Towards an Efficient Implementation of MOCUS

A. Rauzy
IML, UPR CNRS 9016
163, avenue de Luminy – Case 907
F-13288 Marseille cedex 9 – FRANCE
arauzy@iml.univ-mrs.fr

Summary & conclusions

MOCUS is probably the most famous algorithm to compute minimal cutsets of fault trees. It has been proposed by Fussel and Vesely in 1972. It is at the present the core method of many fault tree assessment tools. Despite its wide use, textbooks and articles give only few details about how to implement it.

In this article, we describe data structures as well as several improvements and heuristics that make MOCUS robust and efficient. We introduce the notion of shadow variables in order to deal with success branches of event trees.

We report experiments on a benchmark of the 1819 event tree sequences that were generated during a PSA study. These results show that MOCUS is a good alternative to Binary Decision Diagrams.

Keywords: Fault Trees, Event Trees, MOCUS

1 Introduction

MOCUS is probably the most well known algorithm to compute minimal cutsets of fault trees, event trees, block diagrams, ... It has been proposed by Fussel and Vesely in 1972 [1]. It is representative of the class of top-down algorithms. It is at the present the core method of many fault tree assessment tools, including Risk Spectrum [2] and IRRAS (SAPHIRE) [3] that are both widely used by in nuclear enginering. Each of these tools introduces its own refinements on the original method, but the basis remains the same.

Despite the wide use of top-down algorithms, textbooks and articles devoted to this topic give only few details about how to implement them. For instance, the question of data structures is almost never discussed although it plays a central rôle in the efficiency of the methods. In this article, we describe a full implementation of MOCUS, including data structures that make it robust and efficient. We try to point out all the places where design decisions must be taken and we discuss our choices.

As a benchmark, we used the 1819 event tree sequences that were generated during a PSA study. Some of these sequences contain more than one thousand input variables, twice more gates, among which a significant proportion are replicated. To cope with such large models, we introduced new ideas, such as the notion of shadow variables. This article presents these improvements. We show also that MOCUS is sensitive to the way formulae are written and we propose a rewriting heuristic that overcomes this problem.

During last decade, the framework of fault tree assessment algorithms has been deeply impacted by the introduction of Binary Decision Diagrams (BDDs) based methods (see for instance [4, 5] for the very first contributions). BDDs [6, 7] are the state-of-the-art data structure to encode and to manipulate Boolean functions. In most of the cases, they outperform the other methods. However, BDDs also are subject to combinatorial explosion. For large models, such as the largest sequences of our benchmark, the BDD that encodes the model may be not computable within reasonable amounts of time and computer memory. The problem comes that, conversely to MOCUS-like algorithms, the BDD technology does not support easily approximations (although some progresses have been done recently in this direction [8]). Therefore, BDDs may be unable to provide any result, which is indeed unacceptable. This is a strong motivation to develop efficient alternative algorithms. The experimental results we obtained on our benchmark show that MOCUS is a good candidate to be such an alternative method.

The remainder of this article is organized as follows. Some preliminary definitions are given section 3. The section 4 describes an efficient implementation of MOCUS. The section 5 discusses several improvements. Finally, experimental results are reported section 6.

2 Notations

Throughout this article, we shall use the following notations.

- $\mathcal{E} = \{e_1, \ldots, e_N\}$ denotes the set of basic events.
- $\mathcal{G} = \{g_1, \ldots, g_M\}$ denotes the set of gate variables.
- F_1, F_2, ... denote Boolean formulae.
- $var(F)$ denotes the set of variables occurring in F.
- $MCS[F]$ denotes the set of minimal cutsets of F.
- $p(F)$ denotes the probability of a F.

3 Preliminaries

3.1 Sets of Equations

In what follows, we consider Boolean models that are given as sets of (Boolean) equations. Structure functions of fault trees, event trees, block diagrams can be viewed as such sets of equations.

