Reliability Engineering and System Safety **40** (1993) 203–211

New algorithms for fault trees analysis

Antoine Rauzy

LaBRI, CNRS-Université Bordeaux I, 351, cours de la Libération 33405 Talence Cedex, France

(Received 20 February 1992; accepted 2 September 1992)

In this paper, a new method for fault tree management is presented. This method is based on binary decision diagrams and allows the efficient computation of both the minimal cuts of a fault tree and the probability of its root event. We show on a set of benchmarks that our method results in a qualitative and quantitative improvement in safety analysis of industrial systems.

1 INTRODUCTION

The fault tree is one of the most commonly used methods for safety analysis of industrial systems. A difficulty in this method is the computation of the minimal cuts of the tree, i.e. the minimal combinations of elementary faults that make the whole system fail. This computation is NP-hard. In this paper, we propose an algorithm, based on Bryant's trees, which can be used to compute and memorize these cuts very efficiently.

Binary decision diagrams (BDDs) were introduced by Bryant[1,2] and are now used in many domains, for instance the symbolic verification of digital circuits[3,4] and the implementation of constraint logic programming languages.[5]

A BDD is a graph encoding Shannon's decomposition of a formula. BDDs have two important features: (1) the graphs are compacted by sharing equivalent subgraphs, and (2) the results of operations performed on BDDs are memorized and thus a job is never performed twice. These two features make BDDs one of the most efficient methods for Boolean formulae management.

The algorithm we propose consists in computing from the BDD encoding a fault tree, a second BDD encoding its minimal cuts. The procedures we have defined use all the functionalities of a BDD package, in particular the ability to memorize all the operations. The power of this mechanism allows an improvement in efficiency of several orders of magnitude with respect to 'classical' methods.

In addition, since a BDD encodes the Shannon's decomposition of a fault tree, the computation of the probability of the root event, given the probabilities of the leaves (terminal events) becomes very easy.

This paper is organized as follows: some definitions and basic properties of Boolean formulae and functions are given in the first section. In the second section BDDs are presented. In the third section, the algorithm computing minimal cuts is described. Finally, the computation of the probability of the root event is presented.

Reliability Engineering and System Safety 0951-8320/93/$06.00

2 FAULT TREES

2.1 Assignments, Boolean functions and formulae

In this section we will briefly introduce the vocabulary used.

Definition 1: Let $X = \{x_1, \ldots, x_n\}$ be a set of Boolean variables. An *assignment* of X is a mapping from X into $\{0, 1\}$.

An assignment σ can be viewed as the characteristic function of a subset of X ($\sigma(x) = 1$ iff x belongs to the subset). It follows that the set Σ_X of assignments on X is isomorphic to $P(X)$ the powerset of X. Thus, we can define a partial order $\leq$ on Σ_X as the inclusion in $P(X)$.

Definition 2: A *Boolean function f* on X is a mapping from Σ_X into $\{0, 1\}$.

Thus, in the same way as above, the set Φ_X of the Boolean functions on X is isomorphic to $P(P(X))$, and we can define a partial order $\leq$ on Φ_X as the inclusion in $P(P(X))$.

Assignment and function are semantic objects. A

unique Boolean function corresponds to each Boolean formula built over the set X of variables and a given set of connectives (of course, the converse is not true). In order to distinguish between functions and formulae we will denote functions with lower-case letters and formulae with upper-case letters.

By abuse of notations, we will sometimes write that a variable belongs to an assignment or that an assignment belongs to a function when the corresponding relations are verified in the corresponding powersets.

Definition 3: An assignment σ is a *solution* of a Boolean formula F if σ belongs to the corresponding Boolean function f.

Definition 4: A solution σ of a Boolean formula F is *minimal* if for any other assignment σ', $\sigma' < \sigma$ implies that $\sigma'(f) = 0$.

The set of solutions of F is denoted by $\mathrm{Sol}(F)$, the set of its minimal solutions is denoted by $\mathrm{Sol}_{\min}(F)$.

Definition 5: Let f be a Boolean function. f is monotonic if for any assignment σ and σ', $\sigma(f) = 1$ and $\sigma \leq \sigma'$ imply that $\sigma'(f) = 1$.

By considering an assignment as the conjunction of the variables to which it gives the value 1, it is possible to characterize the monotonic functions:

Property 1: A function f is monotonic iff it is equivalent to the disjunction of its minimal solutions.

2.2 Fault trees

The presentation of fault trees we give here is, of course, incomplete. The interested reader should see Ref. 6 for a detailed monograph.

