Assessment of Large Automatically Generated Fault Trees by means of Binary Decision Diagrams

J. Gauthier, X. Leduc[1] and A. Rauzy[2]

Abstract: Dassault Aviation developed a reliability workbench based on the high level formal description language AltaRica. The workbench includes a compiler of AltaRica models into fault trees. The fault trees generated for the largest industrial systems involve up to a thousand basic events and several dozens thousands gates. Moreover, they are non-coherent. The assessment of such large formulae is challenging, even for Binary Decision Diagrams, the state-of-the-art data structure to encode and to manipulate Boolean functions. In this article, we describe the various heuristics and strategies we used to make it tractable.
Keywords: AltaRica, Fault Trees, Binary Decision Diagrams, formulae rewritings

1. Introduction

The Fault Tree method [VGRH81] is the most commonly used method to analyze the risk in avionic industry. Fault trees are a good trade-off between modeling capacities and the practical efficiency of assessment algorithms. However, fault tree models are disconnected from the functional architecture of the systems under study. Therefore, even a minor change in the latter may induce dramatic changes in the formers. An idea to circumvent this problem is to derive automatically fault trees from a functional description. To this purpose, Dassault Aviation developed a reliability workbench based on the high level formal description language AltaRica [AGPR00]. The principle of the compilation of AltaRica models into fault trees is described in reference [Rau02].

To assess fault trees, Dassault aviation uses a Binary Decision Diagrams engine. Bryant's Binary Decision Diagrams (BDD for short) [Bry86, BRB90] are the state-of-the-art data structure to encode and to manipulate Boolean functions. Since their introduction in the reliability field [CM93, Rau93], BDD have proved to be one of the most efficient tools to

[1] Dassault Aviation, Flight Control System Certification, Dassault Equipement, Quai Marcel Dassault, 92210 Saint-Cloud, France, {jean.gauthier, xavier.leduc}@dassault-aviation.com

[2] IML/CNRS, 169 Avenue de Luminy, 13288 Marseille Cedex 9, France, arauzy@iml.univ-mrs.fr

assess fault trees. The treatment chain (compilation of AltaRica Models and assessment with BDD) works actually well on small and medium size models. However, it fails to analyze, at least without some additional work, the largest models at hand. The corresponding fault trees involve up to a thousand basic events and several dozens thousands gates. Moreover, they are non coherent, which makes them almost impossible to handle with the classical approach based on minimal cutsets extraction [VGRH81]. In this article, we describe the various techniques we used to make their assessment tractable.

It is well known that the efficiency of the whole BDD technique is very sensitive to the chosen variable (basic event) ordering. Since the determination of the best variable ordering is intractable [FS90, BW96], heuristics are designed to find reasonably good ones (see e.g. [AMS03, FFK88]). These heuristics rely in general on topological considerations and depend on the way formulae are written. Rewriting procedures are thus used both to simplify the trees and to get better variable orderings. The two issues are strongly related. Any complete fault tree analysis BDD strategy should therefore involve such rewritings, as illustrated in references [RA02 and ER05]. It is also commonly admitted that running times spent into this preprocessing step are negligible. With fault trees we had at hand, the picture is dramatically different. First, rewritings must be carefully optimized to avoid unacceptable running times. Second, it turns out that if rewritings are necessary to make models tractable, variable ordering heuristics are surprisingly of a little help. More important is the fine tuning of the management policy of the various BDD tables and of BDD construction.

BDD are often considered as a 0/1 technology: either the construction of the BDD is feasible and the technique is efficient, or the computer runs out of memory and nothing can be done. However, approximations are possible through the notion of truncated BDD [Rau01]. Truncated BDD have the nice property that the minimal cutsets of a BDD truncated at order k, are the minimal cutsets of order k or less of the original formula. By means of truncated BDD we were able to extract minimal cutsets of the largest model at hand up to the order 4, which was sufficient for practical purposes.

The contribution of this article is manifold. First, we report successful experiments on very large industrial models. Second, we recall the principle of the compilation of the AltaRica

language into Fault Trees. This principle applies actually to any states/events formalism, beyond the specific case of AltaRica. We propose here an improvement on the compilation of algorithm of reference [Rau02]. Third, we discuss the notion of truncated BDD. Fourth, we describe efficient implementations for several formula rewritings and we suggest a strategy to combine them. Last, we show how to tune a BDD engine to handle very large models.

The remainder of this document is organized as follows. Section 2 presents the AltaRica language and discusses it compilation into fault trees. Section 3 recalls basics about (truncated) BDD and ZBDD and describes preprocessing algorithms and strategies. Section 4 reports experimental results. Finally, section 5 concludes the article.

2. The AltaRica Language and its compilation into Fault Trees

2.1. Informal presentation of the AltaRica language

AltaRica is a high level formal description language dedicated to reliability and dependability studies. It has been used in different industrial contexts (see e.g. [BRDS04 and BBCH04]). It relies on the notion of mode automata [MR98, Rau02]. Mode automata are input/output state machines. They generalize formalisms like finite state machines (FSM), Mealy machines, Moore machines, Petri nets... AltaRica can be seen just as a convenient way to describe and to combine mode automata.

To illustrate the notion of mode automata, consider the valve pictured Figure 1 (a). Its behavior can be described by the mode automaton Figure 1 (b). The state of the valve is described by means of two Boolean variables “open” and “stuck”. State variables play a role similar to places of Petri nets. In our example they are Boolean. They could be integral, real, or in some enumerated sets. Initially, the mode automaton is for instance in the state “open=false, stuck=false”. It may change of state when an event occurs. In our example, there are three possible events: “close”, “open” and “fail”. Input and output flows are also described by means of variables. In our example, there is a single input flow “inFlow” and a single output flow “outFlow”. The values of output flows are fully determined by the values of input flows and state variables, as illustrated Figure 1. Note that in our example both flow variables belong to the enumerated set “{null, low, high}”. The AltaRica code for this mode automaton is given Figure 2.