Let $\mathcal{E} = \{e_1, \ldots, e_N\}$ and $\mathcal{G} = \{g_1, \ldots, g_M\}$ be two distinct sets of Boolean variables. $\mathcal{E}$ is the set of basic events. $\mathcal{G}$ is the set of gates. An equation over $\mathcal{E} \cup \mathcal{G}$ is an equality of the following form.

$$g \;=\; F$$

where $g \in \mathcal{G}$, and F is a boolean formula built over $\mathcal{E} \cup \mathcal{G}$ and the usual logical connectives "." (and), "+" (or), "$^-$" (not), k/n (k-out-of-n).

A set of equations is assimilated with the conjunct of its elements. A set E of equations is said hierarchical if it fulfils the following requirements.

- For each g in $\mathcal{G}$, E contains exactly one equation of the form $g = F$. For the sake of simplicity, we denote F_i $(i = 1, \ldots, M)$ the unique formula such that the equation $g_i = F_i$ is in E.
- There exists a one-to-one function ι from $\mathcal{G}$ to $[1, M]$ (i.e. a total order over $\mathcal{G}$) such that for any two gates g_i and g_j the following holds.

$$\iota(g_i) \leq \iota(g_j) \;\;\Rightarrow\;\; g_i \notin var(F_j)$$

- E is uniquely rooted, i.e. there is a unique gate g_{top} that occurs in no F_i.

In what follows, we shall consider only hierarchical sets of equations.

Any (hierarchical) set of equation E can be rewritten into an equivalent singleton $\{g_{top} = F'_{top}\}$. To obtain F'_{top}, it suffices to substitute the F_i's for the g_i's according to the order ι. For this reason, E can be assimilated with the formula F'_{top}. For the sake of simplicity, we shall do systematically this assimilation. For instance, we shall write "the minimal cutsets of E" instead of "the minimal cutsets of F'_{top} where F'_{top} is the formula such that ... ".

3.2 Minimal Cutsets

A literal is either a variable v or its negation $\overline{v}$. A product is a conjunct of literals that does not contain both a literal and its negation (v and $\overline{v}$). A minterm is a product that contains a literal for each variable of $\mathcal{E}$. Products and minterms are assimilated with sets of literals.

A minterm π can be seen also as an assignment, i.e. a function λ_π from $\mathcal{E}$ to $\{0, 1\}$.

$$\lambda_\pi(e) \;=\; \begin{cases} 1 & \text{if } e \in \pi \\ 0 & \text{if } \overline{e} \in \pi \end{cases}$$

For the sake of simplicity, we shall just write $\pi(e)$ instead of $\lambda_\pi(e)$. Assignments can be lifted up to formulae by structural induction and according to the usual laws.

$$\pi(\overline{F}) = 1 - \pi(F) \qquad \pi(F.G) = min(\pi(F), \pi(G)) \qquad \pi(F + G) = max(\pi(F), \pi(G))$$

Let π be a product, we denote by π^c the minterm obtained by complementing π with negative literals built over the events that do not occur in π.

A positive product π is a minimal cutset of a set of equations E if the following conditions hold.

- π is a cutset of E, i.e. $\pi^c(E) = 1$.
- π is minimal, i.e. for any product $\rho \subset \pi$, $\rho^c(E) = 0$.

The reference [8] gives an algebraic framework for the notion of minimal cutsets.

The set of minimal cutsets of E in denoted by $MCS[E]$.

3.3 And-Or-Trees

MOCUS takes And-Or-Trees as input formulae. An And-Or-Tree is a set of equations such that each F_i is made with a single connective and only events can be negated. Any set of equations can be efficiently rewritten as an And-Or-Tree. The algorithm is as follows.

1. Flatten equations, i.e. a new equation is created for each nested connective. E.g.

$$g = a + b.c + d \quad \longrightarrow \quad \left\{ \begin{array}{l} g = a + g_{new} + d \\ g_{new} = b.c \end{array} \right.$$

2. Expand k-out-of-n gates. This can be done in $\mathcal{O}(k.n)$ by applying the following decomposition.

$$g = k/n(v_1, \ldots, v_n) \quad \longrightarrow \quad \left\{ \begin{array}{l} g = g_{new1} + g_{new2} \\ g_{new1} = x_1 . g_{new3} \\ g_{new2} = k/n - 1(v_2, \ldots, v_n) \\ g_{new3} = k - 1/n - 1(v_2, \ldots, v_n) \end{array} \right.$$

Note that most of the authors (e.g. [3, 2]) suggest to expand k-out-of-n gates as a sum of products. When $k \approx n/2$ and n is not too small, this may be costly. The process we propose works even for large values of n.