The fault trees we deal with are defined by means of sets of equations in the form:

$$\begin{aligned} x &= y_1 \text{ or } y_2 \text{ or } \ldots \text{ or } y_n \\ \text{or} \quad x &= y_1 \text{ and } y_2 \text{ and } \ldots \text{ and } y_n \\ \text{or} \quad x &= \text{at-least } k \text{ in } y_1, y_2, \ldots, y_n. \end{aligned}$$

such that: (1) and x, y_1, $y_2, \ldots, y_n$ are Boolean variables (events), (2) a variable cannot appear as the left-hand member of two equations, (3), all the variables but one (the root event) occur at least once in a right-hand member, (4) there is no cycle, i.e. no subset of equations in the form: $x_1 = \mathrm{op}(\ldots, x_2, \ldots)$, $x_2 = \mathrm{op}(\ldots, x_3, \ldots), \ldots, x_n = \mathrm{op}(\ldots, x_1, \ldots)$.

Example 1.
Figure 1 is defined by three equations:

$$\begin{aligned} a &= (b1 \text{ or } b2) \\ b1 &= (c1 \text{ and } c2) \\ b2 &= (c2 \text{ and } c3) \end{aligned}$$

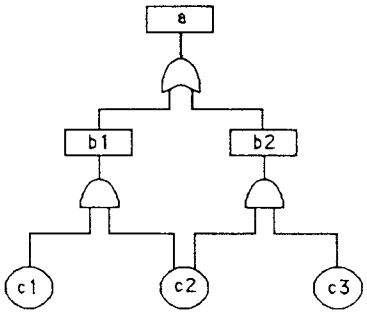

Fig. 1. A fault tree.

A complete tree is given in the appendix. This set of equations describes a directed acyclic graph (DAG). The internal nodes are either connectives or events. A connective has only one ascendant. The sink nodes (or leaves) are terminal or elementary events.

This fault tree encodes the following Boolean function:

$$F(c1, c2, c3) = (c1 \wedge c2) \vee (c2 \wedge c3)$$

There are eight possible assignments of the three variables $c1$, $c2$, $c3$, denoted, in terms of sets by:

$$\begin{aligned} \varnothing&: c1 = 0, c2 = 0, c3 = 0, \\ \{c1\}&: c1 = 1, c2 = 0, c3 = 0, \\ \{c2\}&: c1 = 0, c2 = 1, c3 = 0, \\ \{c3\}&: c1 = 0, c2 = 0, c3 = 1, \\ \{c1, c2\}&: c1 = 1, c2 = 1, c3 = 0, \\ \{c1, c3\}&: c1 = 1, c2 = 0, c3 = 1, \\ \{c2, c3\}&: c1 = 0, c2 = 1, c3 = 1, \\ \{c1, c2, c3\}&: c1 = 1, c2 = 1, c3 = 1, \end{aligned}$$

$F(c1, c2, c3)$ admits three solutions: $\{c1, c2\}$, $\{c2, c3\}$ and $\{c1, c2, c3\}$. The first two are minimal since there is no subset of these two sets that is a solution. $F(c1, c2, c3)$ is monotonic since any superset of a solution is also a solution. In fact, the connectives and, or and at-least are monotonic thus the following property holds.

Property 2: The fault trees we deal with encode monotonic functions.

Let us recall that the solutions of a fault tree are also called *cuts*.

3. BINARY DECISION DIAGRAMS

We review here basic definitions and properties of BDDs for reference. For a complete presentation, the reader is referred to two papers by Bryant.[1,2]

3.1 Shannon's decomposition

Theorem 3: Shannon's decomposition: let f be a Boolean function on X, and x be a variable of X, then $f = (x \wedge f_{\{x=1\}}) \vee (\neg x \wedge f_{\{x=0\}})$, where f evaluated in $x = v$ is denoted by $f_{\{x=v\}}$.

In order to make the notations correspond with the intuitive notion of binary tree induced by the Shannon's decomposition of a function, we introduce the ite (If–Then–Else) connective:

Definition 6: $\text{ite}(F, G, H) = (F \wedge G) \vee (\neg F \wedge H)$.

Definition 7: Let F be a Boolean formula. F is said to be in Shannon's form if either it is a constant or it is in the form $\text{ite}(x, F, G)$ where x is a variable and F and G are formulae in Shannon's form in which x doesn't occur.

Example 2
Let $F: (a \vee c) \wedge (b \vee c)$. Then, G: $\text{ite}(a, \text{ite}(b, 1, \text{ite}(c, 1, 0)), \text{ite}(c, 1, 0))$ is a formula in Shannon's form, equivalent to F.

Property 4: For any formula F a formula G exists in Shannon's form equivalent to F.

