Combining different binary decision diagram techniques for solving models with multiple failure states

M Pock[1,2,3]*, **O Malassé**[2,3], and **M Walter**[1]
[1] Arts et Métiers Paristech, Metz, France
[2] Technische Universität München, Lehrstuhl für Rechnertechnik und Rechnerorganisation, France
[3] Centre de Recherche en Automatique de Nancy, France

The manuscript was received on 22 September 2009 and was accepted after revision for publication on 8 June 2010.

DOI: 10.1177/1748006XJRR284

Abstract: This article explains information flow diagrams (IFDs) in great detail. With these models, scenarios leading to dangerous failures as well as spurious shutdowns can be generated, taking into account several failure modes for basic components. These scenarios can be evaluated in a qualitative and quantified way. For an efficient solution different BDD-techniques were combined, mainly zero suppressed BDDs and binary expression diagrams. It is shown how these techniques were used for the model and how a large BDD is created by assembling several smaller diagrams, so that even large and complex systems can be described and evaluated in a compact and efficient way.

Keywords: safety modelling, binary decision diagrams, binary expression diagrams

1 INTRODUCTION

Assessing fail-safe systems is a difficult but important task. Failures can have severe consequences such as loss of life or grave injuries. Today, the assessment is done by modelling the system and analysing these models with simulation or analytical methods. There are several modelling methods used for dependability analysis in general, some of them are also used for fail-safe systems. These can be distinguished in two classes: combinatorial methods, describing systems on a high level, and state-based methods describing systems on a low level. Combinatorial methods such as fault trees can be used to calculate the overall system failure probability for a top event using basic system components and their redundancy structure. They are easy to understand, to use and to solve, but they cannot describe all of the important properties of fail-safe systems in a satisfying way. They cannot describe several failure modes or dependencies between subsystems or single components.

State-based methods such as Petri nets or Markov chains are much more powerful, but not very intuitive. Often it is not possible to create such a model in a hierarchical way, and the model itself can get very large even for small systems. Owing to these reasons, modelling complex systems with such methods can be quite unpractical and error-prone, besides the solution needs much more computing power. Therefore a specialized method is presented for fail-safe systems which combines intuitivity and important properties of such systems.

A special problem for fail-safe systems is that they have different failure modes. Conventional models cannot take these into account properly, so a new model has recently been presented: information flow diagrams [**1**] (IFDs). These can describe fail-safe systems and their different failure modes with one hierarchical model which is both powerful and easy to apply. In order to estimate the probability of failure on demand (PFD) and the probability of spurious trips (PFS) the widely used binary decision diagram (BDD) techniques are employed. As the present model has

**Corresponding author: A3SI, ENSAM Metz, 4 Rue Augustine Fresnel, Metz 57078, France.*
email: Michael_Pock@gmx.de

certain properties, the conventional BDD methodology has to be adapted. While BDDs in general just support two modes (failed and working) for every component, up to four have to be handled: failed dangerously, failed non-dangerously, loss of signal, and working. Furthermore, the construction process of the BDD is optimized. Instead of transforming a large Boolean model into a BDD, several smaller models are combined. This will increase the efficiency of the BDD construction enormously.

This article gives a detailed presentation of the used BDD-techniques, including the variations of zero-suppressed BDDs and binary expression diagrams (BEDs). A description is given of how a combination of different variants can be used to solve complex models qualitatively and quantitatively.

This paper is organized as follows: section 2 briefly presents IFDs. Section 3 explains different BDD-techniques in general. In section 4, the application of the BDD-techniques for IFDs is explained in detail. Section 5 presents related work, followed by the conclusions in section 6.

2 THE INFORMATION FLOW DIAGRAM

Information flow diagrams (IFD, [**1**]) were developed to describe fail-safe systems with two failure modes. With Boolean models such as fault trees, it would be necessary to create two models for each one. Furthermore, these failure modes would not be stochastically independent as a system can only be in one state at one time. These dependencies have to be included in the different models making modelling difficult and error prone. The present paper focuses on two different events of the system, leading to two different failure modes:

(a) dangerous incidents which could lead to accidents (failure mode D);
(b) non-dangerous spurious trips (failure mode S).

