Efficient extraction of the structure formula from reliability block diagrams with dependent basic events

M Pock[1]* and **M Walter**[2]
[1]Arts et Métiers, Paristech, Metz, France
[2]Lehrstuhl für Rechnertechnik und Rechnerorganization, Technische Universität München, Munich, Germany

The manuscript was received on 15 October 2007 and was accepted after revision for publication on 22 April 2008.

DOI: 10.1243/1748006XJRR139

Abstract: Traditional solution methods for fault trees or reliability block diagrams can only be applied as long as the failure and repair events of all components of the system are stochastically independent. In this paper, an efficient algorithm is presented based on binary decision diagram-like data structures that are able to evaluate systems including intercomponent dependencies between arbitrary components. The algorithm, which was integrated into the software package OpenSESAME, is shown to be as good as traditional algorithms for the special case of independent components. The performance of the algorithm in the more general case was empirically analysed using some real-world examples. Very short running times and a moderate memory consumption were observed.

Keywords: availability modelling, inter-component dependencies, binary decision diagrams (BDDs), Shannon decomposition

1 INTRODUCTION

The availability of fault tolerant systems can be estimated by the use of stochastic models, which calculate the availability of the system from the properties of its components. Two types of stochastic model are distinguished. The first type is so-called combinatoric models such as reliability block diagrams (RBDs) or fault trees (FTs). Combinatoric models are very intuitive and easy to handle, but they usually imply that there are no dependencies between the failure and repair events of the components. Consequently, it is not possible to model important system properties such as common cause failures, limited repair personnel, failure propagation, imperfect failure-detectors, and fail-over times. This may lead to overoptimistic results. The second type of model is state-based models (SBMs) such as Markov chains, stochastic process algebras, or stochastic Petri nets. They can handle dependencies, but lack usability. In comparison to FTs or RBDs, they are not very intuitive, do not support modular or hierarchic models, and are hard to modify [**1**].

**Corresponding author: A3SI, Arts et Métiers, 4, Rue Augustine Fresnel, Paristech, F-57070 Metz, France. email: Michael.POCK-3@etudiants.ensam.eu*

To overcome this problem, the authors have developed the tool OpenSESAME, which allows for an inclusion of intercomponent-dependencies into combinatorial availability models (see section 2.2 of this paper). Several other approaches exist, based on similar principles (see section 6). Because traditional solution methods cannot be applied in all cases, the models are automatically transformed into an SBM. This usually involves two steps:

1. An SBM structure (e.g. a Markov chain or a Petri net) is created, representing the behaviour of the components of the system. The structure and the number and kind of its states depends on the intercomponent dependencies.
2. A Boolean reward function is derived from the RBD or FT that tells the solver of the SBM which combinations of failed components result in an available or unavailable system respectively. A reward function is a transformation of a Boolean expression especially for SBMs where the Boolean variables are equated with certain states in the SBM.

In practice, the size of the state-space grows exponentially with the number of components. It is therefore better to create not a single SBM, but several models,

one for each so-called set of interdependent components (SICs). The components are grouped into SICs in such a way that there are no dependencies between two components, if they are not in the same SIC.

As a result, several small SBMs can be solved instead of a single large one, which greatly reduces the computational demands. However, this also implies that the reward function must be divided according to the SIC decomposition of the model.

In this paper, an algorithm is proposed and evaluated that creates a symbolic representation of the structure formula from an RBD and decomposes the formula in accordance with a given set of SICs. Obtaining the SICs and creating the state-based model, as well as solving the SBM, is outside the scope of this paper as these issues are described in reference [**2**] and related work (section 6).

The paper is structured as follows: section 2 gives an overview on RBDs and the tool OpenSESAME and contains a more formal problem statement. The proposed algorithm and its implementation are described in sections 3 and 4. An evaluation of the proposed algorithm can be found in section 5. Section 6 contains the related work, and the paper is summarized in section 7.

2 PRELIMINARIES

2.1 Reliability block diagrams

RBDs are undirected graphs whose edges are labelled with components of which the system consists (Fig. 1). In the general case investigated here, a component may occur more than once in the graph. There are no limitations on the structure of the graph. In particular, the authors do not restrict themselves to series-parallel diagrams and also allow 'bridges' in the graph. There is a pair of two special nodes *s* and *t*, called the 'terminal nodes'. If there is a path between *s* and *t* through the RBD that contains only edges labelled with available components, the whole system is available; otherwise it is unavailable.

An RBD can be transformed into a Boolean expression by finding all simple paths from *s* to *t*. A simple path from *s* to *t* is a set of edges which connects *s* with *t* and does not contain an edge more than once. Finding these paths is an NP-complete problem [**3**]. The probability that this Boolean expression is true equals the availability of the system. Determining this probability implies determining the satisfiability of the function and is NP-complete, too.

In practice, systems are often available only if at least *k* out of *n* subsystems are available (or good) at the same time. Using traditional RBDs, $\binom{n}{k}$ parallel paths were required to describe such a system. To avoid such a large number of edges, RBDs are

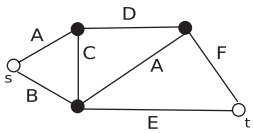

Fig. 1 A reliability block diagram (RBD)

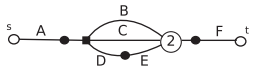

Fig. 2 An RBD containing a 2-out-of-3: G edge with the subsystems: *B, C,* and $D \wedge E$

therefore often extended by so-called *k*-out-of-*n*: G edges as shown in Fig. 2.

