FAST BDD TRUNCATION METHOD FOR EFFICIENT TOP EVENT PROBABILITY CALCULATION

WOO SIK JUNG*, SANG HOON HAN and JOON-EON YANG
Korea Atomic Energy Research Institute,
P.O. Box 105, Yuseong, Daejeon, 305-600, Korea
*Corresponding author. E-mail : woosjung@kaeri.re.kr

Received January 4, 2008
Accepted for Publication July 31, 2008

A Binary Decision Diagram (BDD) is a graph-based data structure that calculates an exact top event probability (TEP). It has been a very difficult task to develop an efficient BDD algorithm that can solve a large problem since it is highly memory consuming. In order to solve a large reliability problem within limited computational resources, many attempts have been made, such as static and dynamic variable ordering schemes, to minimize BDD size. Additional effort was the development of a ZBDD (Zero-suppressed BDD) algorithm to calculate an approximate TEP. The present method is the first successful application of a BDD truncation. The new method is an efficient method to maintain a small BDD size by a BDD truncation during a BDD calculation. The benchmark tests demonstrate the efficiency of the developed method. The TEP rapidly converges to an exact value according to a lowered truncation limit.

KEYWORDS : Binary Decision Diagram, BDD, Fault Tree, Truncation, Top Event Probability

1. INTRODUCTION

1.1 BDD Algorithm

A Binary Decision Diagram (BDD) [1-3] provides an efficient representation and manipulation of Boolean formulae, and it was shown that a BDD is effective in the diverse fields of computer science and reliability [4]. Bryant [3] popularized the use of a BDD by developing a set of algorithms for an efficient construction and manipulation of BDDs. The BDD algorithm was applied to a reliability analysis [5,6] and has been investigated to solve large fault trees and importance measures [7-10]. The BDD algorithm has become a very popular method to calculate an exact top event probability (TEP) of a small or intermediate size reliability problem [5-10].

Figure 1 displays the relations among fault tree solving methods, such as a BDD algorithm, a ZBDD (Zero-suppressed BDD) algorithm, and an algorithm based on traditional Boolean algebra. The BDD algorithm generates a BDD structure by solving a fault tree. On the other hand, an algorithm based on traditional Boolean algebra generates minimal cut sets (MCSs). An MCS is a minimal combination of basic events that causes a top event. The MCSs could be expressed by a ZBDD structure, which is a factorized form of MCSs [11]. As shown in Fig. 1, a BDD structure could be reduced to a ZBDD structure by truncating and subsuming the BDD structure. However, there had been no method to generate a ZBDD structure directly from a fault tree before the development of the ZBDD algorithm [12].

The BDD algorithm calculates an exact TEP since it does not employ any approximations such as a truncation, a rare-event approximation, and a delete-term approximation. All details regarding a BDD and a ZBBD are explained in Sections 1.2 and 1.3, and the delete-term approximation is illustrated in Appendix A. Since the BDD algorithm is highly time and memory consuming, especially for large problems, it has been difficult to solve large reliability problems such as fault trees for accident sequences in a Probabilistic Safety Assessment (PSA). In order to solve a large complex problem with limited computational resources, numerous attempts have been made to minimize BDD size, which is measured by the number of nodes in the BDD structure (see Sections 1.2 to 1.4). The BDD size is heavily dependent on the choice of the variable ordering for a BDD construction (see Section 1.3 and Appendix B). As shown in Fig.1, a BDD truncation cannot be applied to the BDD algorithm (see Section 2.2).

1.2 ZBDD Algorithm

ZBDD structure was proposed as an efficient data structure that encodes minimal cut sets (MCSs) [11]. The

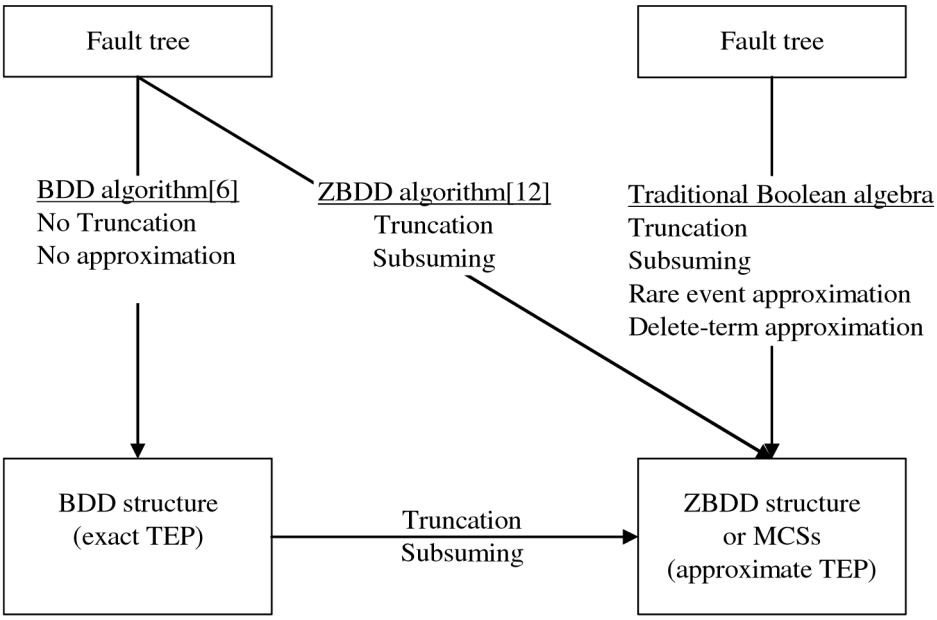

Fig. 1. Fault Tree Solving Methods

ZBDD structure is interpreted as a factorized form of MCSs. By optimally choosing the factorization order, that is, a ZBDD variable ordering, the ZBDD size could be significantly minimized. A new ZBDD algorithm was developed by using special formulae of Boolean operations on two ZBDDs [12]. The ZBDD algorithm is an important variation of the BDD algorithm since it solves a large fault tree with a truncation limit in a short time. If an MCS probability is less than a truncation limit, the MCS is deleted during the calculation. The ZBDD algorithm and its software FTREX (Fault Tree Reliability Evaluation eXpert) were developed to overcome the massive memory requirements and a long run time [12-14]. Benchmark tests [12,13] were performed to demonstrate the efficiency of the ZBDD algorithm over an algorithm that is based on traditional Boolean algebra. The ZBDD algorithm showed a fast calculation speed for various fault trees and truncation limits. Due to the nature of the ZBDD, MCSs could be calculated easily with a given truncation limit. In terms of computation time and memory usage, the ZBDD algorithm is much more efficient than MCS algorithms that are based on classical Boolean algebra.

