Analytical Solutions of Large Fault Tree Models using BDD: New Techniques and Applications

Olivier Nusbaumer[a*], Wolfgang Kröger[b], and Enrico Zio[c]
[a]Leibstadt Nuclear Power Plant, Leibstadt, Switzerland
[b]Swiss Federal Institute of Technology, Zürich, Switzerland
[c]Polytechnic of Milan, Milan, Italy

Abstract: Most tools available for quantifying large linked Fault Tree models as used in Probabilistic Safety Assessment (PSA) are unable to produce analytically exact results. The algorithms of such quantifiers are designed to neglect sequences when their likelihood decreases below a predefined truncation limit. In addition, the rare event approximation is typically implemented to the first order, ignoring success paths. In the last decade, new quantification algorithms using the mathematical concept of Binary Decision Diagram (BDD) have been proposed to overcome these deficiencies. Since a BDD analytically encodes Boolean expressions, exact failure probabilities can be deduced without approximation or truncation. However, extended effort is required when converting a given Fault Tree to its BDD form; this turns out to be an optimization problem of NP-complete complexity. Several innovative optimization techniques are developed and investigated as a case study on the fullscope PSA model of the Leibstadt Nuclear Power Plant. We succeeded in converting the Leibstadt PSA model into a BDD with more than 1'500'000 nodes, for a total of 3650 basic events. The BDD covers a complete Event Tree sequence that includes reactor shutdown and cooling with all Emergency Core Cooling Systems and support systems, enabling objective comparisons between quantification tools.

Keywords: PSA, BDD, Optimization, Quantification.

1. INTRODUCTION

The algorithms of quantification tools used for Probabilistic Safety Assessment (PSA) are often designed to neglect sequences when their likelihood decreases below a predefined truncation limit. In addition, the rare event approximation is typically implemented to the first order, ignoring success paths. This is only justified for analyses where the probabilities of basic events are low. When the basic events in question are failures, the first order rare event approximation is always conservative, resulting in wrong interpretation of risk importance measures.

A novel approach using the mathematical concept of Binary Decision Diagram (BDD) has been proposed to overcome these deficiencies [1]. BDDs have some remarkable properties. First, they have a size that is not related to the number of prime implicants of the encoded Boolean expression. Even small BDDs can encode several trillions of prime implicants. Secondly, they provide an analytical representation of Boolean expressions and can be efficiently used for all kinds of logical operations. The extension of this mathematical concept to Fault Tree applications is straightforward.

However, extended effort is required when converting a given Fault Tree to its BDD form; the complexity associated with the conversion can be considerably reduced by optimizing the order of the basic events in the BDD. In order to evaluate optimization techniques in practice, a software tool (*NeuralSpectrum*) was developed as part of this study. The software is an integrated Fault Tree - BDD tool that features a Fault Tree package, a BDD engine and a Minimal Cutset engine, with dedicated Fault Tree to BDD conversion and optimization routines.

Finally, the impact of the different approximations used in the classical approach is evaluated using the Leibstadt PSA model, by comparing approximated results to exact BDD results for different large system Fault Trees.

* olivier.nusbaumer@kkl.ch

2. BINARY DECISION DIAGRAMS

This section attempts to provide a succinct introduction to the mathematical concept of BDD and to practical applications.

2.1. Shannon Expansion

Let $x \rightarrow y_0, y_1$ be the *if-then-else* operator *ite* defined by:

$$x \rightarrow y_0, y_1 = ite(x, y_0, y_1) = (x \wedge y_0) \vee (\bar{x} \wedge y_1)$$

Hence, $x \rightarrow y_0, y_1$ is *true* if x and y_0 are *true*, or if x is *false* and y_1 is *true*. In other words, if x is *true*, the logical state of y_0 is returned, otherwise the logical state of y_1 is returned.

An *If-Then-Else Normal Form* (*INF*) is a Boolean expression built entirely from *ite* operators and the constants *0* (*false*) and *1* (*true*). If we denote *t[0/x]* the Boolean expression obtained by replacing x with *0* in t, then it is not hard to see that the following equivalence holds:

$$x \rightarrow t[1/x], t[0/x] \Leftrightarrow (x \wedge t[1/x]) \vee (\bar{x} \wedge t[0/x])$$

This is known as the Shannon expansion (or Shannon decomposition) of t with respect to x. This simple equation has a lot of useful applications and is the basis for constructing BDDs.

2.2. Binary Decision Diagrams

A Binary Decision Diagram (BDD) is a graph that depicts the Shannon expansion of a Boolean expression. It is a data structure that has been used extensively in the Computer-Aided Design (CAD) community. It is able to represent and manipulate large Boolean expressions very efficiently, e.g. with a minimum of nodes resp. node operations. The effectiveness of BDD in its various applications is very dependent on its variable ordering, as we will see in the next sections.

