Exact and Truncated Computations of Prime Implicants of Coherent and Non-Coherent Fault Trees within Aralia

Yves Dutuit and Antoine Rauzy [1]

LADS and LaBRI
Université Bordeaux I, 351, cours de la Libération
33405 Talence cedex, France
dutuit@iuta.u-bordeaux.fr and rauzy@labri.u-bordeaux.fr

Aralia is a Binary Decision Diagram (BDD) package extended to handle fault trees. It is currently developed at the University of Bordeaux as a part of a partnership between university laboratories and several french companies.
BDD's are the state of the art data structure to handle boolean functions. They have been recently used with success in the framework of safety and reliability analysis.
The aim of this paper is to present how prime implicants (minimal cuts) of coherent and non-coherent fault trees are computed within Aralia. The used algorithms are mainly those proposed by J.C. Madre and O. Coudert one the one hand and A. Rauzy on the other hand. We introduce the notion of minimal p-cuts that is a sound extension of the notion of minimal cuts to the case of non-coherent fault trees. We propose two BDD based algorithms to compute them.
We show how to modify these algorithms in order to compute only prime implicants (or minimal p-cuts) whose orders are less than a given constant or whose probabilities are greater than a given threshold. We report experiments showing that this improves significantly the methodology for this allows fast, accurate and incremental approximations of the desired result.

Key words: Coherent and non-coherent Fault Trees, Prime Implicants, Minimal (p-)Cuts, Binary Decision Diagrams.

[1] This works has been partly supported by the Aralia project that is a partnership between our two laboratories — the Laboratoire d'Analyse de Défaillances des Systèmes (LADS) and the Laboratoire Bordelais de Recherche en Informatique (LaBRI) — and the following compagnies: Commissariat à L'Énergie Atomique (LETI), Dassault Aviation, Électricité de France, Elf Aquitaine, Institut de Protection et de Sûreté Nucléaire, Renault, Schneider Electric, Société Générale pour les techniques Nouvelles (COGEMA) and Technicatome.

1 Introduction

The fault tree method is a well known engineering techniques that is widely used to perform safety analyses of embedded systems [VGRH81,LGTL85]. It is nowadays well understood by most practitioners. However, problems still remain to exploit it fully because of the computational complexity of the basic operations it requires. These difficulties appear both for qualitative treatment (computation of sets of failures of elementary components that induce a failure of the whole system) and for quantitative treatment (determination of the probability of failure of the whole system given the probabilities of failure of its elementary components). Classical techniques — described in the cited books — fail to handle large size non-decomposable fault trees because they cannot avoid combinatorial explosions in both time and space.

Since fault trees are essentially boolean formulae, a way to limit combinatorial explosion is to encode them by means of Binary Decision Diagrams (BDD's). BDD's are the state-of-the-art data structure to encode and manipulate boolean functions. They have been introduced by Akers [Ake78] and improved by Bryant [Bry86,BRB90]. They are nowadays used in a wide range of areas, including hardware design and verification, protocol validation and automated deduction (see [Bry92] for a survey of BDD's and their applications).

This paper presents BDD based algorithms that perform qualitative and quantitative analyses of fault trees. We focus on the computation of prime implicants of non-coherent fault trees. Prime implicants, that are often called minimal cuts in the reliability engineering literature, are of a great interest for qualitative analyses: they can be seen as minimal sets of failures of elementary components that induce a failure of the whole system. Even medium size fault trees may have a very large number of prime implicants. Following Madre and Coudert's [CM92a], Minato's [Min93] and Rauzy's [Rau93] ideas, it is possible to encode them by means of BDD's. The point is that the size of the BDD encoding a set of prime implicants is not directly related to the number of elements of the set. Very large sets can be encoded by means of small BDD's. This is a major difference with the classical sum-of-products representation. Two algorithms have been proposed to compute a BDD encoding prime implicants of a function from the BDD encoding this function. The first one by J.C. Madre and O. Coudert in [CM92b], the second one by one of the authors in [Rau93]. Both use the same inductive decomposition principle. The latter works only for monotone functions (coherent fault trees) but it is more efficient than the former [Ara94]. Both outperform by orders of magnitude already proposed methods. Both are implemented in the BDD package Aralia [Ara94,Ara95]. This tool has been realized as a part of a collaboration between

two laboratories of the Université Bordeaux I and several French companies [2] grouped in a research team of the French institute for system dependability (ISdF [3]).

In this paper, we present Madre and Coudert's algorithm as well as two algorithms to compute minimal p-cuts. Minimal p-cuts are a new kind of implicants we introduce here. They can be seen as a sound extension of the notion of minimal cuts to the case of non-coherent fault trees. Intuitively, they consists of minimal positive parts of prime implicants. They are of interest for two reasons: first, practitioners are often more interested in positive information (what is failed) than in negative information (what works correctly). Second, more accurate approximations can be obtained by computing truncated minimal p-cuts than by computing truncated prime implicants. We shall develop this point later.

We performed with Aralia a number experiments on real-life fault trees. These fault trees are almost impossible to handle by means of classical techniques because they are not only very large (several hundred of gates and primary events) but also non decomposable (without modules). Thanks to some preprocessing of the trees, we always succeeded in computing the BDD encoding them. This allowed us to get the exact probability of their top events. For some of them however, we did not succeed in computing the BDD's encoding their prime implicants and minimal p-cuts, even with computer memories allowing the allocation of several millions of BDD nodes. To tackle this problem, we propose here a modification of the inductive decomposition principle that allows the computation of only prime implicants and minimal p-cuts whose orders are less than a given constant or whose probabilities are greater than a given threshold. The new algorithms are complete in this sense that all of the prime implicants and minimal p-cuts verifying the property are actually obtained. This makes a major difference with classical decomposition/modularization techniques for fault trees, such as those proposed in [CY85]. These techniques, despite of their interest, cannot be made complete in the above sense (excepted of course by computing all of the prime implicants or minimal p-cuts). Moreover, it is possible to compute a BDD encoding a function whose prime implicants (or minimal p-cuts) are exactly those obtained by the truncated computation. This BDD is used to compute the exact probability of this function, which makes it possible to evaluate how accurate the approximation is.

We show by means of examples that the algorithms we propose improve significantly the methodology for they allow fast, accurate, and incremental approximations of the desired results.

[2] See first page.
[3] Institut de Sûreté de Fonctionnement.

The remainder of this paper is organized as follows. In the next section, we first briefly recall basics on fault trees and prime implicants. We present BDD's in section 3. We present Madre and Coudert's algorithm as well as the ones to compute minimal p-cuts in section 4. We propose the algorithms computing truncated prime implicants and minimal p-cuts in section 5. Finally, we report experimental results in section 6.

2 Fault Trees and Prime Implicants

In this section, we recall basic definitions about fault trees and boolean expressions.

2.1 Fault Trees

For the purpose of this paper, *fault trees* are essentially considered as *boolean formulae*, i.e. terms inductively built over the two constants 0 and 1, a set of variables $\mathcal{X}$, and usual logical connectives $\wedge$ (and), $\vee$ (or), $\neg$ (not), ...

Fig. 1 depicts a small fault tree that will be used throughout this paper. g_1 is the *root event* of the tree, g_2 and g_3 are *internal events*, and a, b and c are *primary events*. Primary events are also called *terminal* events. The boolean formula associated with g_1 is $((a \wedge b) \vee (\neg a \wedge c))$. This fault tree is not coherent because of the negation, or in other words because the boolean formula it describes is not monotone. In what follows we simply say g_1 for "the boolean formula associated with g_1".

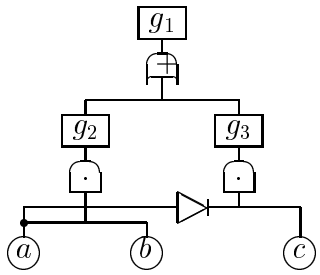

Fig. 1. A non coherent fault tree

2.2 Prime Implicants and Minimal P-Cuts

In order to introduce formally the notion of prime implicants, we need the following definitions.

A *literal* is either a boolean variable x or its negation $\neg x$. x and $\neg x$ are said *opposite*.

A *product* is a set of literals that does not contain both a literal and its opposite. A product is assimilated with the conjunction of its elements.

An *assignment* over $\mathcal{X}$ is any mapping from $\mathcal{X}$ to $\{0,1\}$. Assignments are extended inductively into mappings from boolean formulae into $\{0,1\}$ according to the usual rules: let σ be an assignment and let f and g be two formulae, then $\sigma[f \vee g] = max(\sigma[f], \sigma[g])$, $\sigma[f \wedge g] = min(\sigma[f], \sigma[g])$ and $\sigma[\neg f] = 1 - \sigma[f]$. An assignment σ satisfies a formula f if $\sigma[f] = 1$, otherwise it falsifies f. For instance $[a \leftarrow 1, b \leftarrow 1, c \leftarrow 1]$ is an assignment over $\{a, b, c\}$ that satisfies g_2 and g_1 and that falsifies g_3.

Let f and g be two formulae. If any assignment satisfying f satisfies g as well, f *implies* g, which is denoted by $f \models g$. If both f implies g and g implies f, f and g are said equivalent, which is denoted by $f \equiv g$.

Let $f(x_1, \ldots, x_n)$ be a boolean function and let σ be any assignment that satisfies f. Then, f is *monotone* if for any variable x_i such that $\sigma[x_i] = 0$, the assignment σ', such that $\sigma'[x_i] = 1$ and $\sigma'[x_j] = \sigma[x_j]$ for $j \neq i$, satisfies f as well. If the property holds only for some variable x_i, we say that f is *monotone in* x_i.

The notion of monotone formulae is the mathematical basis of coherent fault trees. It is clear that a formula made only of variables and connectives $\wedge$, $\vee$ and k-out-of-n is monotone.

Let f be a boolean formula and π be a product. π is an *implicant* of f if $\pi \models f$, i.e. if any assignment satisfying π satisfies f.

Let f be a boolean formula and π be an implicant of f. π is *prime* if there is no implicant ρ of f strictly included in π. We denote by $PI[f]$ the set of all the prime implicants of the formula f.

For instance the formula g_1 of the Fig. 1 admits 7 implicants $\{a, b\}$, $\{a, b, c\}$, $\{a, b, \neg c\}$, $\{\neg a, b, c\}$, $\{\neg a, c\}$, $\{\neg a, \neg b, c\}$ and $\{b, c\}$ and 3 prime ones $\{a, b\}$ and $\{\neg a, c\}, \{b, c\}$.

Prime implicants of coherent fault trees are often called minimal cuts in the reliability engineering literature.

The *order* of a product (an implicant, a prime implicant) is the number of literals it contains.

As said in the introduction, there may exist an exponential number of prime

implicants with respect to the number of variables occurring in the formula under study [Qui59,CM78]. In practice, even though this worst case is seldom reached, the actual number of prime implicants is often very large (several thousands) and classical approaches [VGRH81,LGTL85] fail to handle large size non-decomposable fault trees because they cannot avoid combinatorial explosions in both time and space.

In practice, one is often only interested in positive parts of prime implicants. This is because only positive literals represent failures [4]. In order to formalize this, we introduce here the notion of p-cut.