Remark: the Shannon's form of a formula encodes its truth table.

3.2 Binary decision diagrams

Definition 8: Let $<$ be a total order on variables. A *binary decision diagram* (BDD) is a directed acyclic graph (DAG). A BDD has two leaves: 0 and 1 encoding the two corresponding constant functions. Each internal node encodes an ite connective, i.e. it is labelled with a variable x and has two out-edges. These two edges are called 0-edge and 1-edge. An internal node descendant from a node labelled by a variable x is labelled by a variable y such that $x < y$.

BDDs compactly encode formulae in Shannon's form by means of subtree sharing. A set of formula can thus be represented with a single multirooted BDD.

Example 3
The BDD associated with the formula:

$$F = (a \vee c) \wedge (b \vee c)$$
$$= \text{ite}(a, \text{ite}(b, 1, \text{ite}(c, 1, 0)), \text{ite}(c, 1, 0)$$

is shown in Fig. 2.

3.3 Memory management for BDDs

The practical efficiency of BDDs derives from the way in which the memory is managed. A major idea, due to Bryant, consists in using a hashtable to store the ite

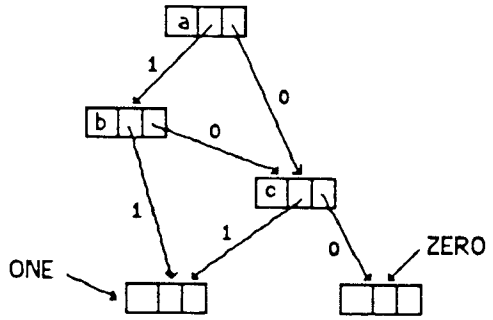

Fig. 2. The BDD associated with $(a \vee c) \wedge (b \vee c)$.

nodes: the ite-table. When, during a computation, a node $\text{ite}(x, F, G)$ is required—where x is a variable and F and G are two addresses in the table—the table is consulted and the node is created only if it is not yet stored.

The use of such a table warrants the encoding of two equivalent Boolean formulae at the same address. The sharing of equivalent subtrees to one or several BDDs is thus automatically performed.

3.4 Logical operations on BDDs

In order to compute the BDD associated with a formula F, the following principle is applied: if F is a constant then one associates with F the corresponding leaf, if $F = x$ where x is a variable then one associates with F the BDD $\text{ite}(x, 1, 0)$, finally if $F = G \lozenge H$, where $\lozenge$ is a binary connective and G and H are formulae* then one computes the BDDs associated with G and H and then performs the operation $\lozenge$ on these two BDDs. Given two Boolean functions g and h encoded by the BDDs $G = \text{ite}(x, G1, G2)$ and $H = \text{ite}(y, H1, H2)$, it is possible to compute directly on G and H any logical operation between g and h by means of the following property:

Property 5: Let x and y be two variables $(x < y)$, let $G1$, $G2$, $H1$, $H2$ be four formulae and let be $\lozenge$ any binary connective $(\wedge, \vee, \Leftrightarrow, \oplus, \ldots)$, then the following equalities hold:

$$\text{ite}(x, G1, G2) \lozenge \text{ite}(x, H1, H2) = \text{ite}(x, G1 \lozenge H1, G2 \lozenge H2) \quad (5.1)$$

$$\text{ite}(x, G1, G2) \lozenge \text{ite}(y, H1, dH2) = \text{ite}(x, G1 \lozenge \text{ite}(y, H1, H2), G2 \lozenge \text{ite}(y, H1, H2)) \quad (5.2)$$

* If $\lozenge$ is not a binary connective then F is rewritten implicitly in a equivalent formula with a binary connective as root connective. For instance, let F be at-least $(k, F1, \ldots, Fn)$ then F is rewritten implicitly in $(F1 \wedge \text{at-least } (k-1, F2, \ldots, Fn)) \vee \text{at-least } (k, F2, \ldots, Fn)$.

```
computation (op, F, G) ≡
   if ((F=0) or (F=1) or (G=0) or (G=1))
      return op(F, G) /* call to the truth table of op */
   else if (computation-table has entry {⟨op, F, G⟩, R})
      return R
   else let x be the least variable of F and G
      U←computation (op, F_{x=1}, G_{x=1}),
      V←computation (op, F_{x=0}, G_{x=0})
      if (U=V) return U
      else R←find-or-add-ite-table (x, U, V)
         insert-in-computation-table ({⟨op, F, G⟩, R})
         return R
```

Fig. 3. BDDs: the main algorithm.

These two equalities suggest clearly a recursive procedure (Fig. 3).