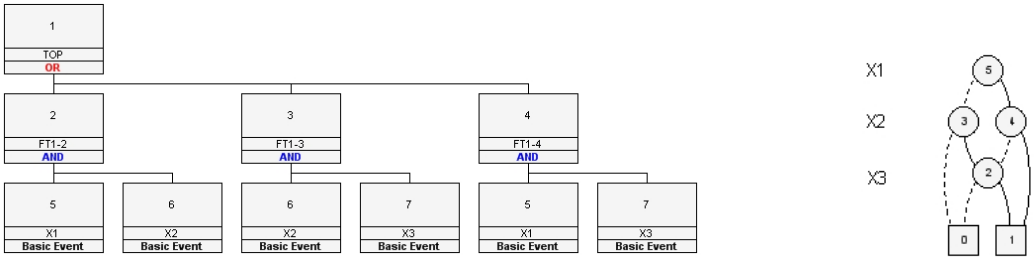
Definition (BDD): A Binary Decision Diagram (BDD) is a rooted Directed Acyclic Graph (DAG) with two terminal leaves labeled *0* (*0-terminal*) and *1* (*1-terminal*), encoding the two corresponding constant functions, and a set of variable nodes u with two outgoing edges, called *low(u)* and *high(u)*, connected to other nodes or leaves.

BDDs have some remarkable properties. First, they have a size that is not related to the number of prime implicants of the encoded Boolean expression. Even small BDD can encode several trillions of prime implicants. Secondly, they provide an analytical representation of Boolean expressions and can be efficiently used for all kinds of logical operations. This is based on the fact that for any function there is exactly one BDD representing it.

Remark: Since a BDD encode Boolean expression in its most compact form, the extension of this mathematical concept to Fault Tree applications is straightforward.

As an example, the Fault Tree and BDD representations for a 2 out of 3 system model given by $(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$ are shown in Picture 1. The dotted branches of the BDD denote the *low* decision branches (variable is *false*), while the solid branches denote the *high* decision branches (variable is *true*). The end states are given by the squared terminal leaves *0* and *1*, the *false* and *true* assignments, respectively. Note that variable x_1, x_2 and x_3 hold 1, 2 and 1 nodes respectively, i.e. *var(5)* = x_1, *var(4)* = *var(3)* = x_2 and *var(2)* = x_3.

Picture 1: Two out of 3 System Fault Tree and its BDD Form



Remark: At this stage, it is important to notice that the BDD yields a unique, complete and compact encoding of the entire truth table corresponding to the Fault Tree. The dotted (*low*) branches of the BDD denote the "*1-p*" terms that play a central role in PSA model quantification when dealing with high probabilities. By following all paths leading to *1-terminal* and summing them up, the analytical top event probability can be directly evaluated as (exact solution):

$$p_{top} = p(x_1) \cdot p(x_2) + p(x_1) \cdot (1 - p(x_2)) \cdot p(x_3) + (1 - p(x_1)) \cdot p(x_2) \cdot p(x_3)$$

This exactness is impossible to be directly derived from the Fault Tree representation, or even from its prime implicants (e.g. Minimal Cutsets). Using Minimal Cutsets, one typically neglects the "*1-p*" terms and thus only gets an upper bound approximation (if the model is coherent).

2.3. Computational Aspects

Since BDDs can have several millions of nodes, it is important to use dynamic memory allocation for efficient computer memory management. We propose to use and implement the concept of *Generics* data objects. *Generics* are powerful C#.Net programming features introduced by Microsoft in 2005. *Generics* allow to define open-ended, type-safe data structures and hashtables, without committing to actual data types, featuring internal garbage collector. This results in significant performance boost and higher quality code.

BDD nodes are typically represented by numbers *0, 1, … , n*, with *0* and *1* reserved for the terminal nodes. The variables in the ordering $x_1 < x_2 < \ldots < x_n$ are represented by their indices 1, 2, … , n. Variable number *0* is reserved for the terminal nodes, e.g. *var(0) = var(1) = 0*. The BDD is stored in a table called the BDD table *T: u → (var, low, high)*, which maps a node *u* to its vertex of three attributes *var(u) = var, low(u) = low* and *high(u) = high*.

In order to ensure that the BDD being constructed is reduced, it is necessary to check from a vertex (*var, low, high*) whether a node *u* with *var(u) = var*, *low(u) = low*, *high(u) = high* already exists or not. For this purpose, we implement a hashtable *H: (var, low, high) → u* mapping node vertex *(var, low, high)* to node *u*. This hashtable is actually the inverse of table *T*. It is often called the unique table of the BDD.

An object oriented class that uses *Generics* is proposed for efficiently encoding a BDD structure. The class is shown in Table 1.

Table 1: BDD Class

Object Oriented BDD Class	Comment
public class BDD {	Node structure
public struct Node {	
public struct Vertex {	Vertex structure
public int var, low, high	Vertex attributes
}	
public Vertex vertex	Vertex
public int ref	Number of nodes pointing
}	to this node (reference counter)
public int top	Top node of the BDD

public List <Node> node Dictionary <Vertex, int> H List <int> varorder }	Dynamic list of nodes (Generic list) Unique table (Generic hashtable) Variable order (Generic list)

The easiest way to build a BDD from a given Fault Tree is to use a brute force Depth-First Left-Most (DFLM) approach. The idea is to construct the BDD starting from the bottom of the Fault Tree and process it upwards. The basic events or gates below a parent gate are processed starting from the left to the right and partial BDDs are assembled sequentially. When no dynamic optimization is used, the resulting variable ordering is referred to as "natural ordering" since variables are numbered in the order they appear. The dynamic algorithm (*Apply*) [2] to logically assemble two BDDs with Boolean operator $\Diamond$ is based on a corollary of the Shannon expansion. It is reproduced in Table 2.