2.2 OpenSESAME

The 'simple but extensive, structured availability modelling environment' allows for an easy-to-use modelling of high-availability systems including intercomponent dependencies. The redundancy structure of the system is specified by a monotone RBD. In the editor of OpenSESAME, RBD-edges may be labelled with sub-RBDs instead of single components. This allows for a hierarchic arrangement of RBDs and avoids RBDs which become too large to be readable. Furthermore, in top-level RBDs, more than one pair of terminal nodes may be specified to allow for a simultaneous evaluation of several availability measures.

Intercomponent dependencies can be specified in several ways. First, non-zero fail-over times stemming from fault detection and reconfiguration of stand-by redundant components can be specified at the *k*-out-of-*n*: G edges. For dependencies concerning the failure behaviour of the components, OpenSESAME provides so-called 'failure dependency diagrams'. They can be used to define failures with a common cause, different kinds of failure propagation, imperfect fault detection, and so on. Finally, by defining repair groups, the modellers may specify which components are repaired by a group of repair personnel and how many repairs can be performed in parallel by each group.

For a quantitative analysis of the model, it is transformed into a set of state-based models. Alternatively, either stochastic Petri nets or process-algebraic models are created. The Petri nets are solved by the tool DSPNexpress [**4**] whereas CASPA [**5**] is used to analyse the algebraic models. Both steady-state and transient availabilities can be computed. More information on the solution process can be found in reference [**1**].

2.3 Formal statement of the problem

A given set of components *C* is classified into disjunct sets of interdependent components (SICs). Let a_i be

the probability that component i is available. For n components $i_1, i_2, \ldots, i_n$ from n different SICs the probability $a_{i_1 \wedge i_2 \wedge \ldots \wedge i_n}$ that all components are available at the same time is $a_{i_1} \cdot a_{i_2} \cdot \ldots \cdot a_{i_n}$ as all components are jointwise independent. It is further assumed that $a_{j_1 \wedge j_2 \wedge \ldots \wedge j_m}$ is known for all cases where $j_1, j_2, \ldots, j_m$ belong to the same SIC. In practice, this probability is computed by the state-based solver. Similarly, a_i can be computed for every component i.

Furthermore, a monotone RBD with terminal nodes s and t is given whose edges are labelled with components from C. The RBD may contain k-out-of-n: G edges. What is the terminal pair availability of the RBD?

3 THE MERGE-AND-DELETE ALGORITHM

3.1 Overview

In this section, an efficient algorithm to extract the Boolean expression from a general RBD is proposed. The expression stores all simple paths of the RBD by using a so-called edge expansion diagram (EED). In contrast to, for example, storing the term in conjunctive normal form (CNF), the EED avoids an explicit enumeration of all paths or cut sets and is therefore much more efficient in the average case.

The proposed algorithm, called Merge-and-Delete, is a generalization of the work proposed in reference [**6**]. There, EED were defined as directed, acyclic graphs. After executing the Merge-and-Delete algorithm, the nodes of the EED contain RBDs (Fig. 3) and their related Boolean expressions (Fig. 4), whereas the edges are labelled with Boolean variables. An EED is constructed by step-wisely contracting the original RBD. To simplify the algorithm, it is first assumed that there are no k-out-of-n: G edges in the RBD. The algorithm is extended to RBDs including k-out-of-n: G edges in section 3.4.

3.2 Creating the EED

At the beginning of the algorithm, an EED-node attributed to the original RBD is created. This node is referred to as the 'root' of the EED. One hash table is also constructed for the whole EED that maps the components to a set of respective edges in the root-RBD. This hash table is referred to as the 'component index'. A second hash table is created which maps the RBDs to EED-nodes. At the beginning, there is only one entry, mapping the original RBD to the root. This hash table is called the 'node map'. The algorithm works as follows:

1. The contraction is started at s. Assume that the component called A of an edge adjacent to s is available. In this case, it is possible to go from s to the other end of this RBD-edge. Both RBD-nodes can be merged; the edge in between is included in a possible simple path. If the RBD has more edges containing A, their adjacent RBD-nodes will be merged, too. These edges can be found in constant time using the component index created before. During the search for simple paths, it is not allowed to include a node of the RBD in a path twice, so all the other edges adjacent to s can be deleted.
2. Slings and inner nodes with degree one in the RBD are deleted because they cannot be part of a simple path, either.