In the algorithm described below, a second hashtable (the computation table) has been introduced in order to memorized quadruples $\{\langle \text{op}, F, G\rangle, R\}$, where op is a logical operation and F, G and R are addresses in the ite-table. Thus, F op G will be performed only if the computation table does not have an entry for this operation. With this table the algorithm 'learns' in some sense from the computations it peforms: the more computations it has done, the more efficient it is.

Remark: Bryant notes that he has obtained best performances with a hashcash rather than a hashtable for the computation table (a hashcash is a hashtable without collision chain).[2] We have made a different choice: in or implementation all the operations are memorized.

Example 4

Assume that we have built the two BDDs $G = \text{ite}(a, 1, \text{ite}(c, 1, 0))$ and $H = \text{ite}(b, 1, \text{ite}(c, 1, 0))$ encoding respectively the subformulae $(a \vee c)$ and $(b \vee c)$ of the formula of example 2. Now, in order to compute the BDD associated with the formula itself, we need to compute $G \wedge H$.

$$
\begin{aligned}
G \wedge H &= \text{ite}(a, 1 \wedge \text{ite}(b, 1, \text{ite}(c, 1, 0)), \text{ite}(c, 1, 0) \\
&\quad \wedge \text{ite}(b, 1, \text{ite}(c, 1, 0)) \\
&= \text{ite}(a, \text{ite}(b, 1, \text{ite}(c, 1, 0)), \text{ite}(c, 1, 0) \\
&\quad \wedge \text{ite}(b, 1, \text{ite}(c, 1, 0))) \\
&= \text{ite}((a, \text{ite}(b, 1, \text{ite}(c, 1, 0)), \text{ite}(b, 1 \\
&\quad \wedge \text{ite}(c, 1, 0), \text{ite}(c, 1, 0) \wedge \text{ite}(c, 1, 0))) \\
&= \text{ite}(a, \text{ite}(b, 1, \text{ite}(c, 1, 0)), \text{ite}(b, \text{ite}(c, 1, 0), \\
&\quad \text{ite}(c, 1, 0))) \\
&= \text{ite}(a, \text{ite}(b, 1, \text{ite}(c, 1, 0)), \text{ite}(c, 1, 0))
\end{aligned}
$$

i.e. The BDD shown in Fig. 2.

3.5 Complexity

To conclude this presentation of BDDs, let us give the following two complexity results:

Property 6: The maximal size of a BDD encoding a formula on n variables is $2^n/n$; the complexity in the worst case of $F \Diamond G$, where F and G are two BDDs and $\Diamond$ any binary operation is in $O(|F| \,.\, |G|)$, where $|F|$ denotes the size of F.

Of course, these worst case complexity results are not very significant; in practice the complexity is far better. At the moment, no definitive results on average complexity are known.

4 COMPUTATION OF MINIMAL SOLUTIONS

4.1 Solutions defined by the paths in a BDD

Let f be a Boolean function and F the BDD associated with f. Each path from the root of F to leaf 1 defines a solution of f ($x \in \sigma$ iff the path goes through a node labelled by x and goes out of this node on the 1-edge). The set of solutions defined by the paths of a BDD F is computed by the procedure shown in Fig. 4. This procedure must be called with $\sigma = \varnothing$).

All these solutions are not minimal. Nevertheless, the following property holds:

Property 7: Let f be a Boolean function encoded by the BDD F and let σ be a solution of f, then a path exists from the root of F to leaf 1 which defines a solution δ of f such that δ is included in σ.

```
solutions (F, σ) ≡
   if (F=0) return ∅
   else if (F=1) return {σ}
   else /* F=ite(x, F1, F2) */
      S←solutions(F1, σ∪{x}),
      T←solutions(F2, σ),
      return S∪T
```

Fig. 4. Solutions defined by a BDD.

Proof: three cases are possible:

(1) $F = 0$ and f doesn't have any solution.
(2) $F = 1$ and f is a tautology. Thus, the empty assignment (all the variables assigned to 0) is a solution of f included in σ.
(3) $F = \text{ite}(x, G, H)$ and two cases are possible:

$x \in \sigma$ and σ and $\sigma \backslash \{x\}$ is a solution of G
$x \notin \sigma$ and σ is a solution of H

It follows that the property is demonstrated by induction on the size of F.

Corollary 8: If σ is a minimal solution of f, then there exists only one path from the root of F to leaf 1 defining σ.

Example 5
In the BDD shown in Fig. 2, there are three paths from the root to leaf 1: $\{a, b\}$, $\{a, c\}$ and $\{c\}$. $\{a, b\}$ and $\{c\}$ correspond to the two minimal solutions of the function. $\{a, c\}$ is not minimal.