1.3 Comparison of the BDD and ZBDD Algorithms

If a fault tree in Fig. 2 is solved in a bottom-up way by the BDD algorithm with an alphabetical variable ordering $a < b < c < d < e$, the final BDD structure becomes

$$a\left(b+\bar{b}\left(c+\bar{c}\left(d+\bar{d}e\right)\right)\right)+\bar{a}b. \quad (1)$$

The derivation of Eq. (1) is explained in Appendix B.

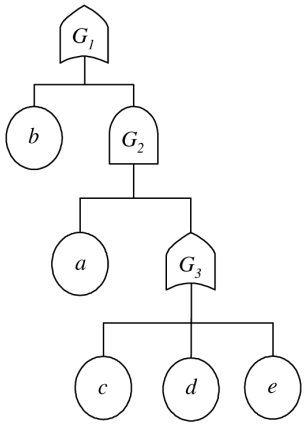

Fig. 2. Sample Fault Tree

The BDD structure in Eq. (1) is depicted in Fig. 3. Variables a and e are located in the highest and lowest positions in the BDD structure according to the variable ordering $a < b < c < d < e$, respectively.

As shown in Fig. 3, the BDD structure consists of nested nodes. The BDD algorithm solves the fault tree in Fig. 2 in a bottom-up way and generates the BDD structure in Fig. 3. Each BDD node that has a circular shape in Fig. 3 has a pair of failed and successful states. The solid and dashed lines from a node denote the failed and successful states of a node, respectively. The terminal nodes are always one of two terminal nodes labeled 0 or 1, and they are represented by square boxes in Fig. 3.

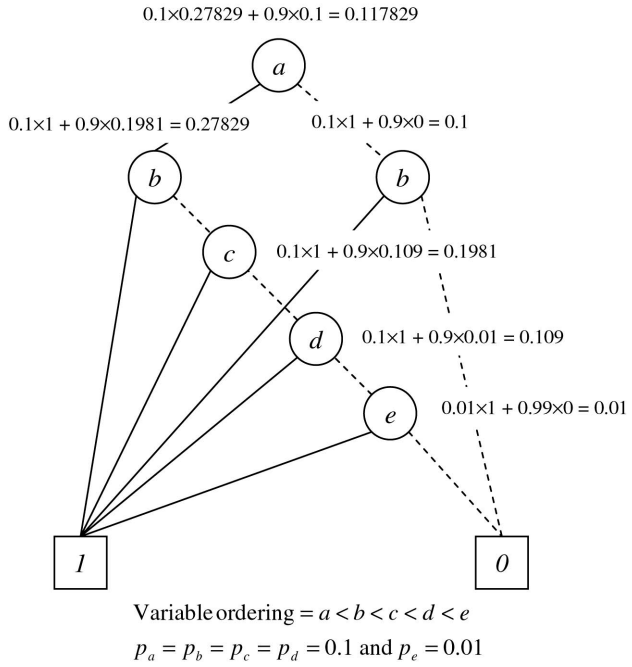

Fig. 3. BDD Solution

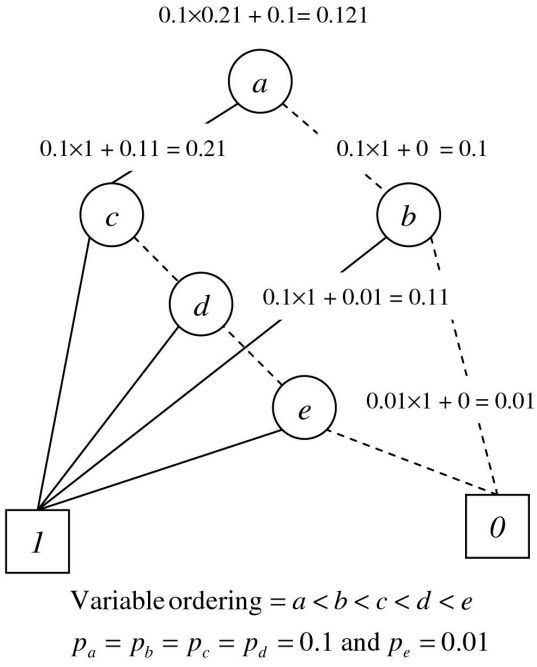

Fig. 4. ZBDD Solution

Here, 1 and 0 denote the failed and successful states, respectively. As shown in Fig. 3, the first node *a* is connected to two child nodes with solid and dashed lines. The solid and dashed lines from the node *a* denote failed Boolean symbols *a* and $\bar{a}$, respectively. The terminal nodes are depicted with square boxes in Fig. 3.

As illustrated in Fig. 3, the probability of the BDD structure is calculated recursively [6], and the resultant probability is an exact TEP. The mathematical expression for the recursive probability calculation is

$$\begin{aligned} &P\left(ab + a\bar{b}c + a\bar{b}\bar{c}d + a\bar{b}\bar{c}\bar{d}e + \bar{a}b\right) \\ &= p_a p_b + p_a q_b p_c + p_a q_b q_c p_d + p_a q_b q_c q_d p_e + q_a p_b \\ &= p_a\left(p_b + q_b\left(p_c + q_c\left(p_d + q_d p_e\right)\right)\right) + q_a p_b \ , \end{aligned} \tag{2}$$

where p_x=$P(x)$ and q_x=1-p_x.