3. Push down negations. A new gate variable g_{not} is created (when needed) for the gate variable g. g_{not} encodes the negation of g. The equation of g_{not} is obtained by means of de Morgan's laws, e.g.

$$\left. \begin{array}{l} h = \overline{g} + d \\ g = a.b.c \end{array} \right\} \quad \longrightarrow \quad \left\{ \begin{array}{l} h = g_{not} + d \\ g_{not} = a_{not} + b_{not} + c_{not} \end{array} \right.$$

This process is achieved in one top-down pass over the set of equations.

```
$P \leftarrow \emptyset$, $R \leftarrow \{g_{top}\}$
while $R \neq \emptyset$ do
    select a product $\pi$ in $R$, $R \leftarrow R \setminus \{\pi\}$
    if $\pi$ is terminal
        then $P \leftarrow P \cup \mathrm{ExtractMcs}(\rho)$
        else
            select a gate $g$ in $\pi$: $\pi = g.\rho$
            if $g = \Pi_{i=1}^{k} l_i$
                then insert $\rho.\Pi_{i=1}^{k} l_i$ in $R$
            if $g = \sum_{i=1}^{k} l_i$
                then if $\exists\ l_i \in \rho$
                    else insert $\rho$ in $R$
                    else insert $\rho.l_1, \ldots, \rho.l_k$ in $R$
```

Figure 1: The pseudo-code for the MOCUS algorithm

4 MOCUS

4.1 Principle of the Algorithm

MOCUS works with two sets of products. The set R of products it remains to process and the set P of already found minimal cutsets. R is initialized with $\{g_{top}\}$, where g_{top} is the root of the And-Or-Tree under study. P is initialized with the empty set. Then, each product π of R is considered in turn.

- If π is terminal, i.e. if it contains only basic events, minimal cutsets of π are extracted and inserted in P. This process is discussed in section 4.2.
- Otherwise, a gate variable is selected in π and replaced by its definition, possibly giving several new products. The obtained products are reinserted in R.

The pseudo-code for the algorithm is given Fig. 1.

Let π be a non-terminal product considered at a given step of the algoritm. Let g be the gate variable selected in π ($\pi = g.\rho$). Finally, let $g = F$ be the equation that defines g in E. There are two cases.

- If $F = \Pi_{i=1}^{k} l_i$, then π is rewritten in $\rho.F$. $\rho.F$ is reinserted in R if it contains no pair of opposite literals.
- If $F = \sum_{i=1}^{k} l_i$, then π is rewritten in $\sum_{i=1}^{k} \rho.l_i$. If ρ contains one of the l_i, this sum is equivalent to ρ, which is reinserted in R. If ρ contains the opposite of l_i, the product $\rho.l_i$ is discarded. The remaining products $\rho.l_i$ are reinserted in R.

Such simplifications have been described already by several authors (see for instance [3]).

4.2 Extraction of Minimal Cutsets

Let π be a terminal product considered at a given step of the algoritm. Let $\rho = \Pi_{i=1}^{k} e_i$ be the positive part of π. By construction, ρ^c is a cutset of E. However, it may be non minimal. Minimal cutsets of ρ are extracted as follows.

The products $\rho_i = \rho \setminus \{e_i\}$ are considered in turn. If all of these products are such that $\rho_i^c(E) = 0$, ρ is minimal. Otherwise, ρ is not minimal and minimal cutsets are extracted from the ρ_i's such that $\rho_i^c(E) = 1$. In order to avoid to do the same job twice, a cache can be used to store already considered products (the same goal can be achieved by considering the e_i's always in the same order). Note also that to change ρ_i into ρ_{i+1} it suffices to flip the values of e_i and e_{i+1}.

