# An Algorithm to Find Minimal Cuts of Coherent Fault-Trees with Event-Classes, Using a Decision Tree

**Juan A. Carrasco**, *Member IEEE*
Universitat Politècnica de Catalunya, Barcelona

**Víctor Suñé**
Universitat Politècnica de Catalunya, Barcelona

**Key Words** — Fault tree, Minimal cuts, Decision tree.

**Summary & Conclusions** — A new algorithm (CS-MC) for computing the minimal cuts of $s$-coherent fault trees is presented. Input events of the fault tree are assumed classified into classes, where events of the same class are indistinguishable. This allows capturing some symmetries which some systems exhibit. CS-MC uses a decision tree. The search implemented by the decision tree is guided by heuristics which try to make CS-MC as efficient as possible. In addition, an irrelevance test on the inputs of the fault tree is used to prune the search. The performance of CS-MC is illustrated and compared with the basic top-down and bottom-up algorithms using a set of fault trees, some of which are very difficult. The CS-MC performs very well even in the difficult examples, and the memory requirements of CS-MC are small.

## 1. INTRODUCTION

*Acronyms*[1]

| | |
|---|---|
| ATPG | automatic test pattern generation (automatic generation of input vectors for the test of digital circuits) |
| CS-MC | Carrasco-Suñé minimal-cuts algorithm (the new algorithm in this paper) |
| DT | decision tree |
| BDD | binary decision diagram |

Fault trees are a very popular tool in reliability engineering. The knowledge of the minimal cuts[2] allows the designer to analyze the criticality of the basic events and to improve the reliability of the modeled system. It is well-known that computation of all minimal cuts of an arbitrary fault tree is NP-hard [21]. In spite of this theoretical difficulty, there exist algorithms which perform reasonably well in many practical cases. Older algorithms can be classified in 2 categories: *top-down* and *bottom-up*. All minimal cuts of a 'fan-out free' fault tree (a fault tree without repeated basic events or gates branching out to more than one gate input) can be computed very easily by traversing the fault tree in a top-down fashion. In the basic top-down algorithm [11], a set of cuts (often called the superset) is obtained as if the fault tree were fan-out free. The set of minimal cuts is then obtained by using in each cut the reduction rule $xx \rightarrow x$ and keeping those cuts which are not properly contained in any other cut. The algorithm involves $N \cdot (N-1)$ inclusion tests, where $N$ is the cardinality of the superset. These inclusion tests can be performed very efficiently by assigning different prime numbers to the basic events, and representing the cuts by the product of the constituent basic events [23]. However, even using these techniques, reduction of the superset is expensive if $N$ is large. Also, some fault trees with a manageable number of minimal cuts have $N$ so large that it is impossible to keep in memory the superset[3].

Some improvements to the basic top-down algorithm have been proposed. In [2] the size of the superset and the number of required inclusion tests is reduced by eliminating repeated events which only fan-out to OR gates. The algorithm in [19] stops the top-down expansion process at OR gates with basic-event inputs, substitutes OR gates with repeated basic-event inputs by one of those repeated basic-events, and performs reduction after each substitution step. The algorithm was to some extent faster than the basic top-down algorithm in almost all cases. Ref [16] proved that cuts of the superset without repeated basic events are all minimal and that the test for inclusion can be performed within the remaining cuts.

Bottom-up algorithms try to avoid the potentially large superset of the top-down algorithms. In the basic bottom-up algorithm [3], the fault tree is traversed from the inputs to the top event, obtaining at each step the set of minimal cuts associated with a given gate of the fault tree from the set of minimal cuts associated with the gates in its fan-in. In general, each step of the bottom-up algorithm requires reducing the superset associated with the processed gate. The reduction is usually not very expen-

[1] The singular & plural of an acronym are always spelled the same.

[2] The term 'minimal cut' is used instead of the more common 'minimal cutset' because the minimal cuts in this paper are bags.

[3] Thus, for instance, the example EDF of this paper has $N \approx 8.76 \cdot 10^{24}$ but only 2463 minimal cuts.

0018-9529/99/$10.00 

sive for OR gates. For a 2-input AND gate, the trivial procedure involves $n_1 \cdot n_2 \cdot (n_1 \cdot n_2 - 1)$ tests, where $n_1$, $n_2$ are the number of minimal cuts at the inputs. However, a more sophisticated algorithm [17] reduces the number of tests to $n_1 \cdot n_2 \cdot (n_1 + n_2 - 1)$ in the worst case. A very recent [14] elaboration of the bottom-up algorithm includes a preprocessing step which yields reduced level fault trees which are processed in descending-level order. This new algorithm can appreciably speed up the basic bottom-up algorithm when the fault tree has gates with fan-out.

The concept of module [7] can be exploited to reduce appreciably, for some fault trees, the computation cost of finding the minimal cuts. A module is a portion of the fault tree having basic events as inputs, and a single gate (the module output) fanning-out of the module. Efficient algorithms have been proposed to find modules [9, 15, 22, 26]. The analysis of a fault tree with modules can be reduced to the analysis of each module and of the fault tree obtained by substituting each module by an independent input.

More recently, algorithms based on BDD representations [4] of the logic function implemented by the fault tree have been proposed. In [20, 24] the fault tree is assumed to be $s$-coherent, and a BDD is constructed for it. That BDD is then transformed to obtain another BDD such that each path from the root to the leaf 1 represents a minimal cut. The MetaPrime tool [8] uses a similar approach which can deal with non $s$-coherent fault trees; the BDD encoding the minimal cuts is called metaproduct. More recently [18], another algorithm has also been developed for non $s$-coherent fault trees, which in many cases gives more compact BDD representations than the metaproducts obtained by MetaPrime. All these algorithms are typically much faster than the early top-down and bottom-up algorithms.

This paper develops CS-MC (a new algorithm) to compute the minimal cuts of $s$-coherent fault trees. CS-MC considers generalized fault trees with basic event classes, wherein the elements of each class are indistinguishable. This is an useful generalization, because some systems have indistinguishable basic events. CS-MC was motivated by bounding methods [5, 6, 25], which require the list of minimal cuts to be found. With classes, the fault tree no longer represents a logic function with binary arguments. Also, cuts and minimal cuts are no longer 'sets' but 'bags'. CS-MC uses a DT to generate cuts of the fault tree. Cuts thus obtained are not guaranteed to be minimal and they must be tested for minimality. However, instead of inclusion tests, CS-MC makes an independent minimality test for each generated cut. Since cuts are generated one by one, this allows writing sequentially in secondary storage minimal cuts, making the CS-MC memory requirements small, and independent of the number of minimal cuts. These small memory requirements are a unique feature of CS-MC (BDD representations of fault trees are, in the worst case, of exponential size with the number of basic events). Using examples, some of which are quite difficult, this paper shows that the CS-MC performance is good. Also, the number of cuts which are processed is usually not large compared with the number of minimal cuts. This is in sharp contrast with top-down and bottom-up algorithms.

Section 2 establishes preliminary theoretical results. Section 3 describes CS-MC and illustrates it using a small example. Section 4 presents experimental results for a set of large fault trees and compares CS-MC with the top-down and bottom-up algorithms.