As shown in Fig. 1, the ZBDD structure can be obtained in two ways. First, it could be calculated directly by solving a fault tree by the ZBDD algorithm [12]. Second, it could be obtained indirectly by ignoring the successful events in the final BDD structure and employing the subsuming operation in Ref. 6. If the fault tree in Fig. 2 is solved with the alphabetical variable ordering $a < b < c < d < e$ by the ZBDD algorithm [12], the final ZBDD structure is

$$a\left(c + d + e\right) + b\,. \tag{3}$$

The ZBDD structure in Fig. 4 has the form of factorized MCSs.

Before the development of the ZBDD algorithm [12], the ZBDD structure should be obtained indirectly by the following two steps [6]: (1) a BDD structure is generated by solving a fault tree, and (2) a ZBDD structure is calculated by ignoring the successful events in the BDD structure and employing a subsuming operation [6]. By using these two indirect steps, the BDD structure in Fig. 3 could be reduced to the ZBDD structure in Fig. 4. The BDD structure in Eq. (1) is reduced to $a(b+c+d+e)+b$ by ignoring the successful events. Then, by deleting the subset *ab* of *b*, the equation $a(b+c+d+e)+b$ becomes the ZBDD structure in Eq. (3) as

$$\begin{aligned} &a\left(b + \bar{b}\left(c + \bar{c}\left(d + \bar{d}e\right)\right)\right) + \bar{a}b \rightarrow a\left(b + \left(c + \left(d + e\right)\right)\right) + b \\ &= a\left(c + d + e\right) + b. \end{aligned} \tag{4}$$

1.4 Static and Dynamic Variable Ordering of the BDD Algorithm

In order to solve a large reliability problem within limited computational resources, numerous attempts have been made to minimize a BDD size. The size of a BDD structure (measured by the number of nodes) is drastically dependent on the choice of the variable ordering for a BDD construction (see Appendix B). Finding the optimal variable ordering is an NP-hard (nondeterministic polynomial-time hard) problem [15]. That is, if a fault tree has *N* basic events as variables, all possible 2^N variable orderings should be tried to find and confirm the optimal

variable ordering. Bryant [3] showed that the importance of a desirable variable ordering may lead to a small size of a BDD structure. All BDD methods for finding a better variable ordering are based on static and dynamic variable ordering heuristics [16-21]. These heuristics are based on a decision for a trade-off between a fast calculation and a small BDD structure. Dynamic variable reordering methods using variable sifting [18-21] are considered as significant improvements in the BDD technology. However, the current static and dynamic variable ordering methods are still inefficient for solving large problems.

1.5 Objectives of this Study

In order to calculate an exact TEP of a large fault tree with a BDD algorithm, some approaches have been investigated, such as the static and dynamic variable ordering approaches in Section 1.4. The BDD truncation algorithm in this study is a new approach. It is also well known that a BDD truncation cannot be applied to a BDD algorithm (see Section 2.2). The objective of this study is to develop a special BDD truncation method to accelerate a BDD calculation without sacrificing the solution probability accuracy. The developed method is described in Section 2 and the benchmark test results to demonstrate the efficiency of the developed method are listed in Section 3.

2. METHODS

2.1 BDD Algorithm

A BDD is a directed acyclic graph where a Shannon decomposition is implemented at every node. The Shannon decomposition is succinctly defined in terms of the ternary If-Then-Else (ITE) connectives as

$$\begin{aligned} F &= ite(x, F_1, F_2) = xF_1 + \bar{x}F_2 \\ G &= ite(y, G_1, G_2) = yG_1 + \bar{y}G_2 \end{aligned}, \qquad (5)$$

where x and y are two variables with the variable ordering $x < y$. The Shannon decomposition is a method by which a Boolean function can be represented by the sum of two sub-functions of the original Boolean function. For example, the Boolean function F in Eq. (5) is divided into two sub-functions, xF_1 and $\bar{x}F_2$, where F_1 and F_2 are created from F by setting the variable x to TRUE and FALSE, respectively [12]. The BDD operation is recursively performed on the higher priority variable x as

$$H = F <> G = \begin{cases} ite(x, F_1 <> G_1, F_2 <> G_2) \ if \ x = y \\ ite(x, F_1 <> G, \ F_2 <> G) \ if \ x < y \end{cases}, \qquad (6)$$

where <> is an *AND* or *OR* Boolean operator. The recursive BDD operation in Eq. (6) is illustrated as follows:

$$\begin{aligned} H &= F + G \\ &= ite(a,1,ite(b,ite(c,1,0),0)) + ite(d,1,0) \\ &= ite(a,1 + ite(d,1,0),ite(b,ite(c,1,0),0) + ite(d,1,0)) \\ &= ite(a,1,ite(b,ite(c,1,0) + ite(d,1,0),0 + ite(d,1,0))) \\ &= ite(a,1,ite(b,ite(c,1 + ite(d,1,0),0 + ite(d,1,0)),ite(d,1,0))) \\ &= ite(a,1,ite(b,ite(c,1,ite(d,1,0)),ite(d,1,0))) \ , \end{aligned} \qquad (7)$$

where F=*ite*(a,1,*ite*(b,*ite*(c,1,0)0)) and G=*ite*(d,1,0). An alphabetical variable ordering is used in Eq. (7).