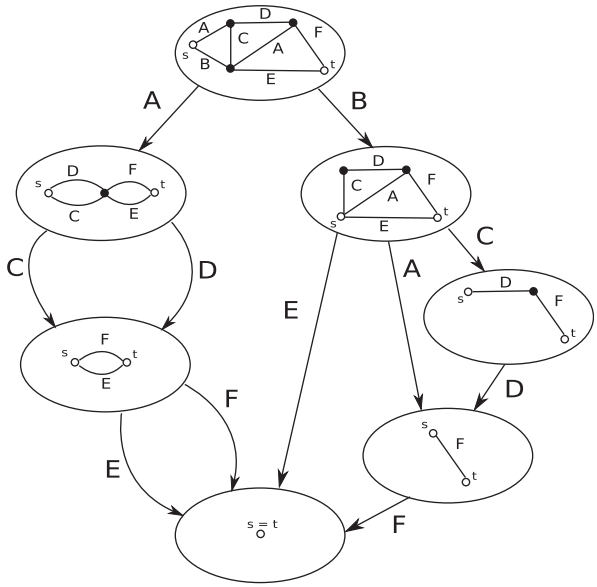

Fig. 3 The construction of the EED

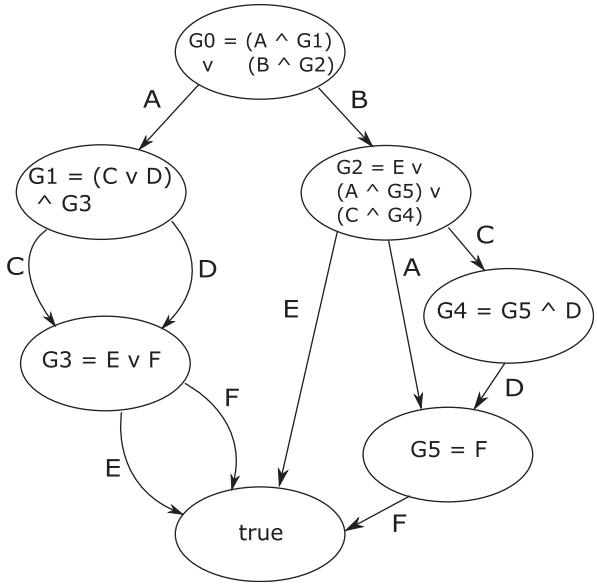

Fig. 4 The extraction of the Boolean expression out of the EED

3. The node map is checked for the modified RBD. In this case, the EED-node found in the map is used. Otherwise, a new EED-node is created, containing the modified RBD. The newly created RBD is put in the node map, with the RBD used as key. Then the EED-node that was found or created is linked with the root, using an edge attributed with A.

This is repeated for all components that are part of the edges adjacent to s. Up to $deg(s)$ new EED-nodes will be created, all linked with the root.

The algorithm is repeated on the newly created EED-nodes. It stops when all edges in the RBD have been removed or both border nodes have been merged. Therefore, the EED has one leaf containing only one border node. Figure 3 shows the algorithm applied to the example RBD of Fig. 1.

3.3 Extraction of the symbolic expression

After creating the EED, the Boolean expressions in the nodes have not been set yet. This is done by beginning with the leaf of the EED; it will be set to *true*. The expression of other nodes can be obtained using the method getExpr(EEDNode n). This algorithm uses several other methods and attributes. The attribute *nodeExpr* stores the expression of this node after it has been calculated once. The method *getComp*() returns an expression that consists just of the component of an EED-edge. The method *getTarget*() returns the target of an EED-edge.

```
Method getExpr(EEDNode n):
01) if nodeExpr != null
02)     return nodeExpr
03) Expression expr = false
04) for all outgoing EEDEdges e_i of n do:
05)     Expression A_i = e_i.getComp()
06)     EEDNode c_i = e_i.target()
07)     Expression G_i = getExpr(c_i)
08)     expr = expr ∨ (A_i ∧ G_i)
09) end for
10) nodeExpr = expr
11) return expr
```

By calling getExpr(*root*) the expressions of all EED-nodes are calculated recursively. The result will not be an operator tree. Instead, a directed acyclic graph (DAG) will be constructed automatically which normally needs less memory than a corresponding tree, as the same subexpressions are stored only once. Figure 4 shows the example EED with attributed expressions.

3.4 *k*-out-of-*n* edges

In the subsections 3.2 and 3.3 it was assumed that there are no k-out-of-n: G edges in the RBD. One way to treat these edges is to replace them by $\binom{n}{k}$ parallel paths each containing k components in the series. However, for the special case that

(a) each configuration of the k-out-of-n: G edge is a single component or a series system, and
(b) none of the components of the k-out-of-n: G edge appears somewhere else in the RBD,

a more efficient solution exists. For this case, the Boolean algebra is extended to be able to describe these edges in one simple expression called 'k-out-of-n: G expression'. Let $A_1, \ldots, A_n$ be Boolean expressions representing series systems, k and n natural numbers. Then a k-out-of-n: G expression is written as follows

$$k \text{ out of } n : G[A_1, A_2, \ldots, A_n]$$

This kind of expression is true if and only if at least k of the n given subexpressions are true. With these expressions, k-out-of-n: G edges can be solved easily in the EEDs. If the Merge-and-Delete algorithm hits a k-out-of-n: G edge, the expression analogous to the edge is created. This expression will be used like the Boolean variable for normal edges. A small example is shown in Fig. 5. The resulting Boolean expression for this RBD $A \vee (2 \text{ out of } 3 : [B, C, D \vee E]) \vee F$. Note that this action only brings a benefit if the SBM-solver is capable of calculating the probability of this expression efficiently, i.e. it must support arbitrary algebraic terms as reward functions and must not be restricted to, for example, the Boolean algebra.