At the end of this process, a number of minimal cutsets are obtained. It remains to discard those that are already in P. The section 4.4 proposes a data structure to do this efficiently.

4.3 Truncations on the Probabilities

It is often the case, when dealing with real-life models, that the set E of equations under study admits a huge number of minimal cutsets. Fortunately, it is in general an accurate approximation to consider only the most important ones, i.e. those that have the highest contribution to the system unreliability.

The contribution of a minimal cutset ρ is defined as follows.

$$\frac{p(\rho)}{\sum_{\pi \in MCS[E]} p(\pi)}$$

The contribution is sometimes called the Fussel-Vesely importance factor of ρ for it has been proposed by these authors in [9] as a risk ranking measure. Note that for any subset P of $MCS[E]$ the following inequality holds.

$$\frac{p(\rho)}{\sum_{\pi \in P} p(\pi)} \geq \frac{p(\rho)}{\sum_{\pi \in MCS[E]} p(\pi)}$$

Therefore, if the contribution of a product estimated only from the already found minimal cutsets is below a given threshold T, its actual contribution is below the threshold as well (and the product can be immediately discarded). The threshold T is often called the relative cutoff.

Now, consider the sequence $\pi_1, \pi_2, \ldots$ of terminal products produced by the algorithm (before the extraction of minimal cutsets). It is easy to verify *ad absurdo* that if ρ is a minimal cutset, then there is a i such that ρ is the positive part of π_i. This strong property is used to discard under process products as soon as their contribution goes below the threshold T. The property ensures that no minimal cutset (of contribution greater than T) is discarded.

This is the reason why it is interesting to produce minimal custets (with the process presented in the previous section) as soon as possible. The earlier minimal cutsets are produced, the more effective the truncation on the probability is.

It is worth noticing that, in case of hard computations, the relative cutoff T can be dynamically adjusted.

4.4 Data structures

Several points remain to discuss in the algorithm described so far.

- One needs two heuristics. The first one to select the next product to process in R. The second one to select the gate variable to expand in the product under process. This question will be discussed section 5.2.
- One needs a data structure to encode the set of equations. This structure has mainly to make efficient the extraction of minimal cutsets.
- Finally, one needs a data structure to encode the sets of products R and P.

The minimal cutsets extraction is based on the bottom-up propagation of the assignment of values to basic events. To make this operation efficient, one creates once for all, for each variable v, the list of equations $g = F$ such that $v \in var(F)$. Moreover, one maintains dynamically, for each equation $g = F$, three counters of the numbers of variables of $var(F)$ that have respectively the value 1, the value 0 and no value. In this way, values are propagated bottom-up only when needed.

It is actually interesting to maintain the current value of variables (gates and events) during the whole algorithm. Let π be the product under process. It can be the case that the value of a gate variable g of π is determined just by propagating the values of basic events of π. If $\pi(g) = 1$, then g can be removed from the product. If $\pi(g) = 0$, then the product contains a contradiction and can be discarded. Another even more important reason to maintain the current assignment will be given in section 5.3 with the notion of shadow variables.

The data structure that encodes the sets of products R and P should be as compact as possible (for huge numbers of products have to be manipulated). Moreover, insertion, selection and removal of a product should be as efficient as possible. Minato's Zero-Suppressed Binary Decision Diagrams [10] are a potential candidate to do the job. We found that Binary Decision Trees are a better tradeoff. Let ι be a total order over $\mathcal{E} \cup \mathcal{G}$. A Binary Decision Tree is a binary tree such that:

- The leaves of the tree encode either the empty product or the empty set of products.
- Each internal node is labelled with a variable v and has two outedges (low and high). If the nodes pointed by low and high encode respectively the sets of products S_0 and S_1, then the node encodes the following set.

$$S_0 \cup \{\{v\} \cup \pi; \pi \in S_1\}$$

- If a node is labelled with a variable v, then either the node pointed by its low outedge (resp. high outedge) is a leaf or it is labelled with a variable w such that $\iota(v) < \iota(w)$.