A recursive BDD operation is listed in the left column of Fig. 5, which is explained only for *AND* or *OR* Boolean operations. For suppressing the repetition of the same operation $H=F<>G$, the BDD operation results {*hash_key*(<>, F, G), H} are stored in an operation hash table (OHT) and reused. Furthermore, a unique ITE is maintained during the calculation by storing and retrieving {*hash_key*(x, H_1, H_2), H} to and from an ITE hash table (IHT). Here, *hash_key*(<>, F, G) and *hash_key*(x, H_1, H_2) are hash functions that map a triple into a single location in the hash tables. For example, the BDD operation *ite*(c,1,0)+*ite*(d,1,0) and its resultant BDD *ite*(c,1,*ite*(d,1,0)) in Eq. (7) are stored in the OHT, and the stored BDD is reused when the BDD operation *ite*(c,1,0)+*ite*(d,1,0) is requested.

A hash table is an important data structure in computer science that associates keys with values. For example, the hash function *hash_key*(<>, F, G) and *hash_key*(x, H_1, H_2) calculate a specific key by using tripe values (<>, F, G) and (x, H_1, H_2), respectively. The key is a specific memory location of a hash table. The BDD operation $H=F<>G$ is stored in the OHT as follows.

1. Calculate a key by the hash function *hash_key*(<>, F, G).
2. Go to the specific location of the hash table that has the key.
3. Save {*hash_key*(<>, F, G), H} at the OHT. That is, the information {<>, F, G, H} is stored in the data storage that has *hash_key*(<>, F, G).

When a BDD operation $F<>G$ is requested, the pre-calculated H is retrieved from the hash table as follows.

1. Calculate a key by the hash function *hash_key*(<>, F, G).
2. Go to the specific location of the hash table that has the key.
3. Retrieve H if the hash table has {*hash_key*(<>, F, G), H}, that is, if the hash table has stored {<>, F, G, H} at the location of *hash_key*(<>, F, G).

2.2 BDD Algorithm with Truncation

The developed recursive BDD operation is listed in the right column of Fig. 5. The BDD truncation and the use of OHT were mutually exclusive, and they could not be used together. That is, the BDD truncation could not be applied to the BDD algorithm.

bdd_operation(<>, F, G)
Output: H = F<>G

```
/* Step 1. terminal_case */
if ( (F=0 or 1) or (G=0 or 1) or (F=G) )
   return terminal_case(<>, F, G)

/* Step 2. OHT has F <> G ? */
k= hash_key(<>, F, G)
if ( OHT_has_member (k,H) )
   return H

/* Step 3. solve F <> G */
if ( x > y ) swap(F,G)
if ( x = y ) {
   H1 ← bdd_operation(<>, F1, G1)
   H2 ← bdd_operation(<>, F2, G2)
} else { /* x < y */
   H1 ← bdd_operation(<>, F1, G)
   H2 ← bdd_operation(<>, F2, G)
}

/* Step 4. IHT */
H ← IHT_find_or_add(x,H1,H2)

/* Step 5. OHT */
if ( OHT_has_member(k, H) ) {
   return H
} else {
   OHT_insert_member(k, H)
   return H
}
```

(a) Traditional BDD algorithm

bdd_operation(<>, F, G, p)
Output: H = F<>G

```
/* Step 1. terminal_case */
if ( p < truncation_limit ) return 0
if ( (F=0 or 1) or (G=0 or 1) or (F=G) )
   return terminal_case(<>, F, G)

/* Step 2. OHT has F <> G ? */
k= hash_key(<>, F, G)
if ( OHT_has_member (k,H,q) )
   if (q >= p) return H

/* Step 3. solve F <> G */
if ( x > y ) swap(F,G)
if ( x = y ) {
   H1 ← bdd_operation(<>, F1, G1, p × p_x)
   H2 ← bdd_operation(<>, F2, G2, p × (1-p_x))
} else { /* x < y */
   H1 ← bdd_operation(<>, F1, G, p × p_x)
   H2 ← bdd_operation(<>, F2, G, p × (1-p_x))
}

/* Step 4. IHT */
H ← IHT_find_or_add(x,H1,H2)

/* Step 5. OHT */
if ( OHT_has_member(k, H,q) ) {
   /* update OHT */
   if (p > q) {k, T, q} ← {k, H, p}
   return H
} else {
   OHT_insert_member(k, H, p)
   return H
}
```

(b) New BDD algorithm for truncation

IHT = ITE hash table, OHT = operation hash table

Fig. 5. Truncation Implementation to the BDD Algorithm

However, in this study, a method to incorporate the truncation limit into the OHT was developed by employing an upper probability p in the *hash_key*(<>, F, G, p). As shown in Step 3, p_x or (1.0-p_x) is multiplied to the upper probability p for a new recursive BDD operation. Here, p_x denotes the probability of an event x. The upper probability is calculated as

1. $p \leftarrow 1$ when the BDD operation starts.
2. $p \leftarrow p \times p_x$ when the first Boolean operation $F_1 <> G_1$ or $F_1 <> G$ of ITEs in Eq. (6) is started.
3. $p \leftarrow p \times (1.0-p_x)$ when the second Boolean operation $F_2 <> G_2$ or $F_2 <> G$ of ITEs in Eq. (6) is initiated.

The BDD operation in Eq. (7) is constructed recursively in a top-down way by constructing a top-level BDD structure first and then solving the remaining low-level BDD structure later. For example, the BDD operation *ite*(b,*ite*(c,1,0),0)+*ite*(d,1,0) in Eq. (7) is performed under the upper logic $\bar{a}$. That is, the BDD operation has an upper probability $P(\bar{a})$. Similarly, the BDD operation *ite*(c,1,0)+*ite*(d,1,0) in Eq. (7) is performed with the upper logic $\bar{a}b$ and the upper probability $P(\bar{a})P(b)$.