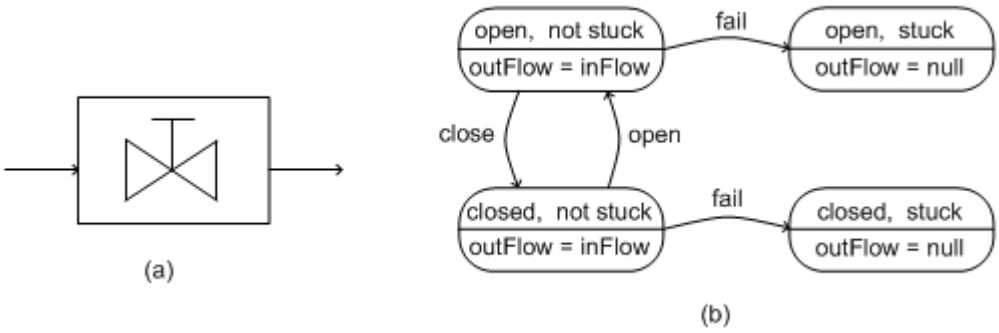

Figure 1. A valve and the mode automaton that describes it.

```
node Valve
  state open:bool; stuck:bool;
  flow  inFlow:{null,low,high}:in; outFlow:{null,low,high}:out;
  event close, open, fail;
  init
    open:=false, stuck:=false;
  trans
    open and not stuck |- close -> open:=false;
    not open and not stuck |- open -> open:=true;
    not stuck |- fail -> stuck:=true;
  assert
    outFlow = if (open) then inFlow else null;
edon
```

Figure 2. The AltaRica code for the valve pictured Figure 1.

Composition operations (free product, connection and synchronization) are defined on Mode Automata. These operations make it possible to assemble them into hierarchies. An AltaRica model is thus a set of nodes organized in a tree like manner – nodes can embed sub-nodes – rooted by a node main. A node can be seen as a reusable modeling component. Several instance of a node may appear in the model. Any such hierarchical description can be flattened into a single mode automaton by means of simple syntactical operations (see for instance [Rau02] for more details). Note that AltaRica is close with that respect to reactive languages such as Lustre [Hal93]. Note also that a complete model is assumed to have no input variables. All input variables of components should actually either set to constant values or connected to output variables of other components.

2.2. Principle of the compilation

A mode automaton describes implicitly a reachability graph, i.e. the graph of states that are reachable from the initial state. From now, reachability graphs we'll consider are assumed to be finite. Some of the reachable states are failure states, the others are working states. The sequence of events that label each path from the initial state to a failed state is a scenario of failure. If we interpret this sequence as a conjunction of events and we consider the disjunction over all failure paths of the corresponding conjunctions, then we get a sum-of-products that encodes all the cutsets of the model. The reachability graph can be actually considered as a reliability network [Shi91]. In practice, the implementation of this idea raises a number of difficulties that we shall examine now.

2.3. Problems and solutions

The first difficulty to consider is indeed the exponential blow up of the number of states in the reachability graph. Consider for instance a system made of n independent binary components in parallel. The reachability graph for this system has 2^n nodes while a hand made fault tree would be just an AND gate of the n basic events. To circumvent this problem, the idea is to detect independent parts in the AltaRica model and to compute separately a reachability graph for each of them. Models we had to deal with are actually decomposable into not too large pieces.

The second difficulty lies in the fact that, in an AltaRica model, states in general and failure states in particular are defined by means of variables and formulae and equations built over these variables. Since AltaRica variables are not necessarily Boolean, we need to create Boolean variables out of them. The idea is therefore to create a Boolean variable X_v for each AltaRica variable X and each value v of the domain of X.

The values of state variables depend on the state of the reachability graph in the following way. A Boolean variable S_i is created for each state i of the reachability graph. Then, for each state variable X and each value v of the domain of X, the Boolean variable X_v is defined as the disjunction of the S_i's such that X takes the value v in the state i.

The values of output flows are defined by means of equations, as illustrated Figure 2. For each assignment of the variables of the left hand side, the equation calculates the value of the right hand side variable. The AltaRica equation can therefore be translated into a set of Boolean equations. Consider for instance the variable "outFlow" of our example. The translation is as follows.

AltaRica equation: outFlow = if open the inFlow else null

Boolean equations: outFlow_null = (not open) or (open and inFlow_null)

outFlow_low = open and inFlow_low

outFlow_high = open and inFlow_high

The same idea can be applied to predicates that define failure states. So, predicates are defined in terms of (Boolean translation) of state variables and output flows, output flows defined in terms of state variables and finally state variables are defined in terms of states of the reachability graph. It remains to define the latter's in terms of events to complete the compilation process.

A first idea is to associate to each state of the reachability graph the disjunction over the paths coming from the initial state of the conjunctions of events that label the path (just as we suggested for failure states). This idea rises however a third subtle difficulty which is illustrated Figure 3. This model is made of two independent components. The first one has two exclusive failure modes, "a" and "b". The second one has only one failure mode, "c". Applying the above principle, we get the following equations (for the sake of the simplicity we give only relevant equations).