For instance, an urgency shutdown for a chemical reactor can be activated unnecessarily, leading to an expensive unavailability of the whole chemical plant (failure mode S). It is also possible that the reactor is not shut down in a critical situation, which could lead to an accident with severe material, personal, or ecological damage (failure mode D). While both scenarios are unwanted, their consequences are completely different, so it is necessary to analyse them separately.

All scenarios leading to one of these two undesirable events of the whole system for quantitative and qualitative analysis are to be extracted. Especially interesting are single points of failure and failure propagation. These kinds of failures are often not obvious, so they can be easily forgotten in a direct attempt to create a fault tree. This problem will be solved by a hierarchical approach. A directed block diagram was used, representing the information flow through the system for high level and special expressions for low-level modelling.

2.1 Information flow diagram

For the IFD, we use different kinds of blocks which represent different functional entities:

(a) WD blocks for watchdogs;
(b) SRC blocks as sources of information;
(c) DEC blocks for logical decisions;
(d) ST blocks for all other functions (storage of information, transformation of information, self-tests, etc.)

Blocks of the type WD are used especially for control units with a watchdog. They have one input and one output and can detect the absence of sensible information in order to react accordingly afterwards by forwarding default or special error values. SRC blocks create the information which flows through the diagram. They represent the sensors in the system and have only one output. ST blocks (standard blocks) are the most versatile blocks. They have one input and one output, and they are used for all functional entities which cannot be represented by the other blocks, e.g. the storage or the transformation of information. The last type of blocks are DEC blocks. They represent logical decision entities. They have several inputs and one output, and describe the behaviour of multiple interconnected sources of information. They do not describe any physical entities.

One block in the diagram, normally a ST- or DEC-block, can be marked as a final block. This block has no output and is used to generate the failure scenarios which will be described in the next sub section. An example of an IFD for the example presented in reference [**1**] is shown in Fig. 1. There are blocks for the different modules of the system, and some extra decision blocks. The information flows from the source blocks to the final block in one general time step t. In the source blocks, the sensors create the information which will flow through the diagram. This information proceeds to the successing blocks where it is processed and proceeded further. The exchange between the blocks is always faultless. In the blocks, faults can occur or faulty states can be detected. This means, that the state of signal can change within a block. Three different erroneous states are distinguished for the signals:

(a) a non-existent failure has been detected (safe failure state S);
(b) an existent failure has not been detected (dangerous failure state D);
(c) a signal is lost (loss of signal failure state L).

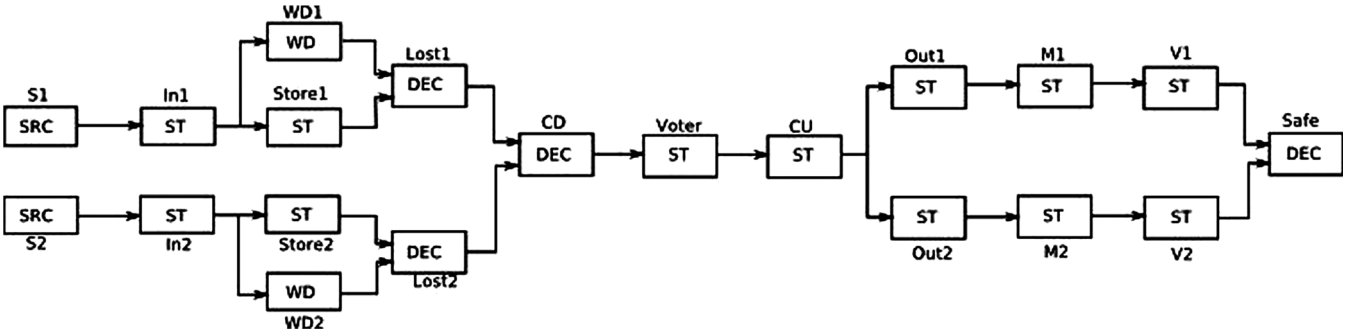

Fig. 1 The IFD for an emergency stop system

To illustrate the presented model, the emergency stop system of a chemical reactor (Fig. 1) will be used, which is described in this section. This system should stop the reaction if the temperature in the reactor is becoming too high by stopping the inflow of the chemicals.

