JAIDM
Journal of AI and Data Mining
Vol.1, No.1, 2013, 19- 25.

An improved algorithm for network reliability evaluation

M. Ghasemzadeh*

Electrical and Computer Engineering Department, Yazd University

Received 19 January 2013; accepted 14 February 2013
**Corresponding author: m.ghasemzadeh@yazd.ac.ir (M. Ghasemzadeh)*

Abstract
Binary Decision Diagram (BDD) is a data structure proved to be compact in representation and efficient in manipulation of Boolean formulas. Using Binary decision diagram in network reliability analysis has already been investigated by some researchers. In this paper, we show how an exact algorithm for network reliability can be improved and implemented efficiently by using a Colorado University Decision Diagram (CUDD).

Keywords: *Network Reliability, Efficienty, CUDD, Binary Decision Diagram.*

1. Introduction
In many systems, such as a computer or electricity networks the connectivity of the network components is of a great concern. Sometimes we are only interested in the connectivity of some components or connectivity of just two special components. In this paper, we review the properties of a Binary Decision Diagram [12,3] which is a modern data structure in representation and manipulation of Boolean formulas, then we see how a network reliability can be measured efficiently by using this data structure. We also consider the CUDD [16], which is a standard open source package for construction and the manipulation of BDD and its variants, such as ZDD, Zero-suppressed Binary Decision Diagram.
The network model is an undirected graph where vertices of the graph stand for the sites and the edges of the graph stand for the links between the sites. In practice each site or link can fail accidentally, but we suppose that sites (vertices) are perfect, but links may independently fail with some known probabilities. The problem of checking the connectivity is known to be NP-hard [1, 2].
There are two classes for computation of network reliability. The first class is for approximate computation while the second class is concerned with exact computation of network reliability computation. The existing algorithms in an exact computation are in two different categories: The first category deals with the enumeration of all the minimum paths or cuts. A path is defined as a set of network components in such a way that if these components are all failure-free, the system remains up. A path is minimal if it has no proper sub-paths. Conversely, a cut is a set of network components such that if any of the components fail, the system goes down. Using the enumeration method, one may only compute the reliability of networks consisting of a small number of components. In the second category, the algorithms are based on reducing the graph representing the network by removing some of its components.
These reductions allow us to compute the reliability in a simpler way[15], that is, decompose the problem into two sub-problems: The first one is assumed the component has failed, and the second one is assumed it functions. These reductions are recursively applied until it reaches very primitive instances. It is shown that the idea of eduction lets solve this problem more efficiently [17].
Binary Decision diagram is the state-of-the-art

data structure in Boolean formula representation and manipulation. It has been successfully used in VLSI CAD and widely integrated in commercial tools[11,4]. As a data structure for representation of Boolean functions it was first introduced by Lee and further popularized by Akers[13]. Bryant[4] introduced its restricted form OBDD (Reduced Ordered BDD), which is a canonical representation. He also proved that OBDDs allow efficient manipulations of Boolean formulas. This data structure and its variants can be implemented efficiently in modern computers using a programming language, such as C. CUDD (Colorado University Decision Diagram) Package, provided at the university of Colorado by Fabio Somenzi [16], is an open source package written in C. This package is known to be the most useful package for construction and manipulation of BDD and its variants.

Using BDD in the reliability analysis framework was first introduced by Madre and Coudert [5], and developed by Odeh and Rauzy [14]. In the network reliability framework, Sekine and Imai [10], and Trivedi [18] have shown how to functionally construct the corresponding BDD. Gary Hardy, Corinne Lucet and Nikolaos Limnios [9] improved existing techniques by using the concept of partitions of network nodes. They presented an exact algorithm for computing the K-terminal reliability of a network graph with perfect vertices.

The rest of this paper is organized as follows. First, we introduce BDD with an emphasis on its brilliant properties. In Section 3, we discuss about the network reliability problem and employ BDD in solving them. We give our CUDD based implementation for constructing the desired BDD in Section 4. Finally, we give conclusions in Section 5.

2. Binary decisions diagram and its variants

Binary Decision diagram (BDD) is the state-of-the-art data structure in Boolean formula representation and manipulation. They have been successfully used in VLSI CAD and widely integrated in commercial tools. In this section we review the basic definitions of BDD and learn about their theoretical and practical aspects. There are several extensions of BDD, of which we are interested in ZDD, shown to be more efficient in solving some related problems [6].