In order to compute and memorize the minimal solutions of a function f, we will compute from the BDD F encoding f, a new BDD G such that the set of paths from the root of G to leaf 1 defines exactly the minimal solutions of f.

For the sake of simplicity, we will abbreviate the path from the root to the leaf 1 by path, when it will be clear from the context.

4.2 Characterization of the minimal solutions

Theorem 9: Let $F = \text{ite}(x, G, H)$ be a monotonic Boolean formula in Shannon's form. Then, the following equality holds:

$$\text{Sol}_{\min}(F) = \{\sigma \mid (\sigma = \delta \cup \{x\}) \wedge (\delta \in \text{Sol}_{\min}(G)) \wedge (\delta(H) = 0)\} \cup \text{Sol}_{\min}(H)$$

Proof (sketch): If ρ is a minimal solution of H then ρ is a solution of F. In addition, ρ is minimal since any other solution of F is either a solution of H (i.e. no smaller than ρ) or a solution of G augmented of x (and ρ doesn't contain x).

Now, if δ is a minimal solution of G, then $\delta \cup \{x\}$ is a solution of F. If, in addition, δ is not a solution of H, then a solution of F smaller than $\delta \cup \{x\}$ doesn't exist and $\delta \cup \{x\}$ is minimal.

Example 6
By means of Theorem 9, the minimal solutions of the BDD: ite($(a$, ite($(b$, 1, ite(c, 1, 0)), ite(c, 1, 0)) (Fig. 2) are: the minimal solutions of the BDD ite(c, 1, 0)—i.e. $\{c\}$—the minimal solutions of the BDD ite(b, 1, ite(c, 1, 0))—i.e. $\{b\}$ and $\{c\}$—that are not solutions of the BDD ite(c, 1, 0)—i.e. $\{b\}$—augmented of a—i.e. $\{a, b\}$. Finally, we have:

$$\text{Sol}_{\min}(\text{ite}(a, \text{ite}(b, 1, \text{ite}(c, 1, 0)), \text{ite}(c, 1, 0))) = \{\{a, b\}, \{c\}\}.$$

4.3 The '\' (without) operator

Let $F = \text{ite}(p, G, H)$ be a BDD. Theorem 9 suggests a recursive algorithm in order to compute the BDD $F_{\min}$ encoding the minimal solutions of F: it consists in computing $G_{\min}$ and $H_{\min}$ encoding the minimal solutions of G and H, then removing from $G_{\min}$ all the paths included in a path of H and finally composing the two obtained BDDs with x in order to obtain $F_{\min}$. The point is then to define the 'remove' operation. We will call this the without operator and denote it by '\'.

Let $F = \text{ite}(x, F1, F2)$ and $G = \text{ite}(y, G1, G2)$ be two BDDs. Three cases are possible:

(1) $x < y$, in this case, the paths of F which are not included in a path of G are, on the one hand, the paths of $F1$ augmented by x which are not included in a path of G, and on the other hand, the paths of $F2$ which are not included in a path of G. Thus we have: $F \backslash G = \text{ite}(x, F1 \backslash G, F2 \backslash G)$.
(2) $x > y$, in this case, the paths of G containing y cannot contain a path of F. Then we have, $F \backslash G = F \backslash G2$.
(3) $x = y$, in this case, let σ be a path of F. Two cases are possible:
$x \in \sigma$ and σ can be included either in a path of $G1$ or in a path of $G2$
$x \notin \sigma$ and σ can be included only in a path of $G2$.

It follows that $F \backslash G = \text{ite}(p, F1 \backslash (G1 \text{ or } G2), F2 \backslash G2)$.

The meaning of 'or' has not yet been defined. Here we will use the monotonicity of the studied functions.

Property 10: Let $F = \text{ite}(x, G, H)$ be a monotonic Boolean formula in Shannon's form. Then, G and H are monotonic.

The proof follows from the definitions

Property 11: Let $F = \text{ite}(x, G, H)$ be a monotonic Boolean formula in Shannon's form. Then, for each solution σ of H a solution δ of G exists included in σ.

Here too, the proof follows from the definitions.

Corollary 12: Let $F = \text{ite}(p, F1, F2)$ be a BDD encoding a monotonic solution. Then, for each path σ of $F2$ there exists a path δ of $F1$ included in σ.

Corollary 13: Let $F = \text{ite}(p, F1, F2)$ and $G = \text{ite}(p, G1, G2)$ be two BDDs such that G encodes a monotonic function. Then, $F \backslash G = \text{ite}(\text{p}, F1 \backslash G1, F2 \backslash G2)$.

