This item was submitted to [Loughborough's Research Repository](#) by the author.
Items in Figshare are protected by copyright, with all rights reserved, unless otherwise indicated.

Improved efficiency in qualitative fault tree analysis

PLEASE CITE THE PUBLISHED VERSION

PUBLISHER

(c) The authors

LICENCE

CC BY-NC-ND 4.0

REPOSITORY RECORD

Sinnamon, Roslyn M., and J.D. Andrews. 2008. "Improved Efficiency in Qualitative Fault Tree Analysis".
figshare. <https://hdl.handle.net/2134/3692>.

This item was submitted to Loughborough's Institutional Repository by the author and is made available under the following Creative Commons Licence conditions.

For the full text of this licence, please go to:
<http://creativecommons.org/licenses/by-nc-nd/2.5/>

Improved Efficiency in Qualitative Fault Tree Analysis

R M Sinnamon and J D Andrews
Department of Mathematical Sciences
Loughborough University of Technology

Summary

The fault tree diagram itself is an excellent way of deriving the failure logic for a system and representing it in a form which is ideal for communication to managers/designers/operators etc. Since the method was first conceived, algorithms to derive the minimal cut sets have worked directly with the fault tree diagram itself using either bottom-up or top-down approaches. These conventional techniques have several disadvantages when it comes to analysing the fault tree. For complex systems an analysis may produce hundreds of thousands of minimal cut sets, the determination of these cut sets can be a very time consuming process. Also, for large fault trees it may not be possible to evaluate all minimal cut sets, and methods to identify those event combinations which provide the most significant contributions to the system failure are evoked. Such methods include probabilistic or order culling which reduce the problem to a practical size, but can also create considerable inaccuracies when it comes to evaluating top event probability parameters.

This paper describes how the Binary Decision Diagram method can be employed to evaluate the minimal cut sets of a fault tree, efficiently and without the need to use approximations such as order culling.

1. Introduction

The Binary Decision Diagram (BDD) method was utilised by Bryant (Ref 1) and later developed by Rauzy (Ref 2) to analyse fault trees. Rather than analysing the fault tree directly as with conventional approaches (Refs 3,4), the BDD method 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.

The BDD is constructed by applying an If-Then-Else (*ite*) technique to each of the gates in the fault tree. This *ite* technique is such that the resulting *ite* structure for the top gate in the fault tree encodes Shannons' formula (Ref 5). The importance of Shannons' formula to the quantification of

the fault tree is discussed in a paper entitled "Improved Accuracy in Quantitative Fault Tree Analysis" (Ref 6).

From the BDD both the qualitative and quantitative analysis can be achieved. The size of the resulting BDD is determined by the ordering that has to be given to the basic events in the fault tree before the BDD is constructed. This ordering has further implications for the analysis. If the BDD is not in a minimal form, then it must first undergo a minimising procedure before the minimal cut sets can be obtained, this minimising technique is discussed in section 5.

2. Binary Decision Diagrams

A BDD is a directed acyclic graph. All paths through the BDD terminate in one of two states; either a 1 state, which corresponds to system failure, or a 0 state which corresponds to system success. A BDD is composed of terminal and non-terminal vertices, which are connected by branches or edges. Terminal vertices have the value 0 or 1 and non-terminal vertices correspond to the basic events of the fault tree. Each non-terminal vertex has a 0 branch, which represents the basic event non-occurrence (works), and a 1 branch, which represents basic event occurrence (fails). Consider the if-then-else structure – $\text{ite}(X1, f1, f2)$, which means **if** $X1$ fails **then** consider function $f1$ **else** consider function $f2$. Therefore, in the BDD $f1$ lies on the 1 branch of $X1$ and $f2$ on the 0 branch. The diagram for this is the one given in figure 1.

Figure 1. BDD for $\text{ite}(X1, f1, f2)$

3. Binary Decision Diagram Construction

Once the basic events in the fault tree have been given an ordering, the following procedures are then used to construct the BDD from the fault tree. Rauzy (Ref 4) 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 $\langle \text{op} \rangle$ corresponds to the Boolean operation of the logic gates in the fault tree, so if the gate is an AND gate $\langle \text{op} \rangle$ will be the dot or product symbol ($\cdot$), and if the gate is an OR gate $\langle \text{op} \rangle$ will be the sum symbol ($+$).