The sensors $S1$ and $S2$ measure the current temperature of the chemicals in the tank and transport their results to the controller. The controller reads this result via inputs $In1$ and $In2$ and will store these values for synchronization in its memory ($St1$, $St2$). To avoid a loss of information, the watchdogs $Wd1$ and $Wd2$ supervize the inputs. If an input is lost or arrives too late, the watchdog will pass a default value to the *Voter*. Afterwards, the *Voter* decides, if there is a dangerous situation. If after the voting process the control unit *CU* decides to shut down the system, this information is proceeded to the output modules $Out1$ and $Out2$ and passed to the motors $M1$ and $M2$ which get the order to close the valves $V1$ and $V2$. If at least one of these valves is closed, the shutdown was successful.

For this system, there are two possible kinds of failures in general. Either the emergency system is not available (dangerous failure), or it shuts down the system in a safe state leading to a unnecessary unavailability of the whole reactor (spurious shut-down).

A failure of the whole emergency system can be caused by several different failures of its components. These failures are classified as dangerous (D), non-dangerous (S), or omission (L), which means that the component does not have an output.

The sensors can either measure a value which is too high (S), too low (D), or return no value at all (L). The input modules can either lose the data of the sensors (L) or change it to a higher (S) or lower (D) value. In the memory the stored data can be distorted by a bit-flip in either a dangerous (D) or non-dangerous (S) way. It can happen that the watchdogs do not detect a missing input (D), or that they report such a missing input although there was one (S). The control unit can decide to start a shutdown in a safe state (S) or to not start a shutdown in a dangerous state (D). The output modules can fail to give the orders to their motors to close the valves (D), while the motors can fail to start (D). Finally, the valves can be blocked in an open (D) or closed (S) position.

It is possible to discriminate the single components further, but in this section the explanation is limited to the general outline of the system.

2.2 Low-level model

For each block in the high-level model a low-level model is created. It defines three expressions for three different failure modes: S for non-dangerous failures, D for potential dangerous failures, and L for lost signals. For ST, SRC and WD blocks these expressions consist of different types of subexpressions:

(a) general hardware or software failure state of a component c: $c = s$ with $s \in \{S, D, L, 0\}$;
(b) bit-flip of a component c: $bf(c)$;
(c) failure propagation of the predecessor block: input(s) with $s \in \{S, D\}$ for ST- and $s \in \{L\}$ for WD blocks

The failure modes S, D, and L are analogous to the failure modes of the high-level model, the mode 0 means that no failure has occurred. For instance, the block Store1 could be defined by the following expressions:

$$S(\text{Store1}) = (mem = S) \vee (input(S) \wedge (mem = 0)) \vee (input(D) \wedge (mem = 0) \wedge bf(m))$$

$$D(\text{Store1}) = (mem = D) \vee (input(D) \wedge (mem = 0)) \vee (input(S) \wedge (mem = 0) \wedge bf(m))$$

$$L(\text{Store1}) = (mem = L)$$

This means that the block Store1 is in state S if and only if either the component mem is in state S, mem is in state 0 and the predecessor block is in state S or mem is in state 0, the predecessor block is in state D and a bit-flip of the bit-flip component m occurs. The expressions for D and L can be interpreted analogously.