4 DECOMPOSITION OF THE EXPRESSION

The next step after gaining the structure formula from the RBD in the form of an attributed EED is to derive the reward functions for the individual SICs. For each SIC, an SBM is created which has to be attributed with the respective reward functions. Finally, the computed results for each function are used to compute the overall availability of the system.

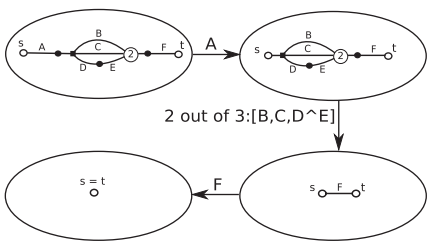

Fig. 5 An EED for an RBD with a k-out-of-n: G edge

4.1 Concept

Thus, the next step is to decompose the redundancy structure into independent subexpressions. For example, assuming that the system redundancy structure ϕ (gained from an RBD not shown in this paper) is defined by

$$\phi = (A \vee B) \wedge (C \vee D \vee E) \wedge (A \vee F)$$

Assuming further that there are three SICs, one containing A and B, the second containing C and D, the third one E and F. By applying the 'Shannon Decomposition' twice, on variables A and B, it obtains

$$\begin{aligned}\phi = {} & (A \wedge B) \ \wedge \ \phi_{A=true,\,B=true} \ \vee \\ & (A \wedge \overline{B}) \ \wedge \ \phi_{A=true,\,B=false} \ \vee \\ & (\overline{A} \wedge B) \ \wedge \ \phi_{A=false,\,B=true} \ \vee \\ & (\overline{A} \wedge \overline{B}) \ \wedge \ \phi_{A=false,\,B=false}\end{aligned}$$

The terms on the right contain neither the variable A nor B any more, as these were substituted by constants. Because the subsystems AB and CDEF are stochastically independent and all conjunction terms are disjoint, it holds

$$\begin{aligned}P(\phi) = {} & P(A \wedge B) \cdot P(\phi_{A=true,\,B=true}) \\ & + P(A \wedge \overline{B}) \cdot P(\phi_{A=true,\,B=false}) \\ & + P(\overline{A} \wedge B) \cdot P(\phi_{A=false,\,B=true}) \\ & + P(\overline{A} \wedge \overline{B}) \cdot P(\phi_{A=false,\,B=false}) \\ = {} & P(A \wedge B) \cdot P(C \vee D \vee E) \\ & + P(A \wedge \overline{B}) \cdot P(C \vee D \vee E) \\ & + P(\overline{A} \wedge B) \cdot P\big((C \vee D \vee E) \wedge F\big) \\ & + P(\overline{A} \wedge \overline{B}) \cdot P(\mathit{false})\end{aligned}$$

The decomposition can be continued analogous on the expressions $C \vee D \vee E$ and $(C \vee D \vee E) \vee F$.

The whole process can be easily generalized. Let $SD(\phi, SIC_i)$ be the Shannon Decomposition for the SIC SIC_i applied on the expression ϕ. For an expression ϕ with n SICs, the decomposition for all SICs can be described as follows

$$SD(\ldots\{SD[SD(\phi, SIC_1), SIC_2], \ldots\}, SIC_n)$$

The result is an algebraic term with several probabilities as its operands. As every probability contains only variables from one SIC, it can be associated with the respective SBM as a reward function. After the evaluation of the SBM, the probabilities are known and the overall availability can be computed.

4.2 Efficient implementation

The decomposition can be represented graphically by a decision tree whose size grows exponentially with the number and the size of the SICs. The tree of the example above would need 17 nodes. In larger trees, many of these nodes would be equal, leaving room for optimization. For instance, even in the small example mentioned before, the expression $C \vee D \vee E$ appears twice.

A well-known technique to efficiently handle binary trees is reduced ordered binary decision diagrams (ROBDDs) [**7**]. As this tree is not binary, this technique cannot be applied directly; the ROBDDs have to be generalized. This kind of decision diagram is called a 'multiple binary decisions diagram' (MuBDD) throughout this paper, as several binary decisions are made in each step, depending on the size of the SIC. The MuBDD for the given example is shown in Fig. 6.