Whenever the upper probability is less than the truncation limit during the recursive BDD operation, the remaining recursive BDD operations are cancelled and the calculation returns with a terminal node 0. The new BDD operation in Fig. 5 can be summarized as

1. If the probability p is less than the truncation limit, the operation is stopped and returns with a terminal node 0.
2. If {*hash_key*(<>, F, G), H, q} exists in the OHT and the stored upper probability q is larger than or equal to the current upper probability p, the current operation $F<>G$ is not performed and the calculation returns with H. The condition $q > p$ guarantees that the stored H is bigger than the BDD to be calculated since the stored H was calculated with a larger upper probability q. That is, the stored H is a superset of the BDD to be calculated. Thus, the OHT is maintained at a small size. By maintaining supersets in the OHT when a truncation is applied, the correct solution could be obtained.
3. The calculated $H=F<>G$ and p are stored in the

OTH if {*hash_key*(<>, *F*, *G*), *H*, *p*} is not in the hash table.

4. The resultant $H=F<>G$ and p updates the stored values in the OHT if they satisfy the inequality condition $q < p$. T and q are replaced with H and p, respectively, since the calculated H is a superset of the stored T.

As explained, the BDD truncation is applied efficiently to all BDD operation levels by using the upper probabilities. The algorithm in this Section provides a new BDD truncation algorithm where the BDD truncation is applied efficiently to all BDD operation levels and the BDDs are stored properly in the OHT. The new method guarantees that the correct BDD calculations could be performed in a short time with less memory.

If the calculations and comparisons of the upper probabilities are not performed and the calculated BDDs are stored to the OHT during the recursive BDD operations, the BDDs that are reused from the OHT might be much smaller or unnecessarily larger than the required BDDs. If smaller BDDs are reused, the correct BDDs cannot be calculated. In most cases, unnecessarily large BDDs are frequently stored in the hash table and reused, and a large number of unnecessary recursive BDD calculations should be performed. Hence, these operations consume all the available memory or take infinite calculation time and the calculations.

3. BENCHMARK TESTS

Two benchmark problems were solved, and the results are listed in Tables 1 and 2 and plotted in Figs. 6

Table 1. Benchmark Test A

Fault tree = CEA9601 (http://iml.univ-mrs.fr/~arauzy/aralia/benchmark.html)
201 gates, 186 events, 26 negates, 4 complemented events
All event probabilities = 0.001
Without fault tree restructuring and modules
Pentium 4 3.0 GHz, 2 GB RAM

Truncation	BDD			FTREX [14]	
	TEP	BDD size	Run time (seconds)	TEP	Cutset number
1.00E-11	1.059240E-06	12,349	0.88	1.143904E-06	1,144
1.00E-12	1.092776E-06	19,090	1.47	1.170891E-06	28,132
1.00E-13	1.176633E-06	54,280	1.77	1.197163E-06	54,436
1.00E-14	1.180200E-06	154,728	3.11	1.197163E-06	54,436
1.00E-15	1.181040E-06	200,157	4.36	1.198107E-06	999,676
1.00E-16	1.182503E-06	320,805	4.67	1.198723E-06	1,615,876
1.00E-17	1.182611E-06	703,816	7.77	1.198723E-06	1,615,876
1.00E-18	1.182618E-06	811,113	9.94	1.198723E-06	6,390,196
Exact TEP	1.182622E-06	1,250,725	6.22		

Table 2. Benchmark Test B

Fault tree = HPSI3.FTP
571 gates, 421 events, 0 negates, 0 complemented events
With fault tree restructuring and modules
Pentium 4 3.0 GHz, 2 GB RAM

Truncation	BDD			FTREX [14]	
	TEP	BDD size	Run time (seconds)	TEP	Cutset number
1.00E-11	1.076139E-03	130,013	2.15	1.082535E-03	5,096
1.00E-12	1.076293E-03	274,187	3.78	1.082557E-03	11,354
1.00E-13	1.076325E-03	573,908	6.76	1.082561E-03	24,381
1.00E-14	1.076332E-03	1,131,067	12.60	1.082562E-03	45,688
1.00E-15	1.076334E-03	2,051,615	21.87	1.082562E-03	86,748
Exact TEP	1.076334E-03	2,497,172	21.31		