2.1. Definition and examples of BDD

Mostly BDD is meant to be an ordered BDD or OBDD. An OBDD is a graphic description of an algorithm for the computation of a Boolean function. The following definition describes the syntax of OBDD, i.e., the properties of the underlying graph. The semantics of OBDD, i.e., the functions represented by OBDD, are specified by the following definitions.

Definition 1: An OBDD G representing the Boolean functions $f^1,\ldots,f^m$ over the variables $x_1,\ldots,x_n$ is a directed acyclic graph with the following properties:

1. For each function f^i there is a pointer to a node in G.
2. The nodes without outgoing edges, which are called *sinks* or *terminal nodes*, are labeled by 0 or 1.
3. All non-sink nodes of G, which are also called *internal* nodes, are labeled by a variable and have two outgoing edges, a *0-edge* and a *1-edge*.
4. On each directed path in the OBDD, each variable occurs at most once as the *label of a node*.
5. There is a variable ordering π, which is a permutation of $x_1,\ldots,x_n$, and on each directed path the variables occur according to this ordering. This means, if x_i is arranged before x_j in the variable ordering, then it must not happen that on some path there is a node labelled by x_j before a node labeled by x_i.

In Figure 1, we draw sink nodes as squares and internal nodes as circles. We always assume that edges are directed downwards. 0-edges are drawn as dashed lines while 1-edges are drawn as solid lines. Figure 1 shows an OBDD G_f with the variable ordering x_1, x_3, x_2 and an OBDD G_g with the variable ordering x_1, y_1, x_0, y_0.

Definition 2: Let G be an OBDD for the functions $f^1,\ldots,f^m$ over the variables $x_1,\ldots,x_n$, and let $a=(a_1,\ldots,a_n)$ be an input. The *computation path* for the node v of G and the input a is the path starting at v obtained by choosing at each internal node labelled by x_i the outgoing a_i-edge.

Each node v represents a function f_v,

where $f_v(a)$ is defined as the value of the sink at the end of the computation path starting at v for the input a. Finally, f^j is defined as the function represented at the head of the pointer for f^j. Definition 2 can be seen as the description of an algorithm to obtain the computation path and therefore the value of $f^j(a)$ for each function f^j and each input a.

In the OBDD G_f in Figure 1, the computation path for the input $(x_1, x_2, x_3) = (1,1,0)$ passes from x_1 in the root, then from the right x_3, then from the right x_2 and finally goes to the 0-Sink. Furthermore, for each node v of G_f the function f_v represented at v is given. Definition 2 is easy to verify that the OBDD G_f in Figure 1, represents the function $f(x_1, x_2, x_3) = x_1 \oplus x_2 \oplus x_3$ and the OBDD G_g represents function $g(x_1, y_1, x_0, y_0) = (s_2, s_1, s_0)$, where (s_2, s_1, s_0) is the sum of the two 2-bit numbers (y_1, y_0) and (x_1, x_0).

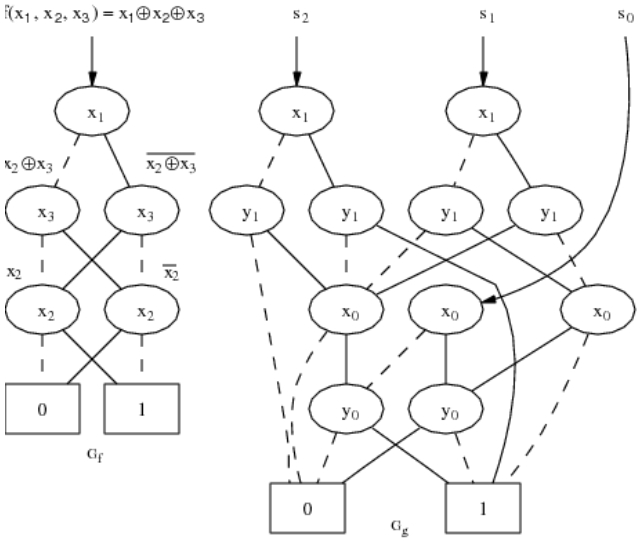

Figure 1. Examples of OBDDs.