*Notation*[4]

| | |
|---|---|
| $C$ | set of basic event classes of the fault tree |
| $c_i$ | basic event class, $1 \le i \le k$ |
| $c_1[n_1]c_2[n_2]\ldots c_k[n_k]$ | bag with $n_i > 0$ instances of $c_i$, ; $c_i[n_i]$ is part of the bag; |
| $I$ | set of inputs of the fault tree; each input is a bag $c[n]$, $c \in C$, $n \ge 1$, *viz*, the realization of at least $n$ instances of event class $c$ |
| $G$ | set of gates of the fault tree |
| $g_r$ | root (top) gate |
| type($g$) | type of gate $g$; either 'AND' or 'OR' |
| val($\cdot$) | value of an implied input or gate; either '0' or '1' |
| $\lvert\cdot\rvert$ | cardinality of a set |

*Definitions*

- bag: collection of possibly repeated elements.
- node: an input or a gate.
- fo($x$): fan-out of node $x$: set of gates fed by $x$.
- fi($g$): fan-in of gate $g$: set of gates/inputs that feed $g$.
- irrelevant: a node $x$ is irrelevant iff all edges branching out of $x$ are irrelevant; an edge $e$ is irrelevant iff either the gate $g$ to which $e$ goes is irrelevant or has been previously implied by a node connected to $g$ by another edge,
- dfo($x$): dynamic fan-out of an unimplied node $x$: set of relevant edges fed by $x$.
- $f_x$: fan-out excess of an unimplied and relevant node $x$: $|\,\text{dfo}(x)| - 1$.
- $f$: fan-out excess of the fault tree:
$$\sum_{\substack{x \in I \cup G \\ x \text{ unimplied and relevant}}} f_x.$$
- $\delta_x$: for an unimplied and relevant gate, $\delta_x = f_x$; for an unimplied and relevant input $c[n]$,
$$\delta_x = \sum_{\substack{c[n'] \in I,\ n' \le n \\ c[n'] \text{ unimplied and relevant}}} f_{c[n']}.$$
- cut: bag with elements in $C$ whose realization implies $g_r$ at 1.
- minimal cut: cut $m$ such that no $m' \subset m$ is a cut.
- compatible: the existence of the assignment $(c[n], 1)$ implies the existence of the assignments $(c[n'], 1)$, $n' < n$, $c[n'] \in I$; the existence of the assignment $(c[n], 0)$ implies the existence of the assignments $(c[n'], 0)$, $n' > n$, $c[n'] \in I$.

[4]Other, standard notation is given in "Information for Readers & Authors" at the rear of each issue.

· input pattern: any compatible combination of assignments of 0, 1 values to inputs of the fault tree.

· reduction of $l$: for a set of inputs implied at 1, $l = \{c_1[n_1], c_2[n_2], \ldots, c_k[n_k]\}$ generation of bag $b$ by traversing $l$ and putting into $b$ each $c[n] \in l$ such that no $c[n']$, $n' > n$ is in $l$.

· cont($x$): controllability of an unimplied node $x$:

if $x \in I$, cont($x$) = 1;

if $x \in G$ and type($x$) = OR,

$$\text{cont}(x) = \min_{\substack{x' \in \text{fi}(x) \\ x' \text{unimplied}}} \{\text{cont}(x')\};$$

if $x \in G$ and type($x$) = AND,

$$\text{cont}(x) = \sum_{\substack{x' \in \text{fi}(x) \\ x' \text{unimplied}}} \text{cont}(x').$$

## 2. BACKGROUND

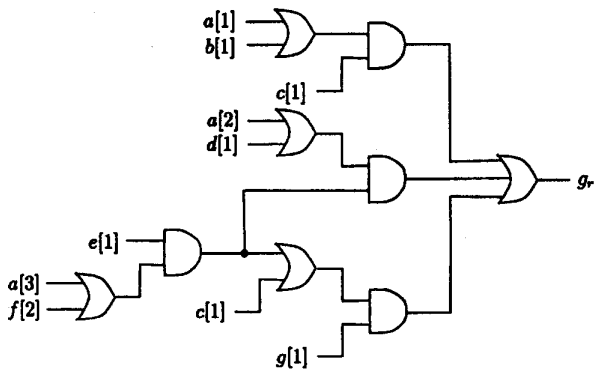


Figure 1: Example Fault-Tree

We consider $s$-coherent fault trees involving OR and AND gates. Figure 1 gives a small example which is used for illustration.

We begin by clarifying several concepts. Generalization to basic-event classes introduces $s$-dependences among the inputs related to the same event class.

An input $c_i[n_i]$ is implied at:

· 1 if $n_i$ events of class $c_i$ are realized,

· 0 if $n_i$ events of class $c_i$ are not realized.

Then,

· val($c_i[n_i]$) = 1 implies: val($c_i[j]$) = 1 for all $j < n_i$ because the realization of $n_i$ basic events of class $c_i$ implies the realization of any number $j < n_i$ of basic events of that class. Similarly

· val($c_i[n_i]$) =0 implies: val($c_i[j]$) = 0 for all $j > n_i$.

Implications in the fault tree are performed from inputs to $g_r$ in the usual way:

· an input of an:

· OR gate at 1 implies the output of the gate at 1,

· AND gate at 0 implies the output of the gate at 0;

· the output of an:

· OR gate is implied at 0 when all inputs are implied at 0,

· AND gate is implied at 1 when all inputs are implied at 1.

Since the fault tree is $s$-coherent and all gates are either OR or AND, such a procedure is enough to know the implication state of $g_r$. To see that it suffices, note that:

· 0's imply 0's,

· 1's imply 1's,

and, therefore, an unimplied output $g_r$ can be implied at 0 or 1 by simply implying the unimplied inputs of the fault tree at 0 or 1, respectively. Therefore, when the procedure does not imply $g_r$, then $g_r$ is really unimplied.

For the example fault-tree in figure 1,

$I = \{a[1], a[2], a[3], b[1], c[1], d[1], e[1], f[2], g[1]\}$.

Two possible input patterns are:

$l_1 = \{(a[1], 1), (a[2], 1), (a[3], 0), (b[1], 0), (c[1], 0), (d[1], 0), (e[1], 1), (f[2], 1), (g[1], 0)\}$;

$l_2 = \{(a[1], 0), (a[2], 0), (a[3], 0), (b[1], 0), (c[1], 0), (d[1], 0), (e[1], 1), (f[2], 1), (g[1], 0)\}$.

For $l_1$, $g_r$ is implied at 1. Then, that input pattern "contains" the cut $m = a[2]e[1]f[2]$ obtained by reducing the set of inputs implied at 1 $\{a[1]a[2]e[1], f[2]\}$.

The problem is to find all minimal cuts of a given $s$-coherent fault tree. Such a problem can be viewed as a search in a finite space. The search space is given by theorem 1.

*Theorem* 1. All minimal cuts are of the form:
$c_1[n_1]c_2[n_2]\ldots c_k[n_k]$, where each $c_i[n_i] \in I$. ◄

The proof is in appendix A.1.

From the irrelevance definitions in section 1, the value of an irrelevant node does not affect val($g_r$). Theorem 2 relates irrelevance & minimality, and is used in the algorithm.

*Theorem* 2. Let $S$ be the set of implied fault-tree inputs. If there exists $c_i[n_i] \in S$ which is implied at 1 and is irrelevant, and there does not exist any $c_i[j] \in I$ with $j > n_i$, then no input pattern obtained by implying more inputs contains a minimal cut. ◄

The proof is in appendix A.2.