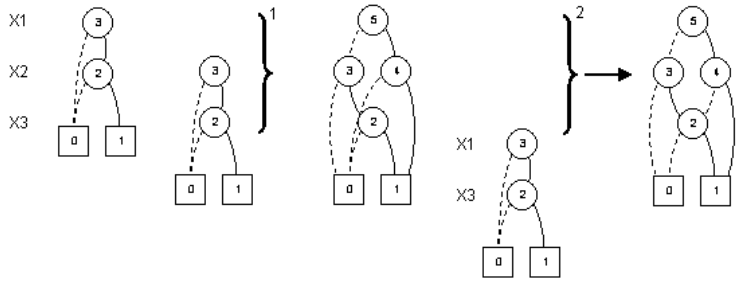
Table 2: Algorithm Apply

Algorithm Apply
Require: u_1 and u_2 the top nodes of the BDDs to assemble
Ensure: The resulting BDD *G*
1: if $G(u_1, u_2) \neq \emptyset$ then 2: return $G(u_1, u_2)$ 3: else if $u_1 \in \{0, 1\}$ and $u_2 \in \{0, 1\}$ then 4: $u = u_1 \Diamond u_2$ 5: else if $var(u_1) = var(u_2)$ then 6: u = new node($var(u_1)$, apply($low(u_1)$, $low(u_2)$), apply($high(u_1)$, $high(u_2)$)) 7: else if $var(u_1) < var(u_2)$ then 8: u = new node($var(u_1)$, apply($low(u_1)$, u_2), apply($high(u_1)$, u_2)) 9: else 10: u = new node($var(u_2)$, apply(u_1, $low(u_2)$), apply(u_1, $high(u_2)$)) 11: end if 12: $G(u_1, u_2) \leftarrow$ u {Add to computation table} 13: return u {Returns node index}

With this in hand, we are in a situation where any Fault Tree model can theoretically be converted into BDD and subsequently be exactly quantified, i.e. without numerical approximation or truncation.

Using the Fault Tree example above, the resulting BDD is computed recursively as shown in Picture 2 (for simplicity, the constructions of the three underlying conjunctive clauses $(x_i \wedge x_j)_{i<j}$ are omitted):

Picture 2: BDD Construction



The two computational steps are thus as follows:

$$\underbrace{\underbrace{(x_1 \wedge x_2) \vee (x_1 \wedge x_3)}_{1} \vee (x_2 \wedge x_3)}_{2}$$

2.4. Variable Ordering and Complexity

The size of a BDD for a given Boolean expression may vary from linear to exponential (e.g. impossible to produce any BDD at all for larger models), depending upon the ordering of the variables chosen; it is therefore of crucial importance to care about variable ordering when using BDDs in practice.

Even worse, the problem of finding the best variable ordering in a BDD has been shown to be of NP-complete complexity [3]. NP-complete problems are the most difficult problems to solve in nature.

In the next section, we develop and study different variable ordering heuristics and dedicated optimization methods for efficient BDD-based Fault Tree analysis.

3. ADVANCED OPTIMIZATION ALGORITHMS

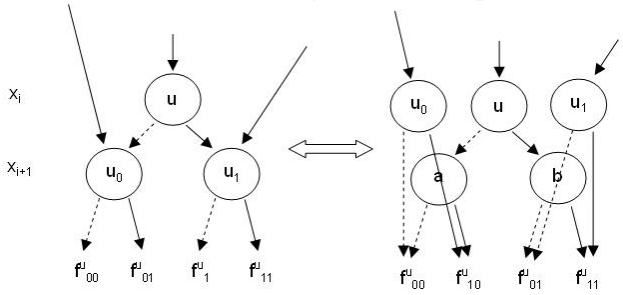
After having briefly described the basics of Fault Tree to BDD conversion, we now turn to more advanced techniques for efficiently converting large PSA model, as employed by the nuclear industry. We briefly discuss some static optimization methods and then turn to more advanced techniques that produced remarkable results.

→ In order to evaluate algorithm effectiveness in practice, the Leibstadt PSA model is used as a test platform. The Leibstadt PSA model is a multistate (fullpower, low power and shutdown) large linked Fault Tree model including internal, external and area events. It consists of about 8000 basic events, 1500 Fault Trees and 300 Event Trees (≈ Initiating Events). It is one of the largest integrated PSA models worldwide.

3.1. Regular Sifting

Sifting [4] is a well known and efficient optimization technique to reduce the size of a BDD. The key concept behind sifting is to sequentially move up and down every variable in the ordering. At each position, the resulting BDD size is recorded and at the end the variable is moved back to its best position, i.e. the one that yields the smallest BDD size. The variable movements are performed by successive local swaps of variables which are adjacent in the ordering.