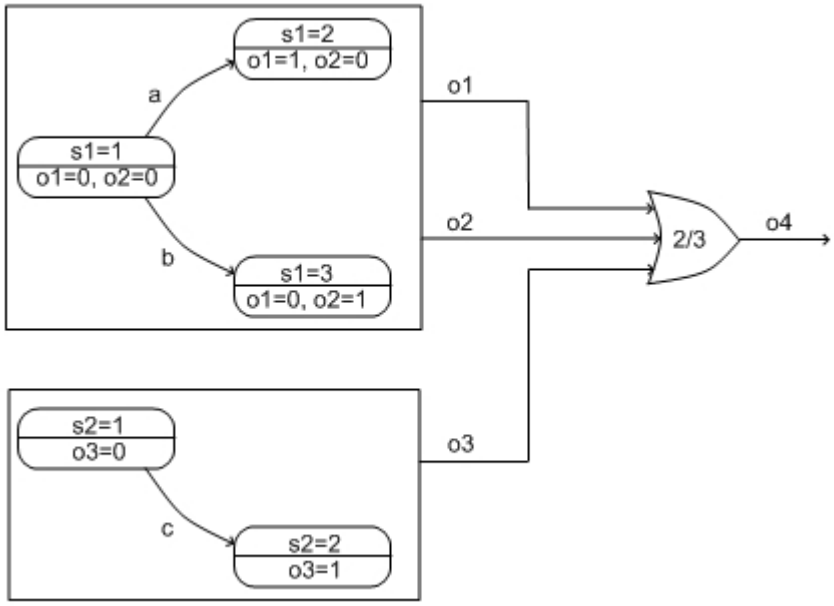

Figure 3. A mode automaton that causes non coherency of fault trees

o4_1 = (o1_1 and o2_1) or (o1_1 and o3_1) or (o2_1 and o3_1)

o1_1 = s1_2 o2_1 = s1_3 o3_1 = s2_2

s1_2 = a s1_3 = b s2_2 = c

It is easy to verify that the minimal cutsets of "o4_1" are "ab", "ac" and "bc". However, "ab" is not possible for "a" and "b" are exclusive events. In order to tackle this problem the only (but sufficient) solution consists in introducing negations. The formula associated with each path is the conjunction of the events that label the path and of the negation of events that do not label the path. In our example, "s1_2" and "s1_3" are redefined as follows.

s1_2 = a and not b s1_3 = not a and b

The compilation algorithm we just sketched (and that has been first proposed in reference [Rau02]) works well for models that are close to a block diagram representation. Many real-life models are actually conceptually simple, nevertheless hard to assess because of their sizes. The models we had at hand fall typically into that category. Their compilation never takes more than few minutes, although they are made of hundred of components and thousands of variables. Note that the algorithm can generate fault trees for several failure conditions at once, which is of great interest in practice.

2.4. Treatment of Sums-of-Products

The compilation algorithm generates many sums-of-products: for states of the reachability graph, for state variables, for flow variables and for predicates. For the largest models, these sums-of-products involve many terms (up to several hundreds) and cause some troubles to BDD algorithms (especially to the variable ordering heuristics). A substantial improvement in the treatment was obtained by factorizing these sum-of-products at the compilation level according to an idea of reference [RCDB03]. Given a sum-of-products S, we select the variable x with the maximum number of occurrences and split S into three subsets as follows.

S = (x and S1) or (not x and S2) or S3

Where S1, S2 and S3 are sums-of-products made respectively with the products that contain x, the products that contain not x, and finally the products that contain neither x nor not x. The same decomposition is applied recursively on S1, S2 and S3.

3. BDD and ZBDD

3.1. BDD

Binary Decision Diagrams are a compact encoding of the truth tables of Boolean formulae [Bry86, BRB90]. The BDD representation is based on the Shannon decomposition: Let *F* be a Boolean function that depends on the variable *v*, then the following equality holds.

$$F = v.F[v \leftarrow 1] + \overline{v}.F[v \leftarrow 0]$$

By choosing a total order over the variables and applying recursively the Shannon decomposition, the truth table of any formula can be graphically represented as a binary tree. The nodes are labeled with variables and have two outedges (a *then*-outedge, pointing to the node that encodes F[v←1], and an *else*-outedge, pointing to the node that encodes F[v←0]). The leaves are labeled with either 0 or 1. The value of the formula for a given variable assignment is obtained by descending along the corresponding branch of the tree. The Shannon tree for the formula $F = ab + \overline{a}c$ and the lexicographic order is pictured Figure 4 (dashed lines represent *else*-outedges).

Indeed such a representation is very space consuming. It is however possible to shrink it by means of the following two reduction rules.

– Isomorphic sub-trees merging. Since two isomorphic sub-trees encode the same formula, at least one is useless.
– Useless nodes deletion. A node with two equal sons is useless since it is equivalent to its son ($F = v.F + \bar{v}.F$).

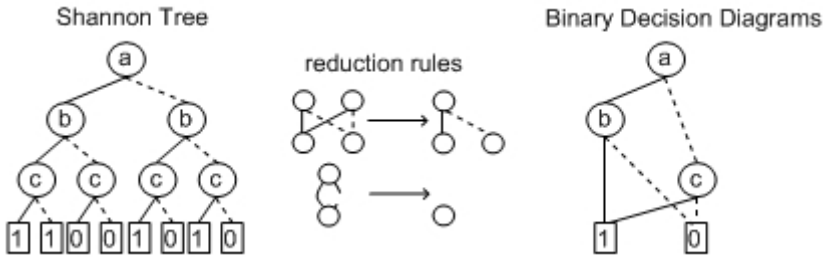

Figure 4. From the Shannon Tree to the BDD

By applying these two rules as far as possible, one gets the BDD associated with the formula. A BDD is therefore a directed acyclic graph. It is unique, up to an isomorphism. This process is illustrated Figure 4.

3.2. Logical operations

Logical operations (and, or, not, ...) can be directly performed on BDD. This results from the orthogonality of the Shannon decomposition with usual connectives:

$$(v.F_1 + \bar{v}.F_0) \oplus (v.G_1 + \bar{v}.G_0) \quad = \quad v.(F_1 \oplus G_1) + \bar{v}.(F_0 \oplus G_0)$$

Where $\oplus$ stands for any binary connective.

Among other consequences, this means that the complete binary tree is never built and then shrunk: the BDD encoding a formula is obtained by composing the BDD encoding its sub-formulae. Hash tables are used to store nodes and to warranty, by construction, that each node represents a different function. Moreover, a caching principle is used to store intermediate results of computations. This makes the usual logical operations (conjunction, disjunction) polynomial in the sizes of their operands. The complete implementation of a BDD package is described in reference [BRB90].