The function represented at the sink labelled by $c \in \{0,1\}$ is the constant function c. Now let v be an internal node which is labelled by x_i. Let v_0 be the 0-successor of v, i.e., the node reached via the 0-edge leaving v, and let v_1 be the 1-successor of v. We consider the computation of f_v for some input. If in the input the value of x_i is 0, then by definition 2 we may obtain f_v by evaluating f_{v_0} and, if the value of x_i is 1, by evaluating f_{v_1}. This can be expressed by the equation:

$$f_v = \bar{x}_i . f_{v_0} \vee x_i . f_{v_1} \quad (1)$$

Using equation 1 we may compute the functions represented at the nodes of an OBDD in a bottom-up fashion. However, the opposite is also true. If a node v labelled by x_i represents the function f_v, then the 0-successor of v represents the subfunction (sometimes called cofactor) $f_{v|x_i=0}$ and the 1-successor the subfunction $f_{v|x_i=1}$. In other words, at v the function f_v is decomposed using *Shannons decomposition rule*:

$$f_v = \bar{x}_i . f_{v|x_i=0} \vee x_i . f_{v|x_i=1} \quad (2)$$

We point out that there are variants of OBDDs where Shannons decomposition rule is replaced by a different decomposition rule. Equation 2 shows that we can decompose the function f_v in different ways by choosing different variables x_i for decomposition. Hence, we may get different OBDDs for the same function if we use different variable orderings. Later on, we will see that the size of an OBDD usually depends strongly on the chosen variable ordering.

2.2. Synthesis

Synthesis is probably the most important operation, because it is needed in almost all applications. The usual way of generating a new BDD is to combine existing BDDs with connectives like AND, OR, EX-OR. If we want to make an OBDD for a given Boolean function. First, we make OBDDs for each variable of the Boolean function, and then we parse the Boolean function and combine the existing OBDDs to make OBDDs for the needed sub functions. Finally the OBDD representing the whole given Boolean function would be obtained. As suggested by Brace, Rudell, and Bryant [3], in OBDD packages, the synthesis algorithm is usually called an *ITE* ("if-then-else") where:

$$ite(f,g,h) = f.g \vee \bar{f}.h$$

The *ITE()* procedure receives OBDDs for two Boolean functions f and g, builds the OBDD for $f < op > g$. In fact, it receives three arguments: *I,T,E* which are OBDDs and returns the OBDD representing: $(I \wedge T) \vee (I \wedge E)$. All binary Boolean operations can be simulated by the ite-operator, e.g.: $f \vee g = ite(f,1,g)$, $f \wedge g = ite(f,g,0)$ or $f \oplus g = ite(f,g,g)$. *ITE()* is a combination of depth-first traversal and dynamic programming. (A recursive, Bottom-up procedure with tabulation). The basic idea of *ITE()* comes from the expansion theorem:

$$F < op > G = v(F_v < op > G_v) + v'(F_{v'} < op > G_{v'})$$

ITE() maintains a table called *Computed Table* to avoid computing the same combination repeatedly. It also maintains another table called *Unique Table* to avoid producing subgraphs representing the same sub-function. The benefit of this technique is the important result that *ITE()* becomes polynomial rather than exponential. Figure 2 displays the pseudocode for the *ITE* operator.

If f and g are given by OBDDs with different variable orderings, the *ITE()* procedure would not work, because for the simultaneous traversal, the variables have to be encountered in the same ordering in both OBDDs. In this situation the synthesis method would be much harder.

2.3. The variable ordering problem for OBDDs

OBDDs share a fatal property with all kinds of representations of switching functions: The representation of almost all functions need exponential space. Bryant [4] discovered that OBDD size strongly depends on the chosen variable ordering. Figure 3, shows the effect of variable ordering for a switching function. Notice that both OBDDs represent the same Boolean function: $F = (a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$.

```
ITE(f, g, h)
  {
    If (f == 1) return g;
    If (f == 0) return h;
    If (g == h) return g;
    if(
p=IN_COMPUTED_TABLE(f,g,h))
return p;
    v = TOP_VARIABLE(f, g, h );
    fn = ITE(fv0,gv0,hv0);
    gn = ITE(fv1,gv1,hv1);
    if(fn == gn) return gn;

if(!p=IN_UNIQUE_TABLE(v,fn,gn))
      p = CREATE_NODE(v,fn,gn);

NSERT_COMPUTED_TBL(p,HASH(
f,g,h));
    return p;
  }
```

Figure 2. The ITE algorithm for ROBDDs.