By sharing the prefixes of products, this data structure saves a lot of memory. It is to see that insertion, removal and selection of a product are linear, in the worst case, in the number of variables. It is worth noticing that it is not necessary to test whether a minimal cutset belongs to the set before inserting it. Both operations are performed at once.

5 Further Improvements

5.1 Preprocessing

As pointed out by many authors, some preprocessing of the set of equations may increase by orders of magnitude the efficiency of algorithms. The book by Kovalenko, Kuznetsov and Pegg [11] proposes twelve transformations to simplify the formulae. These transformations are used by the FAMOCUTN algorithm [12]. They are representative of what is suggested in the literature. They can be grouped in three categories.

- Module detection. A module is a subformula that acts as a super-component, i.e. that is independant from the rest of the formula. The notion of module was introduced by Birnbaum and Esary in [13]. Modules already present in the formula can be detected in linear time [14]. However, modules are often hidden and some rewritings are necessary to extract them.

- Coalescing of gates of the same type, e.g. $F_1 + (F_2 + F_3) \longrightarrow F_1 + F_2 + F_3$. It is clearly interesting to perform coalescing as much as possible because this transformation does once for all operations that MOCUS will do anyway, possibly several times.

- Other transformations based on Boolean algebra laws. For instance, the following transformations may be applied.

$$\begin{array}{lcll} (F.G_1) + (F.G_2) & \longrightarrow & F.(G_1 + G_2) & \text{factoring} \\ F.v & \longrightarrow & F[1/v].v & \text{constant propagation} \end{array}$$

  These transformations aim to simplify the formula, for instance by detecting hidden modules. In [15], Niemelä has suggested also a promising rewriting technique based on propagation of variable values.

The danger with the latter category of preprocessings is that they may be costly. Their application should therefore result of a tradeoff between what they consume and what they save. In our implementation, we limit preprocessing to coalescing and module detection.

5.2 Heuristics

As already said, MOCUS requires two heuristics. The first one to select the next product to process. The second one to select the gate to expand in the product under process.

Both heuristics have the same goal: make terminal products (and therefore minimal cutsets) appear as soon as possible in order to increase the influence of the probabilistic cutoff.

For the second heuristic, Camarinopoulos and Yllera suggest in [16] to develop first the gate with the greatest number of basic events as arguments. This idea is uneasy to justify with respect to the above rule. We rather suggest to select the first and-gate, if any. The idea is that by developing and-gates first, we increase the expectation of a probabilistic cutoff. The order in which and-gates are expanded does not really matter, since all of them have to be expanded before an or-gate is developped. If the product contains only or-gates, the first one is developped.

For the first heuristic, we select the first product in the binary decision tree order.

Both heuristics depend strongly on the order defined among variables. It is clear for the first one, since this order determines the structure of the binary decision tree. It is true also for the second one, because under process products are encoded as sorted vectors of literals, according to the chosen order (sorted products make easier operations such inclusion testing, element searching, merging, ...).

The order among variables is determined by means of a depth-first left-most traversal of the set of equations. Before applying this procedure, gate arguments are sorted in increasing order of their weights. The weight $\omega(v)$ of a variable v is as follows.

$$\omega(v) = \begin{cases} 1 & \text{if } v \text{ is an input variable} \\ \omega(w) & \text{if } v = \overline{w} \\ \sum_{i=1}^{k} \omega(w_i) & \text{if } v = \sum_{i=1}^{k} w_i \\ \Pi_{i=1}^{k} \omega(w_i) & \text{if } v = \Pi_{i=1}^{k} w_i \end{cases}$$

As we shall see section 6.4, this heuristics is important for the robustness of the whole process.

We found experimentally that this static way of selecting products to process and variables to develop is a better tradeoff than the various dynamic heuristics we tried. However, there is certainly still room for improvements in this part of our implementation.