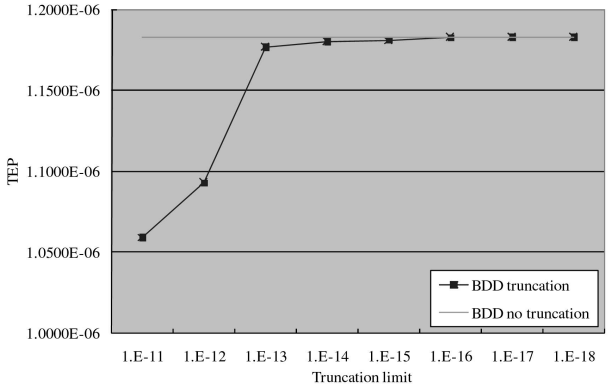

Fig. 6. TEP of Benchmark Test A

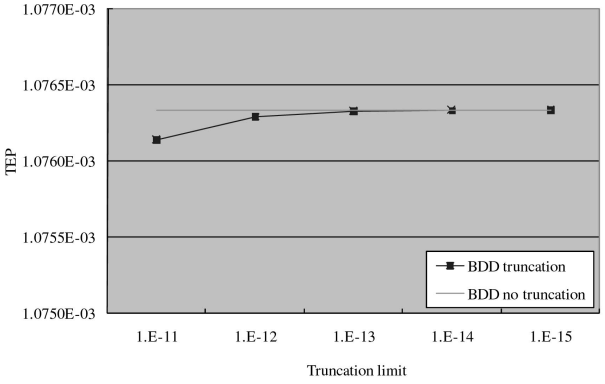

Fig. 8. TEP of Benchmark Test B

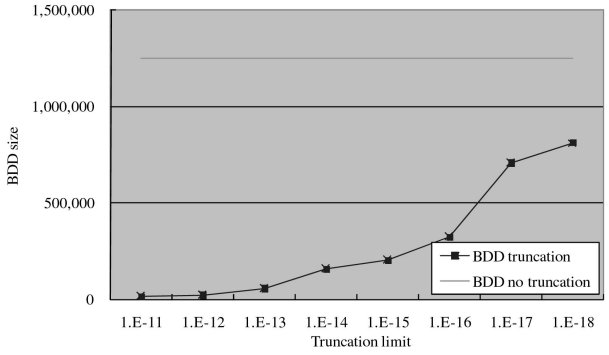

Fig. 7. BDD Size of Benchmark Test A

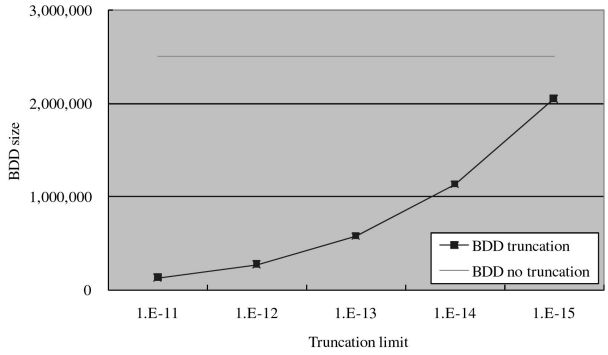

Fig. 9. BDD Size of Benchmark Test B

to 9. The first problem in Table 1 is a non-coherent fault tree and the second problem in Table 2 is a relatively large coherent fault tree. The fault tree in Table 1 is known as one of the most difficult and complex Benchmark fault trees since it has a relatively large number of negates. The fault tree in Table 2 is a fault tree for a High Pressure Safety Injection (HPSI) system in a nuclear power plant. It is one of the biggest system-level fault trees since it has many supporting-system fault trees. The exact TEPs are calculated by the BDD algorithm and listed in the bottom rows of Tables 1 and 2. The other results are calculated by the new BDD algorithm with various truncation limits. The BDD node number denotes the required computational memory. The BDD sizes in Tables 1 and 2 are measured by the number of nodes in a final BDD structure.

The TEPs in the second columns of Tables 1 and 2 are depicted in Figs. 6 and 8, respectively. Figs. 6 and 8 show that the TEPs calculated with the nonzero truncation limits are very close to the exact TEPs. As shown in Fig. 6, the TEP at the truncation limit 1.0E-13 is very close to the exact TEP. As shown in Fig. 8, the TEP at the truncation limit 1.0E-11 has a negligible difference with the exact TEP. Thus, these Benchmark tests A and B show that the exact TEPs could be estimated with a relatively high truncation limit.

The BDD sizes in the third columns of Tables 1 and 2 are depicted in Figs. 7 and 9, respectively. As shown in Figs. 7 and 9, the proposed method uses much less memory than the BDD algorithm. Thus, these benchmark tests show that the exact TEPs could be estimated with a relatively small amount of memory.

As listed in the fourth columns of Tables 1 and 2, the TEPs are calculated rapidly by the proposed method. The calculation results with FTREX [14], which is a cutest-based ZBDD algorithm, are listed in the last two columns in Tables 1 and 2. The approximate TEPs by the cutest-based algorithm never converge to the exact TEPs.

As listed in Table 3, further tests were performed. Fifteen difficult fault trees were selected with two criteria from the problems in the website http://iml.univ-mrs.fr/~arauzy/aralia/benchmark.html. First, a fault tree should be solved with a BDD algorithm. Second, the running time should exceed one second. The fault trees are solved with three different calculations:

1. Exact TEP calculations by the BDD algorithm. The results are listed in the eighth to tenth columns of

Table 3. Benchmark Test C

Fault trees (http://iml.univ-mrs.fr/~arauzy/aralia/benchmark.html)
All event probabilities = 0.001
Without fault tree restructuring and modules
Pentium 4 3.0 GHz, 2 GB RAM

Fault tree	Proposed BDD truncation algorithm (truncation limit = 1.0E-05 × exact TEP)			Proposed BDD truncation algorithm (truncation limit = 1.0E-10 × exact TEP)			BDD algorithm for an exact TEP			Relative error (*d*)	Relative error (*e*)
	TEP (*a*)	BDD size	Run time (seconds)	TEP (*b*)	BDD size	Run time (seconds)	TEP (*c*)	BDD size	Run time (seconds)		
DAS9601	4.651006E-05	1,756	0.78	4.652488E-05	7,415	1.14	4.652500E-05	21,192	1.55	0.032%	0.000%
EDF9202	1.289224E-01	671	0.55	1.304794E-01	10,388	0.88	1.304825E-01	9,600	2.30	1.196%	0.002%
EDF9203	4.308621E-02	9,410	0.73	4.391692E-02	116,340	2.98	4.392265E-02	306,265	5.86	1.904%	0.013%
EDF9204	3.797168E-02	6,300	0.67	3.833556E-02	117,569	3.30	3.833615E-02	209,855	14.01	0.951%	0.002%
EDFPA14B	2.004268E-02	6,500	0.55	2.027975E-02	63,577	1.00	2.028067E-02	224,303	3.56	1.173%	0.005%
EDFPA14O	2.013442E-02	6,916	0.69	2.033033E-02	112,427	1.36	2.033145E-02	931,057	11.90	0.969%	0.006%
EDFPA14P	6.226406E-03	3,538	0.55	6.238446E-03	78,100	0.88	6.238447E-03	201,626	2.74	0.193%	0.000%
EDFPA14Q	2.007533E-02	2,078	0.53	2.028978E-02	48,157	0.98	2.029115E-02	273,308	3.72	1.064%	0.007%
EDFPA14R	1.108374E-03	617	0.50	1.112773E-03	25,009	0.69	1.112774E-03	163,657	1.86	0.395%	0.000%
EDFPA15B	2.359133E-02	3,367	0.63	2.383538E-02	33,926	0.77	2.383546E-02	49,865	1.30	1.024%	0.000%
EDFPA15O	2.355147E-02	7,525	0.58	2.384528E-02	74,399	1.08	2.384561E-02	187,230	3.32	1.234%	0.001%
EDFPA15P	6.148855E-03	1,982	0.52	6.154896E-03	27,989	0.86	6.154896E-03	51,078	1.14	0.098%	0.000%
EDFPA15Q	2.351724E-02	985	0.64	2.383475E-02	24,718	0.83	2.383546E-02	143,016	1.92	1.335%	0.003%
EDFPA15R	1.089150E-03	409	0.52	1.091837E-03	9,968	0.55	1.091837E-03	59,665	1.14	0.246%	0.000%
JBD9601	1.042598E-01	1,347	0.61	1.084826E-01	38,445	0.72	1.084934E-01	81,371	1.01	3.902%	0.010%

(*d*) Relative error = ((*c*)-(*a*)) / (*c*) × 100
(*e*) Relative error = ((*c*)-(*b*)) / (*c*) × 100

Table 3.