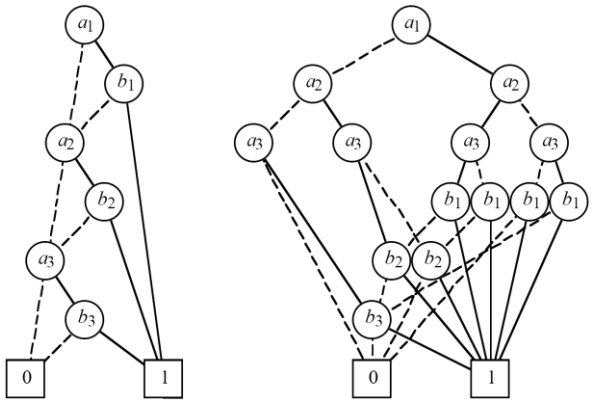

Figure 3. Effect of variable ordering on OBDD size (Bryant, 1986).

Different functions have different ordering sensitivities. Some functions have a high and others have a low variable order sensitivity. The practicability of OBDDs strongly depends on the existence of suitable algorithms and tools for minimizing the graphs in the relevant applications. There are many improvements, optimization algorithms, and additions to the basic OBDD model. It is known by experience that:

- Many tasks have reasonable OBDD

representations.

• Algorithms remain practical for up to 100,000 OBDD nodes.

• Most proposed heuristic ordering methods are generally satisfactory.

However, because of the practical applicability of this data structure, investigation and development of new optimization techniques for OBDDs is still a rewarding research topic.

3. Reliability measurment

As mentioned above, the exact method in evaluating the reliability can be achieved in two different methods. In this section we introduce these methods and discuss a how BDD can be used to gain considerable improvements.

3.1. Enumeration method

We consider the model used in most publications and used by Lucet and Limnios [9]. In this regard, a network model is an undirected stochastic graph $G=(V,E)$, where V stands for vertices representing sites, and E stands for edge set representing the links between the sites. Each edge e_i of the graph G is subject to failure with known probability $q_i (q_i \in [0,1])$. The probability that edge e_i functions can be obtained from $p_i = 1-q_i$. In the following, we consider the vertices as perfect. In classical enumerative methods, all the states of the graph are generated, evaluated as a failing state or as a functioning state, then probabilistic methods are used to compute the resulting reliability. Since there are two states for each edge, there are 2^m possible states for the graph G. Let X_i be the binary random variable state of the link e_i in G'', defined by $X_i = 1$ when link e_i is operational, and when $X_i = 0$ link e_i is down. $X = (X_1, X_2, \ldots X_m)$ is the *random network state vector*. A state x of G is denoted by $x = (x_1, x_2, \ldots x_m)$ where x_i stands for the state of edge e_i, $x_i = 0$ if e_i is down and $x_i = 1$ if it works. Probability of x is can be computed by:

$$Pr(X = x) = \prod_{i=1}^{m} (x_i . p_i + (1 - x_i) . q_i)$$

K-terminal network reliability is defined by:

$$R_k(p;G) = \sum_{x \, is \, a \, functioning \, state} Pr(X = x)$$

Because of exponential number of states, if classical methods are applied, the complexity would be $O(m.2^m)$. So, these methods are not applicable in large networks.

3.2. Graph reduction

In order to avoid drawbacks of the enumeration method, Lucet and Limnios [9] define two graph operations: The *edge deletion*, and the *edge contraction*. $G=(V,E)$ is a given graph such that there is an edge $e_i \in E$. G_i is to be the subgraph obtained from G by deleting $e_i, (G\_i = G \setminus e_i)$. If $e_i = (x, y)$ such that $x, y in V$, then edge contraction consists of merging vertices x and y in one single vertex. We denote $G_* i$ to represent the subgraph obtained from G by contracting e_i. When edge e_i fails, the network behavior is equivalent to G_i; and when functions, the network behavior is equivalent to $G_8 i$. According to this decomposition the following result is emerged and could be applied recursively:

$$R_k(p;G) = p_i . R_k(p;G_{*i}) + q_i . R_k(p;G_{-i})$$

if we consider e_i and e_j as two edges of E then $G_{*_{i-j}}$ means subgraph obtained by contracting e_i and deleting e_j. Figure 4 shows how this idea works.

4. Employing BDD in encoding and evaluation

We can learn from the ITE() operation on BDDs that although in its primitive form it is exponential, by embedding the idea of tabulation, its complexity has been reduced to quadratic. When we look at the algorithm of evaluating network reliability based on reducing the graph representing the network by removing some of its components (decompose the problem into two sub-problems) we can realize that the same technique and also be employed for this problem to gain similar benefits. In this section, we compute network reliability by taking advantage of the BDD data structure.