3.3. ZBDD and Minimal Custets

Minato's Zero-Suppressed Binary Decision Diagrams are BDD with a different semantics for nodes (and slightly different reduction rules) [Min96]. They are used to encode sets of minimal cutsets (MCS for short) and prime implicants (PI for short). Nodes are labeled with literals (and not just by variables). Let p be a literal and U be a set of products. By abuse, we denote p.U the set $\{\{p\} \cup \pi; \pi \in U\}$. The semantics of ZBDD is as follows.

- The leaf 0 encodes the empty set: $Set[0] = \varnothing$.
- The leaf 1 encodes the set that contains only the empty product: $Set[1] = \{\{\}\}$.
- A node $\Delta(p,S1,S0)$, where p is a literal and S1 and S0 are two ZBDD encodes the following set of products.

  $Set[\Delta(p,S1,S0)] = p.Set[S1] \cup Set[S0]$

It is possible to encode huge sets of MCS or PI with relatively small ZBDD, provided these MCS or PI have some regularities that can be captured by the sharing of nodes. In the sequel, we denote MCS[F] the MCS of a formula F (respectively PI[F] its PI). Moreover, we denote $MCS_k[F]$ the MCS of F whose order is at most k. The algorithm to compute MCS[F], $MCS_k[F]$, PI[F] or $PI_k[F]$ from the BDD that encodes F is discussed in reference [Rau01].

3.4. Truncated BDD

Large Boolean models have in general many MCS. Often, there are so many MCS that an individual inspection of each of them would be much too time consuming. However, not all MCS are equally likely to occur. Cutoffs on order or equivalently on probability can be applied to keep only MCS that contribute significantly to the risk. As an approximation, MCS of order (number of variables) more than k, with k=4 or 5, can be discarded without much consequences.

The calculation of the ZBDD that encodes $MCS_k[F]$ from a fault tree can be performed in three steps: first, the BDD that encodes F is computed, then the ZBDD that encodes MCS[F] is computed, last this ZBDD is pruned to keep only cutsets of order k or less. The algorithm to compute MCS can be changed however to calculate the ZBDD that encodes $MCS_k[F]$ straight from the BDD that encodes F [Rau01]. This shortcut saves in most of the cases significant amounts of time and space because the ZBDD that encodes $MCS_k[F]$ (with k = 4,

5 or 6) is much smaller that the ZBDD that encodes all MCS. Both processes are illustrated Figure 5. Note that the same remark applies to ZBDD that encode PI[F] and PI_k[F].

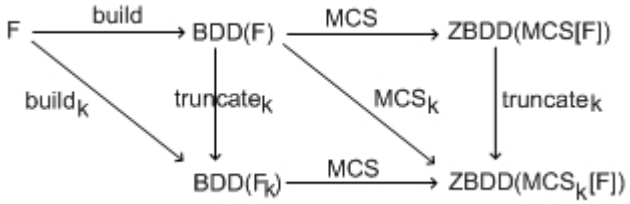

Figure 5. Computations of (truncated) BDD and ZBDD

Now, the process can be further improved (only for MCS). The idea is to consider the intersection of F and minterms$_k$ the set of all minterms that contain less than k positive literals. Recall that a minterm is a product of literals that contains one and only one literal for each variable. Any Boolean function can be seen as a set of minterms. We denote by F_k the function F $\cap$ minterms$_k$. The BDD that encodes F_k can be obtained from the BDD that encodes k by pruning the products or order more that k (in the same way the ZBDD can be pruned). It can also be obtained directly from F by tuning the construction algorithm, as illustrated Figure 5. The fundamental result here is twofold: first, MCS[F_k] = MCS$_k$[F]. Second, the latter computation process is of polynomial complexity while all the others are of exponential worst case complexity [Rau01]. In practice, it saves lots of time and space and makes feasible calculations that would not be tractable otherwise.

3.5. Preprocessing techniques (rewritings)

This section reviews preprocessing techniques we used and discusses their implementation. Fault trees are described by means of sets of Boolean equations in the form g = op(e_1,…, e_k), where g is a gate, op is a Boolean connective (and, or, not, k/n) and the e_i's are either gates or basic events. Such a set of equations contains no loop but gates and basic events can occur more than once. The set is thus encoded as a directed acyclic graph (DAG). Due to space limitation, we shall describe here only the techniques that have proved to be of some interest to handle the models under study. We tried actually many others.

Constant propagation: A fault tree may contain Boolean constants (*true* or *false*). To propagate these constants simplifies the formula, e.g. (*true or F*) reduces to *true*, (*false or F*) reduces to *F*, and so on. A very simple procedure implements constant propagation: to simplify a gate, we first simplify its children, and then simplify the gate itself. The process starts from the top event.

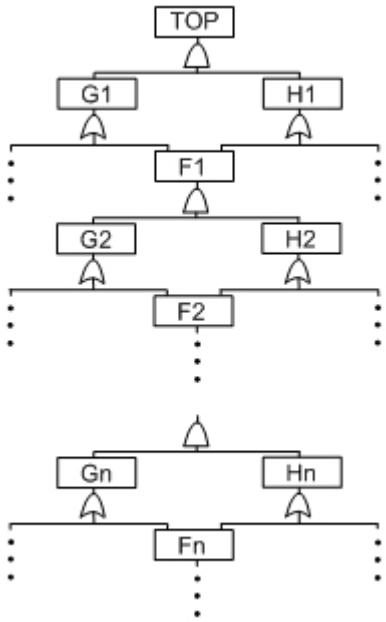

Figure 6. A fault tree with 2^n paths from TOP to Fn.

During the traversal of the tree, the gates that are already simplified must be flagged. In that way, the simplification is linear in the size of the tree. Without flags, it may be exponential as illustrated Figure 1. The number of paths from TOP to Fn, hence the number of recursive calls to the procedure, is 2^n.