To avoid performing inclusion tests, which are time & memory consuming, a criterion to determine when a cut is minimal or not, without knowing any other cut, is useful. Theorem 3 gives such a criterion.

*Theorem* 3. A cut $m = c_1[n_1]c_2[n_2]\ldots c_k[n_k]$ is minimal iff, after implying at 1 each $c_i[n_i]$, each unimplication of $c_i[n_i]$, $1 \le i \le k$, followed by, if such $J_i$ exists, implication of $c_i[J_i]$, where $J_i$ is the greatest integer $< n_i$ with $c_i[J_i] \in I$, leaves $g_r$ unimplied. ◄

The proof is in appendix A.3.

## 3. DESCRIPTION OF CS-MC

From theorem 1, the search space is the space of input patterns of the fault tree. All minimal cuts can be found by exhaustively searching that space and, for each input

pattern for which val($g_r$) = 1, obtaining its associated cut $m$ by reduction of the set of inputs implied at 1, and testing $m$ as theorem 3 indicates. However, CS-MC does not explicitly generate all input patterns, because in some circumstances it detects that no input pattern containing the current set of implications would yield a minimal cut. CS-MC also detects when the fault tree becomes fan-out free ($f$ becomes 0) and uses the top-down algorithm to obtain all potential minimal cuts which can be generated by performing more implications at the inputs of the fault tree.

CS-MC traverses a DT such as the one shown in figure 3. Initially, all nodes of the fault tree are unimplied and the current node is the root of the DT. A backtrace procedure selects an input $c_i[n_i]$. The selected input is implied at $v = 1$, a successor of the current node is constructed with the pair $(c_i[n_i], v)$ assigned to it, and the successor is visited. The process continues in a similar way from that node. After each implication, the relevance status of edges & nodes, $f_x$, $f$, $\delta_x$, cont($x$) are updated. In some cases, the search can be pruned. When the search is pruned, the DT is traversed up to the root until a node $y$ is found that has only a successor with $v = 1$. If no such a node $y$ is found, CS-MC finishes. Otherwise, the search is backtracked up to node $y$. Backtracking involves the deletion of all the implications done as a direct consequence of the input assignments associated with the nodes in the path from the current node to $y$. It also involves restoring the old values of the relevance status of the edges and nodes, $f_x$, $f$, $\delta_x$, cont($x$). Deletion of implications and restoring of old values of the relevance status of edges & nodes, $f_x$, $f$, $\delta_x$, cont($x$) is made easily because CS-MC stores, for each node of the DT, the implications performed and the old values of the relevance status of edges & nodes, $f_x$, $f$, $\delta_x$, cont($x$) which change as a result of the input implication performed at the node. After backtracking, a successor of $y$ associated with the input assignment $(c_i[n_i], 0)$, where $c_i[n_i]$ is the input of the successor of $y$, is created, the assignment implied, and the process continues from that successor. If the implied input value, $v$, is 1 the search can be pruned in the following 4 cases:

- Case 1. val($g_r$) = 1: Extra input implications at 1 give cuts which are guaranteed to be non-minimal.
- Case 2. $f = 0$: The fault tree has been reduced to a fault tree which is fan-out free[5] and all minimal cuts beyond that point of the search must be included in the cuts which are obtained by adding to the current set of inputs implied at 1, the inputs which are found using the basic top-down algorithm on the reduced fan-out free fault tree and reducing those sets of inputs.
- Case 3. An input $c[n]$ implied at 1 is irrelevant, and there does not exist $c[n'] \in I$, $n' > n$. According to theorem 2, no minimal cut can be found by implying more inputs.

For $v = 0$, case 1 is impossible. Besides cases 2 & 3, there is another situation in which no more implications are necessary:

- Case 4. val($g_r$) = 0: Further input assignments do not change the value of $g_r$, and no minimal cut exists from that point.

In case 1, a potential minimal cut is obtained by reducing the set $l$ of inputs implied at 1. In case 2, the top-down algorithm is used to find potential minimal cuts. The cuts thus obtained are checked for minimality using theorem 3, and are recorded if they are minimal.

CS-MC has similarities with some ATPG algorithms [1, 10, 12]. Figure 2 gives a recursive high-level description of CS-MC, which uses a stack to store the path in the DT to the currently processed node. CS-MC is invoked with an empty stack. In the worst case, the stack has $|I|$ cells. This together with the fact that only a minimal cut has to be stored at a given time (possible because the minimality check does not involve inclusion tests) makes the CS-MS memory requirements small. The algorithm description in figure 2 assumes that the fault tree to be solved is fan-out free. To handle the special case in which the fault tree is fan-out free, it suffices to check the value of $f$ before calling *compute_cuts* and invoke the top-down algorithm if $f = 0$.

The backtrace procedure which selects input assignments is crucial for the performance of CS-MC and is inspired in ATPG algorithms. The procedure begins at the output $g_r$ of the fault tree and follows a path to a fault-tree input by selecting at each gate one of its unimplied inputs. Gate inputs are selected using the criteria:

1. Choose the input $x$ with highest $\delta_x / \text{cont}(x)$.
2. Among the inputs with same $\delta_x / \text{cont}(x)$ choose the input connected to the node with lowest cont($x$).

If several inputs are identical according to both criteria, then the gate input is chosen following a predefined ordering (for the fault tree of figure 1 the ordering is from top to bottom of the figure).

The heuristics used in the backtrace procedure try to reach, as soon as possible, nodes with backtracking, either because the fault tree becomes fan-out free or because $g_r$ is implied at 1. The $\delta_x$ is a local measure of how much $f$ is decreased if $x$ is implied at 1. The cont($x$) is a measure of the ease with which the considered-node is implied at 1 (the higher cont($x$) the more difficult) and is taken directly from heuristic measures used to guide ATPG algorithms [13]; the measure in that context is called 1-controllability. Other combinations were tried, such as selecting first according to cont($x$) and then according to $\delta_x$; the chosen heuristics gave better performance.

When the number of minimal cuts is very large CS-MC is not efficient. In some cases, the number of minimal cuts can be kept reasonable by limiting the cardinality of the minimal cuts being searched to a given maximum value $K$ greater or equal to the minimum minimal-cut cardinality. To do this, it suffices to book-keep the cardinality *card* of the current reduced set of inputs implied-at-1 and prune

[5]The reduced fault tree is defined by the relevant and unimplied nodes of the original fault tree.

Algorithm *compute_cuts*
(INPUTS_ASSIGNMENTS_STACK $s$)

```
backtrace(g_r, c_i[n_i]);
v = 1;
end = NO;
while (!end) {
  imply c_i[n_i] at v;
  push((c_i[n_i], v), s);
  if (val(g_r) == 1 || f == 0) {
    if (val(g_r) == 1) {
      get cut m;
      if (test_minimality(m)) store m; /* m is minimal */
    }
    else { /* f == 0 */
      collect potential minimal cuts by adding to the current
      set of inputs implied at 1 the inputs found by the
      top-down algorithm and reducing those sets, for each
      cut perform the minimality test and store the cut if
      it is minimal;
    }
  }
  else if (!(val(g_r) == 0 || an input c[n] implied at 1 has
       become irrelevant and does not exist c[n'] ∈ I, n' > n))
       compute_cuts(s)
  unimply all c_i[j] with either j ≤ n_i (case v = 1) or
  j ≥ n_i (case v = 0) which were implied as a direct
  consequence of setting c_i[n_i] at v;
  pop((c_i[n_i], v), s);
  if (v == 1) v = 0 else end = YES;
}
```

Figure 2: High-Level Description of CS-MC

the search when either $card > K$ or $card = K$ and $g_r$ is unimplied.