Picture 3: Single Node Swap

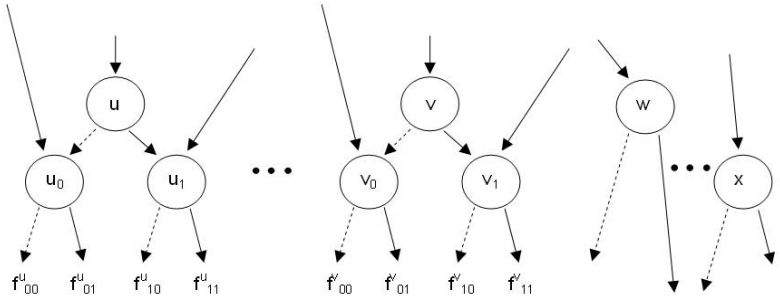


The left side of Picture 3 illustrates the most common case where all cofactors are distinct functions. If some of the cofactors f_{00}, f_{01}, f_{10} or f_{11} are functionally equivalent, the respective nodes are collapsed and thus fewer nodes are involved in the swap. Performing the swap of x_i and x_{i+1}, node u is overwritten with vertex $(x_{i+1}, (x_i, u_{00}, u_{10}), (x_i, u_{01}, u_{11}))$. Possible independent variables on x_{i+1} are translated upwards.

The situation after the swap is illustrated on the right hand side of Picture 3. The reference counters of all nodes involved in the swap are updated accordingly. Node u_0 and u_1 will only vanish if no other external reference besides those from node u existed (e.g. $ref = 0$). New nodes a and b must only be created if nodes with identical vertex do not already exist in the BDD, otherwise they are retrieved

from the unique table *H*. This is a critical issue that becomes relevant when dealing with multiple nodes per layer. In that case, in order to guarantee that the resulting BDD is reduced, the nodes at variable layer x_i that are independent on x_{i+1} must first be identified and moved one variable downwards before swapping other nodes. This ensures that possible equivalent nodes generated during the swaps are not duplicated, but retrieved from the unique table *H*. The general situation is illustrated in Picture 4.

Picture 4: Multiple Nodes Swap



3.2. Static Algorithms (Overview)

A wide range of static optimization algorithms have been developed, implemented and tested as a case study. Those include:

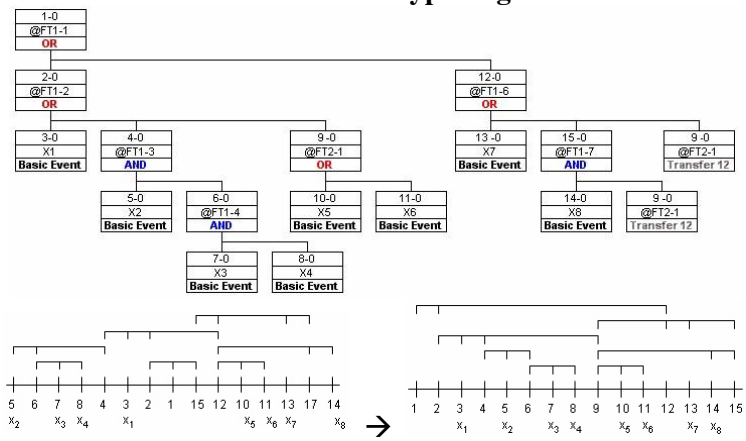
- **Cutset-based Variable Ordering**
  In Fault Tree applications, one could attempt to reduce the size of the resulting BDD by arranging the variables according to the distribution pattern of its Minimal Cutsets. A trivial example is the optimization of the sum of products problem. To optimize this problem, it is easy to identify which variables are connected, and then regroup them accordingly in the ordering. In terms of BDDs, partitioning the problem yields a BDD with a smaller cut, which implies fewer branches and consequently fewer nodes.
- **Coalescing**
  Coalescing a Fault Tree is the action of rewriting its formula so that a gate of a certain type absorbs its children gate(s) if of same type. As long as no transfer gates are involved, this results in AND/OR alternation. By coalescing gates of same type, different variable orders can be obtained by permuting the gate arguments.
- **Occurrence-based ordering**
  This technique consists of sorting the variables according to their occurrence values. The idea is the following: on one hand, it is desired for variables that are related in the formula to appear near to each other in the variable order. On the other hand, variables that are important to the entire formula should appear early in the variable ordering. Different occurrence measures have been investigated.
- **Gate Weight Fanout Permutation**
  The Gate Weight Fanout Permutation technique was first published in [5] for electrical circuit applications. It propagates weights bottom-up in the Fault Tree. A weight *w* is defined as the total number of variables that pertain to a gate. In other words, each basic event is assigned a weight of *1* and the weight of a gate *v* is calculated as the sum of the weights of the children v_i of *v*:

$$w(v) = \begin{cases} 1 \; for \; Basic \; Events \\ \sum_i w(v_i) \; for \; gates \end{cases}$$

  This heuristic can be applied relatively easily to Fault Tree applications. The weights of a transfer gate are passed over to the different transfer locations. For each variable, its children v_i are sorted according to their weights *w*, in increasing or decreasing order, and the Fault Tree can be rewritten. Both sorting orders have been evaluated.