Let f be a boolean function and let π be a product containing only positive literals.

We denote by π_f^c the product obtained by adding to π the negative literals formed over all of the variables occurring in f but not in π. In the sequel, we will omit the subscript f when the referenced formula in clear from the context. For instance, $\{a\}_{g_1}^c = \{a, \neg b, \neg c\}$, where g_1 is the formula of the Fig. 1.

Now, π is a *p-cut* of f if it fulfils the first of the two following requirements, it is *minimal* if it fulfils the second one.

(i) π^c is a implicant of f.
(ii) There is no product $\rho \subset \pi$ such that ρ^c is a implicant of f.

The formula g_1 of Fig. 1 admits two minimal p-cuts, $\{a, b\}$ and $\{c\}$. Note, that the prime implicant (and p-cut) $\{b, c\}$, which is made only of positive literals, does not lead to a minimal p-cut because the other prime implicant $\{\neg a, c\}$ has a shorter positive part.

We denote by $PC[f]$ the set of all the minimal p-cuts of the formula f.

The following property holds that is deduced directly from the definition of monotone functions.

Proposition 1 (Prime implicants of monotone functions) *Let f be monotone function and let π be a prime implicant of f. Then, π contains only positive literals.*

The following proposition comes from the previous one.

Proposition 2 (Prime implicants vs. p-cuts of monotone functions) *Let f be a monotone function. Then, $PI[f] = PC[f]$.*

[4] We would like to thank J. Gauthier, from Dassault Aviation, that showed us the interest of this notion.

3 Binary Decision Diagrams

In this section, we first introduce basic definitions and properties of Binary Decision Diagrams (for a comprehensive survey on that technique, see [BRB90] and [Bry92]). Then, we present the way the algorithm described in the previous section is implemented by means of operations on BDD's.

3.1 Shannon Decomposition

As we will see, the Binary Decision Diagram representation of boolean functions uses mainly the connective *ite* (for *I*f-*T*hen-*E*lse) defined as follows.

Definition 3 (*ite* connective) *Let f, g and h three boolean functions, then*

$$ite(f, g, h) \stackrel{\text{def}}{=} (f \wedge g) \vee (\neg f \wedge h)$$

Any binary connective can be expressed by means of an *ite* and possibly a negation. For instance, $f \vee g \equiv ite(f, 1, g)$ and $f \oplus g \equiv ite(f, \neg g, g)$. It is possible to express the negation by means of an *ite* connective ($\neg f \equiv ite(f, 0, 1)$), however, as we will see, binary decision diagrams manage the negation in a more specific way.

Moreover, given boolean function $f(x_1, \ldots, x_n)$, the following equivalence holds, so-called Shannon decomposition.

Theorem 4 (Shannon decomposition)

$$\begin{aligned} f(x_1, \ldots, x_n) &\equiv (x_1 \wedge f(1, x_2, \ldots, x_n)) \vee (\neg x_1 \wedge f(0, x_2, \ldots, x_n)) \\ &\equiv ite(x_1, f(1, x_2, \ldots, x_n), f(0, x_2, \ldots, x_n)) \end{aligned}$$

By applying this principle recursively, one can rewrite any function as an equivalent one that is built only with variables, connectives *ite* and boolean constants 0 and 1.

For instance, the formula g_1 of Fig. 1 is equivalent to $ite(a, ite(b, 1, 0), ite(c, 1, 0))$.

Another interesting property of the connective *ite* is that it is orthogonal with the usual connectives.

Definition 5 (Orthogonality) *Let Op be an n-ary operation. Let $f^1, \ldots, f^n$*

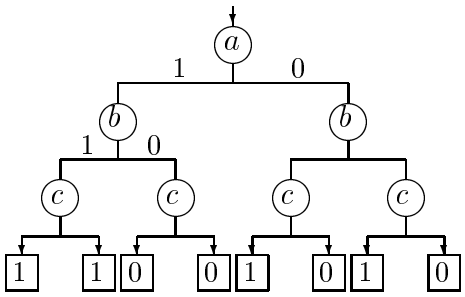

Fig. 2. The decision tree encoding g_1 for the order $a < b < c$

be n formulae and let x be a variable, Op is said to be orthogonal with ite if the following equivalence holds.

$$Op(f^1, \ldots, f^n) \equiv ite(x, Op(f^1_{x=1}, \ldots, f^n_{x=1}), Op(f^1_{x=0}, \ldots, f^n_{x=0}))$$

where $f_{x=v}$ denotes the formula f in which the constant v has been substituted for all occurrences of x.

Proposition 6 (Orthogonality) *$\neg$ (not), $\vee$ (or), $\wedge$ (and), $\Leftarrow$ (imply), $\Leftrightarrow$ (if and only if) and $\oplus$ (exclusive or) are orthogonal with ite.*

3.2 Informal Presentation of Binary Decision Diagrams

In order to provide the reader with an intuition of what binary decision diagrams are, let us consider again the function g_1 of the Fig. 1. Thank to the theorem 4, it is possible to build a decision tree encoding the truth table g_1. Given a total order over the variables of the function under study, a decision tree is a complete binary tree whose leaves are labeled with the constant 0 or 1 and whose iternal nodes are labeled with variables. The nodes at level i in the tree are labeled with the variable of rank i in the considered order. Each node has two outedges, labeled respectively with 0 and 1, that correspond to the two possible valuations of the variable it is labeled with. The tree associated with g_1 for the order $a < b < c$ is pictured Fig. 2. The value of the function for a given variable assignment is obtained by descending along the corresponding branch of the tree. Intuitively, each node of the tree encodes a formula in the form $ite(x, f_1, f_0)$, where x is the variable it is labeled with and f_1 and f_0 are the formulae encoded respectively by the trees pointed by its 1- and 0-outedges.

Now, it is clear that such an encoding for the truth table of a function is very expensive: if the function depends on n variables, the associated decision tree has $2^n - 1$ internal nodes and 2^n leaves. However, it is possible to make the representation far more compact by applying the two following operations.

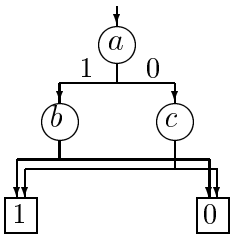

Fig. 3. The BDD encoding g_1 (for the order $a < b < c$)

(i) *Isomorphic subtrees merging.* Since two isomorphic subtrees have the same semantics, at least one is useless. This is especially the case for leaves: two leaves suffice, one labeled with 0, the other one by 1.
(ii) *Useless nodes deletion.* If a node encodes a formula in the form $ite(x, f, f)$ it is useless since $ite(x, f, f) \equiv f$. So pointers to that node can be replaced by pointers to the node encoding f.

By applying these two rules as far as possible, one gets the binary decision diagram associated with the formula. As we will see, it is unique (up to an isomorphism). The binary decision diagram associated with g_1 (for the order $a < b < c$) is pictured Fig 3. The point is that it is possible to get the BDD associated with a formula directly, i.e. without constructing the whole binary decision tree and then reducing it.

3.3 Formal Definition

Let $x_1, \ldots, x_n$ be n boolean variables. A *binary decision diagram* over x_1,...,x_n is a directed acyclic graph G verifying conditions (i) and (ii).

(i) G has two sink nodes labeled respectively with 0 and 1.
(ii) Each internal node of G is labeled with a variable x_i and has 2 outedges labeled respectively with 0 and 1.

Let $<$ be a total order over $\mathcal{X}$. A binary decision diagram G is said to be *ordered* if condition (iii) holds.

(iii) For any pair of nodes (α,β) labeled respectively with the variables x_i and x_j, if one of the two out-edges of α points to β then $x_i < x_j$.

An ordered BDD is said *reduced* if conditions (iv) and (v) hold.

(iv) It contains no internal node whose outedges are both pointing to the same sub-graph.
(v) It contains no two isomorphic sub-graphs.

$$
\begin{aligned}
Shannon[\![0]\!] &\stackrel{\text{def}}{=} 0 \\
Shannon[\![1]\!] &\stackrel{\text{def}}{=} 1 \\
Shannon[\![\Delta(x, \alpha_1, \alpha_0)]\!] &\stackrel{\text{def}}{=} ite(x_i, Shannon[\![\alpha_1]\!], Shannon[\![\alpha_0]\!])
\end{aligned}
$$

Fig. 4. The standard semantics for BDD's

In what follows, we will abbreviate reduced ordered binary decision diagram in ROBDD, or even BDD. Moreover, we will say the BDD α for the BDD whose nodes are those reachable from the node α. Finally, we will denotes by $\Delta(x, \alpha_1, \alpha_0)$ the node α labeled with the variable x and whose 1- and 0-outedges point respectively to the nodes α_1 and α_0 (since, by definition, there is no two isomorphic sub-graphs this node is unique).

3.4 Shannon Semantics for ROBDD's

BDD's, as we defined them in the previous section, are a data structure, just a data structure. As such, they have no intrinsic semantics. So, we have to define function that associates a mathematical objet, namely a boolean function, to each BDD. An usual way to describe such a function and more generally to define operations on BDD's is to write a set of recursive equations. The set of recursive equations defining the standard, or Shannon, semantics for BDD's is given Fig. 4 (there are actually other semantics for BDD's [Rau96]).

Thanks to the Shannon decomposition (theorem 4), it is clear that for any function f there exists at least one ROBDD encoding a function equivalent to f. Moreover, the following property holds.

Proposition 7 (Canonicity of ROBDD's) *Let f be a boolean formula. For given a variable order, the ROBDD associated with f is unique up to an isomorphism [Bry86].*

3.5 Management of BDD's

As a consequence of their canonicity, BDD's can be managed in such way that there is never two distinct nodes encoding the same function. The principe is as follows. BDD's are always created in a bottom-up way, i.e. that the node $\Delta(x, \alpha_1, \alpha_0)$ is always created after the creation of the nodes α_0 and α_1. Nodes are kept in a table and are created only through the function $find\_or\_add\_bdd$. This function looks up the table and creates new nodes only when necessary,

$$
\begin{aligned}
\texttt{build}(1) &\stackrel{\text{def}}{=} 1 \\
\texttt{build}(0) &\stackrel{\text{def}}{=} \bullet 1 \\
\texttt{build}(x_i) &\stackrel{\text{def}}{=} \Delta(x_i, 1, \bullet 1) \\
\texttt{build}(\neg f) &\stackrel{\text{def}}{=} \bullet\texttt{build}(f) \\
\texttt{build}(f_1 \star f_2) &\stackrel{\text{def}}{=} \texttt{apply}(\star, \texttt{build}(f_1), \texttt{build}(f_2))
\end{aligned}
$$

Fig. 5. Equations defining the `build` function.

i.e. when the table does not already contain the wanted node. Technically, *find_or_add_bdd* is implemented by means of hashing techniques [BRB90].