Procedures

(1) Assign each basic event, X_i in the fault tree the **ite** structure $\text{ite}(X_i, 1, 0)$.

(2) For each gate event: $X < Y$

Let $J = \text{ite}(X, F1, F2)$ and $H = \text{ite}(Y, G1, G2)$ then;

$J < \text{op} > H = \text{ite}(X, F1 < \text{op} > H, F2 < \text{op} > H)$

(3) If $X = Y$:

i.e., $J = \text{ite}(X, F1, F2)$ and $H = \text{ite}(X, G1, G2)$ then;

$J < \text{op} > H = \text{ite}(X, F1 < \text{op} > G1, F2 < \text{op} > G2)$

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

$1 < \text{op} > H = 1$ if $< \text{op} >$ is an OR gate

$1 < \text{op} > H = H$ if $< \text{op} >$ is an AND gate

$0 < \text{op} > H = H$ if $< \text{op} >$ is an OR gate

$0 < \text{op} > H = 0$ if $< \text{op} >$ is an AND gate

To illustrate the application of this method consider the fault tree in figure 2.

Figure 2. Example Fault Tree

Take the ordering for the basic events:

$$X1 < X2 < X3 < X4 \quad (1)$$

The gates are considered in a bottom up manner.

$$\begin{aligned} G2 &= \text{ite}(X3, 1, 0) \cdot \text{ite}(X4, 1, 0) \\ &= \text{ite}(X3, \text{ite}(X4, 1, 0), 0) \end{aligned}$$

$$\begin{aligned} G1 &= \text{ite}(X1, 1, 0) \cdot \text{ite}(X2, 1, 0) \cdot \text{ite}(X3, 1, 0) \\ &= \text{ite}(X1, \text{ite}(X2, 1, 0), 0) \cdot \text{ite}(X3, 1, 0) \\ &= \text{ite}(X1, \text{ite}(X2, 1, 0) \cdot \text{ite}(X3, 1, 0), 0) \cdot \text{ite}(X3, 1, 0) \\ &= \text{ite}(X1, \text{ite}(X2, \text{ite}(X3, 1, 0), 0), 0) \end{aligned}$$

$$\begin{aligned} \text{Top} &= G1 + G2 \\ &= \text{ite}(X1, \text{ite}(X2, \text{ite}(X3, 1, 0), 0), 0) + \text{ite}(X3, \text{ite}(X4, 1, 0), 0) \\ &= \text{ite}(X1, \text{ite}(X2, \text{ite}(X3, 1, 0), 0) + \text{ite}(X3, \text{ite}(X4, 1, 0), 0), 0) + \text{ite}(X3, \text{ite}(X4, 1, 0), 0) \\ &= \text{ite}(X1, \text{ite}(X2, \text{ite}(X3, 1, 0) + \text{ite}(X3, \text{ite}(X4, 1, 0), 0), 0) + \text{ite}(X3, \text{ite}(X4, 1, 0), 0)), \\ &\quad \text{ite}(X3, \text{ite}(X4, 1, 0), 0) \\ &= \text{ite}(X1, \text{ite}(X2, \text{ite}(X3, 1 + \text{ite}(X4, 1, 0), 0 + 0), \text{ite}(X3, \text{ite}(X4, 1, 0), 0)), \\ &\quad \text{ite}(X3, \text{ite}(X4, 1, 0), 0) \\ \text{Top} &= \text{ite}(X1, \text{ite}(X2, \text{ite}(X3, 1, 0), \text{ite}(X3, \text{ite}(X4, 1, 0), 0)), \text{ite}(X3, \text{ite}(X4, 1, 0), 0)) \end{aligned}$$

This top event **ite** structure represents the BDD shown in figure 3. Notice that the node or vertex 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.

Figure 3. BDD for fault tree shown in figure 2

Every path through the BDD starts from the root vertex, and proceeds down through the diagram to a terminal vertex. Paths which terminate at a 1 vertex yield the minimal cut sets. However only basic events for which the path leaves their vertex on a 1 branch on the way to a terminal 1 vertex are included in the cut sets. For example the cut sets, of the fault tree shown in figure 3 are:

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

Obviously the resulting BDD for this ordering is not minimum, as it produces one redundant cut set. To obtain only the minimal cut sets the BDD needs to undergo a minimising procedure.