Dereferenciation: Another, usually small, simplification can be obtained by deferencing equations, i.e. by replacing a couple of equations *G1 = G2, G2 = G3* by *G1 = G3*. The implementation of this simplification is very similar to the one of constant propagation and is linear in the size of the tree as well.

Isomorphic gate merging: Automatically generated fault trees often embed isomorphic gates, i.e. with the same connective and same list of children. To check the existence a couple of such gates would take a lot of time with a brute force algorithm. For instance, an algorithm

that sorts gates before comparing them would be inefficient because to sort a list with dozens thousands objects takes time, even if sort algorithms are in O(n.log n). The first version of our algorithm, based this sorting idea, took more than twenty minutes on the large example under study in section 4. To make this simplification efficient, we had to use (large) hash tables to store gates (two isomorphic gates have the same hash code) and running times reduced to few seconds.

Greedy Saturation of Sum-Of-Products: As explained in section 2.4, the compilation algorithm creates many sums of products and products of sums. The idea of the next simplification is to saturate these sums of products, by applying the consensus rule: $F.G + -F.H$ gives $G.H$. This process may produce a lot of new products. Therefore, it is applied in a greedy way: the consensus rule is applied only when either $G \subseteq H$ or $H \subseteq G$. In that way, the initial formula is simplified at each step of the algorithm, which is linear in the size of the formula. This rewriting improves usually a bit the performance, but not that much (and even less useful when the compilation heuristics of section 2.4 is applied). It should be noticed however that it is of interest for variable ordering heuristics. These heuristics are actually based on topological and counting considerations. So, they could be possibly misled by too much gate duplications.

Factorization of common factors: The fault trees generated by the compiler may contain products of sums or sums of products that can be factored ($F.G + F.H$ gives $F.(G + H)$). Such a factorization is important for at least two reasons: it makes a bit easier the BDD calculation and it improves the accuracy of most of the variable ordering heuristics. This simplification is easy to implement. However, most of the time is spent in useless attempts.

Other rewritings: Other rewritings are proposed in the literature, like coalescing, propagation of variables (e.g. $x + F(x)$ reduces to $x + F(0)$) or modularization (see e.g. [CY85, Nie94 and RA02]). None of these techniques were successful on the models we had to deal with (which indeed does not mean that they are useless).

3.6. Variable Ordering Heuristics

It is known, since the very first uses of BDD, that the chosen variable ordering has a great impact on the size of BDD, and therefore on the efficiency of the whole methodology [Bry86]. Finding the best ordering, or even a reasonably good one, is a hard problem (see e.g. [FS90, BW96]). Two kinds of heuristics are used to determine which variable ordering to apply. Static heuristics are based on topological considerations and select the variable ordering once for all (see e.g. [AMS03, FFK88 and MIY90]). Dynamic heuristics change the variable ordering at some points of the computation (see e.g. [Rud93 and PS95]). They are thus more versatile than the formers, but the price to pay is a serious increase of running times. Sifting is the most widely used dynamic heuristics [Rud93].

Static variable ordering heuristics: The most simple, and often one of the best heuristics, consists in numbering basic events by means of a depth first left most (DFLM for short) traversal of the formula. DFLM heuristics is "good" because it preserves, at least to a certain extent, the locality of basic events. For instance, it numbers basic of events of modules consecutively. Static variable ordering heuristics fall into two categories: those consists in a rearrangement of fanins of gates, followed by the DFLM numbering (see e.g. references [FFK88, MIY90]) and those that produce interleaved orders (see e.g. references [AMS93, FOH93]). Heuristics of the first category can bee seen as pre-processing of the formula. For instance, the heuristics proposed in reference [MIY90] consists in associating a weight with each variable (or more exactly literal) as follows.

- The weight of input variables and constants is 1.
- The weight of a gate equals the sum of the weights of its fanins.

Then, fanins of gates (and, or, k/n) are sorted by increasing values of their weights. We tried systematically half dozens such heuristics that were proposed in the literature. None of them gives better perfomance than the simple DFLM heuristics. The shuffling strategy [BBR97, DPRT00] has been designed in order to measure the potential of this category of heuristics. It gave excellent results on Fault Trees/Event Trees of the nuclear industry. It works as follows. An amount of resource (time or space) is chosen. Then, for a given number of tries, gates' fanins are shuffled at random and DFLM heuristics is applied. If at least one of shuffled formulae is computable, then the process is stopped. Otherwise, a new round is performed, with more resources. In our implementation, fanins are not shuffled at complete random.

Rather, gates are numbered at random then fanins are sorted according to that numbering. This algorithm has proved to be much better than the purely random one. We shall see in the next section the results of this strategy on the models under study.

The second category of heuristics produce interleaved orders. Heuristics proposed in the two references [AMS03, FOH93] aim to reduce the "distance" between gates that are related in the formula, typically a gate and its fanin. To this purpose, basic events and gates are numbered and the formula is rearranged in such way the sum of distances between related gates is minimized. This distance is called a linear arrangement of the underlying graph. This idea is seducing. However, it relies on the assumption that the size of BDD is related to the linear arrangement, which is not proved so far, even experimentally. Moreover, the problem of finding an optimal linear arrangement is known to be NP-complete, and even hard amongst NP-complete problems [Hor97]. Anyway, both heuristics gave poor results on the models at hand.

Dynamic variable ordering heuristics: In the circuit analysis framework (from where BDD are issued), static variable ordering heuristics have been replaced by dynamic reordering [Rud93, RBKM91]. The idea of the dynamic reordering is to change the variable order while computing the BDD. Hence, the variable order is continuously optimized. The most popular (by far) reordering strategy is called sifting [Rud93]. It consists in examining the variables in turn. Each variable is "sifted" up and down in the variable order and the best location is chosen on the way. It is well known that dynamic reordering is very time consuming. Nevertheless, in the framework of circuit analysis it made it possible to handle formulae that were not tractable otherwise. However formulae that represent circuits are very different than those issued from reliability models. The latter have much more variables and are much less compact. So the dynamic variable ordering has never proved useful for them, except as a post processing (to reduce the size of the BDD of the top event).