In order to compute the negation of a BDD one has just to exchange its 0 and 1 leaves. This comes from the fact that $\neg$ and *ite* are orthogonal (property 6). This requires *a priori* a BDD traversal, which is at least linear in the size of the considered BDD. However, a programming trick makes possible a negation in constant time and reduces memory consumption. It consists in putting a flag on each edge. This flag indicates whether the pointed BDD is to be considered positively or negatively. As a consequence, only one leaf remains necessary, say for instance the leaf 1, since $0 \equiv \neg 1$. The canonicity of the representation is maintained by storing only nodes with a positive 1-edge. The orthogonality of *ite* and $\neg$ is used to manipulate only such nodes. The following equations describe the replacement rules. Negated edges are denoted with a $\bullet$.

$$
\begin{aligned}
\Delta(x, \bullet\alpha_1, \alpha_0) &\longrightarrow \bullet\Delta(x, \alpha_1, \bullet\alpha_0) \\
\Delta(x, \bullet\alpha_1, \bullet\alpha_0) &\longrightarrow \bullet\Delta(x, \alpha_1, \alpha_0) \\
\bullet\Delta(x, \bullet\alpha_1, \alpha_0) &\longrightarrow \Delta(x, \alpha_1, \bullet\alpha_0) \\
\bullet\Delta(x, \bullet\alpha_1, \bullet\alpha_0) &\longrightarrow \Delta(x, \alpha_1, \alpha_0)
\end{aligned}
$$

3.6 Computation of the BDD Associated with a Formula

Let f be a formula over the variable $x_1, \ldots, x_n$. The computation of the BDD α encoding f (for the variable order $x_1 < \ldots < x_n$) is performed by means of the function `build` that can be described through recursive equations to be applied on f and its subformulae. These equations are given Fig. 5. Their left members are formulae — x is a variable, f, f_1 and f_2 denote any formulae, $\star$ stands for any binary connective $\vee$ (or), $\wedge$ (and), $\oplus$ (exclusive or), ...—, while right members are BDD's. When f is of the form $f_1 \star f_2$ the computation of the BDD α encoding f is performed in two steps:

$$
\begin{aligned}
\texttt{apply}(\star, \Delta(x_i, \alpha_1, \alpha_0), \Delta(x_i, \beta_1, \beta_0)) &\stackrel{\text{def}}{=} \Delta(x_i, \texttt{apply}(\star, \alpha_1, \beta_1), \texttt{apply}(\star, \alpha_0, \beta_0)) \\
\texttt{apply}(\star, \Delta(x_i, \alpha_1, \alpha_0), \Delta(x_j, \beta_1, \beta_0)) &\stackrel{\text{def}}{=} \Delta(x_i, \texttt{apply}(\star, \alpha_1, \beta), \texttt{apply}(\star, \alpha_0, \beta)) \\
\texttt{apply}(\vee, 1, \beta) &\stackrel{\text{def}}{=} 1 \\
\texttt{apply}(\vee, 0, \beta) &\stackrel{\text{def}}{=} \beta \\
\texttt{apply}(\vee, \alpha, \alpha) &\stackrel{\text{def}}{=} \alpha \\
&\vdots
\end{aligned}
$$

Fig. 6. The equations defining the `apply` function.

(i) The BDD's α_1 and α_2 encoding respectively f_1 and f_2 are computed,
(ii) α_1 and α_2 are composed by means of the `apply` function.

The recursive equations describing `apply` are given Fig. 6. There are mainly three cases:

- The main case, in which `apply` is called on two BDD's whose root nodes are labeled with the same variables.
- A degenerated case, in which the function is called on two BDD's whose root nodes are not labeled with the same variables (on Fig. 6, it is assumed that $i < j$ and $\beta = \Delta(x_j, \beta_1, \beta_0)$).
- The terminal case, that is the only one that depends on the connective $\star$. In this case, the truth table of $\star$ allows an immediate decision of the value of the function (On Fig. 6, we just give some examples for $\vee$).

Such a decomposition in three cases — the main one, a degenerated one and a terminal one — appear in most of the equations describing operations on BDD's.

A second table, so-called the *computed_table*, is used. In this table, 4-tuples $\{\star, \alpha, \beta, \gamma\}$ are kept, where α, β and γ are BDD's such that $\gamma = \alpha \star \beta$. Before any computation of $\alpha \star \beta$, one checks if the result is not already in the table. If it is, the result γ is returned, otherwise it is actually computed and then stored in the table. In order to allow a fast access to stored tuples, the *computed_table* is also managed by means of hashing techniques [BRB90].

The use of a table in which intermediate results are stored plays a central role in the efficiency of BDD's. From a theoretical point of view, it ensures bounds on complexities of operations (these bounds are given table 1). From a practical point of view, it dramatically decreases running times. As a matter of fact, none of the fault trees presented section 6 could be processed without this memorization.

In table 1, $|\alpha|$ denotes the size, i.e. the number of nodes, of the BDD α.

$find\_or\_add\_bdd$	$\mathcal{O}(1)$
$\texttt{apply}(\star, \alpha, \beta)$	$\mathcal{O}(\|\alpha\| \times \|\beta\|)$
$\texttt{pr}(\alpha)$	$\mathcal{O}(\|\alpha\|)$

Table 1
Computational costs of operations on BDD's

3.7 Variable Ordering

The size of the BDD encoding a function strongly depends on the chosen variable ordering. Finding the best variable ordering is untractable (the best known algorithm is in $\mathcal{O}(3^n)$), so heuristics are used to determine good ones. This is a complicated matter which is outside of the scope of this paper. The interested reader should see for instance [Rau96] for a discussion as well as on a quite complete bibliography on that topics.

3.8 Computation of the Probability of the Root Event

As a first illustration of the use of BDD's in the reliability analysis framework, let us show briefly how to compute the *exact* probability of the top event of a fault tree given the probabilities of its primary events. This is performed by means of a BDD traversal, the Shannon decomposition being applied on each node of the BDD.

Theorem 8 (Shannon's Decomposition) *Let $f = ite(x, f_1, f_0)$ be a formula, where x is a variable and f_1 and f_0 are formulae in which x does not occur. Then the following equality holds.*

$$p(f) = p(x).p(f_1) + (1 - p(x)).p(f_0)$$

where $p(x)$ denotes the probability that $x = 1$ and $p(f)$ the probability that $f = 1$.

It is easy to induce an effective algorithm from the above theorem (see [Rau93] for more details). Recursive equations that describe this algorithm are given Fig. 7. Thanks to the memorization of intermediate results, the complexity of the algorithm is linear in the size of the BDD, as mentioned in table 1.

For instance, the probability of the formula g_1 of the Fig. 1, is computed as follows.

$$p(g_1) = p(ite(a, ite(b, 1, 0), ite(c, 1, 0))$$

$$\begin{aligned}
\mathtt{pr}(1) &\stackrel{\text{def}}{=} 1 \\
\mathtt{pr}(\bullet\alpha) &\stackrel{\text{def}}{=} 1 - \mathtt{pr}(\alpha) \\
\mathtt{pr}(\Delta(x, \alpha_1, \alpha_0)) &\stackrel{\text{def}}{=} p(x).\mathtt{pr}(\alpha_1) + (1 - p(x)).\mathtt{pr}(\alpha_0)
\end{aligned}$$

Fig. 7. The equations defining the function computing the probability of a BDD.

$$\begin{aligned}
&= p(a).p(ite(b, 1, 0)) + (1 - p(a)).p(ite(c, 1, 0)) \\
&= p(a).p(b) + (1 - p(a)).p(c)
\end{aligned}$$

4 Exact Computations Prime Implicants and Minimal P-Cuts

4.1 Decomposition Theorems for Prime Implicants and Minimal P-Cuts

The first idea of algorithms computing prime implicants and minimal p-cuts from BDD's is as follows.

Theorem 9 (Decomposition Theorem for Prime Implicants) *Let $f(x_1, \ldots, x_n)$ be a boolean function. Then, the set of prime implicants of $f(x_1, \ldots, x_n)$ can be obtained as the union of three sets.*

$$PI[f(x_1, \ldots, x_n)] = PI_{10} \cup PI_1 \cup PI_0$$

where, PI_{10}, PI_1 and PI_0 are defined as follows.

$$\begin{aligned}
PI_{10} &\stackrel{\text{def}}{=} PI[f(1, x_2, \ldots, x_n) \wedge f(0, x_2, \ldots, x_n)] \\
PI_1 &\stackrel{\text{def}}{=} \{\{x_1\} \cup \pi; \pi \in PI[f(1, x_2, \ldots, x_n)] \setminus PI_{10}\} \\
PI_0 &\stackrel{\text{def}}{=} \{\{\neg x_1\} \cup \pi; \pi \in PI[f(0, x_2, \ldots, x_n)] \setminus PI_{10}\}
\end{aligned}$$

where $\setminus$ stands for the set difference.

A formal proof of the theorem 9 is given in appendix. Intuitively, it is justified as follows. A prime implicant π of $f(x_1, \ldots, x_n)$ may contain either x_1 or $\neg x_1$ or none of these two literals. In this latter case, π must still be a prime implicant of f whatever constant is substituted for x_1. Thus, π is a prime implicant of $f(x_1, \ldots, x_n)$ that does not contain x_1 nor $\neg x_1$ if and only if it is a prime implicant of $\forall x_1 f(x_1, \ldots, x_n) = f(1, x_2, \ldots, x_n) \wedge f(0, x_2, \ldots, x_n)$. Now, a product $\{x_1\} \cup \pi$ is a prime implicant of $f(x_1, \ldots, x_n)$ if it is a prime implicant of $f(1, x_2, \ldots, x_n)$ and π is not already a prime implicant of $f(x_1, \ldots, x_n)$, i.e. if π does not belong to $PI[f(1, x_2, \ldots, x_n) \wedge f(0, x_2, \ldots, x_n)]$.

The decomposition gives an inductive principle to compute prime implicants. Note that this principle is different from the Quine's consensus method [Qui52].

In our example $g_1(a,b,c) = (a \wedge b) \vee (\neg a \wedge c) = ite(a, ite(b,1,0), ite(c,1,0))$,

$$g_1(1,b,c) = ite(b,1,0) = b,$$
$$g_1(0,b,c) = ite(c,1,0) = c,$$
$$g_1(1,b,c) \wedge g_1(0,b,c) = ite(b, ite(c,1,0), 0) = b \wedge c$$

It is clear that $PI[b] = \{\{b\}\}$, $PI[c] = \{\{c\}\}$ and $PI[b \wedge c] = \{\{b,c\}\}$.

Thus, $PI[ite(a, ite(b,1,0), ite(c,1,0))] = PI_{10} \cup PI_1 \cup PI_0$, where,

- PI_{10} is $PI[b \wedge c] = \{\{b,c\}\}$.
- PI_1 is the set of products $\{a\} \cup \pi$ where π is in $\{\{b\}\}$ and not in $\{\{b,c\}\}$. Thus, $PI_1 = \{\{a,b\}\}$.
- PI_0 is the set of products $\{\neg a\} \cup \pi$ where π is in $\{\{c\}\}$ and not in $\{\{b,c\}\}$. Thus, $PI_0 = \{\{\neg a, c\}\}$.

Finally, we get $PI[g_1] = \{\{a,b\}, \{\neg a, c\}, \{b,c\}\}$.

Similarly, the following theorem holds that gives an inductive principle to compute minimal p-cuts.