4. Minimal Cut Set Evaluation (The 'without' operator)

In the given example the resulting BDD was not minimum as it produced a redundant cut set. Therefore the BDD must first undergo a minimising procedure to produce the minimal BDD which encodes only **minimal** cut sets. From the unminimised BDD the minimising algorithm of Rauzy (Ref 4) creates a new BDD that defines exactly the minimal cut sets of the fault tree. For example,

consider a general node F in a BDD. If $F = \text{ite}(x, G, H)$ then 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, $\text{sol}_{\min}(F)$, will include the minimal solutions of H so:

$$\text{sol}_{\min}(F) = \{\sigma\}$$

where:

$$\sigma = [\{\delta\} \cap x] \cup [\text{sol}_{\min}(H)]$$

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 apply this algorithm to the BDD in figure 3 we 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 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 $G_{\min}$ the minimal solutions of F2. In this case the solutions of G are minimum, therefore we remove from $G_{\min}$ all the paths that are included in a path of H. This is performed by removing F4 from the 0 branch of F2 and replacing it by a terminal 0 vertex [Refer to figure 4]. No other nodes alter in applying the minimising procedure which results in the BDD shown in figure 4. To increase the efficiency when applying the minimising procedure the following two results can be applied:

- (1) **without** (F, F) = { }
- (2) $\text{sol}_{\min}[\text{ite}(x, F, F)] = F$

The details of this algorithm are given in Ref 4 and Ref 7.

Figure 4. Minimum BDD

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

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

5. Conventional v BDD

To test the efficiency of the BDD method, 10 example fault trees were analysed using the BDD approach and the results compared to the analysis using a conventional Fault Tree Analysis package. A top-down ordering was used for the BDD analysis. The results are given in table 1 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.

Tree	No. of Gates	No. of basic events	No. of repeated events	No. of minimal cut sets	BDD Time (s)	FTA Time (s)	% Improvement
1	19	19	2	27	0.5	1.0	50
2	17	11	7	43	0.5	1.0	50
3	29	61	0	7,471	0.2	1.0	80
4	60	57	41	11,934	0.6	-	100
5	19	19	1	63	0.6	0.9	33
6	21	21	1	75	0.5	0.8	38
7	58	57	21	36,990	0.6	1.5	60
8	70	68	26	4,892	0.8	1.1	27
9	21	40	0	416	0.2	0.9	78
10	26	16	11	20	0.5	1.0	50

Table 1. 10 example fault trees BDD v FTA

It is evident from the results in table 1 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.

6. Variable Ordering

The ordering placed on the basic events will determine the size of the resulting BDD. BDD's produced using a simple top-down ordering of variables are frequently inefficient since they produce a large number of non-minimal cut sets. The non-minimum BDD must undergo the minimising procedure to obtain the minimal cut sets, which can cause an undesirable increase in computation time. It is therefore beneficial to achieve an ordering which is optimal in terms of the resulting size of the BDD.

Cut set redundancy arises when the fault tree contains repeated basic events. It may be beneficial to consider the number of repetitions of the events when placing them in an ordering. The most commonly used ordering of variables is produced by listing them on a top-down basis from the original fault tree structure. An alternative ordering discussed in Ref 7 lists repeated events first within the top-down ordering. In this case, once a repeated event has been ordered due to its occurrence at a high level in the fault tree, subsequent occurrences lower down the tree are ignored.

Also if a gate has more than one repeated event as an input then the most repeated event is ordered first. If they occur the same number of times then the events are taken in list order to break the tie.

To investigate the effects that different ordering schemes produce, three different ordering options are considered. Within each of these three schemes there is also an option to list repeated events first. This will produce six different ordering schemes which have been investigated.

The different ordering schemes considered are best illustrated by an example fault tree, refer to figure 5. The list of gates whose inputs are ordered first is given for each ordering.