5.3 Shadow Variables

MOCUS-like algorithms have been designed to assess both fault trees and event trees. An event tree sequence is assimilated with the conjunct of success and failure branches that appear along the sequence [17]. Consider for instance the small event tree pictured Fig. 2. This event tree is made of an initiating event I, two fault trees F_A and F_B that describe respectively the failures of safety systems A and B, and three consequences C_1, C_2 and C_3. The sequence $I - C_2$ corresponds to the scenario where A fails and B achieves its mission. It is therefore assimilated with the formula $I.F_A.\overline{F_B}$.

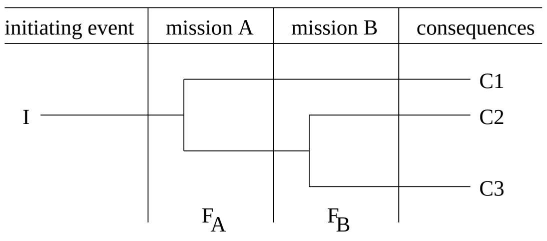

Figure 2: An event tree

It is in general the case that F_B is a coherent fault tree (i.e. F_B contains no negation). Therefore, the only rôle of $\overline{F_B}$ in the conjunct is to discard some of the minimal cutsets of $I.F_A$. This has been already noticed by several authors (e.g. [3]).

Let g_{notB} be the gate variable that encodes $\overline{F_B}$ in the model. We call variables such as g_{notB} "shadow variables". Shadow variables are never expanded. They are however involved in value propagations. If, at a given step of the algorithm, the product under process falsifies one of its shadow variables then this product contains a contradiction (as if it was containing both a literal and its opposite). It is therefore discarded. A product is now considered as terminal if it contains only basic events and shadow variables. The latters are just ignored by the minimal cutsets extraction.

This way of dealing with success branches is much more efficient than those proposed already that consists in computing minimal cutsets of failure and success branches separately and then to compare them. For two reasons. First, shadow variables introduce a new cutoff. Second, there is no more need to post-process the cutsets.

Some of the fault trees of our benchmark are only almost coherent, i.e. that they contain few negated variables. We observed that, even in this case, it is always an accurate approximation to shadow success branches.

It is worth noticing that a significant proportion of the sequences that will be studied in section 6 cannot be handled without this notion of shadow variables, because of the combinatorial explosion of the number of products to process.

6 Experimental Results

6.1 1819 Event Tree Sequences as a Benchmark

As a benchmark, we considered the 1819 event tree sequences generated for a PSA study.

The table 1 gives some statistics about these sequences. I denotes the number of basic events. G denotes the number of gate variables. R denotes the number of replicated variables. A variable is replicated if it occurs more than once in the set of equations. Finally, S denotes the number of singular input variables. A variable is singular if there is only one path to goes from the variable to the top event. We grouped the sequences according to their number of basic events: less than 100, between 100 and 200, between

Table 1: Statistics about the 1819 sequences

I	< 100	< 200	< 300	< 400	< 500	< 600	< 700
#sequences	408	42	274	100	42	344	92
mean G/I	1.45	1.59	1.70	1.33	1.30	1.34	1.38
mean $R/(I+G)$	0.01	0.01	0.14	0.14	0.14	0.15	0.18
mean S/I	0.65	0.51	0.42	0.57	0.61	0.58	0.40

I	< 800	< 900	< 1000	< 1100	< 1200	< 1300
#sequences	113	236	12	116	32	8
mean G/I	1.54	1.64	1.69	1.85	1.80	1.91
mean $R/(I+G)$	0.17	0.17	0.22	0.20	0.20	0.22
mean S/I	0.35	0.38	0.26	0.27	0.28	0.18

Table 2: Running Times

threshold for	#sequences assessed among the 1819							highest
the contribution	$< 1s$	$< 5s$	$< 10s$	$< 30s$	$< 1m$	$< 10m$	$\geq 10m$	running time
10^{-1}	1772	38	8	1	0	0	0	10.24
10^{-2}	1743	60	8	8	0	0	0	16.77
10^{-3}	1688	88	20	21	2	0	0	31.28
10^{-4}	1634	101	40	26	11	7	0	89.89
10^{-5}	1576	111	40	56	12	24	0	248.77
10^{-6}	1486	163	40	68	19	39	4	623.33
BDD	1705	60	18	13	11	12	0	304.22

200 and 300 and so on.