Theorem 10 (Decomposition Theorem for Minimal P-Cuts) *Let $f(x_1, \ldots, x_n)$ be a boolean function. Then, the set of minimal p-cuts of $f(x_1, \ldots, x_n)$ can be obtained as the union of two sets.*

$$PC[f(x_1, \ldots, x_n)] = PC_0 \cup PC_1$$

where, PC_1 and PC_0 are defined as follows.

$$PC_0 \stackrel{\text{def}}{=} PC[f(0, x_2, \ldots, x_n)]$$
$$PC_1 \stackrel{\text{def}}{=} \{\{x_1\} \cup \pi; \pi \in PC[f(1, x_2, \ldots, x_n) \vee f(0, x_2, \ldots, x_n)] \setminus PC_0\}$$

A formal proof of the theorem 10 is given in appendix.

In our example $g_1(a,b,c) = (a \wedge b) \vee (\neg a \wedge c) = ite(a, ite(b,1,0), ite(c,1,0))$, and thus $g_1(1,b,c) \vee g_1(0,b,c) = b \vee c$. Clearly, we have $PC[b \vee c] = \{\{b\}, \{c\}\}$.

Thus, $PC[ite(a, ite(b, 1, 0), ite(c, 1, 0))] = PC_1 \cup PC_0$, where,

- PC_0 is $PC[c] = \{\{c\}\}$.
- PC_1 is the set of products $\{a\} \cup \pi$ where π is in $PC[b \vee c] = \{\{b\}, \{c\}\}$ and not in $PC_0 = \{\{c\}\}$. Thus, $PC_1 = \{\{b\}\}$.

Finally, we get $PC[g_1] = \{\{a, b\}, \{c\}\}$.

Notice that for the special case of monotone functions (coherent fault trees), the two decomposition theorems can be simplified and, once simplified, become actually identical. Let $f(x_1, \ldots, x_n)$ be a monotone function. Then, it is easy to verify that the two following equalities hold.

$$f(1, x_2, \ldots, x_n) \wedge f(0, x_2, \ldots, x_n) = f(0, x_2, \ldots, x_n) \quad (1)$$
$$f(1, x_2, \ldots, x_n) \vee f(0, x_2, \ldots, x_n) = f(1, x_2, \ldots, x_n) \quad (2)$$

Now by simplifying decomposition theorems according to equations 1 and 2, we get exactly the same decomposition principle (excepted that the identifier PI is used in the first case and the identifier PC is used in the second one).

4.2 Meta-Products

A second key idea is to represent sets of products by means of boolean formulae. More precisely, a function $MP(\Sigma)$ is associated with each set of products Σ in such a way that there is a one-to-one correspondence between products of Σ and variable assignments that satisfy $MP(\Sigma)$. Functions $MP(\Sigma)$ are called meta-products by Madre and Coudert in [CM92a]. Note that meta-products are not the only way to encode set of products by means of boolean functions. Minato in [Min93] and Rauzy in [Rau93] proposed an alternative way to do so. For the sake of conciseness, we only consider here Madre and Coudert's representation.

4.2.1 Definition

Let x be a variable and π be a product. Then three cases are possible: either $x \in \pi$ or $\neg x \in \pi$ or neither x nor $\neg x$ belong to π.

The idea is to associate two variables p_x and s_x with each variable x of the original formula. p_x is used to encode the presence of the variable x in the product. s_x is used to encode the sign of x in the product, if it is actually present.

The *meta-product* encoding π, denoted by $MP(\pi)$, is the conjunction, over the variables x occurring in the formula, of $mp(\pi, x)$, where $mp(\pi, x)$ is defined as follows:

$$mp(\pi, x) \stackrel{\text{def}}{=} \begin{cases} (p_x \wedge s_x) \text{ if } x \in \pi, \\ (p_x \wedge \neg s_x) \text{ if } \neg x \in \pi, \\ \neg p_x \text{ if neither } x \text{ nor } \neg x \text{ belongs to } \pi. \end{cases}$$

For instance, the meta-product associated with the prime implicant $\pi = \{\neg a, c\}$ of g_1 is $MP(\pi) = (p_a \wedge \neg s_a \wedge \neg p_b \wedge p_c \wedge \neg s_c)$. a occurs negatively in π, thus the variable p_a encoding the presence of a must be true and s_a the variable encoding the sign of a must be false. b does not occur in π thus p_b the variable encoding the presence of b must be false and s_b may take any value (and thus is not present at all in the meta-product). Finally, c occurs positively in π, thus the variable p_c encoding the presence of c must be true and s_c the variable encoding the sign of c must be true.

4.2.2 Operations on Sets of Products

A set of products is encoded by the disjunction of the meta-products encoding its elements. The formula encoding a set of products Σ is also denoted by $MP(\Sigma)$.

For instance $PI[g_1]$, of prime implicants of the function g_1 of Fig. 1 is encoded by means of the following function.

$$\begin{aligned} MP(PI[g_1]) = MP(\{\{a,b\},\{\neg a,c\},\{b,c\}\}) & \\ (p_a \wedge s_a \wedge p_b \wedge s_b \wedge \neg p_c) & \quad /* = MP(\{a,b\}) * / \\ = \vee\ (p_a \wedge \neg s_a \wedge \neg p_b \wedge p_c \wedge s_c) & \quad /* = MP(\{\neg a,c\}) * / \\ \vee\ (\neg p_a \wedge p_b \wedge s_b \wedge p_c \wedge s_c) & \quad /* = MP(\{b,c\}) * / \end{aligned}$$

More generally, operations on sets of products can be performed through logical operations on the corresponding meta-products.

Proposition 11 (Operations on meta-products) *Let Σ_1 and Σ_2 be two sets of products built over the variables $x_1, \ldots, x_n$. Let $MP(\Sigma_1)$ and $MP(\Sigma_2)$ be the two meta-products built over the variables $p_{x_1}, s_{x_1}, \ldots, p_{x_n}, s_{x_n}$, encoding*

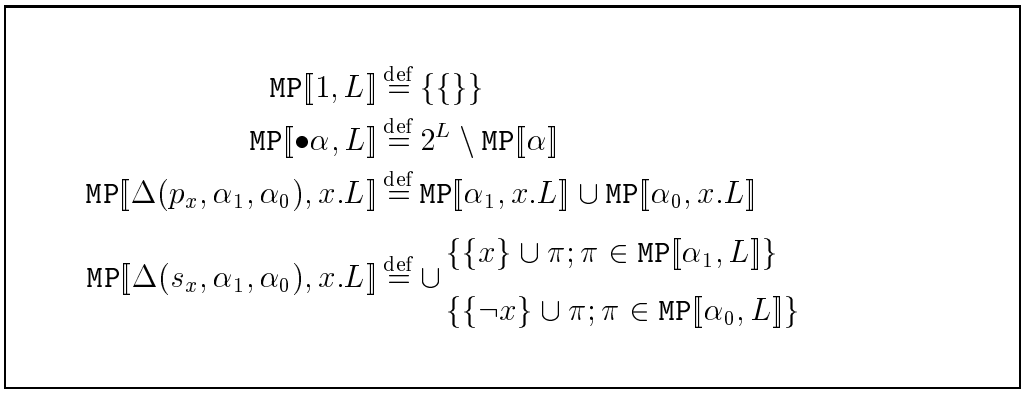

Fig. 8. The equations defining the semantics of BDD's encoding metaproducts.

respectively Σ_1 *and* Σ_2.

- *The empty set of products is encoded by* 0 *(false).*
- *The set of all of the possible products is encoded by the* 1 *(true).*
- *The empty product is encoded by the function* $(\neg p_{x_1} \wedge \ldots \wedge \neg p_{x_n})$.
- $\Sigma_1 \cup \Sigma_2$ *is encoded by the function* $MP(\Sigma_1) \vee MP(\Sigma_2)$.
- $\Sigma_1 \cap \Sigma_2$ *is encoded by the function* $MP(\Sigma_1) \wedge MP(\Sigma_2)$.
- *The complement* $\overline{\Sigma_1}$ *to* Σ_1 *in the set of all of the possible products is encoded by the function* $\neg MP(\Sigma_1)$.

These properties are demonstrated very easily by considering the one-to-one correspondence between satisfying variable assignments and encoded products [CM92a].

Note that the set of products that belong to Σ_1 but not to Σ_2, i.e. the set $\Sigma_1 \cap \overline{\Sigma_2}$, is encoded by the function $MP(\Sigma_1) \wedge \neg MP(\Sigma_2)$.

4.3 Computation of the BDD Encoding Prime Implicants

The third idea is to use BDD's to encode meta-products. The recursive equations defining the semantics of such a BDD are given Fig. 8. In these equations, the ordered list of the variables the products are built over is given as an argument. While not strictly necessary, this simplifies the definition for negated BDD's. Moreover, it will be usefull to set equations defining the algorithm itself. 2^L denotes the set of all of the products that can be built over the variables of the list L. This implementation assumes, as suggested in [CM92a], that the variables p_x and s_x are consecutive in the BDD order.

The recursive equations defining the algorithm that computes the BDD encoding the prime implicants of a function from the BDD encoding this function is sketched Fig. 9. This is basically an implementation of the recursive principle described section 4.1. In these equations, [] denotes the empty list (of vari-

$$
\begin{aligned}
\texttt{MPPI}(\bullet 1, L) &\stackrel{\text{def}}{=} \bullet 1 \\
\texttt{MPPI}(1, []) &\stackrel{\text{def}}{=} 1 \\
\texttt{MPPI}(1, x.L) &\stackrel{\text{def}}{=} \Delta(p_x, \bullet 1, \texttt{MPPI}(1, L)) \\
\texttt{MPPI}(\bullet\Delta(x, \alpha_1, \alpha_0), L) &\stackrel{\text{def}}{=} \texttt{MPPI}(\Delta(x, \bullet\alpha_1, \bullet\alpha_0), L) \\
\texttt{MPPI}(\Delta(y, \alpha_1, \alpha_0), x.L) &\stackrel{\text{def}}{=} \Delta(p_x, \bullet 1, \texttt{MPPI}(\Delta(y, \alpha_1, \alpha_0), L)) \\
\texttt{MPPI}(\Delta(x, \alpha_1, \alpha_0), x.L) &\stackrel{\text{def}}{=} \Delta(p_x, \Delta(s_x, \beta_1, \beta_0), \beta_{10})
\end{aligned}
$$

$$
\text{where,} \quad \begin{cases}
\beta_{10} &= \texttt{MPPI}(\texttt{apply}(\wedge, \alpha_1, \alpha_0), L) \\
\beta_1 &= \texttt{apply}(\wedge, \texttt{MPPI}(\alpha_1, L), \bullet\beta_{10}) \\
\beta_0 &= \texttt{apply}(\wedge, \texttt{MPPI}(\alpha_0, L), \bullet\beta_{10})
\end{cases}
$$

Fig. 9. The equations defining the algorithm MPPI.