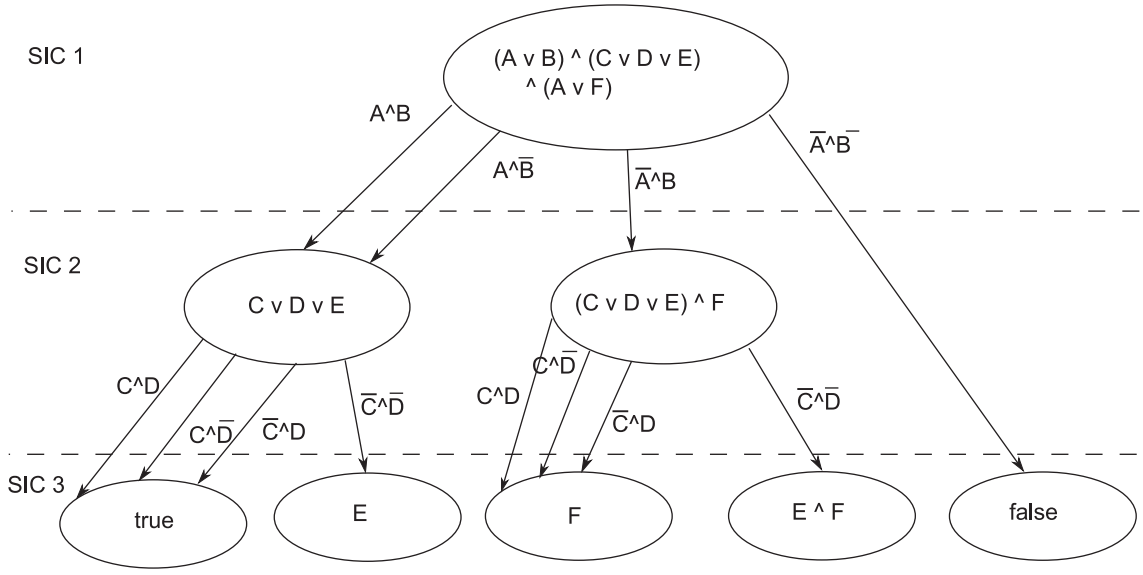

Fig. 6 The MuBDD for $\phi = (A \vee B) \wedge (C \vee D \vee E) \wedge (A \vee F)$

A MuBDD is, again, a directed acyclic graph. A node v contains a Boolean expression, stored as an EED, which can be manipulated by substituting the Boolean variables with constants. For every node, only the variables of one SIC are substituted at a time. For a SIC with n variables, 2^n different substitutions exist. For every possible substitution, an outgoing edge is added. The substitution is stored in the edge as a conjunction term of the respective variables. This edge leads to a node u which is attributed to the Boolean expression obtained by substituting the variables. In most cases, such a node already exists, and the edge will be connected to an already existing node. To check whether such a node exists, the nodes are saved in a hash table, where the expressions are used as keys. If no node with the appropriate expression can be found, a new node is created and the hash table is updated.

Two minimization rules – analogous to the minimization rules of regular ROBDDs – can be applied to the MuBDD, which are shown in Fig. 7. The first rule is that two nodes are merged, if they are equivalent. This rule is enforced implicitly by the hash table. The second rule is for nodes, one of them called v with outgoing edges that all lead to the same target node u. v can be deleted, its incoming edges being linked with u instead.

Calculating the overall probability can be done recursively. Firstly, the probabilities of the expressions stored in the leafs are calculated by the SBM-solver, or simply set if they are trivial. The results are stored in the nodes. For the remaining nodes, probabilities of the expressions stored in the outgoing edges are computed. As all variables in these expressions are elements of the same SIC, the SBM-solver is used to do that. Then the probability of every edge is multiplied with the probability of the node at the end of the edge. The sum of these products yields the probability of the node.

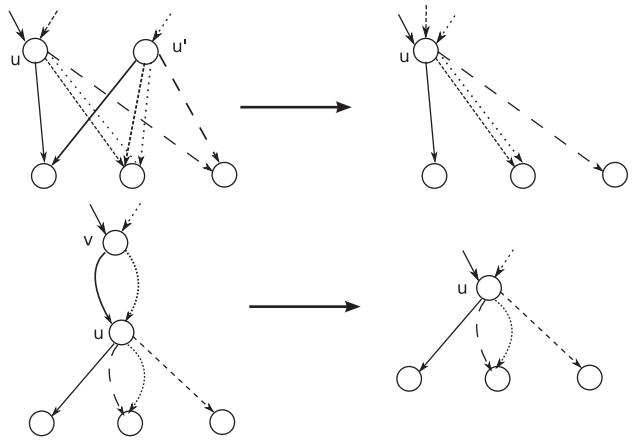

Fig. 7 Minimization of a MuBDD

For example, the probability of the expression in the root of the MUBDD shown in Fig. 6 is obtained by adding four summands, one for each outgoing edge

$$\begin{aligned} P(\phi) = {} & P(A \wedge B) \cdot P(C \vee D \vee E) \\ & + P(A \wedge \overline{B}) \cdot P(C \vee D \vee E) \\ & + P(\overline{A} \wedge B) \cdot P[\,(C \vee D \vee E) \wedge F\,] \\ & + P(\overline{A} \wedge \overline{B}) \cdot P(\mathit{false}) \end{aligned}$$

This is equivalent to the decomposition shown before. The difference to a naive implementation is that $P(C \vee D \vee E)$ is computed only once, so less time and memory are needed.

4.3 The heuristics

An important issue for ROBDD data structures in general is choosing a good variable ordering. Likewise, for MuBDDs, a good ordering for the SICs must be chosen. By experiments, the following heuristic was determined to deliver the best results in most cases:

1. The smaller SICs are handled first.
2. If two or more SICs have the same size, the SIC with most component appearances in the expression is chosen first.
3. If there are two or more SICs with the same size and the same number of component appearances, a breadth-first ordering is used.