The mystery of BDD is merging equivalent subfunctions of a Boolean formula to get compact representation of the entire formula. All main operations on BDD, such as the ITE() function, perform this kind of merging in a recursive manner and in a systematic way to get the most benefits of it, while preserving the canonicity of representation. In has been shown that the recursive network reliability relation in BDD can

be formulated as:

$$\forall i \in [1...m]: \; R_k(p;G) = Pr(f = 1)$$

$$R_k(p;G) = p_i.Pr(f_{X_{i=1}} = 1) + q_i.Pr(f_{X_{i=0}} = 1)$$

We obtain values of $f_{X_{i=0}}$ and $f_{X_{i=1}}$ recursively untill it reaches the sink nodes. The probability is stored in each internal node. We may use the CUDD package to implement the corresponding algorithm and operations.

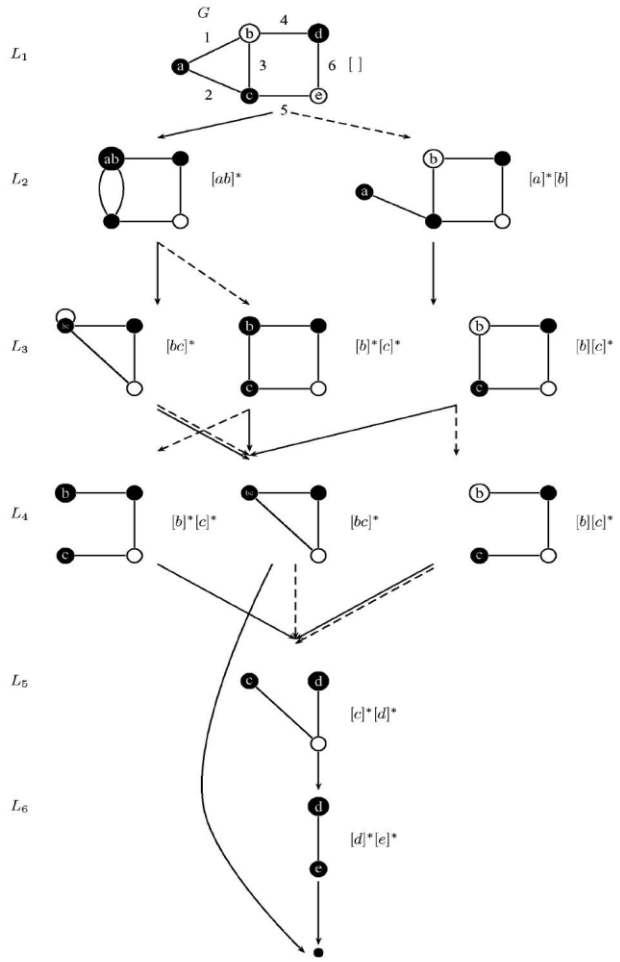

Figure 4: Network decomposition based on edge deletion/contraction (Lucet and Limnios).

4.1. Implementation in the CUDD Package

In this section, we introduce the CUDD package, which is known to be the best open source package for manipulating the BDD and its variants. Content of this subsection is prepared very briefly from its help manual [16].

The CUDD package provides functions to manipulate Binary Decision Diagram (BDD), Algebraic Decision Diagram (ADD), and Zero-suppressed Binary Decision Diagram (ZDD). BDDs are used to represent switching functions; ADDs are used to represent function from $\{0,1\}^n$ to an arbitrary set. ZDDs represent switching functions like BDDs; however, they are more efficient than BDDs when the functions to be represented are characteristic functions of cube sets, or in general, when the ON-Set of the function to be represented is sparse. They are inferior to BDDs in other cases.

The CUDD package can be used in three ways:

- As a black box. The application program that needs to manipulate decision diagrams only uses the exported functions of the package. The rich set of functions included in the CUDD package allows many applications to be written in this way. An application written in terms of the exported functions of the package needs not concern itself with the details of variable reordering, which may take place behind the scenes.
- As a clear box. When writing a sophisticated application based on decision diagrams, efficiency often dictates that some functions be implemented as direct recursive manipulation of the diagrams, instead of being written in terms of existing primitive functions.
- Through an interface. Object oriented languages like C++ can free the programmer from the burden of memory management. A C++ interface is included in the distribution of CUDD. It automatically frees decision diagrams that are no longer used by the application. Almost all the functionality provided by the CUDD exported functions is available through the C++ interface, which is especially recommended for fast prototyping.