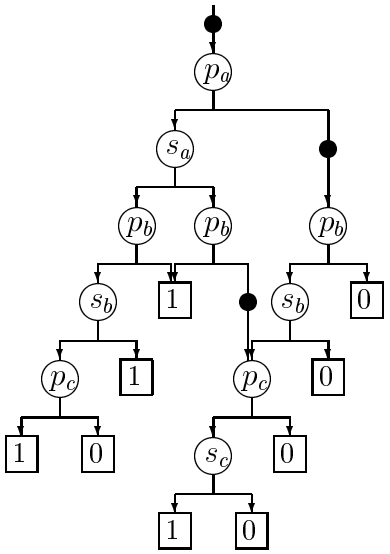

Fig. 10. The BDD encoding $MP(PI[g_1])$ (negated edges are flagged with a black dot). For the sake of clarity, the leaf is duplicated and negated pointers to the leaf are replaced by pointers to a leaf 0.

ables) and the variable x is assumed to be before the variable y in the chosen order.

Note that intermediate results are memorized in this procedure as well. Despite of this memorization, the worst case computational cost of MPPI is exponential with respect to the number of variables.

The BDD encoding $MP(PI[g_1])$ is pictured Fig. 10.

$$
\begin{aligned}
\texttt{MPPC}(\bullet 1, L) &\stackrel{\text{def}}{=} \bullet 1 \\
\texttt{MPPC}(1, []) &\stackrel{\text{def}}{=} 1 \\
\texttt{MPPC}(1, x.L) &\stackrel{\text{def}}{=} \Delta(p_x, \bullet 1, \texttt{MPPC}(1, L)) \\
\texttt{MPPC}(\bullet\Delta(x, \alpha_1, \alpha_0), L) &\stackrel{\text{def}}{=} \texttt{MPPC}(\Delta(x, \bullet\alpha_1, \bullet\alpha_0), L) \\
\texttt{MPPC}(\Delta(y, \alpha_1, \alpha_0), x.L) &\stackrel{\text{def}}{=} \Delta(p_x, \bullet 1, \texttt{MPPC}(\Delta(y, \alpha_1, \alpha_0), L)) \\
\texttt{MPPC}(\Delta(x, \alpha_1, \alpha_0), x.L) &\stackrel{\text{def}}{=} \Delta(p_x, \beta_1, \beta_0)
\end{aligned}
$$

$$
\text{where, } \begin{cases} \beta_0 = \texttt{MPPC}(\alpha_0, L) \\ \beta_1 = \texttt{apply}(\wedge, \texttt{MPPC}(\texttt{apply}(\vee, \alpha_1, \alpha_0), L), \bullet\beta_0) \end{cases}
$$

Fig. 11. The equations defining the algorithm `MPPC`.

4.4 Computation of the BDD Encoding Minimal P-Cuts

In a similar way, the recursive equations that defined the algorithm `MPPC` that computes the BDD encoding the minimal p-cuts of a function from the BDD encoding this function are given Fig.11. They encode the decomposition principle given by the theorem 10. There is however one difference: since variable occur always positively in p-cuts, the variables s_x's are useless. They can be safely removed, i.e. all of the properties of proposition 11 still hold in this simplified representation.

4.5 Another Way to Compute the BDD Encoding Minimal P-Cuts

There is a subtle difference between `MPPI` and `MPPC`. In `MPPI`, the computation of $\alpha_1 \wedge \alpha_0$ is mandatory. In `MPPC`, the computation of $\alpha_1 \vee \alpha_0$ is performed only to remove from $PC[\alpha_1]$ the products that are included in products of $PC[\alpha_0]$. The point is that this operation can be performed directly on meta-product BDD's (we call it `without` in the sequel), which is in some cases more efficient, especially when the number of minimal p-cuts is small. This leads to a second algorithm to compute minimal p-cuts, so called `MPQC`, which is almost the same than `MPPC` excepted that β_1 is computed as follows :

$$\beta_1 = \texttt{without}(\texttt{MPQC}(\alpha_1), \beta_0)$$

Equations defining `without` are given on Fig.12 (in these equations, it is assumed that $x < y$). This algorithm is very similar to the one given by Rauzy in [Rau93].

$$
\begin{aligned}
\texttt{without}(\bullet 1, \beta) &\stackrel{\text{def}}{=} \bullet 1 \\
\texttt{without}(\alpha, 1) &\stackrel{\text{def}}{=} \bullet 1 \\
\texttt{without}(\alpha, \alpha) &\stackrel{\text{def}}{=} \bullet 1 \\
\texttt{without}(\alpha, \bullet 1) &\stackrel{\text{def}}{=} \alpha \\
\texttt{without}(1, \Delta(x, \beta_1, \beta_0)) &\stackrel{\text{def}}{=} \Delta(x, \bullet 1, \texttt{without}(1, \beta_0)) \\
\texttt{without}(\bullet\Delta(x, \alpha_1, \alpha_0), \beta) &\stackrel{\text{def}}{=} \texttt{without}(\Delta(x, \bullet\alpha_1, \bullet\alpha_0), \beta) \\
\texttt{without}(\alpha, \bullet\Delta(x, \beta_1, \beta_0)) &\stackrel{\text{def}}{=} \texttt{without}(\alpha, \Delta(x, \bullet\beta_1, \bullet\beta_0)) \\
\texttt{without}(\Delta(y, \alpha_1, \alpha_0), \Delta(x, \beta_1, \beta_0)) &\stackrel{\text{def}}{=} \Delta(x, \bullet 1, \texttt{without}(\alpha, \beta_0)) \\
\texttt{without}(\Delta(x, \alpha_1, \alpha_0), \Delta(y, \beta_1, \beta_0)) &\stackrel{\text{def}}{=} \Delta(x, \texttt{without}(\alpha_1, \beta), \texttt{without}(\alpha_0, \beta)) \\
\texttt{without}(\Delta(x, \alpha_1, \alpha_0), \Delta(x, \beta_1, \beta_0)) &\stackrel{\text{def}}{=} \Delta(x, \gamma_1, \gamma_0)
\end{aligned}
$$

$$
\text{where, } \begin{cases} \gamma_0 = \texttt{without}(\alpha_0, \gamma_0) \\ \gamma_1 = \texttt{without}(\alpha_1, \texttt{apply}(\vee, \beta_1, \beta_0)) \end{cases}
$$

Fig. 12. The equations defining the algorithm `without`.

Indeed, the following property holds for any BDD α.

$$\texttt{MPPC}(\alpha) = \texttt{MPQC}(\alpha)$$

5 Truncated Computations of Prime Implicants and Minimal P-Cuts

5.1 Algorithms

As mentioned at the section 2.2, a function may have an exponential number of prime implicants with respect to its number of variables. As an illustration, consider the function $\#(n, n, [x_1, \ldots, x_{2n}])$ that is true if and only if exactly n of the x_i's are true. In practice, large real-life fault trees we dealt with have actually many prime implicants (say several millions). Up to some preprocessing, we succedded almost always in computing the BDDs encoding these fault trees within reasonable running times and amounts of memory. There are some case however where we did not succedded in computing the BDDs encoding their prime implicants (or minimal p-cuts). In order to tackle this difficulty, we designed variations on `MPPI`, `MPPC` and `MPQC` that compute only prime implicants whose order is less than a given constant or whose proba-

bility is greater than a given threshold. From a practical point of view, these implicants are interesting because they concentrate in general the most probable failures. Good assessments on reliability (and other measures of the risk) can be obtained by considering them only.

From a technical point of view, variations we designed are based on the observation that, at any step of the algorithms, we know a lower bound on the order and a upper bound on the probability of prime implicants (or minimal p-cuts) to be built. Namely, assume that at the given step the formula under study is $f = ite(x, f_1, f_0)$ and that we know that prime implicants to be built are at least of order k and at most of probability p, then according to the decomposition theorem (theorem 9):

- The prime implicants of f containing x are at least of order $k+1$ and at most of probability $p.p(x)$.
- The prime implicants of f containing $\neg x$ are at least of order $k+1$ and at most of probability $p.(1-p(x))$.
- The prime implicants of f containing neither x or $\neg x$ are at least of order k and at most of probability p.

Similarly, according to theorem 10:

- The minimal p-cuts of f containing x are at least of order $k+1$ and at most of probability $p.p(x)$.
- The minimal p-cuts of f that do not contain x are at least of order k and at most of probability p.

The idea is thus simply to stop the computation when the current order k or the current probability p reach the predefined bounds. Let us call $\texttt{TrMPPI}(f, k, p)$, $\texttt{TrMPPC}(f, k, p)$ and $\texttt{TrMPQC}(f, k, p)$ (the prefix Tr stands for truncated) the algorithms computing respectively the prime implicants and the minimal p-cuts of f whose order is less than k and whose probability is greater than p. Equations defining TrMPPI are given Fig. 13. k_{max} and p_{min} denote respectively the predefined upper bound on the order and lower bound on the probability of the considered products. TrMPPI should be called with initial values of k and p set respectively to 0 and 1.

Equations defining TrMPPC and TrMPQC can be stated in a very similar way, as shown Fig. 14 for TrMPQC.

The following properties hold.

Proposition 12 (Completeness of TrMPPI) *TrMPPI produces all of the prime implicants of the function under study whose order is less than the given lower bound and whose probability is greater than the given upper bound.*

$$
\begin{aligned}
\texttt{TrMPPI}(\alpha, L, k, p) &\stackrel{\text{def}}{=} \bullet 1 \text{ if } k > k_{max} \text{ or } p < p_{min} \\
\texttt{TrMPPI}(\bullet 1, L, k, p) &\stackrel{\text{def}}{=} \bullet 1 \\
\texttt{TrMPPI}(1, [], k, p) &\stackrel{\text{def}}{=} 1 \\
\texttt{TrMPPI}(1, x.L, k, p) &\stackrel{\text{def}}{=} \Delta(p_x, \bullet 1, \texttt{TrMPPI}(1, L)) \\
\texttt{TrMPPI}(\bullet\Delta(x, \alpha_1, \alpha_0), L, k, p) &\stackrel{\text{def}}{=} \texttt{TrMPPI}(\Delta(x, \bullet\alpha_1, \bullet\alpha_0), L, k, p) \\
\texttt{TrMPPI}(\Delta(y, \alpha_1, \alpha_0), x.L, k, p) &\stackrel{\text{def}}{=} \Delta(p_x, \bullet 1, \texttt{TrMPPI}(\Delta(y, \alpha_1, \alpha_0), L, k, p)) \\
\texttt{TrMPPI}(\Delta(x, \alpha_1, \alpha_0), x.L, k, p) &\stackrel{\text{def}}{=} \Delta(p_x, \Delta(s_x, \beta_1, \beta_0), \beta_{10}) \\
\text{where, } &\left\{ \begin{array}{l}
\beta_{10} = \texttt{TrMPPI}(\texttt{apply}(\wedge, \alpha_1, \alpha_0), L, k, p) \\
\beta_1 \;\; = \texttt{apply}(\wedge, \texttt{TrMPPI}(\alpha_1, L, k+1, p.p(x)), \bullet\beta_{10}) \\
\beta_0 \;\; = \texttt{apply}(\wedge, \texttt{TrMPPI}(\alpha_0, L, k+1, p.(1-p(x))), \bullet\beta_{10})
\end{array} \right.
\end{aligned}
$$

Fig. 13. The equations defining the algorithm `TrMPPI`.

$$
\begin{aligned}
\texttt{TrMPQC}(\alpha, L, k, p) &\stackrel{\text{def}}{=} \bullet 1 \text{ if } k > k_{max} \text{ or } p < p_{min} \\
\texttt{TrMPQC}(\bullet 1, L, k, p) &\stackrel{\text{def}}{=} \bullet 1 \\
\texttt{TrMPQC}(1, [], k, p) &\stackrel{\text{def}}{=} 1 \\
\texttt{TrMPQC}(1, x.L, k, p) &\stackrel{\text{def}}{=} \Delta(p_x, \bullet 1, \texttt{TrMPQC}(1, L, k, p)) \\
\texttt{TrMPQC}(\bullet\Delta(x, \alpha_1, \alpha_0), L, k, p) &\stackrel{\text{def}}{=} \texttt{TrMPQC}(\Delta(x, \bullet\alpha_1, \bullet\alpha_0), L, k, p) \\
\texttt{TrMPQC}(\Delta(y, \alpha_1, \alpha_0), x.L, k, p) &\stackrel{\text{def}}{=} \Delta(p_x, \bullet 1, \texttt{TrMPQC}(\Delta(y, \alpha_1, \alpha_0), L, k, p)) \\
\texttt{TrMPQC}(\Delta(x, \alpha_1, \alpha_0), x.L, k, p) &\stackrel{\text{def}}{=} \Delta(p_x, \beta_1, \beta_0) \\
\text{where, } &\left\{ \begin{array}{l}
\beta_0 = \texttt{TrMPQC}(\alpha_0, L, k, p) \\
\beta_1 = \texttt{without}(\ \texttt{TrMPQC}(\alpha_1, L, k+1, p.p(x)),\ \beta_0\ )
\end{array} \right.
\end{aligned}
$$

Fig. 14. The equations defining the algorithm `TrMPQC`.