There are several reasons for these heuristics. Take the smallest SICs first, as the number of children of the corresponding MuBDD-node grows exponentially. The calculation time and the required memory to create a node for a SIC of size n is $O(2^n)$, as 2^n outgoing edges have to be created. The only exception are the leaves – children do not have to be created there; this means that the required calculation time for creating the node is $O(1)$. Instead, the remaining expressions can be calculated directly by the SBM. It therefore makes sense to solve the largest SIC as the last one, to spare the most time and memory.

While substituting certain variables, it can happen that the whole expression is reduced so much that it can be solved trivially. The probability for this event is higher, if variables are substituted at more occurences. This is the reason for choosing the second rule. With this rule, big, unnecessary branches of the MuBDD must be cut off as soon as possible.

If there are SICs that have the same size and the same number of component occurences, a breadth-first search (BFS)-ordering [**8**] is taken to choose the sequence of these SICs. To make a BFS-order of the SICs a BFS-order of all edges is made. After that every SIC is given the minimum order of all its edges. This is a good choice as many practical systems are,

in general, series systems with parallel subsystems. If there is a serial system of parallel subsystems, it is clear that the whole system is unavailable if only one subsystem is unavailable. With a BFS-ordering, this will be detected very soon and can cut off large branches of the MuBDD earlier.

5 EVALUATION AND COMPARISON

From a theoretical point of view, the algorithm has to look at the whole RBD and the whole Boolean expression, so will at least need linear time and space. For the best case, a complexity of $O(n)$ is expected, where n is the number of components in the RBD.

In the worst case, it is possible that there are no already-existing nodes while constructing the EED and the MuBDD. For these cases, the EED and the MuBDD will be real trees instead of directed acyclic graphs. Therefore, the worst-case complexity is $O(2^n)$.

In this section are presented the results obtained by measurements when applying the proposed algorithm to several examples.

5.1 Examples with independent components

Firstly, the authors compare examples without intercomponent dependencies and compare the results with the results of reference [**6**]. For these examples, outlined in Fig. 8, the inner nodes of the MuBDDs always have two outgoing edges, which means that the MuBDD is a traditional ROBDD. The authors want to show that their algorithm is as efficient as the known algorithms for examples without dependencies. The comparison is restricted to the number of EED- and MuBDD/ROBDD-nodes as there was no access to the code of reference [**6**], and its runtime and memory requirements could not be measured on modern hardware.

Table 1 shows the number of EED- and MuBDD-nodes in our implementation, and compares it with the number of EED- and ROBDD-nodes given in reference [**6**] with a breadth-first-ordering of components. For most examples, the number of EED-nodes is the same; in two examples it is even lower. Likewise, the number of MuBDD/ROBDD-nodes is nearly the same for both implementations. The discrepancies result mainly from the different heuristic ordering used. Overall, both versions can handle these kinds of example efficiently.

5.2 Models including dependencies

The main goal of the algorithm proposed in this paper is the possibility to handle models which include intercomponent dependencies. To analyse the efficiency of the algorithm in that case, two

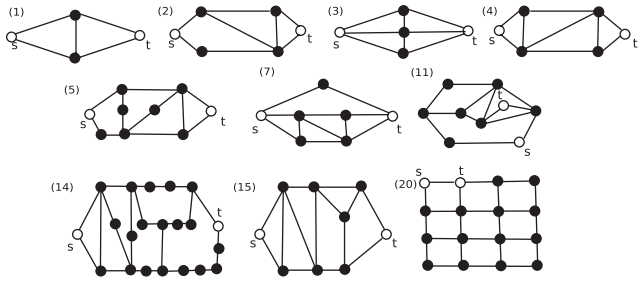

Fig. 8 Some examples taken from reference [**6**]. Every edge contains its own component

Table 1 The number of nodes in our implementation compared with the number of nodes from reference [**6**]

	Proposed algorithms		KLY99	
Example	EED-nodes	MuBDD-nodes	EED-nodes	ROBDD-nodes
KLY-1	5	11	5	10
KLY-2	9	16	9	15
KLY-3	8	29	8	26
KLY-4	11	23	11	22
KLY-5	16	29	16	50
KLY-7	12	28	17	51
KLY-11	20	41	20	48
KLY-14	45	99	45	126
KLY-15	21	42	21	40
KLY-20	76	146	85	177

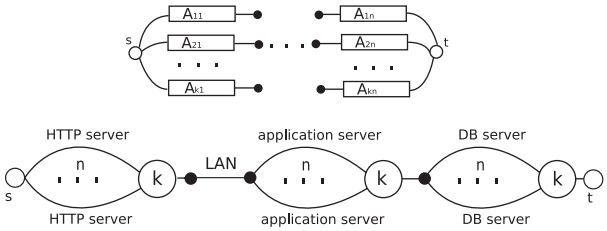

Fig. 9 RBDs of ParSer(k,n) (above) and Web(k,n) (below)

examples were analysed with a varying number of components. The first example, ParSer(k,n), is a parallel system of k serial subsystems, each containing n edges (Fig. 9). The system comprises $k \cdot n$ components, $A_{11}, \ldots, A_{kn}$, every component appears exactly once in the redundancy structure. Two versions of this example are investigated. In version A, there are k SICs, each containing all components of one path. In version B, there are n SICS, each SIC containing one component from each path.