To illustrate CS-MC, figure 3 gives the DT corresponding to the fault tree of figure 1. Nodes are numbered following the creation order. For each node, except the root, the corresponding input assignment is given. For nodes with backtracking, the reason for backtracking is given; for nodes in which the fault tree became fan-out free, or $g_r$ was implied at 1, the cuts found ('mc' indicates the minimal ones) are given. The fault tree has only 7 minimal cuts.

Figure 4 illustrates the backtrace procedure where there is no input assignment (node 0 of the DT). The input selected is c[1]. Figure 5 gives the implication status of the fault tree when c[1] is implied at 1. That implication makes irrelevant the edge marked as i, making the fault tree fan-out free ($f = 0$). Figure 6 shows the corresponding reduced fault tree. Use of the top-down algorithm on the reduced fault-tree gives the sets of inputs:
$\{a[1]\}$, $\{b[1]\}$, $\{a[2], e[1], a[3]\}$, $\{a[2], e[1], f[2]\}$,
$\{d[1], e[1], a[3]\}$, $\{d[1], e[1], f[2]\}$, $\{g[1]\}$.

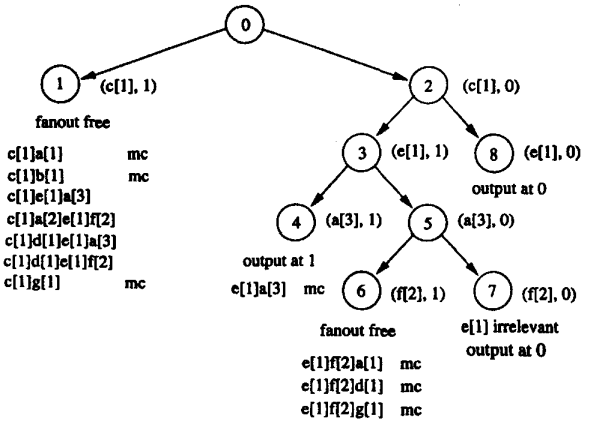


Figure 3: DT for Example of Figure 1

The input currently implied at 1 ($c[1]$) is added to each set, and the resulting sets are reduced, yielding the cuts in figure 3. To illustrate the minimality test, figure 7 gives the implication status for the cut $c[1]e[1]a[3]$, showing crossed the unimplications which result when $a[3]$ is unimplied, keeping implied $a[2]$ & $a[1]$. Since $g_r$ remains implied at 1, according to theorem 3, the cut is not minimal. Finally, to illustrate backtracking by detection of an irrelevant input implied at 1, figure 8 gives the implication status and irrelevant nodes & edges corresponding to node 7 of the DT. Input $e[1]$ is implied at 1, is irrelevant and there does not exist any other input $e[j]$ with $j > 1$. Thus, according to theorem 2, no minimal cut can be found from that point, and the search can be backtracked. At that same node of the DT, the search could also be backtracked for $g_r$ being implied at 0.

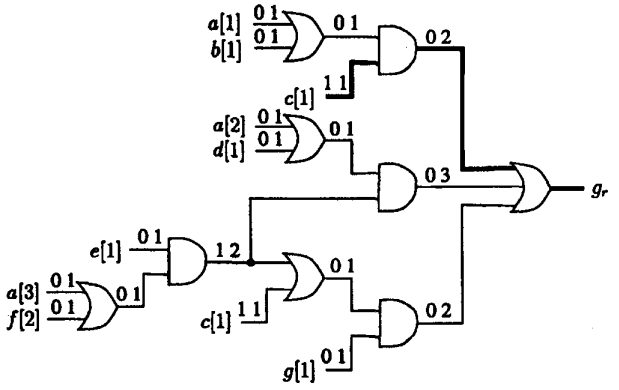


Figure 4: Illustration of Backtrace Procedure

[This applies when there is no input assignment (node 0 of the DT of figure 3). The $\delta_x$ & cont($x$) are written next to each node from left to right, respectively.]

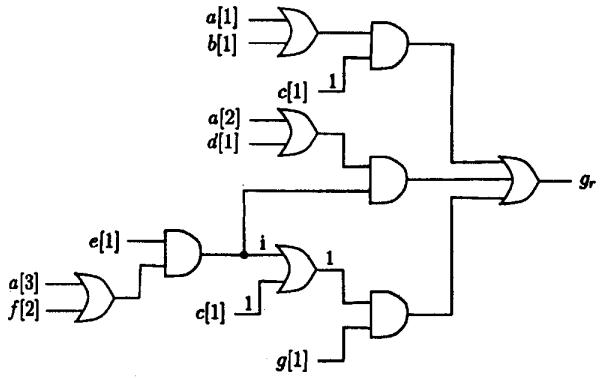


Figure 5: Implication Status at Node 1

[This is for the DT of figure 3, with indication of irrelevant edges.]

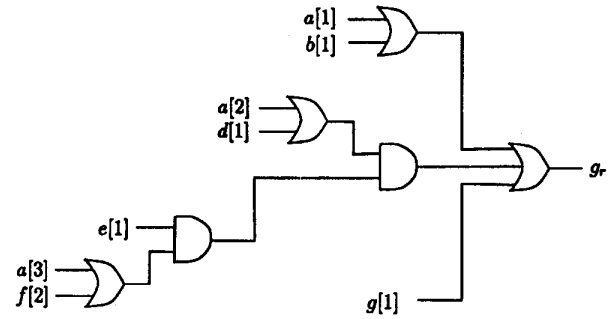


Figure 6: Reduced Fault-Tree

[This corresponds to node 1 of the DT of figure 3.]

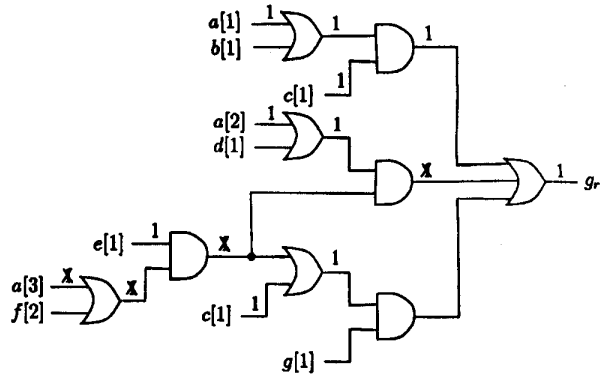


Figure 7: Implication Status

[This corresponds to cut $c[1]e[1]a[3]$ and un-implication of $a[3]$, leaving $a[2]$ & $a[1]$ implied for the fault tree of figure 1.]

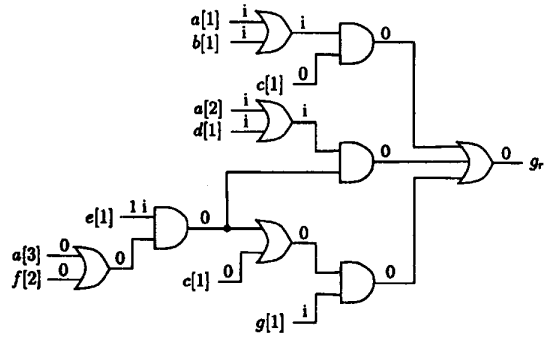


Figure 8: Implication Status and Irrelevance Status

[This corresponds to the node 7 of the DT of figure 3.]