Decision blocks use other kind of expression than ST, SRC, and WD blocks. A DEC block includes the multiple predecessor blocks and their failure states

to define its own failure states. For the final block of the IFD shown in Fig. 1, the following rules are chosen

$$S(\text{Safe}) : V1 = S \vee V2 = S$$
$$D(\text{Safe}) : V1 = D \wedge V2 = D$$
$$L(\text{Safe}) : \textit{false}$$

A spurious trip will occur if at least one of the two valves will fail spuriously. A dangerous failure will only occur if both valves will fail dangerously.

The present paper aims to generate all scenarios for dangerous failures and spurious trips of the final block which are equivalent to the according failures of the global system. At first, two expressions S and D are constructed. To extract these expressions, the expressions $D(B_f)$ and $S(B_f)$ of the final block B_f are created in order to connect them with the local expressions of all the other blocks of the IFD. For ST and WD blocks $input(y)$ is substituted with the expression y of the predecessor block. For DEC blocks with a predecessor block B, $B = y$ with $y \in \{S, D, L\}$ is substituted with the expression $y(B)$. This is continued recursively until the substitution process arrives at the SRC blocks.

3 DECISION DIAGRAMS

It is quite obvious that using this method directly will lead to an exponential growth of the global expressions. This is a severe problem as it will limit the usability of the proposed method. Therefore the size of the created list has to be reduced. In order to reach this aim the technique of BDD [**2**, **3**], are used to control the combinatorial explosion. The widely known BDDs are often used in the domain of reliability analysis [**4**–**7**] to avoid the combinatorial explosion. Multiple variations of BDDs exist. In the current case, two different approaches will be combined: zero suppressed BDDS (z-BDD) [**8**] and Boolean expression diagrams (BED) [**9**]. In this section, a short overview of z-BDDs and BEDs is given. Afterwards, an explanation is given of why and how these techniques are used for our special applications.

3.1 z-BDDs

z-BDDs are a modification of the widely used BDDs and based on binary decision tree (BDT, [**2**]). A BDT is a tree with:

(a) a finite set of Boolean variables *Var* with a given order;
(b) a finite set of nodes V_{NT}, each containing one variable $v \in Var$ as attribute;
(c) two different types of leaves V_1 (*One-Node*) and V_0 (*Zero-Node*);
(d) a root $V_r \in V_{NT}$.

Every node has two children: a high child and a low child.

A BDT can be converted into a BDD by applying two reduction rules:

(a) equivalent nodes, i.e. nodes with identical children and the same variable, are unified;
(b) a node v where the high and the low child are identical (*Don't-Care-Nodes*) are removed, their parents are linked with the only child of v instead.

A BDD is an equivalent representation of the BDT, but instead of a tree a directed acyclic graph (DAG) is used to store the information which is much more memory efficient than a tree.

For z-BDDs, other rules are applied:

(a) equivalent nodes, i.e. nodes with identical children and the same variable, are unified;
(b) a node v with the node V_0 as high child are removed, their parents are linked with the low child of v instead.

z-BDDs can lead to a significant size reduction compared with a common BDD if there are a lot of nodes with V_0 as high child. Figure 2 shows a BDT, a BDD, and a z-BDD of the Boolean expression $(a \wedge \overline{b} \wedge c) \vee (\overline{a} \wedge c)$. All three diagrams are an equivalent representation of the same expression.

3.2 BEDs

BEDs, originally used for verifying circuit implementations, are an expansion of BDDs. In general, BEDs

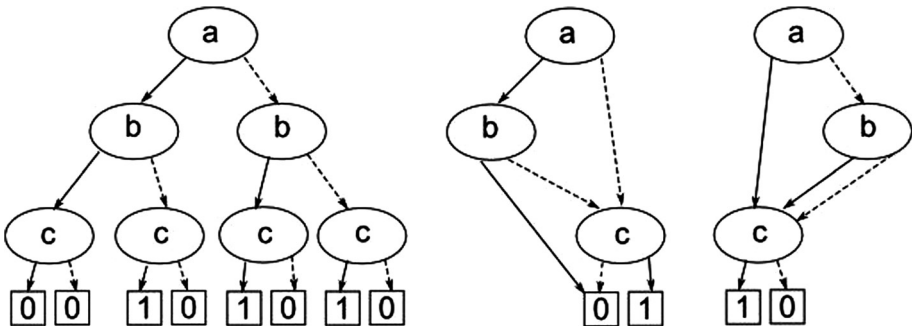

Fig. 2 A BDT (left), BDD (middle), and z-BDD (right) for the Boolean expression $(a \wedge \overline{b} \wedge c) \vee (\overline{a} \wedge c)$