4. Experimental Results

One of the AltaRica models we had to handle, and that will serve here as a test case, is made of 884 state variables, 20 571 flow variables, 1 008 events and 1 010 transitions. Up to the section 4.4, all the experiments reported here were performed with our first compilation algorithm (that produces an explicit representation of Sums-of-Products). Unless otherwise

specified, the remarks apply anyway to both algorithms. For each operation, we shall give its running time. These running times were obtained of a PC with a Intel Pentium 4 processor cadenced at 3.4GHz and with 2,00 Go of RAM. They should be taken with care: we observed that, if the computer runs other programs than our fault tree processing tool, even a simple Web browser, it may slow down the process significantly. The fault tree for this model has been generated in less than five minutes. It is made of 85 788 gates and 918 basic events (it occupies 13 362Ko).

4.1. Preliminary simplifications

The first simplification we applied on the tree was constant propagation. This simple rewriting reduces the tree to 57 553 gates and 918 basic events (7 115Ko), hence achieving a substantial improvement. The process takes 3.20 seconds.

The second simplification we applied was dereferenciation. It reduces the fault tree to 56 982 gates and indeed still 918 basic events (7 035Ko). The simplification process takes 5.28 seconds.

The third simplification we applied was isomorphic gates merging. It reduces the fault tree to 47 324 gates and still 918 basic events (5 882Ko). The simplification process takes 2.54 seconds. The improvement is again substantial. Note that dereferenciation makes its more efficient in terms of both simplification and running times.

After these three simplifications, we were able to compute the truncated BDD (and the corresponding ZBDD) up to the order 3. We used a depth first left most ordering of basic events. This basic heuristics gives good results in many cases for it preserves the “locality” of variables (see e.g. [BBR97] for a discussion on that issue). Table 1 presents some statistics about these computations. The last row gives the number of BDD nodes involved in the computation (without any garbage collection process). A BDD node is encoded within 4 machine words (16 bytes), so the storage of 27 millions of nodes occupies about a half Giga byte. Since BDD are truncated, possibly not all the basic events show up MCS. It is therefore of interest to know which part of the model is actually considered. The fifth row gives the number of basic events that show up for the different orders. Even at order 3, a third of the basic events vanish. This is noticeable, although not necessarily significant from a probabilistic viewpoint.

Table 1. Some statistics after preliminary simplifications

order	1	2	3	all
Sizes of the BDD	240	17 398	671 914	?
Number of cutsets	6	275	10 470	?
Sizes of the ZBDD	7	273	2 668	?
Basic events involved in MCS	6	196	634	918
Number of BDD nodes	588 643	3 853 202	27 014 557	?
Running times	1.67	8.67	79	?

The fourth and fifth simplifications we applied were respectively greedy saturation of sums-of-products and the factorization of common factors. They take respectively 2.18 seconds and 10.01 seconds (all running times throughout this section are given in seconds). The first one reduces the fault tree to 43 283 gates and still 918 basic events (5 501Ko), the second one increases slightly the size of the formula (45 149 gate, 918 and 5 633Ko). Sizes of BDD and ZBDD after these rewritings are given Table 2. The fourth row gives the numbers of BDD nodes created during the computation. The fifth row gives the numbers of nodes that remain after a garbage collection process at the end of the computation. The former gives an idea of how many nodes are necessary to perform the computation. The latter tells how many nodes are required to store the BDD and the ZBDD. Combined together, these rewritings save almost three millions BDD nodes during the computation. On the other hand, they take significant amounts of time. They are however certainly worth applying.

Table 2. Some statistics after more advanced rewritings

order	1	2	3
Sizes of the BDD	234	16 948	654 721
Sizes of the ZBDD	7	273	2 668
Number of BDD nodes before GC	521 052	3 509 320	24 308 317
Number of BDD nodes after GC	277 437	1 959 728	14 150 568
Running times	1.54	7.78	56.26

4.2. Variable ordering heuristics

As said in section 3.6, variable ordering heuristics gave quite deceptive results on the fault trees under study. Table 3 reports the results we obtained over 1000 tries with the shuffling strategy (without any limitation of time or space for out objective was just to measure the potential of heuristics). For each order and each quantity of interest, it gives the minimum, the maximum and the mean over the tries of the ratio between the value of the try and the value for the original formula (as given Table 2). Therefore, the cell at the second row and third column should be read as follows. Table 3 gives hints that not much can be expected from static variable ordering heuristics. Except for what concerns the size of the truncated BDD of the top event, none of the tries improved significantly the performance.

Table 3. Statistics about the shuffling strategy.

order		size BDD	size ZBDD	#nodes (1)	#nodes (2)	running times
1	min	0.833	1	1	1	1
	max	1.11	1	1.59	1.46	1.27
	mean	0.981	1	1.39	1.21	1.18
2	min	0.446	0.912	1	0.955	1
	max	1.58	1.02	1.56	1.39	1.49
	mean	0.968	0.974	1.32	1.16	1.3
3	min	0.285	0.821	1	1	1
	max	1.6	1.06	2.24	1.9	2.85
	mean	0.924	0.96	1.54	1.3	1.86

In order to make a complete study, we apply sifting as a post-processing to the formula at hand (with the parameters suggested by reference [Rud93]). Table 4 gives the results we obtained. A comparison of these results with those of Table 3 makes clear that dynamic reordering is awfully time consuming and provides no improvement. To apply sifting actually while constructing the BDD would lead to even more time consumption and certainly no improvement.