2. TEP calculations by the proposed BDD truncation method with truncation limits that are 10^{-5} times smaller than the exact TEPs. The truncation limits are calculated by the multiplication of the scaling factor 10^{-5} and the exact TEPs. For example, a truncation limit 4.6525E-10 for a fault tree DAS9601 is obtained by the multiplication of a scaling factor 10^{-5} and the exact TEP 4.6525E-05. The approximate TEP calculation results are listed in the second to fourth columns of Table 3.
3. TEP calculations by the present BDD truncation method with truncation limits that are 10^{-10} times smaller than the exact TEPs. The truncation limits are calculated by the multiplication of a scaling factor 10^{-10} and the exact TEPs. The calculation results are listed in the fifth to seventh columns of Table 3.

As shown in the last two columns of Table 3, the proposed method estimates the exact TEPs with negligible errors. Furthermore, as shown in the third and sixth columns, the proposed method uses much less memory than the BDD algorithm for an exact TEP calculation. As listed in the fourth and seventh columns, the current BDD truncation method estimates the exact TEPs very quickly. Thus, these Benchmark tests in Table 3 demonstrate the efficiency of the developed method where the exact TEPs of various fault trees could be estimated with a negligible error rate in a short time.

As listed in the bottom rows of Tables 1 and 2 and in the ninth and tenth columns of Table 3, the BDD algorithm is highly time and memory consuming. Thus, it shows that it is a very difficult task to solve a large fault trees with a BDD algorithm. When it is impossible to calculate an exact TEP of a huge fault tree with a BDD algorithm, the proposed BDD truncation algorithm could be an excellent method to estimate an exact TEP. The amount of truncated information is negligible from the perspective of the TEPs.

4. CONCLUSIONS

In order to solve a large reliability problem with limited computational resources, numerous attempts have been made to minimize BDD size. The traditional approach was to find out an optimal variable ordering by using some heuristics. An additional attempt was the development of a ZBDD algorithm to calculate an approximate TEP.

This study presents an efficient method to maintain a small BDD size by a BDD truncation during a BDD calculation. The new method provides an accurate TEP in a reasonably short time. The present method is the first successful application of a BDD truncation. The benchmark tests in Section 3 demonstrated the efficiency of the developed method. The TEP converged rapidly to an exact value when the truncation limit was lowered. The present method is very fast and uses much less memory than the BDD algorithm.

For future research, the effect of a BDD truncation on the exact importance measures should be investigated. Furthermore, these truncated BDD importance measures should be compared with those from cutest based methods.

APPENDIX A. DELETE-TERM APPROXIMATION

A fault tree $T=G_1\bar{G}_2$, $G_1=abc+bcd$, and $G_2=ab+ac$ has an exact solution

$$\begin{aligned} T &= (abc+bcd)\overline{(ab+ac)} = (abc+bcd)\overline{a(b+c)} \\ &= (abc+bcd)(\bar{a}+\bar{b}\bar{c}) = \bar{a}bcd, \end{aligned} \tag{A.1}$$

where DeMorgan's law is applied when $\overline{a(b+c)}$ is expanded. Cutsets that have impossible state combinations such as $a\bar{a}$ and $b\bar{b}$ are deleted when $(abc+bcd)(\bar{a}+\bar{b}\bar{c})$ is expanded.

Instead of the complex Boolean algebra in Eq. (A.1), a delete-term approximation [22] is employed in the conventional fault tree analysis for the fast calculation. The delete-term approximation is based on the fact that the top event $T=G_1\bar{G}_2$ cannot occur when G_2 is in a TRUE state. The following are delete-term approximations.

1. The G_1 cutest $\{abc\}$ is deleted since its propagation makes G_2 in a TRUE state and the top event T in a FALSE state. When the G_1 cutest $\{abc\}$ is propagated to the fault tree, G_2 is in a TRUE state since G_2 has a cutset $\{ab\}$ and $\{ac\}$.
2. The G_1 cutest $\{bcd\}$ is selected as a cutest of the top event T since it does not make G_2 in a TRUE state. Finally, $\{bcd\}$ remains as a final cutest as

$$T = (abc+bcd)\overline{(ab+ac)} \approx bcd \tag{A.2}$$

It is a typical example of a delete-term approximation.