The second example, Web(k,n), is a model of a fault-tolerant web server previously published in reference [**9**]. The server consists of n HTTP servers, n application servers, and n database servers. The servers are connected by a network. The whole system is available if at least k servers of every type are working properly and if the network is available. There are

four SICs, one for each type of server and one for the network. The RBD for this example is shown in Fig. 9.

Table 2, illustrated by the left side of Fig. 10, shows the calculation time and the number of nodes in the MuBDD for ParSer(k,n) versus the parameter k. Regarding the EEDs, there is no difference between the versions A and B. For constructing the EED, only the structure of the RBD is important; the dependencies do not have an influence.

Regarding the size of the MuBDDs, there is a significant difference between versions A and B. For versions A, the number of nodes and the calculation time grows linearly with k. In this case, the complexity of the algorithm is perfect. However, the calculation time and the number of MuBDD-nodes in version B grow exponentially. In general it holds that systems where the dependent components are concentrated on local subsystems have a much better performance than systems where the dependent components are spread widely over the whole RBD. The SICs influence the structure of the MuBDDs very strongly, as they define which variables have to be substituted at the same time.

Table 3, illustrated by the right side of Fig. 10, shows the measured times for the creation of the EEDs, MuBDDs, and the overall runtime for the example Web(k,n). While the time for the EEDs and the MuBDDs is constant, the overall computation time grows exponentially with n. The reason for this is that the parameters k and n do not influence the EED or the MuBDD at all, as the k-out-of-n: G edges are interpreted as single edges. Independently of k and n, there are five EED-nodes and six MuBDD-nodes. However, the number of states of the SBMs for each type of server grows exponentially with n, so more time is needed to solve the SBM.

Table 2 The number of EED- and MuBDD-nodes needed in ParSer(k,n) for version A and version B

Example	EED-nodes	MuBDD-nodes A	MuBDD-nodes B
ParSer(2,5)	11	4	15
ParSer(3,5)	15	5	31
ParSer(4,5)	19	6	63
ParSer(5,5)	23	7	127
ParSer(6,5)	27	8	255
ParSer(7,5)	31	9	511

Very good results were also obtained for two other real-world models of a distributed database [**10**] and a fault-tolerant water supply system [**2**]. In both cases, the memory demands and compute times were negligible.

6 RELATED WORK

The work presented here is based on the idea of transforming high-level, user-friendly input diagrams into low-level, formal, state-based models. Several other research groups have also developed methods and tools for reliability/availability assessment based on this idea. The developed methods can be categorized using the following:

(a) the kind of input used for defining the redundancy structure (e.g. fault trees or reliability block diagrams);
(b) the kind of supported intercomponent dependencies or other dynamic system properties, and how they can be defined by the modeller;
(c) the low-level representation generated, i.e. which back-end tools can be used to analyse the model;
(d) whether and how the tool identifies independent submodels to decrease the computational demands.

The focus of this paper is on the last aspect. In OpenSESAME, the overall model is divided into submodels corresponding to the set of interdependent components. This can be done independently of the authors' redundancy structure. To the best of knowledge this is a unique feature of OpenSESAME. In other tools, the model can only be divided according to the redundancy structure. In an FT, for example,

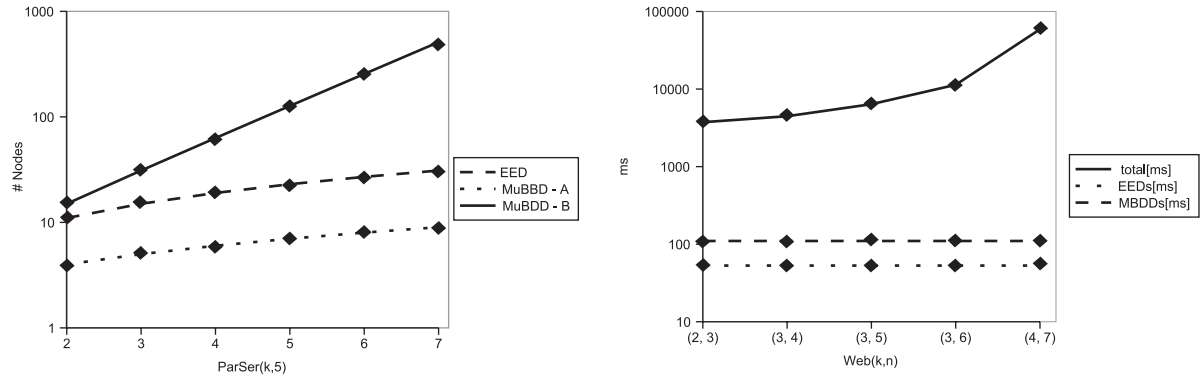

Fig. 10 (Left) Number of EED- and MuBDD-nodes needed of ParSer(k,5); (right) calculation time needed for Web(k,n)

Table 3 The computation times for the EEDs, MuBDDs, and the total solving process including the solving of the SBMs for the example Web(*k*,*n*)

Example	EEDs [ms]	MuBDDs [ms]	Total [ms]
Web(2,3)	47	121	3737
Web(3,4)	44	106	4464
Web(3,5)	53	109	6381
Web(3,6)	56	103	11 347
Web(4,7)	52	105	58 965

this means that two branches of a tree cannot be solved separately, as soon as two arbitrary basic events of the two branches are interdependent, even if all the other events are independent from each other.