- **Dual Hypergraph Netlength Optimization**
  The Dual Hypergraph Netlength optimization technique is a more sophisticated heuristic. This technique develops an ordering not only for basic events, but also for gates, by trying to group ”connected” variables and gates together. For a given ordering, the netlength is defined as the maximal difference in positions of gates and basic events on the net. Hyperedges are developed according to the gate input and output (Picture 5). The ordering is then optimized using linear optimization algorithms. For this study, the FORCE linear optimization algorithm was chosen [6]. It was shown that the dual hypergraph technique gives better result for Fault Tree by calculating the Center Of Gravity (COG) of hyperedges using the geometric mean instead of the standard arithmetic mean. The non-optimized resp. optimized dual hyperedges are shown in Picture 5.

Picture 5: Fault Tree Hyperedges



- **Dual Hypergraph Gate Weightening**
  The dual hypergraph gate weighting technique was developed as part of this study based on the insights gained by the weighting and the dual hypergraph techniques. The idea is to use linear optimization information to decide in which order gates are going to be processed. In this new approach, the gate placement suggested by the dual hypergraph optimization is retained, while the basic event ordering is processed normally using a DFLM approach. This results in a rewriting of the gate order, exclusively.

All those algorithms have been programmed in our code *NeuralSpectrum* and evaluated in practice. They showed mixed results, with size optimization not exceeding 50%, obviously insufficient for most practical applications on large Fault Trees. Substantially better results were achieved on single safety division Fault Trees (e.g. single trains), but results degraded when applied on more global Fault Trees. Detailed descriptions and benchmarks of the algorithms have to be found in [7].

3.3. Dynamic Implementation of Sifting (DS)

During Fault Tree to BDD conversion, intermediate BDDs can be optimized using dynamic methods on the fly. Dynamic optimization refers to the use of BDD-based optimization methods that take place during Fault Tree processing. The central idea is to perform sub-problem optimization of intermediate BDDs as soon as they become problematic in terms of size and/or complexity. This optimization step results in a new global variable order and thus to smaller BDD sizes for the locally optimized parts. The drawback is that intermediate variable re-arrangements often cause perturbations to other parts of the problem. The central question is to determine objective criteria as to when optimization should be launched or not.

One effective launch criterion is based on the increase in BDD complexity. Sub-problem optimization is performed when the following characteristics are met after a Boolean operation on two BDDs:

1. the ”complexity” increases abnormally after a Boolean operation, and

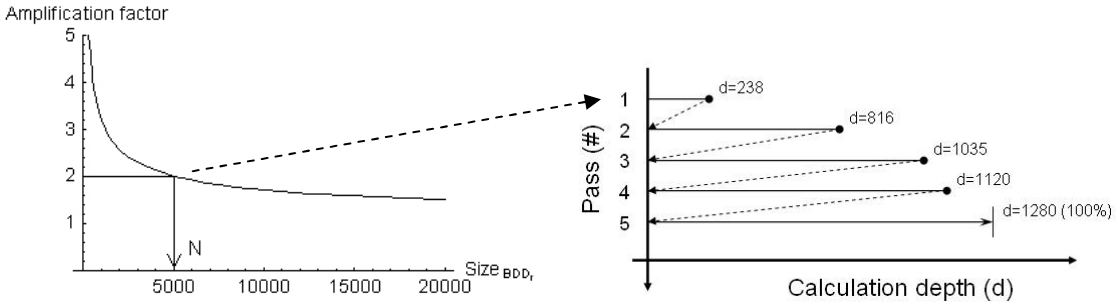
2. the size of the resulting BDD becomes a problem for further calculation.

It remains to answer the question of measuring the increase in complexity for a given operation on two BDDs f_1 and f_2. We introduce an *amplification factor* β that is itself a function of the resulting BDD size. This amplification factor determines how much the resulting BDD size can deviate from the ideal resulting BDD size of $size(f_1)+size(f_2)$. By making this factor a function of the resulting BDD size, we allow the algorithm to be more restrictive when dealing with large BDDs, and less restrictive when the BDD sizes are small. This ensures that optimization is only performed when it is really required, thus limiting perturbation. The proposed launch criterion is:

$$size(f_R) > \underbrace{\left(\left(\frac{size(f_R)}{N}\right)^{\alpha} + 1\right)}_{\beta} \cdot \left(size(f_1) + size(f_2)\right)$$

Parameters used in practice yield $N = 5000$ and $\alpha = -0.50$.

Picture 6: Amplification Factor and Passes



If the problem complexity increases above the defined criterion, the intermediate BDD is optimized on the global variable ordering and the Fault Tree to BDD conversion is restarted from the beginning using the new order. In order to prevent loops during calculation, we disallow optimization if the calculation depth is less or equal than the calculation depth at the position of the last optimization, as illustrated in the example of Picture 6. Depth is defined as the number of Boolean operations performed.