Figure 5. Example BDD for different orderings

Orderings

- (1) Top-down ordering (Top, g1, g2, g5, g3, g4):

$$a < b < e < c < g < f < d$$

- (2) Top-down ordering with each sub-tree ordered first (Top, g1, g3, g4, g5, g2):

$$a < f < e < b < d < c < g$$

- (3) As (2) and gates with only basic event inputs ordered first (g1, g5, g3, g4, g2, Top):

$$c < g < f < e < b < d < a$$

The above orderings can vary depending on which gates are chosen at each level in the fault tree. For example, ordering (3) may have considered the gates top, g2, g4, g1, g5, g3 which gives the basic event ordering $a < b < e < d < c < g < f$, call this ordering (4).

For this fault tree, ordering number 3 directly creates a minimum BDD, removing the need to apply the **without** operator. Also, listing repeated events first for ordering (4) gives:

$$a < e < b < d < g < c < f$$

which reduces the analysis even further by having to make less **ite** computations.

Six example fault trees were analysed using the three different ordering options. Table 2 provides a summary of each fault tree. Also, for the trees with repeated events, a second ordering was assessed which considered the repeated events first within the main ordering scheme which had given the most minimum ordering. The results can be seen in table 3. The number of **ite** calculations before and after minimising the BDD have been entered, along with the difference in these values, which is the extra **ite** calculations that are needed to make the BDD minimum. 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	No. of repeated events	No. of minimal cut sets
Tree1	41	11,934
Tree2	28	35
Tree3	2	2
Tree4	0	8,716
Tree5	0	7,056
Tree6	68	8,179

Table 2. Summary of 6 example fault trees

Trees and orderings	Number of ite calculations before minimising	Number 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
'repeated' 2	1457	1787	330	2.09
Tree2				
1	136	181	45	0.41
2	284	375	91	0.24
3	336	378	42	0.23
'repeated' 3	154	201	47	0.25
Tree3				
1	25	26	1	0.12
2	25	27	2	0.14
3	25	27	2	0.14
'repeated' 1	25	25	0	0.10
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
Tree6				
1	29,867	—	—	—
2	2,582	4,282	1,700	4.09
3	1,687	2,704	1,017	3.35
'repeated' 3	2,982	4,752	1,770	4.20

Table 3. Results of different ordering on 6 example fault trees

The results show that there are vast differences in the number of **ite** computations, when different orderings 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 clear from these examples that each tree has an individual variable ordering that will optimise the size of its BDD. There does not appear to be a general ordering scheme that will be 'best' for all trees. The top-down ordering for tree 6 resulted in a BDD whose minimisation could not be executed in a reasonable time.

Bryant (Ref 1) 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.

7. Conclusions

Conventional top-down and bottom-up techniques of analysing Fault Trees are at times an inadequate means of identifying **all** minimal cut sets **efficiently**. It has been shown that analysis procedures based on Binary Decision Diagrams to represent the system failure logic can produce all minimal cut sets for problems which defeat conventional approaches. Also for problems in which analysis times can be compared the BDD method is favourable.

To improve the efficiency of the analysis even further, alternative ordering schemes can be applied to the basic events of the fault tree to reduce the size of the resulting BDD. This has advantages for both the qualitative and quantitative analysis. However it seems unlikely that a general rule based ordering scheme can be determined which will be optimal for all fault trees.

8. References

1. R.E. Bryant, "Graph-Based algorithms for Boolean function manipulation," *IEEE Trans. Computers*, Vol C-35, No. 8, 1986 Aug, pp677-691.
2. A. Rauzy, "New algorithms for fault tree analysis," *Reliability Engineering and System Safety*, vol 40, 1993, pp203-211.

3. J.B. Fussell and W.E. Vesely, "A new methodology for obtaining cut sets for fault trees", *Trans Am Nucl Soc*, Vol 15, 1972, June, pp262-263
4. W.E. Vesely, "A Time Dependent Methodology for Fault Tree Evaluation", *Nuclear Eng and Des*, 13, 1970, pp337-360.
5. W.G. Schneeweiss, "*Boolean Functions with Engineering Applications and Computer Programs*," Springer-Verlag, 1989.
6. R.M. Sinnamon and J.D. Andrews, "Improved accuracy in Quantitative Fault Tree Analysis", *Advances in Reliability Technology Symposium*, Manchester, 16-17 April 1996.
7. R.M. Sinnamon and J.D. Andrews, "New Approaches to Evaluating Fault Trees," *Proceedings of Esrel'95 Conference*, June, 1995, pp241-254.