The following sections give a short overview on other methods based on the idea of high-level model transformation.

6.1 Boolean-logic-driven Markov processes

In reference [**11**], an innovative approach for combining FTs and Markov models is presented. Each leaf of the FT represents a component of the system which can be described in more detail by a Markov process. Switching between the states of the chain can be triggered by the failure or repair of other components, which allows for modelling intercomponent dependencies. To simplify the task of the modeller, several predefined standard cases can be reused.

6.2 DIFtree

Another tool that transforms high-level input diagrams into state-spaced models is the dynamic innovative fault tree (DIFtree) [**12**]. The input diagrams are so-called dynamic fault trees, which extend traditional FTs by a set of new gates that can handle different kinds of redundancy (e.g. cold, warm, and hot redundancies).

6.3 Dynamic reliability block diagrams, DRBDs

In this approach [**13**] dynamic extensions to reliability block diagrams are introduced. These extensions can be used to model various dependencies between components including several types of redundancy (hot, warm, standby) and failure propagation. Dependencies are specified by their type (order or strong) and the action and reaction events (wake-up, reparation, sleep, and failure). In total, 24 different kinds of dependency are defined.

7 SUMMARY

This paper proposes a new high-level approach to calculate the availability of fault-tolerant systems. The proposed algorithm can handle RBDs that may contain bridges and repeated events. Stochastic dependencies may occur between any two or more components of the system. In addition, the algorithm can cope with *k*-out-of-*n*: G edges without having to transform them into series-parallel diagrams.

As the problem itself is NP-complete, it is possible to create problems which require an exponential amount of computing resources and memory to solve them. However, when applied to some typical real-world examples, very good results were obtained. In many cases the runtime of the algorithm grows only linearly with the number of components. For the special case that all components are independent from each other (or the dependencies only occur sparsely in the system), the proposed algorithm performs as equally well as the algorithms known from the literature.

In our future work, the authors will apply the proposed algorithm to a more extended set of real-world examples. A possible optimization would be to not store the Boolean expressions in the MuBDD explicitly, to save memory. Furthermore, the heuristics for the ordering of the SICs may be reconsidered.

In its current version, the algorithm does not support a NOT-logic: only monotone systems can be modelled. It is therefore planned to extend the algorithm to non-monotone systems, especially important for safety-critical systems.

8 AVAILABILITY

A copy of OpenSESAME containing the proposed algorithm can be obtained from the website http://OpenSESAME.in.tum.de.

REFERENCES

1 **Walter, M.** and **Trinitis, C.** Simple models for high-availability systems with dependent components. In Proceedings of the 2006 European Safety and Reliability Conference, vol. 3, pp. 1719–1726 (Taylor & Francis) 2006.

2 **Walter, M., Siegele, M.,** and **Bode, A.** OpenSESAME – the simple but extensive, structured availability modeling environment. *Reliability Engng and Syst. Saf.*, 2008, **93**, 857–873.

3 **Ball, M. O.** Computational complexity of network reliability analysis: an overview. *IEEE Trans. on Reliability*, 1986, **3**, 230–239.

4 **Lindemann, C.** *Performance modelling with deterministic and stochastic Petri nets*, 1998 (Wiley and Sons).

5 **Kuntz, M., Siegle, M.,** and **Werner, E.** CASPA – a tool for symbolic performance and dependability evaluation. In Proceedings of EPEW'04 FORTE co-located workshop, 2004 (Springer), pp. 293–307, LNCS 3236.

6 **Kuo, S.-K., Lu, S.-K.,** and **Yeh, F.-M.** Determining terminal-pair reliability based on edge expansion diagrams using OBDD. *Trans. on Reliability*, 1999, **48**(3), 234–246.

7 **Brace, K., Rudell, R.,** and **Bryant, R.** Efficient implementation of a BDD package. In Proceedings of the 27th ACM/IEEE Design Automation Conference, 1990 (IEEE Press, Piscataway, NJ), pp. 40–45.

8 **Knuth, D.** *The art of computer programming*, vol. 1, 1997 (Addison-Wesley).

9 **Walter, M.** and **Schneeweiss, W.** *The modeling world of reliability/safety engineering*, 2005 (LiLoLe Verlag).

10 **Sanders, W.** and **Mahlis, L.** Dependability evaluation using composed SAN-based reward models. *J. Parallel and Distributed Comput.* (Special issue on Petri net models of parallel computers), 1992, **15**, 238–254.

11 **Bouissou, M.** and **Bon, J. L.** A new formalism that combines advantages of fault-trees and Markov models: Boolean logic driven Markov processes. *Reliability Engng and Syst. Saf.*, 2003, **82**(2), 149–163.

12 **Dugan, J. B., Sullivan, K. J.,** and **Coppit, D.** Developing a low-cost high-quality software tool for dynamic fault-tree analysis. *IEEE Trans. on Reliability*, 2000, **49**(1), 49ff.

13 **Distefano, S., Scarpa, M.,** and **Puliafito, A.** Modeling distributed computing system reliability with DRBD. In Proceedings of the 25th Symposium on *Reliable Distributed Systems (SRDS 2006)*. IEEE, 2006.