Dynamic optimization was implemented in the *NeuralSpectrum* computer code and was extensively tested on the Leibstadt PSA model. Table 3 summarizes the results (BDD sizes) obtained for different Emergency Core Cooling Systems (ECCS) Fault Trees (smaller is better).

Table 3: Result Summary of Dynamic Sifting

ECCS System	BDD Size using Brute Force (DFLM)	BDD Size using Dynamic Sifting (DS)
High Pressure Core Spray (HPCS)	6545	761
Low Pressure Core Spray (LPCS)	206'503	3050

In practice, this technique produces excellent results. In fact, many very large Fault Trees could be successfully converted using this concept. Due to its efficient selective criteria, this heuristic has runtimes orders of magnitude better than standard heuristics.

3.4. Group Sifting (GS)

A specificity of nuclear PSA models is the parallel alignment of basic events, that is, several parallel failure mechanisms are often found below specific gates. For instance, for a given component, this may include failure to start, failure to run, human errors, electrical supply or logic devices.

All those failure mechanisms are similar in nature, i.e. their basic events have strong affinity; hence, there no reason why they could not be grouped together before or during sifting. During regular

sifting, each basic event representing a single failure mechanism is moved up and down in the ordering. Intuitively however, dependent events should be moved as a group (e.g. chain). This observation is supported by experiments on symmetric sifting [8], in which it was shown that regular sifting tends to put symmetric variables together in the ordering, but that the symmetry groups themselves tend to be in non-optimal positions.

Due to the nature of PSA models, Group Sifting should thus be the preferred sifting technique. Symmetries on two adjacent variable layers x_i and x_{i+1} appear in two distinct types:

- Symmetry on *low* branch (*type* = -1)
    1. $size(x_i)=size(x_{i+1})$ and
    2. $var(high(u))= i+1$ and $ref(high(u))=1$ and $low(u)=low(high(u))$ $\forall$ node $u \in$ layer x_i
- Symmetry on *high* branch (*type* = 0)
    1. $size(x_i)=size(x_{i+1})$ and
    2. $var(low(u))= i+1$ and $ref(low(u))=1$ and $high(u)=high(low(u))$ $\forall$ node $u \in$ layer x_i

The chains of adjacent variables that satisfy the above criteria are stored in an open-ended, jagged matrix. Since the dependency chains may include alternations of *high* and *low* symmetries, some identifiers (*type*) are implemented to determine sub-chain(s) within a chain. A chain can then be represented by one single independent variable (the chain's father).

After having grouped dependent variables using the above criteria and representing them by a father variable, regular sifting is initiated. In this sifting process however, when moving a father variable, the whole group of variables of the dependency chain is virtually moved. The symmetry tests are continuously performed during sifting, so that additional dependent pairs can be identified. Once adjacent variables are identified as dependent (either on *low* or *high*), they are locked together by updating the dependency matrix accordingly. Formally, the dependency matrix looks like:

$$\begin{matrix} id_{father_0} \\ \cdots \\ id_{father_m} \end{matrix} \rightarrow \begin{cases} type_0^0; id_{00}^0 ... id_{0n_0}^0, id_{10}^0 ... id_{1n_1}^0, ..., type_k^0; id_{k0}^0 ... id_{kn_k}^0 \\ \cdots \\ type_0^0; id_{00}^0 ... id_{0n_0}^0, id_{10}^0 ... id_{1n_1}^0, ..., type_k^0; id_{k0}^0 ... id_{kn_k}^0 \end{cases}$$

This technique can be seen as dynamic variable modularization combined with sifting.

Comprehensive benchmarks on different ECCS models confirmed the anticipated effectiveness of the Group-Sifting technique on PSA models. The results were so promising that the technique has been implemented as the standard optimization technique in the *NeuralSpectrum* computer code. Table 4 shows a result summary (BDD sizes) for different ECCS Fault Trees.

Table 4: Result Summary of Group Sifting

ECCS Systems	BDD Size using Brute Force (DFLM)	BDD Size using Regular Sifting	BDD Size using Group Sifting (GS)
High Pressure Core Spray (HPCS)	6545	3204	497
Low Pressure Core Spray (LPCS)	206'503	40'656	7763
Residual Heat Removal (RHR)	306'339	99'945	11'948

3.5. Dynamic Group Sifting using Modularization (DGSM)

Dynamic Group Sifting using Modularization (DGSM) is the most effective dynamic Fault Tree to BDD conversion technique that was developed in this study. It combines the aspects of static optimization, Group Sifting (dynamically implemented) and modularization (as pre-processing). The procedure is depicted in Picture 7. Table 5 summarizes the results (BDD sizes) obtained for different system Fault Trees.