## 4. RESULTS

Table 1 summarizes the characteristics of the fault trees which have been used to test CS-MC. In all cases the number of basic-event classes equals the number of inputs. For each fault tree, we give the number of inputs, number of gates, fan-out excess ($f$) in the unimplied fault tree, depth (maximum number of gates from a fault tree input to $g_r$) and number of minimal cuts.

Fault trees MS5 & MS10 correspond to the master-slave system in figure 9 with $n = 5$ and $n = 10$, respectively. That system is made up of a cluster of redundant master processing units $\mathrm{MPU}_1$ and $\mathrm{MPU}_2$ which are communicated with $n$ clusters of redundant slave processing units $\mathrm{SPU}_{i,1}$ and $\mathrm{SPU}_{i,2}$, $1 \leq i \leq n$. Communication is done through two redundant buses BA & BB to which the master & slave units are connected through dedicated interfaces. The system is operational if some fault-free master processing unit can communicate directly (through one fault-free bus and two fault-free interfaces) with at least one fault-free slave processing unit of each slave cluster. Denote the logical operators AND and OR by '·' and by '+' respectively; name each event class as the component type whose failure models, the expression of the fault tree of that system is:

$$g_r = \left[\sum_{i=1}^{n} \Psi_1 + (\Theta_{A:1} + \Phi_{A:i}) \cdot (\Theta_{B:1} + \Phi_{B:i})\right] \cdot \left[\sum_{i=1}^{n} \Psi_2 + (\Theta_{A:2} + \Phi_{A:i}) \cdot (\Theta_{B:2} + \Phi_{B:i})\right]$$

$\Psi_k \equiv \mathrm{MPU}_k[1]$

$\Theta_{A:k} \equiv \mathrm{IMA}_k[1] + \mathrm{BA}[1]$

$\Theta_{B:k} \equiv \mathrm{IMB}_k[1] + \mathrm{BB}[1]$

$\Phi_{A:i} \equiv (\mathrm{ISA}_{i,1}[1] + \mathrm{SPU}_{i,1}[1]) \cdot (\mathrm{ISA}_{i,2}[1] + \mathrm{SPU}_{i,2}[1])$

$\Phi_{B:i} \equiv (\mathrm{ISB}_{i,1}[1] + \mathrm{SPU}_{i,1}[1]) \cdot (\mathrm{ISB}_{i,2}[1] + \mathrm{SPU}_{i,2}[1])$

Fault trees BR40 & BR80 model the failure of the braided ring system of figure 10 with $n = 40$ & $n = 80$, respectively. The braided ring is composed of stations $\mathrm{S}_i$, $0 \leq i \leq n-1$. There are links $\mathrm{D}_i$ between $\mathrm{S}_i$

Table 1: Fault-Tree Characteristics

| tree | $\|I\|$ | $\|G\|$ | fan-out excess | depth | minimal cuts |
|---|---|---|---|---|---|
| MS5 | 38 | 103 | 92 | 7 | 511 |
| MS10 | 68 | 203 | 192 | 7 | 1911 |
| BR40 | 120 | 42 | 3080 | 2 | 3160 |
| BR80 | 240 | 82 | 12560 | 2 | 12720 |
| DR35 | 112 | 46 | 152 | 10 | 3698 |
| DR70 | 217 | 46 | 257 | 10 | 14157 |
| EDF | 39 | 43 | 9 | 34 | 2463 |
| ELF1 | 61 | 122 | 12 | 147 | 46188 |
| ELF2 | 32 | 65 | 57 | 12 | 4805 |
| ELF3 | 80 | 107 | 87 | 17 | 24386 |

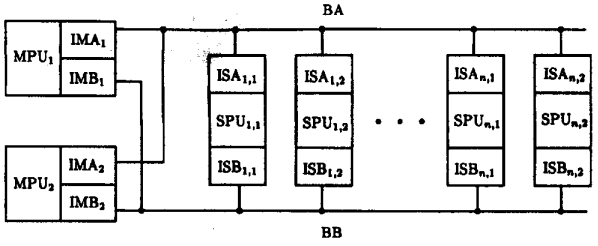


Figure 9: Master-Slave System
[With $n$ clusters of redundant slave processing units]

and $S_{(i+1) \bmod n}$ and links $T_{(i+1) \bmod n}$ between $S_i$ and $S_{(i+2) \bmod n}$. All these links are directed. The system is up if it is possible to build a ring connecting at least $n-1$ fault-free stations $S_i$. The fault-tree expression is:

$$g_r = \left[\sum_{i=0}^{n-1}(S_i[1] + D_i[1])\right] \cdot \prod_{i=0}^{n-1}\left(\sum_{\substack{j=0 \\ j\neq i}}^{n-1} S_j[1] + \sum_{\substack{j=0 \\ j\neq i,(i-1) \bmod n}}^{n-1} D_j[1] + T_i[1]\right). \quad (1)$$

Fault trees DR35 & DR70 correspond to the system of figure 11 with $n = 35$ & $n = 70$, respectively. Two redundant servers $S_1$, $S_2$ are communicated with gateways $G_1$, $G_2$ through a double ring network composed of nodes $N_i$, $0 \le i \le n-1$. There are clockwise links $I_i$ from $N_{(i+1) \bmod n}$ to $N_i$ and counter-clockwise links $D_i$ from $N_i$ to $N_{(i+1) \bmod n}$. Each node has a spare module $SN_i$ that can bypass $N_i$ if $N_i$ has failed. However, those spare components are not connected to servers or gateways. The servers are connected to nodes $N_{\lfloor n/2 \rfloor}$ and $N_{\lfloor n/2 \rfloor+1}$; gateways $G_1$ & $G_2$ are connected to nodes $N_0$ & $N_1$, respectively. The system is operational if communication exists in both directions between at least one fault-free server and one fault-free gateway.

$S_1$ and $S_2$ are indistinguishable;

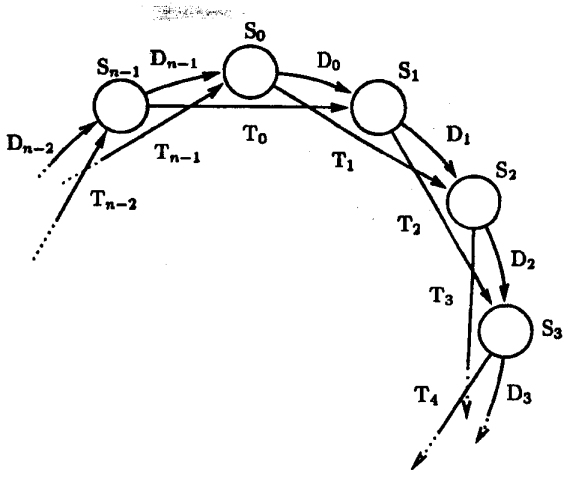


Figure 10: Braided Ring System with $n$ Stations

$N_i$ and $SN_i$, $2 \le i \le \lfloor n/2 \rfloor - 1$, $\lfloor n/2 \rfloor + 2 \le i \le n-1$ are also indistinguishable. Denote by:

- S: the event class that models failure of $S_1$, $S_2$,
- $M_i$: the event class that models failure of $N_i$, $SN_i$, $2 \le i \le \lfloor n/2 \rfloor - 1$, $\lfloor n/2 \rfloor + 2 \le i \le n-1$,
- the components' names: the event classes which model the failure of the other components.