The table 1 shows that the sequences of the benchmark contain a significant proportion of replicated variables. Moreover, the more sequences contain basic events, the less the proportion of singular variables is.

6.2 Running Times

The table 2 gives the number of sequences assessed whithin less than 1s, within more than 1s and less 5s, and so on for different values of the threshold on the contribution. The contribution is (over-)estimated as described section 4.3. The running times were measured on a Pentium III cadenced at 733Mgz and running linux.

The table 2 shows that most of the sequences are assessed whithin less than 30s, even for very low values of the threshold. For comparison purposes, the running times to compute BDDs are given. The advantage seems to go to BDDs. However, it should be said that a highly sophisticated preprocessing and many ad-hoc tricks were used to be able to compute

Table 3: MOCUS errors.

	MOCUS optimistic				MOCUS pessimistic			
	error :	mean	maximum		error :	mean	maximum	
contribution	#seq.	relative	absolute	relative	#seq.	relative	absolute	relative
10^{-1}	115	0.64	$1.29\ 10^{-9}$	1	13	21.5	$2.21\ 10^{-9}$	242
10^{-2}	73	0.32	$2.79\ 10^{-10}$	0.61	55	11.5	$7.32\ 10^{-9}$	539
10^{-3}	29	0.16	$2.44\ 10^{-11}$	0.30	99	8.40	$9.05\ 10^{-9}$	664
10^{-4}	22	0.07	$6.54\ 10^{-12}$	0.12	106	8.98	$1.19\ 10^{-8}$	741
10^{-5}	15	0.02	$9.24\ 10^{-13}$	0.08	113	8.77	$1.28\ 10^{-8}$	762
10^{-6}	4	0.02	$6.89\ 10^{-16}$	0.06	124	8.12	$1.31\ 10^{-8}$	769

these BDDs (these techniques will be presented in a forthcoming article). Moreover, the given running times for BDD do not include the computation of minimal cutsets. For the largest sequences, we are unable at the moment to compute these minimal cutsets. In a word, our implementation of MOCUS is a bit less efficient than our BDD implementation, but it is more robust.

6.3 Accurracy of the Results

It is of interest to compare the unreliabilities estimated by MOCUS with the exact results provided by BDDs. The table 3 gives the mean relative error as well as the maximum absolute and relative errors done by MOCUS for the 128 hardest sequences (i.e. the sequences for which MOCUS with the contribution threshold set at 10^{-6} takes more than 10s). The following conclusions can be drawn from these results.

- MOCUS is rarely optimistic. The smaller the threshold on contribution is, the less optimistic is MOCUS. Moreover MOCUS, is only slightly optimistic.
- MOCUS is often pessimistic, and sometimes very pessimistic (up to a factor 1000). However, this can be corrected by considering more terms in the Sylvester-Poincaré development.

6.4 Influence of Fanin Ordering

Except in [16], the order in which top-down algorithms consider the products and the variables inside the products is not discussed in literature. It has however a great influence. To show this influence, we consider 30 rewritings of the hardest sequence of our benchmark. Each rewriting consists in permuting at random the arguments of gates. For each rewriting, we call MOCUS with the with and without the preprocessing discussed section 5.2. The contribution threshold is fixed at 10^{-4}.