Table 4. Statistics about sifting.

order	1	2	3
sizes of BDD	240	16 986	654 825
sizes of ZBDD	7	273	2 668
#BDD nodes	241 993	1 938 916	14 081 623
Running times	10.18	290	2 605

4.3. Tuning BDD Tables

The BDD technology involves several tables. The most important are the hash tables associated with each variable (and that give access to the BDD nodes labeled with the variable), and the hash cache in which intermediate results are stored. The size of the hash cache is usually set once for all. The sizes of the hash tables are preferably set dynamically because some of the variables have very few BDD nodes associated with, while others have a lot. We made the following experiment. We set the size of the hash cache to 2 097 143 (a prime number close to 2^{21}). The minimum size of the hash tables to 1 023 (2^{10}-1) and their maximum sizes to various values (of the form 2^n-1). Table 5 gives the running times in seconds we observed (for the whole computation BDD + ZBDD, the computation of the ZBDD taking always less than 1 second). This experiment shows that the maximum size should be chosen large enough (which means that some of the variables have very big numbers of BDD nodes associated with).

Table 5. Computation times with different maximum size of hash tables.

	order	1	2	3
maximum size	1023	2.68	41.49	784
	2047	1.28	11.09	167
	16 383	1.59	8.34	74
	65 535	1.67	8.51	65
	131 071	1.93	9.48	63

We then fixed the maximum size of hash tables to 65 535 and made the size of the hash cache varying. Table 6 gives the corresponding running times. This experiment shows that the hash cache must be chosen big (typically close to 2^{21}). But beyond a certain size, no improvement is to be expected by enlarging it. It is worth to notice, that the picture is a bit different if isomorphic gates are not merged. It that case, the hash cache must be chosen larger to achieve the same performance, for the same computations are done several times.

Table 6. Computation times with different size of the hash cache.

	order	1	2	3
hash cache size	524 287	1.67	8.67	79
	1 048 573	1.68	8.60	72
	2 097 143	1.67	8.51	65
	4 194 301	1.68	8.67	63

4.4. Cutsets of order 4!

As explained section 2, the compilation produces many Sums-of-Products. Sets of paths are described by Sums-of-Products. Variable valuations are also described by Sums-of-Products. From the viewpoint of BDD treatment, Sums-of-Products raise at least two problems. First, they induce lots of intermediate computations. Consider for instance, the formula "A.B + A.C". To build the BDD of this formula, three operations are required. If the formula is rewritten as "A.(B+C)", only two operations are necessary. When dealing with large formulae and large Sums-of-Products, this may causes a significant difference in performance, even we end up with the same BDD. Second, the BDD for intermediate formulae may be much larger than the final BDD. To tackle these problems, we introduced two ideas. The first one consists in rewriting Sums-Of-Products as Decision Trees (as explained section 2.4). The second one consists in "forgetting" the intermediate computations as soon as it is possible: the BDD of a gate is freed when all the BDD of all fanouts of this gate are computed. When the garbage collector is invoked, the room necessary to store these

BDD can be collected (at least partly). Using both ideas we were able to compute MCS up to the order 4. Table 7 gives some statistics about this calculation.

Table 7. Statistics about MCS of order 4

order	4
Size of BDD	10 580 386
Size of ZBDD	16 598
#BDD nodes (1)	63 736 093
#BDD nodes (2)	55 427 566
#calls to the garbage collector	6
Running time	1 505" ≈ 25'

To our knowledge, this is the largest BDD calculation ever reported in the literature. It is worth noticing that the garbage collector has been called six times. The total amount of memory allowed to the calculation was about 1.6 Go. With a larger memory, the number of calls to the garbage collector would have been lesser and the running times certainly much better. With more memory, it would be also interesting to enlarge the hash cache and, although certainly at a lesser extent, the maximum size of hash tables.

5. Conclusion

The experiments we reported here are of interest for several reasons. From an industrial view point, they demonstrate the maturity of the technology. The process by which fault trees are derived from a functional description is automatic, formal and much less error prone than a hand design. The treatment chain is now able to handle very large models (actually the largest models designed so far) with a good efficiency and accuracy. This provides strong arguments in favor of the approval of the methodology by regulation authorities.

More technically, we improved the compilation algorithm by a suitable treatment of sums-of-products. The ideas we suggested here can be applied to any states/events formalism (like Petri nets or state charts). We designed also a strategy to build BDD. This strategy involves efficient implementation of rewriting procedures, fine tuning of BDD tables and appropriate approximation through the notion of truncated BDD. We showed, to our own surprise, the

limited usefulness of variable ordering heuristics. No doubt that the ideas we developed here can be reused in other contexts.

It remains certainly room for improvement. Our feeling is that it would be worth exploring at least two ideas. First, the variable ordering of decision trees that handle sums-of-products is probably not optimal. Second, there may be a way to prune the AltaRica description before the compilation. It is also clear that with more RAM, the performance of the whole process would improve significantly. However, it is not possible to get more RAM than we used on 32 bits architecture: we used 1.6 Go while at most 2 Go can be addressed (3 Go under certain conditions). With that respect, the development of BDD engines for 64 bits architectures opens new perspectives, as shown by preliminary experiments. Nevertheless, the improvement of algorithms and heuristics will remain the key issue.

Another key issue for the future of the technology we discussed in this article is the treatment of systems with loops (like electric circuits). The current version of AltaRica does not allow equations that define flow variables to contain loops. To be able to deal with looped system would be a great improvement.

6. References

[AMS03] F.A. Aloul, I.L. Markov and K.A. Sakallah, FORCE: A Fast and Easy-To-Implement Variable-Ordering Heuristic, *Proceedings of GLVLSI'03*, 2003

[AGPR00] A. Arnold, A. Griffault, G. Point, and A. Rauzy. The altarica language and its semantics. *Fundamenta Informaticae*, 34:109–124, 2000.