Proposition 13 (Completeness of `TrMPPC` and `TrMPQC`) *`TrMPPC` and `TrMPQC` produce all of the minimal p-cuts of the function under study whose order is less than the given lower bound and whose probability is greater than the given upper bound (and indeed they give the same result).*

The proof of both properties is done by induction of the structure of the formula encoded by the BDD associated with the function under study.

It is also easy to establish a maximum bound on the number of calls to the

algorithms.

Proposition 14 (Complexity of `TrMPPI`) *The number of recursive calls to the algorithm is in $\mathcal{O}(2^k n^k)$ (when k is small with respect to n).*

Proof by structural induction as well.

Proposition 15 (Complexity of `TrMPPC` and `TrMPQC`) *The number of recursive calls to the algorithm is in $\mathcal{O}(n^k)$ (when k is small with respect to n).*

Proof by structural induction as well.

This means that running times spent in the search for prime implicants can be arbitrarily bound by fixing k accordingly.

5.2 Computation of Probability of Truncated Prime Implicants and Minimal P-Cuts

As shown in section 3.8, it is easy to compute the probability of the root event of a fault tree f once the BDD α encoding f has been computed. On the other hand, the computation of truncated prime implicants and minimal p-cuts raises a question: is the approximation we get by considering only truncated prime implicants or minimal p-cuts good enough ? To answer this question, it would be ideal to design an algorithm that computes the probability that at least one of these prime implicants is true from the BDD β encoding them. Unfortunately, it is not possible to compute this probability directly. The solution consists in computing a BDD γ that encodes a function whose prime implicants are exactly those encoded by β. Once γ has been computed, the desired probability is easily obtained (see section 3.8).

To come back to a standard BDD from a BDD encoding a meta-product, we will use an algorithm that is almost directly induced by the recursive equations given Fig. 8 that define the semantics of meta-product BDD's. The equations defining this algorithm are given Fig. 15.

If the considered BDD encodes a simplified meta-product for minimal p-cuts (as discussed section 4.4), the third rule should be applied on every internal node).

It is easy to verify from equations of Fig.15, that the BDD `ReverseMP`(β) encodes the disjunction of the products encoded by β. Formally, the following

$$
\begin{aligned}
\texttt{ReverseMP}(1) &\stackrel{\text{def}}{=} 1 \\
\texttt{ReverseMP}(\bullet 1) &\stackrel{\text{def}}{=} \bullet 1 \\
\texttt{ReverseMP}(\bullet\Delta(x,\alpha_1,\alpha_0)) &\stackrel{\text{def}}{=} \texttt{ReverseMP}(\Delta(x,\bullet\alpha_1,\bullet\alpha_0)) \\
\texttt{ReverseMP}(\Delta(p_x,\alpha_1,\alpha_0)) &\stackrel{\text{def}}{=} \texttt{apply}(\vee,\texttt{ReverseMP}(\alpha_1),\texttt{ReverseMP}(\alpha_0)) \\
\texttt{ReverseMP}(\Delta(s_x,\alpha_1,\alpha_0)) &\stackrel{\text{def}}{=} \Delta(x,\texttt{ReverseMP}(\alpha_1),\texttt{ReverseMP}(\alpha_0))
\end{aligned}
$$

Fig. 15. The equations defining ReverseMP.

property holds.

Proposition 16 (Semantics of ReverseMP) *Let β be a meta-product BDD and let $\alpha = \mathit{ReverseMP}(\beta)$, then the following equality holds.*

$$Shannon[\![\alpha]\!] \equiv \bigvee_{\pi \in MP(\beta)} \bigwedge_{l \in \pi} l$$

The following property asserts the soundness of this backward computation.

Proposition 17 (Soundness of ReverseMP) *Let α be a BDD, k be an order and p a probability, then the following equalities hold.*

$$\mathit{ReverseMP}(\mathit{MPPI}(\alpha)) = \alpha \tag{3}$$
$$\mathit{MPPI}(\mathit{ReverseMP}(\mathit{TrMPPI}(\alpha,k,p))) = \mathit{TrMPPI}(\alpha,k,p) \tag{4}$$
$$\mathit{TrMPPI}(\mathit{ReverseMP}(\mathit{TrMPPI}(\alpha,k,p)),k,p) = \mathit{TrMPPI}(\alpha,k,p) \tag{5}$$
$$\mathit{MPPC}(\mathit{ReverseMP}(\mathit{TrMPPC}(\alpha,k,p))) = \mathit{TrMPPC}(\alpha,k,p) \tag{6}$$
$$\mathit{TrMPPC}(\mathit{ReverseMP}(\mathit{TrMPPC}(\alpha,k,p)),k,p) = \mathit{TrMPPC}(\alpha,k,p) \tag{7}$$
$$\mathit{MPQC}(\mathit{ReverseMP}(\mathit{TrMPQC}(\alpha,k,p))) = \mathit{TrMPQC}(\alpha,k,p) \tag{8}$$
$$\mathit{TrMPQC}(\mathit{ReverseMP}(\mathit{TrMPQC}(\alpha,k,p)),k,p) = \mathit{TrMPQC}(\alpha,k,p) \tag{9}$$

Equality 3 means that by computing prime implicants of a BDD α and then applying ReverseMP on the obtained BDD, we get α again. Equalities 4-5 extend this result to computations with truncation. Indeed, such a nice property does not hold for minimal p-cuts, since the function encoded by $\texttt{ReverseMP}(\texttt{MPPC}(\alpha))$ is always monotone, even if the one encoded by α is not, for p-cuts contain only positive literals. However, equalities 6-9 indicate that the combination π of one of the algorithms computing minimal p-cuts with ReverseMP works as a projection, in the mathematical meaning of this term, from the space of Boolean functions to the space of monotone Boolean functions whose prime implicants obey constraints on their order and probability. Namely, for any function encoded by a BDD α, we have $\pi(\pi(\alpha)) = \pi(\alpha)$.

Moreover, it is easy to verify the following property.

Proposition 18 (Complexity of `ReverseMP`) *Let β be a BDD, then the number of recursive calls to the algorithm when calling `ReverseMP` on β is in $\mathcal{O}(|\beta|)$, where $|\beta|$ denotes the number of nodes of the BDD $|\beta|$.*

6 Experimental Results and Conclusion

In order to test the algorithms presented in this paper, we considered a number of real-life fault trees that were provided by our industrial partners. We report here results obtained of two of them coming from "Électricité de France" and "Dassault Aviation". The tests reported in the tables below were performed on a SUN workstation ultra spark 1 with a 512 megabytes of memory. It is clear that such a computer is not on every desk ! On the other hand, since most of the research centers are now equiped with such machines, it makes sense to study what can be done at the limits of the current technology.

6.1 A Coherent Fault Tree from Électricité de France

The first tree we consider is a coherent one from Électricité de France. It is referenced under the name `edfpa01` in our benchmark and it is a good representative for a very large class of fault trees. It has 337 gates ($\wedge$ and $\vee$ gates), 306 primary events and only 31 small modules. The computation of the BDD encoding it takes 13.1 seconds on average. This BDD is made of 17836 nodes. The computation of the exact probability of its top event from this BDD takes 0.55 seconds. The following table reports the probability of the top event for different values of the probabilities of the primary events (for this test, we assume that all primary events have the same probability).

	Probabilities				
Primary events	0.5	0.1	0.01	0.001	0.0001
Top event	1	1	0.536048	0.0386668	0.0033621

Note that the two first probabilities of the top event are 1, which simply means that they are too close to 1 to be distinguished from this value with the double precision floating point numbers of our computer.

`edfpa01` admits $7,520,142$ prime implicants (or minimal cuts or minimal p-cuts since these notions are equivalent for monotone functions). The table 2

				Contributions				
Orders	$\#PI$	|ReverseMP|	Times	0.5	0.1	0.01	0.001	0.0001
1	33	34	0.02	1	0.969097	0.526574	0.83993	0.979962
2	6801	1474	0.02	1	0.999984	0.949656	0.996757	0.999958
3	148780	19532	0.12	1	1	0.998515	0.999987	1
4	647556	68890	0.39	1	1	0.999991	1	1
5	507572	185849	1.01	1	1	1	1	1
6	577192	475242	2.54	1	1	1	1	1
7	1575616	583036	3.12	1	1	1	1	1
8	1745584	564808	3.05	1	1	1	1	1
9	1323120	310649	1.68	1	1	1	1	1
10	812520	152699	0.83	1	1	1	1	1
11	166968	106832	0.58	1	1	1	1	1
12	8400	100921	0.55	1	1	1	1	1

Table 2
Numbers of Prime Implicants and their contributions to the probability of the top event of `edfpa01`.

gives, for each order k:

- The numbers of prime implicants of order k.
- The size of the BDD encoding the function whose prime implicants are those of `edfpa01` of order less than k.
- The running times to get the probability of this function from the previous BDD (these times are independant of the probability of primary events).
- The contribution of this function to the probability of `edfpa01` for the different values of the probabilities of the primary events mentioned above (this contribution is defined as the probability of the function divided by the probability of `edfpa01`).

A number of remarks can be made about this table.