Picture 7: Dynamic Group Sifting using Modularization

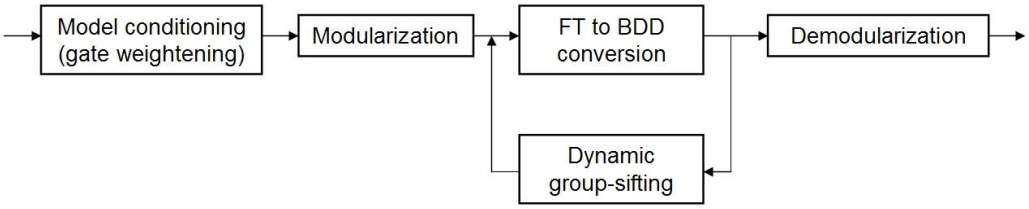


Table 5: Result Summary of Dynamic Group Sifting using Modularization

ECCS Systems	BDD Size using Brute Force (DFLM)	BDD Size using Regular Sifting	BDD Size using DGSM
High Pressure Core Spray (HPCS)	6545	3204	497
Low Pressure Core Spray (LPCS)	206'503	40'656	3053
Residual Heat Removal A (RHR/A)	306'339	99'945	3117
Residual Heat Removal B (RHR/B)	Impossible	52'447	21'177
All ECCS (+ support systems)	Impossible	Impossible	1'781'100

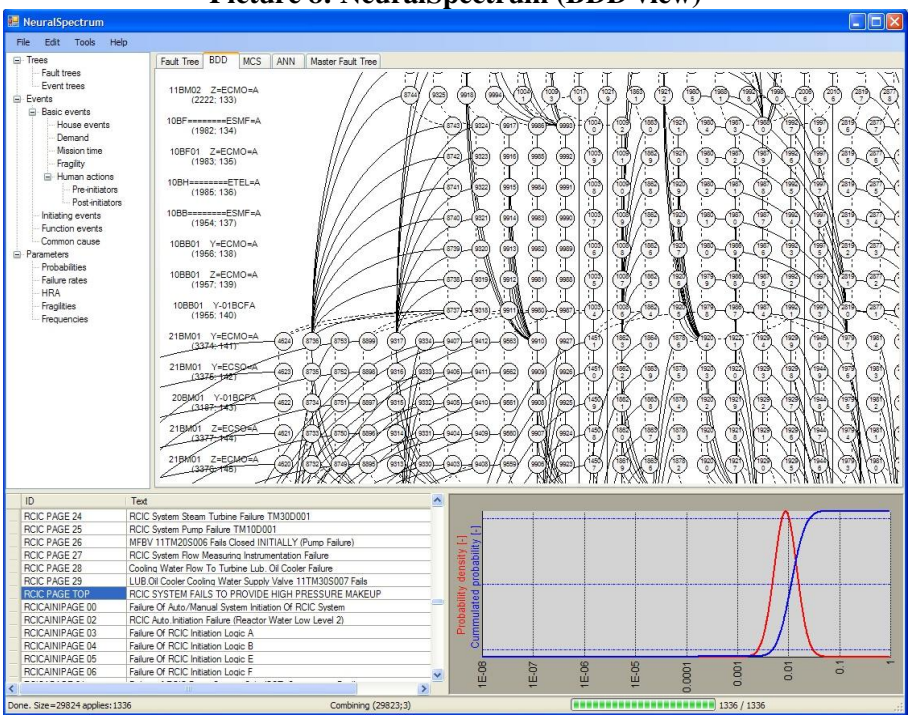
Using this technique, a complete PSA sequence with all ECCS (including their support systems) of the Leibstadt NPP could be converted to BDD, which is a very encouraging result.

4. NEURALSPECTRUM COMPUTER CODE

NeuralSpectrum is a fully integrated Fault Tree - BDD tool that was developed as part of this study. Its development was motivated by the need to evaluate the different heuristics and evaluate them on practical examples. The software is Windows compatible and was programmed in Microsoft C#.Net. The program has about 10'000 lines of code. A view of the software's main window (in BDD view) is shown in Picture 8. The main features of the software are listed below:

- Import and export of PSA model
- Fault Tree rewriting techniques
  - Coalescing
  - Occurrence based variable ordering
  - Fan-out permutation
  - Dual hypergraph netlength optimization
  - Dual hypergraph gate weighting
- Fault Tree statistics
  - Basic event and gate occurrences lists
- Conversion of Fault Trees to BDD
  - Depth-First Left-Most (DFLM)
  - Sifting
  - Group-Sifting (GS)
  - Modularization
  - Dynamic Group Sifting using Modularization (DGSM)
- BDD optimization package
  - Sifting
  - FORCE
  - Group Sifting, Random Group-Sifting
  - Variable affinity based on affinity matrix
- Generation of Minimal Cutsets (MCS) and ZBDD
- Variable orders combination
  - Concatenation
  - Concatenation with swap
  - Linear optimization (minimizes entropy between two variable orders)

Picture 8: NeuralSpectrum (BDD view)



5. BENCHMARKS

Since we succeeded in converting a master Fault Tree to its BDD form, we are now in a position to make numerical comparisons between different quantification techniques and quantifiers. Different Fault Tree top events probabilities (p_{top}) of the Leibstadt PSA model are quantified exactly using BDDs and compared to results obtained from standard Minimal Cutsets (MCS) based analysis, using *RiskSpectrum*. Comparisons are done for internal events, where basic events probabilities are typically low, and for seismic events, where the probabilities are increasing with increasing earthquake magnitudes. The quantification setup is used for the MCS analysis is shown in Table 6. The comparison results for internal events are shown in Table 7 (5% maximum relative error).