It follows that the 'or' operation is useless!

```
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-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-ite-table (x, U, V)
                insert-in-computation-table (⟨without, F, G⟩, R)
                return R
```

Fig. 5. Algorithm for computing $F \backslash G$.

4.4 Algorithms

Theorem 9 and properties 10–13 thus allow the description of an algorithm which computes the BDD encoding the minimal solutions of a monotonic Boolean function from the BDD encoding this function. It is given in Figs 5 and 6.

Example 7

Let us apply the algorithm on ite(a, ite(b, 1, ite(c, 1, 0)), ite(c, 1, 0)):

solmin(ite(a,ite(b, 1, ite(c, 1, 0)), ite(c, 1, 0))) =

$K1$ = solmin(ite(b, 1, ite(c, 1, 0)))

$K11$ = solmin(1) = 1

$U11$ = without(1, ite(c, 1, 0)) = 1

$V11$ = solmin(ite(c, 1, 0) = ite(c, 1, 0)/*here we have shortened */

thus, $K1$ = ite(b, 1, ite(c, 1, 0))

$U1$ = without($K1$, ite(c, 1, 0)) = without (ite(b, 1, ite(c, 1, 0)), ite(c, 1, 0))

$U12$ = without(1, ite(c, 1, 0)) = 1

$V12$ = without(ite(c, 1, 0), ite(c, 1, 0)) = 0

Thus $U1$ = ite(b, 1, 0)

$V1$ = solmin(ite(c, 1, 0)) = $V11$ = ite(c, 1, 0) /* memorization of results! */

Thus, the result is ite(a, ite(b, 1, 0), ite(c, 1, 0)).

It is now easy to verify that the solutions of this BDD obtained by the procedure in Fig. 4 are $\{a, b\}$ and $\{c\}$, i.e. the minimal solutions of the considered formula.

4.5 Practical results

4.5.1 Benchmarks

We have tested our algorithm on 13 fault trees from industrial practice: nine of these were proposed by

```
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)    /* H is monotonic */
      V←minsol(H)
      R←find-or-add-ite-table(x, U, V)
      insert-in-computation-table({⟨minsol, F, _⟩, R})
      return R
```

Fig. 6. Algorithm for computing minimal solutions.

Table 1. Statistical elements on Dassault's trees

	Trees								
	1	2	3	4	5	6	7	8	9
Variables	103	122	51	53	51	121	276	109	49
Connectives	145	82	30	30	20	112	324	73	36
Size	248	204	81	83	71	233	600	182	85

Table 2. Statistical elements on European trees

	Trees			
	Chinese	European 1	European 2	European 3
Variables	25	61	32	80
Connectives	36	84	40	107
Size	61	145	72	187

Dassault Aviation, and four by Professor Y. Dutuit (one of these trees is completely described in the appendix). Tables 1 and 2 lists some elements of these trees.

4.5.2 Results

Tables 3 and 4 gives the computation times, the number of minimal solutions and the size of the obtained BDDs. The running times include the choice of an index for each variable, the computation of the BDD associated with the tree and finally the computation of the BDD encoding its minimal cuts.

In order to give some comparison elements, Table 5 gives the running times obtained for the Dassault's trees with a classical algorithm using de Morgan's laws.

4.5.3 Heuristics

The size of a BDD (and thus the time necessary to achieve its computation) depends greatly on the order chosen for the variables. In the first version of our program, the variables occurred in the set of equations describing the tree. With this rough order, the memory of our machine was insufficient for three of the nine Dassault's trees (i.e. 100 0000 ite nodes and 200 000 quadruples).

Table 4. Running times (minimal cuts) for European trees

	Chinese	European 1	European 2	European 3
Times	0s50	4s36	0s31	6s81
Solutions	392	46 188	4 805	24 386
Size	56	6 044	183	2 590

Table 5. Runing times obtained with a classical algorithm

	Trees								
	1	2	3	4	5	6	7	8	9
Time	10 577	2 470	147	130	74	43 279	84 383	—[a]	—[a]

[a] The computations for the two last trees are not possible in reasonable times.

Thus, we have defined a very simple heuristic in order to find a better ordering: it consists in carrying out a depth-first exploration of the tree and numbering the variables as soon as they appear. The computation times given above have been obtained with this heuristic.

5 PROBABILITY OF THE ROOT EVENT

5.1 Shannon's decomposition

In the previous section, we have shown how BDDs allow the efficient computation of the minimal cuts of a fault tree. It is clear that BDDs allow the probability of the root event of the tree to be computed, given the probabilities of the leaves (terminal events), by means of the Shannon's decomposition.