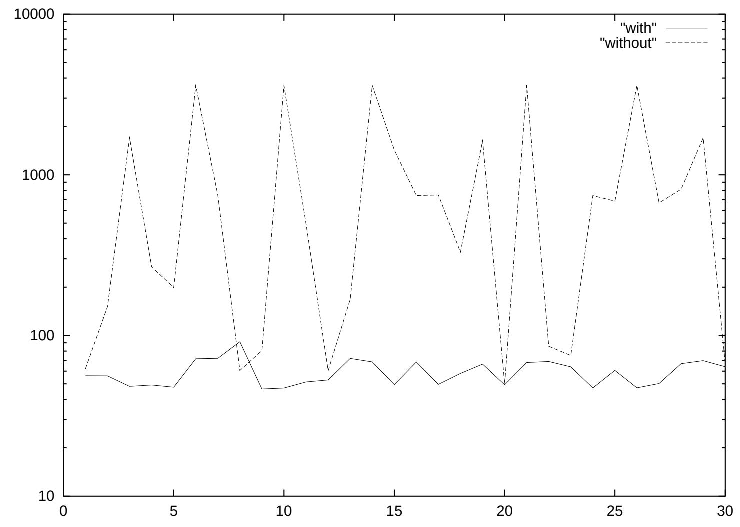

Figure 3: Running times with and without rearrangement of gate arguments.

The figure 3 shows the running times (to make the figure illustrative we joined the points with lines and we draw running times on a logarithmic scale). These two curves illustrate the interest of the heuristic we propose and show that it is robust and efficient.

References

[1] J. Fussel and W. Vesely, “A New Methodology for Obtaining Cut Sets for Fault Trees,” *Trans. Am. Nucl. Soc.*, vol. 15, pp. 262–263, June 1972.

[2] U. Berg, *RISK SPECTRUM, Theory Manual.* RELCON Teknik AB, April 1994.

[3] K. Russel and D. Rasmuson, “Fault tree reduction and quantification – and overview of IRRAS algorithms,” *Reliability Engineering and System Safety*, vol. 40, pp. 149–164, 1993.

[4] O. Coudert and J.-C. Madre, “Fault Tree Analysis: 10^{20} Prime Implicants and Beyond,” in *Proceedings of the Annual Reliability and Maintainability Symposium, ARMS'93*, January 1993. Atlanta NC, USA.

[5] A. Rauzy, “New Algorithms for Fault Trees Analysis,” *Reliability Engineering & System Safety*, vol. 05, no. 59, pp. 203–211, 1993.

[6] R. Bryant, “Graph Based Algorithms for Boolean Fonction Manipulation,” *IEEE Transactions on Computers*, vol. 35, pp. 677–691, August 1986.

[7] K. Brace, R. Rudell, and R. Bryant, “Efficient Implementation of a BDD Package,” in *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pp. 40–45, IEEE 0738, 1990.

[8] A. Rauzy, “Mathematical Foundation of Minimal Cutsets,” *IEEE Transactions on Reliability*, 2000. to appear.

[9] J. Fussel, “How to hand-calculate system reliability characteristics,” *IEEE Transactions on Reliability*, vol. R-24, no. 3, 1975.

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

[11] I. Kovalenko, N. Kuznetsov, and P. Pegg, *Mathematical Theory of Reliability of Time Dependent Systems with Practical Applications.* Wiley Series in Probability and Statistics, John Wiley & Sons, 1997. ISBN 0-471-95060-2.

[12] W. Hennings and N. Kuznetsov, “FAMOCUTN and CUTQ — Computer codes for fast analytical evaluation of large fault trees with replicated and negated gates,” *IEEE Transaction on Reliability*, vol. 44, pp. 368–376, 1995.

[13] Z. Birnbaum and J. Esary, “Modules of coherent binary systems,” *SIAM J. of Applied Mathematics*, vol. 13, pp. 442–462, 1965.

[14] Y. Dutuit and A. Rauzy, “A Linear Time Algorithm to Find Modules of Fault Trees,” *IEEE Transactions on Reliability*, vol. 45, no. 3, pp. 422–425, 1996.

[15] I. Niemelä, “On simplification of large fault trees,” *Reliability Engineering and System Safety*, vol. 44, pp. 135–138, 1994.

[16] 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*, vol. 11, pp. 93–108, 1985.

[17] I. Papazoglou, “Mathematical foundations of event trees,” *Reliability Engineering and System Safety*, vol. 61, pp. 169–183, 1998.

Antoine Rauzy is with the French National Center for Scientific Research (CNRS) and the “Institut de Mathématique de Luminy”. His topics of interest are formal methods and reliability engineering. His background is in computer science (PhD, Habilitation à Diriger des Recherches).