number	No. of Gates	No. of Basic Events	No. of Repeated Basic Events	No. of Minimal Cut Sets
1	79	103	39	3804
2	6	7	3	7
3	3	4	1	2
4	19	16	2	27
5	14	13	2	9
6	17	11	7	43
7	32	63	0	8,716
8	29	61	0	7,471
9	30	60	0	7,056
10	21	40	0	416
11	3	4	1	3
12	21	40	4	84,424
13	19	19	1	63
14	21	21	1	75
15	30	32	1	2,100
16	42	41	21	11,934
17	58	57	21	36,990
18	60	57	41	11,934
19	10	10	1	13
20	6	8	2	6
21	30	72	8	255
22	10	31	2	71
23	25	61	57	7,777
24	12	30	4	61
25	81	199	68	8,179
26	5	7	0	4
27	5	7	3	4
28	11	21	0	36
29	11	20	1	30
30	11	20	1	10
31	70	68	26	4,892
32	30	34	28	35
33	26	16	11	20
34	5	7	1	3
35	4	5	1	2
36	122	61	60	46,188
37	4	6	0	6
38	58	114	114	35,300
39	4	5	1	3
40	5	6	1	3
41	8	8	1	6
42	5	5	3	4
43	7	6	3	2
44	7	6	3	4
45	10	10	2	8
46	153	74	46	340
47	3	4	1	2
48	4	6	1	3
49	3	4	2	4
50	4	5	3	5
51	10	8	4	10