Theorem 14: Shannon's decomposition (2): let f be a function and x a variable occurring in f then the following equality holds:

$$p(f) = p(x = 1) \cdot p(f_{\{x=1\}}) + p(x = 0) \cdot p(f_{\{x=0\}})$$

5.2 Algorithm

The Shannon's decomposition is carried out as shown in Fig. 7. Note that this procedure uses the BDD

Table 3. Running times (minimal cuts) for Dassault's trees

	Trees								
	1	2	3	4	5	6	7	8	9
Times	3s13	0s56	0s03	0s03	0s01	0s65	4s43	0s05	0s05
Solutions	8 060	14 217	16 200	16 704	17 280	19 518	25 988	$\approx 81 \times 10^9$	27 788
Size	1 056	422	62	69	53	466	1 525	161	51

The program was written in C, compiled with the gcc compiler without optimization option and runs onto a SUN IPX workstation with 16 MBytes of memory.

```
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 /* F=ite(x, G, H) */
      R←p(x).probability(G) + (1−p(x)).probability(H)
      insert-in-computation-table({⟨probability, F, _⟩, R})
      return R
```

Fig. 7. Algorithm for computing $p(F)$.

memorization. It follows that its complexity is linear in the size of the BDD and not in the size of the solutions (i.e. number of solutions × number of events in one solution).

5.3 Benchmarks

Below, the running times shown in Tables 6 and 7 are those necessary for choosing an index for each variable, computing the BDD associated with each tree and the probability of its root event.

Table 6. Running times (probability of the root event) for Dassault's trees

	Trees								
	1	2	3	4	5	6	7	8	9
Times	4s76	0s36	0s03	0s05	0s02	0s81	8s25	0s10	0s06
Solutions	8 060	14 217	16 200	16 704	17 280	19 518	25 988	$\approx 81 \times 10^9$	27 788
Size	1 056	4 22	62	69	53	466	1 525	161	51

Table 7. Running times (probability of the root event) for European trees

	Trees			
	Chinese	European 1	European 2	European 3
Times	0s05	1s71	0s21	4s76
Solutions	392	46 188	4 805	24 386
Size	56	6 044	183	2 590

6 RELATED WORK

Recently, Coudert and Madre have proposed a method based on BDDs in order to compute the prime implicants of a formula.[7] Their method is more powerful than the one we have proposed since it can be applied even on non-monotonic formulae. However, it requires duplication of each variable and the efficiency is decreased (at least by a factor of 20 as far as we have compared the two methods). In addition, the use of the BDD obtained to compute the probability of the root event is not so clear.

7 CONCLUSION

In this paper, we have described new algorithms for fault tree management. The main conclusion stands in the tables describing the benchmarks. These tables show that BDDs are very well-adapted for our purpose and allow a qualitative and quantitative improvement in the safety analysis of embedded systems.

It remains for us to extend the work to the entire fault tree method including non-consistent trees, fuzzy trees and so on.

ACKNOWLEDGEMENT

This work is supported by a collaboration between the LaBRI and Dassault Aviation (Cecilia Project).

REFERENCES

1. Bryant, R., Graph based algorithms for Boolean function manipulation. *IEEE Transactions on Computer*, **35**(8) (1987) 677–91.
2. Brace, K., Rudell, R. & Bryant, R., Efficient implementation of a BDD package. Paper presented at the *27th ACM/IEEE Design Automation Conference*, 1990, IEEE 0738.
3. Billon, J. C. & Madre, P., Formal proof of combinatorial digital circuits. IFIP WG 10.2 July 1988.
4. Burch, J. R., Clark, E. M., McMillan, K. L. & Dill, D. L., Sequential circuit verification using symbolic model checking. *Proceedings Workshop DAC* 1990.
5. *Büttner, W. & Simonis, H., Embedding Boolean*

expressions into logic programming. Journal of Symbolic Computation, **4** (1987) 191–205.
6. Limnios, N., Arbres de Défaillances. Traité des Nouvelles Technologies Edition Hermes, Paris 1992. (in French).
7. Coudert, J. C. & Madre, P., A new method to compute prime and essential implicant of Boolean functions. *Rapport de Recherche BULL* RT/91028, October 1991.

APPENDIX: A FAULT TREE

This is one of the fault trees of our benchmarks. The syntax is as follows: $a := f$ defines the event a by the formula f. The symbols | and & stand for 'or' and 'and', finally @(k, [. . .]) stands for the connective 'at-least k in . . .'.