[BBCH04] P. Bieber, Ch. Bougnol, Ch. Castel, J. P. Heckmann, Ch. Kehren, S. Metge, and Ch. Seguin. Safety assessment with altarica - lessons learnt based on two aircraft system studies. In *18th IFIP World Computer Congress, Topical Day on New Methods for Avionics Certification*, Toulouse France, 26 - 26 August 2004. IFIP.

[BRDS04] M. Boiteau, Y. Dutuit, A. Rauzy and J.-P. Signoret, The AltaRica Data-Flow Language in Use: Assessment of Production Availability of a MultiStates System, *Reliability Engineering and System Safety*, Vol. 91, pp 747-755, Elsevier.

[BW96] B. Bollig and I. Wegener. Improving the Variable Ordering of OBDDs is NP-Complete. *IEEE Trans. on Software Engineering*, 45(9):993–1001, September 1996.

[BBR97] M. Bouissou, F. Bruyère, and A. Rauzy. BDD based Fault-Tree Processing: A Comparison of Variable Ordering Heuristics. In C. Guedes Soares, editor, *Proceedings of European Safety and Reliability Association Conference, ESREL'97*, volume 3, pages 2045–2052. Pergamon, 1997. ISBN 0-08-042835-5.

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

[BRB90] K. Brace, R. Rudell, and R. Bryant. Efficient Implementation of a BDD Package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 40–45. IEEE 0738, 1990.

[BRKM91] K.M. Butler, D.E. Ross, R. Kapur, and M.R. Mercer. Heuristics to Compute Variable Orderings for Efficient Manipulation of Ordered BDDs. In *Proceedings of the 28th Design Automation Conference, DAC'91*, June 1991. San Francisco, California.

[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*, Volume 11, pp 93–108, 1985.

[CM93] 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.

[DPRT00] F. Ducamp, S. Planchon, A. Rauzy, and P. Thomas. Handling very large Event Trees by means of Binary Decision Diagrams. In S. Kondo and K. Furuta, editors, in proceedings of the *International Conference on Probabilistic Safety Assessment and Management, PSAM'5*, pages 1447–1452. Universal Academy Press, 2000. ISBN 4-946443-64-9.

[ER05] S. Epstein and A. Rauzy, Can We Trust PRA ?, *Reliability Engineering and System Safety*, Volume 88, Issue 3, pp 195-205, 2005.

[FS90] S.J. Friedman and K.J. Supowit. Finding the Optimal Variable Ordering for Binary Decision Diagrams. *IEEE Transactions on Computers*, 39(5):710–713, May 1990.

[FOH93] H. Fujii, G. Ootomo, and C. Hori. Interleaved Based Variables Ordering Methods for Ordered Binary Decision Diagrams. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 38–41, 1993.

[FFM93] M. Fujita, H. Fujisawa, and Y. Matsugana. Variable Ordering Algorithm for Ordered Binary Decision Diagrams and Their Evalutation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(1):6–12, January 1993.

[FFK88] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams. In *Proceedings of IEEE International Conference on Computer Aided Design, ICCAD'88*, pages 2–5, 1988.

[Hal93] N. Halbwachs. *Synchronous Programming of Reactive System*s. Kluwer Academic Publisher, 1993. ISBN 0-7923-9311-2.

[Hor97] S.B. Horton, *Optimal Linear Arrangement problem: algorithms and approximation*. Thesis, Georgia Institute of Technology, May, 1997.

[MR98] F. Maraninchi and Y. Rémond. Mode-automata: About modes and states for reactive systems. In *European Symposium On Programming*, Lisbon (Portugal), March 1998. Springer verlag.

[Min96] S.-I. Minato. *Binary Decision Diagrams and Applications to VLSI CAD*. Kluwer Academics Publishers, 1996. ISBN 0-7923-9652-9.

[MIY90] S. Minato, N. Ishiura, and S. Yajima. Shared Binary Decision Diagrams with Attributed Edges for Efficient Boolean Function Manipulation. In L.J.M Claesen, editor, *Proceedings of the 27th ACM/IEEE Design Automation Conference, DAC'90*, pages 52–57, June 1990.

[Nie94] I. Niemelä. On simplification of large fault trees. *Reliability Engineering and System Safety*, Volume 44, pp 135–138, 1994.

[PS95] S. Panda and F. Somenzi. Who Are the Variables in Your Neighborhood. In *Proceedings of IEEE International Conference on Computer Aided Design, ICCAD'95*, pages 74–77, 1995.

[Rau01] A. Rauzy. Mathematical Foundation of Minimal Cutsets. *IEEE Transactions on Reliability*, volume 50, number 4, pages 389-396，2001.

[Rau02] A. Rauzy. Mode automata and their compilation into fault trees. *Reliability Engineering and System Safety*, Elsevier, Volume 78, Issue 1, pp 1-12, 2002.

[RCDB03] A. Rauzy, E. Châtelet, Y. Dutuit and C. Bérenguer. A practical comparison of methods to assess sum-of-products. *Reliability Engineering and System Safety*, 79:33-42, 2003.

[RA02] K.A. Reay and J.D. Andrews. A fault tree analysis strategy using binary decision diagrams, R*eliability Engineering and System Safety*. Volume 78, Issue 1, pp 45-56, October 2002.

[Rud93] R. Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In *Proceedings of IEEE International Conference on Computer Aided Design, ICCAD'93*, pages 42–47, November 1993.

[RBKM91] D.E. Ross, K.M. Butler, R. Kapur, and M.R. Mercer. Fast Functional Evaluation of Candidate OBDD Variable Orderings. In *Proceedings of the 2nd ACM/IEEE European Design Automation Conference, ECAD'91*, pages 4–10, February 1991.

[Shi91] D.R. Shier. *Network Reliability and Algebraic Structures*. Oxford Science Publications, 1991.

[SA97b] R.M. Sinnamon and J.D. Andrews. Improved Efficiency in Qualitative Fault Tree Analysis. *Quality and Reliability Engineering International*, 13:293–298, 1997.

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