Remark 19 *The number of prime implicants is very large and it is doubtful that classical methods based on inclusion/exclusion techniques could proceed this tree.*

Remark 20 *Most of the probability of the function* `edfpa01` *is concentrated into prime implicants of low orders, which is in general the case for the coherent fault trees of our benchmark.*

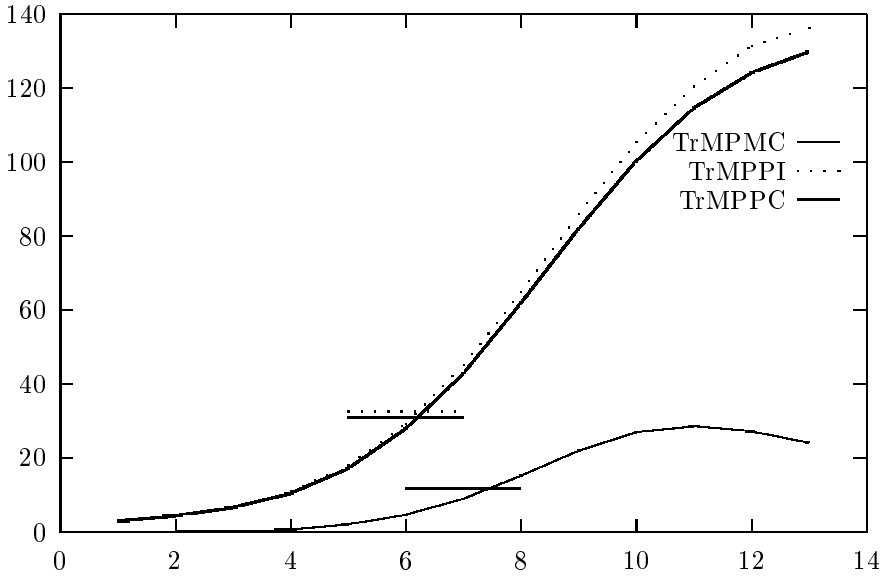

Fig. 16. Running times in seconds (y-axis) of `TrMPMC`, `TrMPPI` and `TrMPPC` on `edfpa01` for different truncation orders (x-axis).

Remark 21 *Even for large BDD's, the computation of the probability of the function encoded by the BDD is achieved very quickly.*

In order to compute prime implicants of `edfpa01`, we tested four algorithms: `MPMC`, the version with truncation of the algorithm one of us proposed in [Rau93], and the versions with truncation of the three algorithms `MPPI`, `MPPC` and `MPQC` presented in this paper. The running times of these algorithms are compared on Fig. 16 and Fig. 17. Horizontal lines give running times for algorithms without truncation.

A number of remarks can be made about these two figures.

Remark 22 *Running times for `MPPI` and `MPPC` (and the versions with truncation of these two algorithms) are almost the same. `MPMC` is faster, while `MPQC` is far slower. It is hard to find convincing arguments to explain that. Clearly, `MPMC` takes advantage it knows that the function is monotone. On the other side, performances of `MPQC` are bad as soon as the number of implicants is large.*

Remark 23 *For all of these four algorithms, there is an order k beyond which it is more costly to compute prime implicants whose order is less than k than to compute all prime implicants. The same remark holds also for the number of nodes of the obtained BDD's and the total number of nodes required to computed them. This can be explained in two ways: first, the memorization of intermediate results is less efficient when a truncation is performed (since the truncation order must be taken into account), and second, there are probably more regularities that can be captured by BDD's when no truncation is per-*

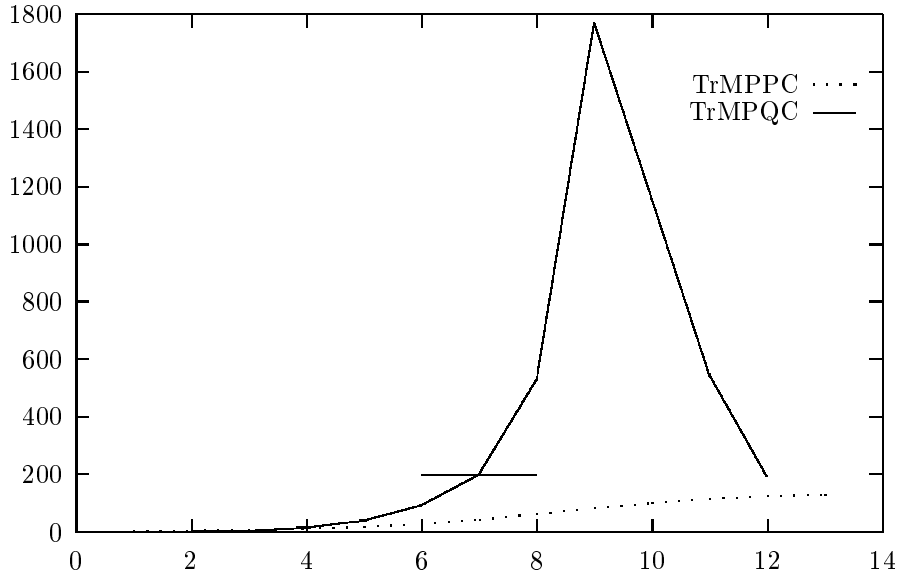

Fig. 17. Running times in seconds (y-axis) of `TrMPPC` and `TrMPQC` on `edfpa01` for different truncation orders (x-axis).

formed. The later argument is enforced by the observation that, at least for `MPMC` *and* `MPQC` *in the* `edfpa01` *case, the computations times decrease as the truncation order increases beyond a certain order.*

6.2 A Non Coherent Fault Tree from Dassault Aviation

The second tree we consider is a non-coherent one from Dassault Aviation. It is referenced under the name `das9605` in our benchmark. It has 287 gates ($\neg$, $\wedge$, $\vee$ and k-out-of-n gates), 122 primary events and only 28 small modules. The computation of the BDD encoding it takes 3.55 seconds on average. This BDD is made of 34913 nodes.

Since this tree is non-coherent, the notions of prime implicants and minimal p-cuts do not coincide. `das9605` admits $1.30977\ 10^{11}$ prime implicants. The BDD encoding them is computed in 173 seconds by `MPPI` and has 897253 nodes. The Fig.18 reports the evolution of the number of prime implicants as a function of the order. A number of remarks should be done about this figure.

- The prime implicants of lowest order contain already 22 literals.
- The did not succeed in computing truncated prime implicants whose orders are greater than 65. To give an idea, for the truncation order 60, `TrMPPI` takes 38 minutes and builds a BDD of $1,189,426$ nodes.

The number of prime implicants of `das9605` ($1.30977\ 10^{11}$) should be compared with its number of minimal p-cuts: 4259. The table 3 gives the number of

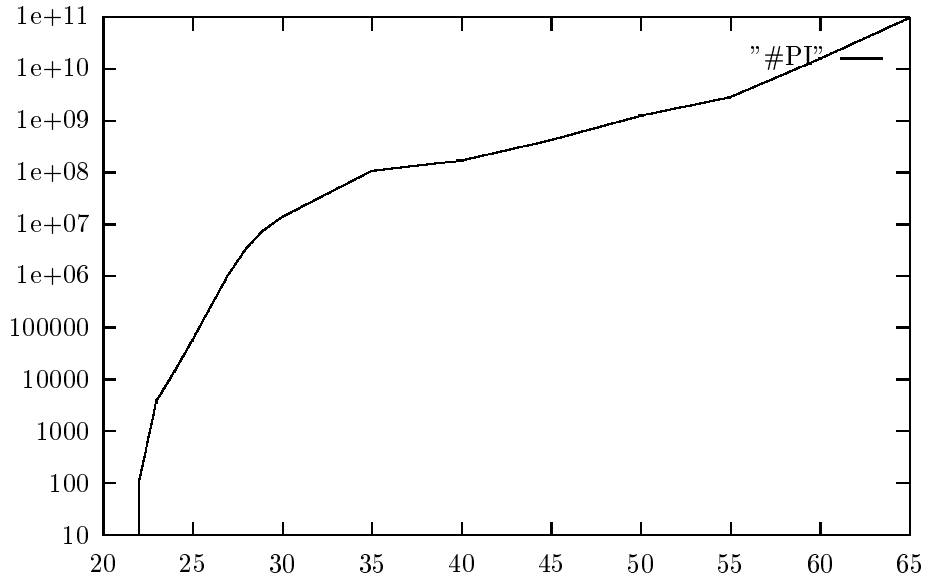

Fig. 18. Number of prime implicants (y-axis) as a function of the order (x-axis) for `das9605`.

Order	1	2	3	4	5	6	7	8	9	All
#p − cuts	0	47	80	319	342	571	580	1168	1152	4259
`TrMPPC`	0.89	5.24	25.20	94.03	288.72	924.34	5006	13,914	?	47,109
`TrMPQC`	0.01	0.06	0.30	0.91	2.03	3.56	5.41	7.28	8.86	2.06

Table 3
Number of minimal p-cuts and running times in seconds of `TrMPPC` and `TrMPQC` on `das9605` for different truncation orders.

minimal p-cuts for each order and the running times in seconds to get the BDD encoding them with `TrMPPC` and `TrMPQC`. `MPPC` and `MPQC` take respectively 47,109 and 2.06 seconds to compute the BDD encoding all minimal p-cuts.

These results lead to the two following remarks.

Remark 24 *Both orders and numbers of minimal p-cuts are reasonnable while those of prime implicants are not. This means that qualitative treatments can be achieved by means of the minimal p-cuts but not by means of prime implicants.*

Remark 25 *Since the number of minimal p-cuts remains small, `MPQC` is far more efficient than `MPPC` (which takes about 13 hours to achieve the computation).*

The computation of the exact probability of the top event of `das9605` (from the BDD encoding this function) takes 0.30 seconds. The table 4 reports the

	Probabilities				
Primary events	0.5	0.1	0.01	0.001	0.0001
Top event	$5.56868\ 10^{-5}$	0.125066	0.0042344	$4.6525\ 10^{-5}$	$4.69524\ 10^{-7}$
Truncation order	Contributions of minimal p – cuts				
2	17942.8	2.56175	1.08091	1.00768	1.00076
3	17955.3	2.81484	1.09866	1.00939	1.00093
4	17957.1	2.89925	1.09936	1.0094	1.00093
5	17957.1	2.90572	1.09937	1.0094	1.00093
6	17957.1	2.90659	1.09937	1.0094	1.00093
7	17957.1	2.90665	1.09937	1.0094	1.00093
8	17957.1	2.90666	1.09937	1.0094	1.00093
9	17957.1	2.90667	1.09937	1.0094	1.00093

Table 4
Numbers of minimal p-cuts and their contributions to the probability of the top event of `das9605`.

probability of the top event for different values of the probabilities of the primary events (for this test, we assume that all primary events have the same probability). Beyond the third row, it reports the contribution of minimal p-cuts to the probability of the top event.

Here, the term contribution should be taken with care: it means the quotient of the probability of the top event by the probability of the function encoded by the BDD `ReverseMP`(`TrMPPC`(`das9605`, k)). The prime implicants of this function are the minimal p-cuts of `das9605` whose orders are less than k. The important point is the following.