European 1

root := (g126 & g138 & g144)
g144 := (g111 | g112 | g143 | c053)
g143 := (g139 & g140 & g141 & g142)
g142 := (g065 | g069 | g118 | g132 | c051)
g141 := (g064 | g067 | g117 | g131 | c050)
g140 := (g063 | g068 | g118 | g130 | c049)
g139 := (g062 | g066 | g117 | g129 | c048)
g138 := (g106 | g119 | g137)
g137 := @(3, [g133,g134,g135,g136])
g136 := (g065 | g132 | c061)
g135 := (g064 | g131 | c060)
g134 := (g063 | g130 | c059)
g133 := (g062 | g129 | c058)
g132 := (g065 | g128 | c057)
g131 := (g064 | g127 | c056)
g130 := (g063 | g128 | c055)
g129 := (g062 | g127 | c054)
g128 := (g116 & g120)
g127 := (g115 & g120)
g126 := (g111 | g112 | g125 | c053)
g125 := @(2, [g121,g123,g122,g124])
g124 := (g069 | g118 | c051)
g123 := (g067 | g117 | c050)
g122 := (g068 | g118 | c049)
g121 := (g066 | g117 | c048)
g120 := (g109 | g110)
g119 := (g107 | g108 | c052)
g118 := (g114 | g047)
g117 := (g113 | g046)
g116 := (g103 & g105)
g115 := (g102 & g104)
g114 := (g091 & g093)
g113 := (g090 & g092)
g112 := @(3, [g098,g099,g100,g101])
g111 := @(3, [g094,g095,g096,g097])
g110 := @(3, [g086,g087,g088,g089])
g109 := @(3, [g082,g083,g084,g085])
g108 := @(3, [g078,g079,g080,g081])
g107 := @(3, [g074,g075,g076,g077])
g106 := @(3, [g070,g071,g072,g073])
g105 := (g069 | c045)
g104 := (g067 | c044)

g103 := (g068 | c043)
g102 := (g066 | c042)
g101 := (g069 | c041)
g100 := (g067 | c040)
g099 := (g068 | c039)
g098 := (g066 | c038)
g097 := (g069 | c037)
g096 := (g067 | c036)
g095 := (g068 | c035)
g094 := (g066 | c034)
g093 := (g069 | c033)
g092 := (g067 | c032)
g091 := (g068 | c031)
g090 := (g066 | c030)
g089 := (g065 | c029)
g088 := (g064 | c028)
g087 := (g063 | c027)
g086 := (g062 | c026)
g085 := (g065 | c025)
g084 := (g064 | c024)
g083 := (g063 | c023)
g082 := (g062 | c022)
g081 := (g065 | c021)
g080 := (g064 | c020)
g079 := (g063 | c019)
g078 := (g062 | c018)
g077 := (g065 | c017)
g076 := (g064 | c016)
g075 := (g063 | c015)
g074 := (g062 | c014)
g073 := (g065 | c013)
g072 := (g064 | c012)
g071 := (g063 | c011)
g070 := (g062 | c010)
g069 := (c001 & c009)
g068 := (c001 & c008)
g067 := (c001 & c007)
g066 := (c001 & c006)
g065 := (c001 & c005)
g064 := (c001 & c004)
g063 := (c001 & c003)
g062 := (c001 & c002)

p(c001)=0.01	p(c013)=0.016	p(c024)=0.016	p(c035)=0.016	p(c046)=0.00052	p(c057)=0.015
p(c002)=0.051	p(c014)=0.0218	p(c025)=0.016	p(c036)=0.016	p(c047)=0.00052	p(c058)=0.0188
p(c003)=0.051	p(c015)=0.0218	p(c026)=0.015	p(c037)=0.016	p(c048)=0.018	p(c059)=0.0188
p(c004)=0.051	p(c016)=0.0218	p(c027)=0.015	p(c038)=0.016	p(c049)=0.018	p(c060)=0.0188
p(c005)=0.051	p(c017)=0.0218	p(c028)=0.015	p(c039)=0.016	p(c050)=0.018	p(c061)=0.0188
p(c006)=0.112	p(c018)=0.015	p(c029)=0.015	p(c040)=0.016	p(c051)=0.018	
p(c008)=0.112	p(c019)=0.015	p(c030)=0.0137	p(c041)=0.016	p(c052)=0.000008	
p(c009)=0.112	p(c020)=0.015	p(c031)=0.0137	p(c042)=0.0038	p(c053)=0.000072	
p(c010)=0.016	p(c021)=0.015	p(c032)=0.0137	p(c043)=0.0038	p(c054)=0.015	
p(c011)=0.016	p(c022)=0.016	p(c033)=0.0137	p(c044)=0.0117	p(c055)=0.015	
p(c012)=0.016	p(c023)=0.016	p(c034)=0.016	p(c045)=0.0117	p(c056)=0.015	