Then the system fault-tree is (there is an expression for $g_r$ and each gate with fan-out):

$$g_r = S[2] + (G_1[1] + \Theta_{0,\lfloor n/2 \rfloor}) \cdot (G_2[1] + \Theta_{1,\lfloor n/2 \rfloor}),$$

$$\Theta_{i,z} \equiv N_i[1] + (N_z[1] + C_{i,z}) \cdot (N_{z+1}[1] + C_{i,z+1}),$$

$$C_{0,\lfloor n/2 \rfloor} = (\Psi_1 + \Psi_{\lfloor n/2 \rfloor+1} + DR) \cdot (\Psi_1 + \Phi_0 + RFR) \cdot (\Psi_{\lfloor n/2 \rfloor+1} + \Phi_{\lfloor n/2 \rfloor} + LFR),$$

$$C_{0,\lfloor n/2 \rfloor+1} = (\Psi_1 + \Psi_{\lfloor n/2 \rfloor} + DR) \cdot (\Psi_1 + \Psi_{\lfloor n/2 \rfloor} + \Phi_0 + \Phi_{\lfloor n/2 \rfloor} + RFR) \cdot LFR,$$

$$C_{1,\lfloor n/2 \rfloor} = (\Psi_0 + \Psi_{\lfloor n/2 \rfloor+1} + DR) \cdot RFR \cdot (\Psi_0 + \Psi_{\lfloor n/2 \rfloor+1} + \Phi_0 + \Phi_{\lfloor n/2 \rfloor} + LFR),$$

$$C_{1,\lfloor n/2 \rfloor+1} = (\Psi_0 + \Psi_{\lfloor n/2 \rfloor} + DR) \cdot (\Psi_{\lfloor n/2 \rfloor} + \Phi_{\lfloor n/2 \rfloor} + RFR) \cdot (\Psi_0 + \Phi_0 + LFR),$$

$$\Psi_z \equiv N_z[1] \cdot SN_z[1], \qquad \Phi_z \equiv I_z[1] + D_z[1],$$

$$DR = \sum_{i=2}^{\lfloor n/2 \rfloor-1} M_i[2] + \sum_{i=\lfloor n/2 \rfloor+2}^{n-1} M_i[2] + \left(\sum_{i=0}^{n-1} I_i[1]\right) \cdot \left(\sum_{i=0}^{n-1} D_i[1]\right),$$

$$RFR = \sum_{i=2}^{\lfloor n/2 \rfloor-1} M_i[2] + \sum_{i=1}^{\lfloor n/2 \rfloor-1} \Phi_i,$$

$$LFR = \sum_{i=\lfloor n/2 \rfloor+2}^{n-1} M_i[2] + \sum_{i=\lfloor n/2 \rfloor+1}^{n-1} \Phi_i.$$

Table 2: Algorithm performance

| tree | minimal cuts | processed cuts | backtracks | CPU time (s) | cuts top-down | cuts bottom-up |
|---|---|---|---|---|---|---|
| MS5 | 511 | 1473 | 1936 | 1.74 | $3.42250 \cdot 10^4$ | $8.28500 \cdot 10^3$ |
| MS10 | 1911 | 5208 | 23135 | 13.60 | $1.36900 \cdot 10^5$ | $2.93450 \cdot 10^4$ |
| BR40 | 3160 | 3161 | 821 | 11.40 | $3.86255 \cdot 10^{77}$ | $3.86255 \cdot 10^{77}$ |
| BR80 | 12720 | 12721 | 3241 | 175.00 | $1.24936 \cdot 10^{178}$ | $1.24936 \cdot 10^{178}$ |
| DR35 | 3698 | 118444 | 60099 | 51.10 | $1.22672 \cdot 10^{26}$ | $2.90796 \cdot 10^7$ |
| DR70 | 14157 | 808953 | 395907 | 536.00 | $8.32281 \cdot 10^{30}$ | $4.75491 \cdot 10^8$ |
| EDF | 2463 | 3435 | 1683 | 4.26 | $8.75983 \cdot 10^{24}$ | $3.81949 \cdot 10^{10}$ |
| ELF1 | 46188 | 112606 | 169278 | 235.00 | $1.26358 \cdot 10^{20}$ | $2.86939 \cdot 10^8$ |
| ELF2 | 4805 | 13754 | 16842 | 23.60 | $4.17538 \cdot 10^{17}$ | $5.48907 \cdot 10^8$ |
| ELF3 | 24386 | 69488 | 150601 | 150.00 | $1.45039 \cdot 10^{16}$ | $7.66257 \cdot 10^7$ |

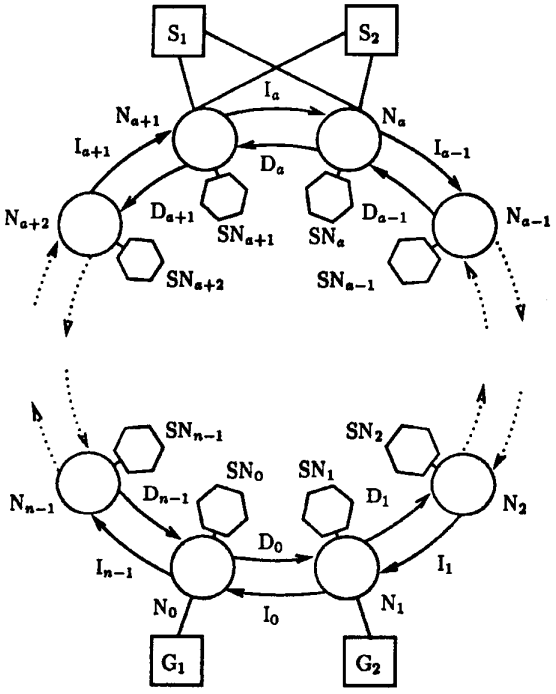


Figure 11: Double ring network ($a = \lfloor n/2 \rfloor$)

Fault-tree EDF models the failure of the communication network with 14 nodes and 25 directed links in figure 12. The network is up if the sender node S and the receiver node R are both fault-free and there is a path of fault-free components from S to R.

Fault trees ELF1, ELF2, ELF3 are approximately the fault-trees with these names in [8].

Table 2 shows the results obtained by CS-MC for each fault-tree. We give number of minimal cuts, number of processed cuts, number of backtracks, CPU times measured in a Sparc10 workstation, and, for the sake of comparison, number of cuts which would be processed by the basic top-down and bottom-up algorithms.

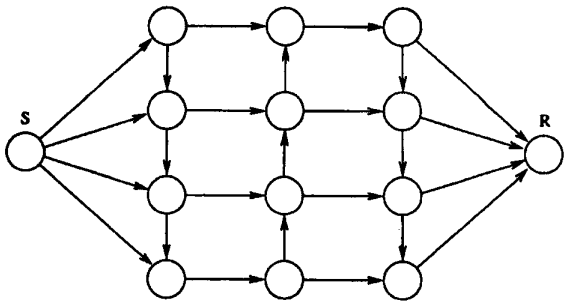


Figure 12: Communication Network for Fault-Tree EDF

For all the fault trees except MS5 & MS10 the use of either top-down or bottom-up algorithm is impractical. For all fault trees, CS-MC outperforms the classical algorithms, with a ratio of number of processed cuts between $5.6 \cdot 10^0$ and $9.8 \cdot 10^{173}$.