Table 6: Quantification Setup

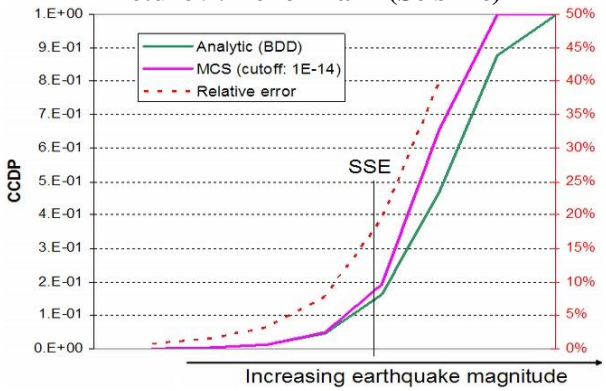
Absolute quantification truncation	10^{-14}
Relative quantification truncation	0
Maximum number of saved MCS	10'000'000

Table 7: Benchmark (Internal Events)

ECCS Systems	**Number of variables**	**p_{top}^{MCS} (*RiskSpectrum*)**	**p_{top}^{Exact} (BDD)**	**Relative Error**
High Pressure Core Spray (HPCS)	569	2.50E-02	2.50E-02	0.03%
Residual Heat Removal A and B (RHR/A+B)	1946	7.83E-04	7.77E-04	0.72%
All ECCS (+ support systems)	3650	2.31E-08	2.20E-8	5.05%

A similar comparison was performed under seismic conditions, where the probabilities increase with increasing earthquake magnitudes. The seismic magnitude is given as a sustained Peak Ground Acceleration (PGA). Nuclear Power Plants are designed for PGA values up to the Safe Shutdown Earthquake (SSE) magnitude. Design reserves provide higher defense capabilities that are typically evaluated as a Conditional Core Damage Probability (CCDP). The detailed comparison with exact BDD-based results is shown in Picture 9 as a function of the seismic magnitude. It is shown that higher seismic magnitudes are difficult to address using MCS-based analysis, because the rare event approximation begins to significantly impact the results (20% relative error at SSE).

Picture 9: Benchmark (Seismic)



6. CONCLUSION

In this study, innovative static and dynamic optimization algorithms to efficiently convert Fault Trees to their analytical BDD form have been developed and investigated as a case study on the full scope PSA model of the Leibstadt Nuclear Power Plant. Since a BDD analytically encodes a Boolean expression, the exact failure probability of the top event can be calculated without numerical approximation or truncation.

A software tool (*NeuralSpectrum*) has been developed. *NeuralSpectrum* is an integrated Fault Tree - BDD tool that features a Fault Tree package, a BDD engine and a Minimal Cutset engine, with dedicated Fault Tree to BDD conversion and optimization routines.

The Leibstadt PSA model was successfully converted to a BDD form of more than 1'500'000 nodes, for a total of 3650 basic events. The BDD covers a complete Event Tree sequence that includes reactor shutdown and reactor cooling with all Emergency Core Cooling Systems, including all their support systems.

The impact of the different approximations and truncations used in the classical approach is evaluated on the Leibstadt PSA model, by comparing the approximated results (*RiskSpectrum*) to exact BDD results. The comparison shows that the classical approach produces accurate results for internal event assessments, but should be used with caution for external event assessments or Level 2 PSA, where the probabilities are typically much higher.

References

[1] A. Rauzy, "*New Algorithms for Fault Trees Analysis*", Reliability Engineering and System Safety, 40, pp. 203–211, (1993).

[2] R. Bryant, "*Graph Based Algorithms for Boolean Function Manipulation*", IEEE Transactions on Computers, 35, pp. 677-691, (1986).

[3] B. Bollig and I. Wegener, "*Improving the Variable Ordering of OBDDs is NP-Complete*", IEEE Transactions on Computers, 45(9), pp. 993-1002, (1996).

[4] R. Rudell, "*Dynamic Variable Ordering for Ordered Binary Decision Diagrams*", In Proc. Of International Conference on Computer Aided Design (ICCAD), pp. 42-47, (1993).

[5] S. Minato et al., "*Shared Binary Decision Diagrams with Attributed Edges for Efficient Boolean Function Manipulation*", In Proc. of the 27th ACM/IEEE DAC, pp. 52-57, (1990).

[6] F.A. Aloul et al., "*FORCE: A Fast and Easy-To-Implement Variable-Ordering Heuristic*", Great Lakes Symposium on VLSI, pp. 116-119, (2003).

[7] O. Nusbaumer, "*Analytical Solutions of Linked Fault Tree Probabilistic Risk Assessments using Binary Decision Diagrams with Emphasis on Nuclear Safety Applications*", ETH, 2007, Zurich.

[8] S. Panda et al., "*Symmetry Detection and Dynamic Variable Ordering of Decision Diagrams",* In Proc. of International Conference on Computer Aided Design (ICCAD), pp. 628-631, (1994).