Remark 26 *None of the functions encoded by the BDD's* `ReverseMP`(`TrMPPC`(`das9605`, k)) *is equivalent to* `das9605`*, even the one encoded by the BDD* `ReverseMP`(`MPPC`(`das9605`)) *is not. These functions are monotone (since p-cuts contain only positive literals) while* `das9605` *is certainly not. This has two consequences:*

- *The contributions we get are approximations of the contributions of prime implicants, which explains that the table 4 reports contributions greater than 1.*
- *These approximations are sound only when the probabilities of primary events are low (and thus that the probabilities of the corresponding negative literals can be assimilated to 1), which explains that for high probabilities of primary events (0.5, 0.1) the obtained results have strictly no meaning.*

Remark 27 *Note however that for low probabilities of primary events ($10^{-3}, 10^{-4}$), approximations are rather good, which makes minimal p-cuts an useful tool both for qualitative and quantitative analysis of non coherent fault trees.*

7 Conclusion

In this paper, we introduced a new notion of implicants of Boolean functions, the minimal p-cuts, that formalizes the idea to keep only positive parts of prime implicants.

We proposed two BDD based algorithms to compute minimal p-cuts. We also proposed variations on these algorithms as well as on known algorithms to compute prime implicants that compute only prime implicants or minimal p-cuts whose order is less than a given value and whose probability is greater than a given threshold.

We provide experimental evidence that these algorithms allow fast, accurate and incremental qualitative and quantitative analyses of very large coherent and non-coherent real-life fault trees. These experiments show that the BDD based technology outperforms by orders of magnitude all previously proposed methods to assess the probability of a Boolean model.

References

[Ake78] B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, 27(6):509–516, 1978.

[Ara94] Groupe Aralia. Arbres de Défaillances et diagrammes binaires de décision. In *Actes du 1^{er} congrès interdisciplinaire sur la Qualité et la Sûreté de Fonctionnement*, pages 47–56. Université Technologique de Compiègne, 1994. The "Groupe Aralia" is constitued by A. Rauzy (LaBRI – Université Bordeaux I), Y. Dutuit (LADS – Université Bordeaux I), J.P. Signoret (Elf-Aquitaine – Pau), M. Chevalier (Schneider Electric – Grenoble), I. Morlaes (SGN – Saint Quentin en Yvelines), A.M. Lapassat (CEA-LETI – Saclay), S. Combacon (CEA-IPSN – Valduc), F. Brugère (Technicatome – Aix-Les Milles), M. Bouissou (EDF-DER – Clamart).

[Ara95] Groupe Aralia. Computation of Prime Implicants of a Fault Tree within Aralia. In *Proceedings of the European Safety and Reliability Association Conference, ESREL'95*, pages 190–202, Bournemouth – England, June 1995. European Safety and Reliability Association.

[BRB90] K. Brace, R. Rudell, and R. Bryant. Efficient Implementation of a BDD Package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*. IEEE 0738, 1990.

[Bry86] R. Bryant. Graph Based Algorithms for Boolean Fonction Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[Bry92] R. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24:293–318, September 1992.

[CM78] A.K. Chandra and G. Markowsky. On the number of prime implicants. *Discrete Mathemathics*, 24:7–11, 1978.

[CM92a] O. Coudert and J.-C. Madre. A New Method to Compute Prime and Essential Prime Implicants of Boolean Functions. In T. Knight and J. Savage, editors, *Advanced Research in VLSI and Parallel Systems*, pages 113–128, March 1992.

[CM92b] O. Coudert and J.-C. Madre. Implicit and Incremental Computation of Primes and Essential Primes of Boolean Functions. In *Proceedings of the 29th ACM/IEEE Design Automation Conference, DAC'92*, June 1992.

[CY85] L. Camarinopoulos and J. Yllera. An Improved Top-down Algorithm Combined with Modularization as Highly Efficient Method for Fault Tree Analysis. *Reliability Engineering and System Safety*, 11:93–108, 1985.

[LGTL85] W.S. Lee, D.L. Grosh, F.A. Tillman, and C.H. Lie. Fault tree analysis, methods and applications : a review. *IEEE Transcations on Reliability*, 34:194–303, 1985.

[Min93] S. Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In *Proceedings of the 30th ACM/IEEE Design Automation Conference, DAC'93*, pages 272–277, 1993.

[Qui52] W.V.O. Quine. The problem of simplifying truth functions. *American Mathematics Monthly*, 59:521–531, 1952.

[Qui59] W.V.O. Quine. On cores and prime implicants of truth functions. *American Mathematics Monthly*, 66:755–760, 1959.

[Rau93] A. Rauzy. New Algorithms for Fault Trees Analysis. *Reliability Engineering & System Safety*, 05(59):203–211, 1993.

[Rau96] A. Rauzy. An Introduction to Binary Decision Diagrams and some of their Applications to Risk Assessment. In O. Roux, editor, *Actes de l'école d'été, Modélisation et Vérification de Processus Parallèles, MOVEP'96*, 1996. Also Technical Report LaBRI number 1121-96.

[VGRH81] W.E. Vesely, F.F. Goldberg, N.H. Robert, and D.F. Haas. Fault Tree Handbook. Technical Report NUREG 0492, U.S. Nuclear Regulatory Commission, 1981.

A Proof of the Decomposition Theorem for Prime Implicants

Let us first recall the theorem to be proved.

Theorem 9 (Decomposition Theorem) *Let $f(x_1, \ldots, x_n)$ be a boolean function. Then, the set of prime implicants of $f(x_1, \ldots, x_n)$ can be obtained as the union of the sets.*

$$PI[f(x_1, \ldots, x_n)] = PI_{10} \cup PI_1 \cup PI_0$$

where, PI_{10}, PI_1 and PI_0 are defined as follows.

$$\begin{aligned} PI_{10} &\stackrel{\text{def}}{=} PI[f(1, x_2, \ldots, x_n) \wedge f(0, x_2, \ldots, x_n)] \\ PI_1 &\stackrel{\text{def}}{=} \{\{x_1\} \cup \pi; \pi \in PI[f(1, x_2, \ldots, x_n) \setminus PI_{10}\} \\ PI_0 &\stackrel{\text{def}}{=} \{\{\neg x_1\} \cup \pi; \pi \in PI[f(0, x_2, \ldots, x_n) \setminus PI_{10}\} \end{aligned}$$

where $\setminus$ stands for the set difference.

In order to prove the above theorem, we first demonstrate the following lemma.

Lemma 28 *Let $f(x_1, \ldots, x_n)$ be a Boolean function and let π be a product built over $x_1, \ldots, x_n$ such that $x_1 \notin \pi$ and $\neg x_1 \notin \pi$. Then, π is a prime implicant of $f(x_1, \ldots, x_n)$ iff it is a prime implicant of $f(1, x_2, \ldots, x_n) \wedge f(0, x_2, \ldots, x_n)$*

Proof: *if part:* Let π a prime implicant of $f(x_1, \ldots, x_n)$ such that $x_1 \notin \pi$ and $\neg x_1 \notin \pi$. Then, π is an implicant of $f(1, x_2, \ldots, x_n)$ and of $f(0, x_2, \ldots, x_n)$. Thus, π is an implicant of $f(1, x_2, \ldots, x_n) \wedge f(0, x_2, \ldots, x_n)$. Now, assume there exists a prime implicant ρ of $f(1, x_2, \ldots, x_n) \wedge f(0, x_2, \ldots, x_n)$ such that $\rho \subset \pi$. Clearly, ρ is an implicant of $ite(x, f(1, x_2, \ldots, x_n), f(0, x_2, \ldots, x_n))$ and thus of $f(x_1, \ldots, x_n)$. But π is prime: a contradiction.

only if part: Now, let π a prime implicant of $f(1, x_2, \ldots, x_n) \wedge f(0, x_2, \ldots, x_n)$ and assume that there is a prime implicant ρ of $f(x_1, \ldots, x_n)$ such that $\rho \subset \pi$. ρ cannot contain x and thus, by the above reasonning, it is an implicant of $f(1, x_2, \ldots, x_n) \wedge f(0, x_2, \ldots, x_n)$. A contradiction. □

In order to complete the proof of the theorem 9, it suffices to remark that to be a prime implicant of $f(x_1, \ldots, x_n)$, a product $\{x_1\} \cup \pi$ should fulfil the two following requirements:

(i) π is a prime implicant of $f(1, x_2, \ldots, x_n)$.
(ii) π is not a prime implicant of $f(x_1, \ldots, x_n)$, i.e. since $x_1 \notin \pi$, $\pi \notin PI[f(1, x_2, \ldots, x_n) \wedge f(0, x_2, \ldots, x_n)]$.

Similarly for prime implicants containing $\neg x_1$. $\square$

B Proof of the Decomposition Theorem for Minimal P-Cuts

Let us first recall the theorem to be proved.

Theorem 10 (Decomposition Theorem for Minimal P-Cuts) *Let $f(x_1, \ldots, x_n)$ be a boolean function. Then, the set of minimal p-cuts of $f(x_1, \ldots, x_n)$ can be obtained as the union of two sets.*

$$PC[f(x_1, \ldots, x_n)] = PC_1 \cup PC_0$$

where, PC_1 and PC_0 are defined as follows.

$$PC_1 \stackrel{\text{def}}{=} \{\{x_1\} \cup \pi; \pi \in PC[f(1, x_2, \ldots, x_n) \vee f(0, x_2, \ldots, x_n)] \setminus PC_0\}$$
$$PC_0 \stackrel{\text{def}}{=} PC[f(0, x_2, \ldots, x_n)]$$

Let us first remark that for any function $f(x_1, \ldots, x_n)$, the function $g(x_1, \ldots, x_n)$ defined as follows:

$$g(x_1, \ldots, x_n) = ite(x_1, f(1, x_2, \ldots, x_n) \vee f(0, x_2, \ldots, x_n), f(0, x_2, \ldots, x_n))$$

is monotone in x_1. Moreover, it is easy to see that

$$PC[g(x_1, \ldots, x_n)] = PC[f(x_1, \ldots, x_n)]$$

It is clear that they have the same minimal p-cuts that do not contain x_1. Let $\{x_1\} \cup \pi$ be a minimal p-cut of $g(x_1, \ldots, x_n)$. π^c should be an implicant of $g(1, x_2, \ldots, x_n)$ but not of $g(0, x_2, \ldots, x_n)$, that is π^c should be an implicant of $f(1, x_2, \ldots, x_n)$.

The reader will easily complete in a way very similar to the one we used to demonstrate theorem 9. $\square$

Notice that it is mandatory to consider minimal p-cuts of $f(1, x_2, \ldots, x_n) \vee f(0, x_2, \ldots, x_n)$ in order to get minimal p-cuts of $f(x_1, \ldots, x_n)$ that contain x_1. The reason is that one should eliminate from minimal p-cuts of $f(1, x_2, \ldots, x_n)$ those that contain strictly a minimal p-cut of $f(0, x_2, \ldots, x_n)$. For instance, let $f(a, b, c) = (a \wedge b \wedge c) \vee (\neg a \wedge b)$. We have $PC[f(1, b, c)] = \{\{b, c\}\}$ and

$PC[f(0,b,c)] = \{\{b\}\}$, and thus $PC[f(1,b,c)] \setminus PC[f(0,b,c)] = PC[f(1,b,c)]$, but $PC[f(1,b,c) \vee f(0,b,c)] = PC[f(0,b,c)]$. Thus, by considering minimal p-cuts of $f(1,b,c) \vee f(0,b,c)$, we eliminate the product $\{b,c\}$ which does not belong to $PC[f(0,b,c)]$ but that contains strictly a product of this set, namely $\{c\}$.