To illustrate the impact of:

1. the heuristics that we used in the backtrace procedure,
2. irrelevance test,

table 3 shows the number of backtracks performed when either the inputs are chosen at random or the test of irrelevance is disabled. The CPU time was limited to 5 hours and results are not given when more than 5 hours were necessary to compute all minimal cuts. In all cases, both the heuristics for input selection and the irrelevance test appreciably reduce the number of backtracks.

Section 3 stated that the performance of CS-MC could be improved substantially in some cases by limiting the cardinality of the minimal-cuts searched to a maximum value $K$ greater than or equal to the minimum minimal-cut cardinality. Table 4 gives the CPU times and number of minimal cuts found for $2 \leq K \leq 6$, and the results when the cardinality of the minimal-cuts is not limited. As anticipated, the performance is improved substantially when most of the minimal cuts are of large cardinality, and it is barely affected when the minimal cuts are all of small cardinality.

Table 3: Algorithm Performance

[Number of backtracks in CS-MC, and when either 'fault-tree inputs are selected at random' or 'the irrelevance test is disabled']

| tree | proposed | w/ random selection | w/o relevance test |
|---|---|---|---|
| MS5 | 1936 | 67619 | 309245 |
| MS10 | 23135 | | |
| BR40 | 821 | 1849 | 3943 |
| BR80 | 3241 | 10985 | 15883 |
| DR35 | 60099 | 233663 | 248414 |
| DR70 | 395907 | 1798840 | 1791888 |
| EDF | 1683 | 266611 | 22159 |
| ELF1 | 169278 | 9115802 | |
| ELF2 | 16842 | 226704 | 101994 |
| ELF3 | 150601 | | |

Table 4: Comparative Behavior

[CPU times in seconds (top row) and numbers of minimal cuts found (bottom row) when the search is limited to a maximum minimal-cut cardinality $K$, and for an exhaustive (Exh) search.]

| | Cardinality $K$ | | | | | |
|---|---|---|---|---|---|---|
| tree | 2 | 3 | 4 | 5 | 6 | Exh |
| MS5 | 0.23 | 0.41 | 0.67 | 0.98 | 1.56 | 1.74 |
| | 7 | 55 | 151 | 151 | 511 | 511 |
| MS10 | 0.31 | 0.85 | 1.84 | 3.58 | 7.51 | 13.6 |
| | 12 | 100 | 291 | 291 | 1911 | 1911 |
| BR40 | 11.46 | 11.70 | 11.70 | 11.70 | 11.70 | 11.70 |
| | 3080 | 3160 | 3160 | 3160 | 3160 | 3160 |
| BR80 | 130 | 130 | 130 | 130 | 130 | 175 |
| | 12560 | 12720 | 12720 | 12720 | 12720 | 12720 |
| DR35 | 1.87 | 15.7 | 51.1 | 51.1 | 51.1 | 51.1 |
| | 550 | 2990 | 3698 | 3698 | 3698 | 3698 |
| DR70 | 8.93 | 110 | 468 | 483 | 509 | 536 |
| | 2318 | 12110 | 14157 | 14157 | 14157 | 14157 |
| EDF | 0.22 | 0.28 | 1.22 | 2.77 | 3.70 | 4.26 |
| | 2 | 2 | 599 | 1631 | 2248 | 2463 |
| ELF1 | 0.23 | 0.31 | 0.75 | 3.07 | 14.0 | 235 |
| | 1 | 2 | 72 | 472 | 2648 | 46188 |
| ELF2 | 0.30 | 0.91 | 3.33 | 8.92 | 19.8 | 23.6 |
| | 6 | 127 | 395 | 1025 | 4805 | 4805 |
| ELF3 | 0.39 | 1.11 | 3.58 | 10.0 | 29.8 | 150 |
| | 22 | 124 | 388 | 1527 | 4979 | 24386 |

## 5. COMMENTARY

CS-MC performs well even in difficult examples. The number of processed cuts is usually not much larger than the number of minimal cuts of the fault trees, and is much smaller in many cases than the number of cuts processed by the basic top-down and bottom-up algorithms. Compared with recent algorithms, based on BDD representations of the fault tree [8, 18, 20, 24], CS-MC seems to be slower. However, CS-MC has small memory requirements whereas in the worst-case the algorithms based on BDD representations require memory that is exponential in the number of basic events. Since CS-MC generates a list of minimal cuts, it becomes impractical when the number of minimal cuts is extremely large. In that case, the cardinality of the minimal cuts searched can be limited, with the beneficial side effect of reducing the time requirements.

## APPENDIX

### A.1 Proof of Theorem 1

By contradiction. Let $m = c_1[n_1]c_2[n_2]\ldots c_k[n_k]$ be a minimal-cut not satisfying the condition. Then, there exists $c_i[n_i]$ part of $m$ such that $c_i[n_i] \notin I$. Assume that there exists $c_i[j] \in I$ with $j < n_i$. Let $J_i$ be the greatest of such integers; consider the bag $m'$ obtained from $m$ by substituting $c_i[n_i]$ by $c_i[J_i]$. If $c_i[j] \in I$ with $j < n_i$ does not exist, let $m'$ be the bag obtained by eliminating $c_i[n_i]$ from $m$. Clearly, $m'$ performs the same implications as $m$ does. Therefore, $m'$ is a cut, but being $m' \subset m$, $m$ is not minimal.

### A.2 Proof of Theorem 2

Let $m$ be any cut obtained by implying more inputs and by reducing the set of inputs implied at 1. Because no $c_i[j] \in I$, $j > n_i$ exists, $c_i[n_i] \in m$. Assume that there exists at least one $c_i[j] \in I$, $j < n_i$, and let $J_i$ be the greatest integer $j < n_i$ with $c_i[j] \in I$. Consider the bag $m'$ obtained from $m$ by substituting $c_i[n_i]$ by $c_i[J_i]$. If there does not exists any $c_i[j] \in I$, $j < n_i$, let $m'$ be the bag obtained from $m$ by eliminating $c_i[n_i]$. Because $c_i[n_i]$ is irrelevant, the value of $g_r$ is not affected by the value of $c_i[n_i]$, and $m'$ is also a cut. Furthermore, $m' \subset m$, implying that $m$ is not minimal.

### A.3 Proof of Theorem 3

Necessity & sufficiency are shown.

· *Necessity:* If there does exist any $c_i[n_i] \in m$ such that its unimplication followed, if existent, by the implication of $c_i[J_i]$, where $J_i$ is the greatest integer $< n_i$ with $c_i[J_i] \in I$, leaves $g_r$ implied at 1, the bag $m'$ obtained from $m$ by either deleting $c_i[n_i]$ or replacing $c_i[n_i]$ by $c_i[J_i]$ is also a cut, and being $m' \subset m$, $m$ is not minimal.