APPENDIX B. BDD CALCULATION

A fault tree is solved in a bottom-up way by using Eq. (6). If a fault tree in Fig. 2 is solved with an alphabetical variable ordering $a < b < c < d < e$, the gate G_3 has the following solution:

$$\begin{aligned} G_3 &= c+d+e \\ &= ite(c,1,ite(d,ite(e,1,0))) \\ &= c+\bar{c}\left(d+\bar{d}e\right) \end{aligned} \tag{B.1}$$

The solution of the gate G_2 is obtained by combining $ite(a,1,0)$ and ITE in Eq. (B.1) as

$$\begin{aligned} G_2 &= a\cdot G_3 \\ &= ite(a,1,0)\cdot ite(c,1,ite(d,ite(e,1,0))) \\ &= ite(a,ite(c,1,ite(d,ite(e,1,0))),0) \\ &= a\left(c+\bar{c}\left(d+\bar{d}e\right)\right) \end{aligned} \tag{B.2}$$

The gate G_1 is solved by combining $ite(b,1,0)$ and ITE in Eq. (B.2) as

$$\begin{aligned} G_1 &= b+G_2 \\ &= ite(b,1,0)+ite(a,ite(c,1,ite(d,ite(e,1,0))),0) \\ &= ite(a,ite(b,1,0)+ite(c,1,ite(d,ite(e,1,0))),ite(b,1,0)) \\ &= ite(a,ite(b,1,ite(c,1,ite(d,ite(e,1,0)))),ite(b,1,0)) \\ &= a\left(b+\bar{b}\left(c+\bar{c}\left(d+\bar{d}e\right)\right)\right)+\bar{a}b. \end{aligned} \tag{B.3}$$

If the fault tree in Fig.2 is solved with the other variable orderings, the final BDD structure varies as

$$b+\bar{b}\left(a\left(c+\bar{c}\left(d+\bar{d}e\right)\right)\right)$$
with a variable ordering $b < a < c < d < e$ (B.4)

$$e\left(b+\bar{b}a\right)+\bar{e}\left(d\left(b+\bar{b}a\right)+\bar{d}\left(c\left(b+\bar{b}a\right)+\bar{c}b\right)\right)$$
with a variable ordering $e < d < c < b < a$. (B.5)

Please note that the three BDD structures in Eqs. (B.3), (B.4), and (B.5) are identical Boolean expressions. The BDD size in Eq. (B.4) is much smaller than those in Eqs. (B.3) and (B.5). As illustrated in these examples, the BDD size is drastically dependent on the choice of the variable ordering for a BDD construction.

REFERENCES

[1] C.Y. Lee, "Representation of switching circuits by binary-decision programs," Bell System Technical Journal, 38, pp. 985-999, 1959.

[2] B. Akers, "Binary Decision Diagrams," IEEE Transactions on Computers, C-27(6), pp. 509-516, 1978.
[3] R. Bryant, "Graph Based Algorithms for Boolean Function Manipulation," IEEE Transactions on Computers, C-35(8), pp. 677-691, August, 1986.
[4] R. Bryant, "Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams," ACM Computing Surveys, 24, pp. 293-318, September 1992.
[5] O. Coudert and J.C. Madre, "Implicit and Incremental Computation of Primes and Essential Primes of Boolean Functions," Proceedings of the 29th ACM/IEEE Design Automation Conference, DAC'92, June 1992.
[6] A. Rauzy, "New Algorithms for Fault Trees Analysis," Reliability Engineering and System Safety, 40, pp. 203-211, 1993.
[7] O. Coudert and J.C. Madre, "Fault Tree Analysis: 1020 Prime Implicants and Beyond," Proceedings of the Annual Reliability and Maintainability Symposium, Atlanta, NC, USA, January 1993.
[8] A. Rauzy and Y. Dutuit, "Exact and Truncated Computations of Prime Implicants of Coherent and Non-coherent Fault Trees Within Aralia," Reliability Engineering and System Safety, 58, pp. 127-144, 1997.
[9] Y. Dutuit and A. Rauzy, "Efficient Algorithms to Assess Component And Gate Importance in Fault Tree Analysis," Reliability Engineering & System Safety, Volume 72, pp. 213-222, May 2001.
[10] S. Epstein, A. Rauzy, "Can we trust PRA," Reliability Engineering & System Safety, Volume 88, pp. 195-205, June 2005.
[11] S. Minato., "Zero-suppressed BDDs for set manipulation in combinatorial problems," Proc. of the 30th Int'l Conf. on Design Automation, pp. 272-277, 1993.
[12] W.S. Jung, S.H. Han, J.J. Ha, "A Fast BDD Algorithm for Large Coherent Fault Trees Analysis," Reliability Engineering and System Safety, Vol. 83, pp. 369–374, 2004.
[13] W.S. Jung, S.H. Han, J.J. Ha, "Development of an Efficient BDD Algorithm to Solve Large Fault Trees," Proceedings of the 7th International Conference on Probabilistic Safety Assessment and Management, June, Berlin, Germany, 2004.
[14] W.S. Jung, S.H. Han, J.J. Ha, "An Overview of the Fault Tree Solver FTREX," 13th International Conference on Nuclear Engineering, Beijing, China, May 16-20, 2005.
[15] B. Bollig and I. Wegener, "Improving the variable ordering of OBDDs is NP complete," IEEE Trans. Comput., Vol 45, pp. 993-1002, September 1996.
[16] S. J. Friedman and K. J. Supowit, "Finding the Optimal Variable Ordering for Binary Decision Diagrams," IEEE Transactions on Computers, Vol. C-39, No. 5, pp. 710-713, May 1990.
[17] N. Ishiura, H. Sawada, and S. Yajima, "Minimization of binary decision diagrams based on exchanges of variables," International Conference on Computer Aided Design, pp. 472-475, November 1991.
[18] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," International Conference on Computer Aided Design, pp. 42-47, November 1993.
[19] S. Panda, F. Somenzi, "Who are the variable in your neighborhood," International Conference on Computer Aided Design, pp. 74-77, November 1995.
[20] C. Meinel and A. Slobodova, "Speeding up variable reordering of OBDD," in Int. Conf. Comput. Design, pp. 338-343, 1997.
[21] R. Drechsler, W. Gunther, and F. Sornenzi, "Using Lower Bounds During Dynamic BDD Minimization," IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, Vol. 20, No. 1, January 2001.
[22] W.S. Jung, J.E. Yang, and J.J. Ha, "A New Method to Evaluate Alternate AC Power Source Effects in Multi-Unit Nuclear Power Plants," Reliability Engineering and System Safety, Vol. 82, pp. 165-172, 2003.