Figure 5 shows the main procedure in CUDD for the algorithm of evaluating the probability of network reliability.

The decomposition shown in Figure 3 can be mapped into BDD construction. Its root corresponds to the original network graph and in each level, one edge is deleted or contracted. Children of a node in each level represent sub-graphs obtained by successive edge deletion or edge contractions.

```
float ComputeNetRel( DdNode *DecBDD )
        {
         float Rk;

         if ( DecBDD==CuddOne) return 1;
         if ( DecBDD==CuddZero) return 0;
         if ( Rk = Computed(DecBDD) )
          return(Rk);
         Pr1 = CaculReliability(
          Cudd_T(DecBDD));
         Pr0 = CaculReliability(
          Cudd_E(DecBDD));
         Rk = (1-q[i]) * Pr1 + q[i] * Pr0;
         InsertComputed( DecBDD , Rk);
          return(Rk);
        }
```

Figure 5. Network reliability evaluation / part of cudd code.

5. Conclusion

Two exact methods for evaluation of network reliability were discussed. We saw how by inspiration from the ITE() operator in BDD construction, an algorithm with lower complexity for evaluation of network reliability can be obtained. In fact employing a variant of BDD called ZDD can lead to even more advantages.

References

[1] Ball, M. O. (1980). Complexity of network reliability computations. Networks, 10,153-165.

[2] Ball, M. O. (1986). Computational complexity of network reliability analysis:An overview. IEEE Trans. Reliability. R-35:230-239.

[3] Karl, S. (1990). Brace and Richard L. Rudell and Randal E. Bryant. Efficient Implementation of a BDD Package. DAC: Design Automation Conf. 40-45.

[4] Randal, E. (1986). Bryant. Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers. 35(8),677-691.

[5] Coudert, O. and Madre, J. C. (1992). Implicit and incremental computation of primes and essential primes of Boolean functions. Proceedings of the 29th ACM/IEEE Design Automation Conference. 36-39.

[6] Drechsler, R. and Sieling, D. (2001). Binary decision diagrams in theory and practice. International Journal on Software Tools for Technology Transfer (STTT). 3(2),112-136.

[7] GhasemZadeh, M., Klotz, V. and Meinel, C. (2004). Embedding Memoization to the Semantic Tree Search for Deciding QBFs. Australian Conference on Artificial Intelligence. 681-693.

[8] GhasemZadeh, M. and Meinel, C. (2005). Splitting Versus Unfolding. 7th International Symposium on Representations and Methodology of Future Computing Technology, Tokyo, Japan, 2005.

[9] Hardy, G., and Lucet, C. and Limnios, N. (2007). K-Terminal Network Reliability Measures With Binary Decision Diagrams. IEEE Transactions on Reliability. 56(3), 506-515.

[10] Imai, H., Sekine, K. and Imai, K. (1999). Computational investigations of allterminal network reliability via BDDs. IEICE Transactions on Fundamentals. E82-A(5),714–721.

[11] Christoph Meinel and Jochen Bern and Anna Slobodová. Efficient OBDD-Based Boolean Manipulation in CAD beyond Current Limits. Design Automation Conf.(DAC), pp 408-413, San Francisco, CA, 1995.

[12] Meinel, C. and Theobald, T. (1998). Algorithms and data structures in VLSI design: OBDD - foundations and applications. Berlin, Heidelberg. New York: Springer-Verlag.

[13] Moret, B. (1982). Decision trees and diagrams. Computer Survey. (14), 593-623.

[14] Rauzy, A. (2003). A new methodology to handle Boolean models with loops. IEEE Trans. Reliability. R-52(1),96-105.

[15] Satyanarayana, A. and Chang, M. K. (1983). Network reliability and the factoring theorem. Networks, 13,107-120.

[16] Somenzi, F. CUDD Package. ftp://vlsi.colorado.edu/pub/.

[17] Theologou, O. and Carlier, J. (1991). Factoring and reductions for networks with imperfect vertices. IEEE Trans. R-40, 210-217.

[18] Zang, X., Sun, H. and Trivedi, K. (1999). A BDD-based algorithm for reliability evaluation of phased mission systems. IEEE Trans. Reliability. R-48(1),50-60.