· *Sufficiency:* Assume that $g_r$ is unimplied for each unimplication, followed, if existent, by implication of $c_i[J_i]$, and assume that there exists a bag $m' \subset m$ that is a cut. Being $m' \subset m$, there exists $m''$, $m' \subseteq m'' \subset m$ such that $m''$ is obtained from $m$ by either deleting $c_i[n_i]$ for some $1 \le i \le k$ or replacing for some $1 \le i \le k$ $c_i[n_i]$ by $c_i[n_i']$ with $n_i' < n_i$. In both cases $g_r$ is not implied at 1 by $m''$ and $m''$ is not a cut, implying that neither is $m'$ a cut. Then, there does not exists any cut $m' \subset m$, and $m$ is minimal.

## ACKNOWLEDGMENT

We are pleased to thank Antoine Rauzy from Université Bordeaux I and Jean C. Madre from Synopsys for providing us the fault trees ELF1, ELF2, ELF3; and to thank Angel Calderón for helping us in the early stages of our work.

## REFERENCES

[1] M. Abramovici, J.J. Kulikowski, P.R. Menon, D.T. Miller, "SMART and FAST: Test generation for VLSI scandesign circuits", *IEEE Design and Test of Computers*, vol 3, 1986 Aug, pp 43 – 54.

[2] 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, 1976 Oct, pp 1442 – 1446.

[3] R.G. Bennets, "On the analysis of fault trees", *IEEE Trans. Reliability*, vol R-24, 1975 Aug, pp 175 – 185.

[4] R. Bryant, "Symbolic Boolean manipulation with ordered binary decision diagrams", *ACM Computing Surveys*, vol 24, 1992, pp 293 – 318.

[5] J.A. Carrasco, "Improving availability bounds using the failure distance concept", *Dependable Computing and Fault-Tolerant Systems*, vol 5, 1995, pp 479 – 497; Springer-Verlag.

[6] J.A. Carrasco, J. Escribá, A. Calderón, "Efficient exploration of availability models guided by failure distances", *ACM Performance Evaluation Review*, vol 24, 1996 no. 1, May, pp 242 – 251.

[7] P. Chatterjee, "Modularization of fault trees: A method to reduce the cost of analysis", *Reliability and Fault Tree Analysis*, 1975, pp 101 – 126; SIAM.

[8] O. Coudert, J. C. Madre, "MetaPrime: An interactive fault-tree analyzer", *IEEE Trans. Reliability*, vol 43, 1994 Mar, pp 121 – 127.

[9] Y. Dutuit, A. Rauzy, "A linear-time algorithm to find modules of fault trees", *IEEE Trans. Reliability*, vol 45, 1996 Sep, pp 422 – 425.

[10] H. Fujiwara, T. Shimano, "On the acceleration of test generation algorithms", *IEEE Trans. Computers*, vol C-32, 1983 Dec, pp 1137 – 1144.

[11] J.B. Fussell. W. E. Vesely, "A new methodology for obtaining cut sets for fault trees", *Trans. American Nuclear Society*, vol 15, 1972 Jun, pp 262 – 263.

[12] P. Goel, "An implicit enumeration algorithm to generate tests for combinational logic circuits", *IEEE Trans. Computers*, vol C-30, 1981 Mar, pp 215 – 222.

[13] L.H. Goldstein, "Controllability/observability analysis of digital circuits", *IEEE Trans. Circuits and Systems*, vol CAS-26, 1979 Sep, pp 685 – 693.

[14] W. Hennings, N. Kuznetsov, "FAMOCUTN & CUTQN: Programs for fast analysis of large fault trees with replicated & negated gates", *IEEE Trans. Reliability*, vol 44, 1995 Sep, pp 368 – 376.

[15] T. Kohda, E.J. Henley, K. Inoue, "Finding modules in fault trees", *IEEE Trans. Reliability*, vol 38, 1989 Jun, pp 165 – 176.

[16] N. Limnios, R. Ziani, "An algorithm for reducing cut sets in fault-tree analysis", *IEEE Trans. Reliability*, vol R-35, 1986 Dec, pp 559 – 561.

[17] K. Nakashima, Y. Hattori, "An efficient bottom-up algorithm for enumerating minimal cut sets of fault trees", *IEEE Trans. Reliability*, vol R-28, 1979 Dec, pp 353 – 357.

[18] K. Odeh, N. Limnios, "A new algorithm for fault trees prime implicant computations", *ESREL'96*, Crete (Greece), 1996 Jun, pp 1085 – 1090.

[19] D.M. Rasmuson, N.H. Marshall, "FATRAM – A core efficient cut-set algorithm", *IEEE Trans. Reliability*, vol R-27, 1978 Oct, pp 250 – 253.

[20] A. Rauzy, "New algorithms for fault trees analysis", *Reliability Engineering and System Safety*, vol 40, 1993, pp 203 – 211.

[21] A. Rosenthal, "A computer scientist looks at reliability computations", *Reliability and Fault Tree Analysis*, 1975, pp 133 – 152; SIAM.

[22] A. Rosenthal, "Decomposition methods for fault tree analysis", *IEEE Trans. Reliability*, vol R-29, 1980 Jun, pp 136 – 138.

[23] S. N. Semanders, "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.

[24] R.M. Sinnamon, J.D. Andrews, "Fault tree analysis and binary decision diagrams", *Proc. Ann. Reliability and Maintainability Symp*, 1996, pp 215 – 222.

[25] V. Suñé, J.A. Carrasco, "A method for the computation of reliability bounds for non-repairable fault-tolerant systems", *Proc. 5th Int'l Symp. on Modeling, Analysis and Simulation of Computers and Telecommunication Systems* (MASCOTS'97), 1997 Jan, pp 221 – 228, Haifa.

[26] J.M. Wilson, "Modularizing and minimizing fault trees", *IEEE Trans. Reliability*, vol R-34, 1985 Oct, pp 320 – 322.

## AUTHORS

Prof. Juan A. Carrasco; Dep't d'Enginyeria Electrònica, UPC; Diagonal 647 plta. 9; 08028 Barcelona SPAIN.
*Internet (e-mail):* carrasco@eel.upc.es

**Juan A. Carrasco** (Member IEEE) has been 'Profesor Titular' at the Dep't d'Enginyeria Electrònica of UPC (Polytechnical University of Catalonia) since 1988. His research is focused on modeling of digital systems and, more specifically, fault-tolerant systems. He received the Engineer (1981) degree in Industrial Engineering from UPC; the MSc in Computer Science (1987) from Stanford Univ; and the DocEng (1987) from UPC. He is coleading a research group on Reliability and Fault-Tolerance of Electronic Systems at the Dep't d'Enginyeria Electrònica of UPC. He has published more than 30 papers in high-quality journals and conferences and has been on the program committee of several IEEE international conferences.

Víctor Suñé; Dep't d'Enginyeria Electrònica, UPC; Diagonal 647 plta. 9; 08028 Barcelona SPAIN.
*Internet (e-mail):* sunye@eel.upc.es

**Víctor Suñé** has been Research Assistant at the Dep't d'Enginyeria Electrònica of UPC (Polytechnical University of Catalonia) since 1996. His research is focused on bounding techniques for reliability estimation of fault-tolerant systems, which is his dissertation topic. He received the Engineer (1995) degree in Industrial Engineering from UPC.

Manuscript TR97-105 received: 1997 July 30;
revised: 1998 March 10, July 17
Responsible editor: J.B. Dugan
Publisher Item Identifier S 0018-9529(99)03924-X