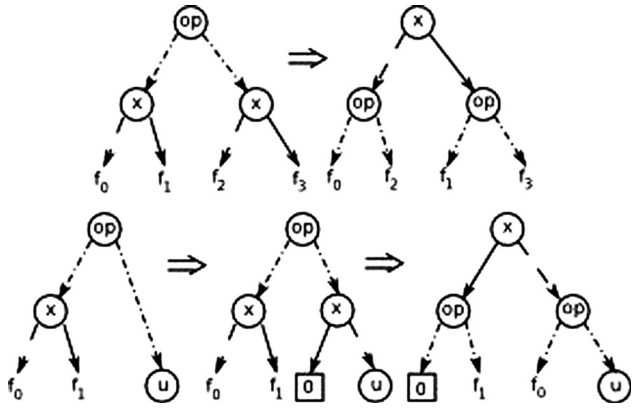

Fig. 3 Reduction rules for BEDs

are a BDD enriched with different kinds of operator nodes for Boolean operations. For this paper just two of them will be used: *OR-Nodes* and *AND-Nodes*. It is possible to transform the BED into a BDD with the same complexity as creating the BDD directly [**9**]. As BEDs are originally developed for standard BDDs an alternation of the original transformation rules is necessary, although leading to *zero-suppressed BEDs* (ZBED). These rules are shown in Fig. 3.

4 APPLICATION FOR IFDS

For the problem of this article z-BDDs will be combined with BEDs. Furthermore, advantage is taken of several characteristics of the IFD in order to create a decision diagram describing the whole model. First, the method for serial IFDs with simplified assumptions is presented. Afterwards, extensions are introduced for DEC blocks and general IFDs. Finally, the quantitative evaluation is discussed.

4.1 Boolean interpretation of local expressions

Decision diagrams are used for representing Boolean expressions. To apply them for describing IFDs, it is necessary to transform the local expressions of the blocks into Boolean expressions. Bit flip- and fault test-resources can have two states, so it is no problem to see them as simple Boolean variables. Hardware resources, however, have four states. For these three different Boolean variables (for example x_S, x_D, and x_0) can be defined for every resource x. Note that three variables are enough, a variable x_L is not necessary. As x can only be in one state at one time, the value of x_L can be deduced from the values of the other three variables. The expression $x_{S1} \wedge x_{S2} = \mathit{false}$ holds for $S1 \neq S2$. The *Input*-expression will not be changed at first. This will be replaced during the construction process of the z-BDD.

After such a transformation the local expressions are close to Boolean expressions, only the *Input*-expressions remain. For example, $\{((x = S) \wedge bf(e)) \vee (\mathit{Input}(S) \wedge (x = L))\}$ is interpreted as $(\overline{x_0} \wedge x_S \wedge \overline{x_D} \wedge e) \vee (\mathit{Input}(S) \wedge \overline{x_0} \wedge \overline{x_S} \wedge \overline{x_D})$.

4.2 DDs for simple serial systems

In this subsection, two assumptions are made:

(a) the IFD does not contain any DEC blocks;
(b) each Boolean variable only occurs in the lists of one block.

With these assumptions, it is very easy to use the structuring of the IFD in order to create a z-BDD very efficiently. It is possible to create a z-BDD for the final block based on the local expressions. This z-BDD can include two extra leaves ($\mathit{Input}(S)$ and $\mathit{Input}(D)$) in the case of ST-blocks, and one extra leaf for WD blocks($\mathit{Input}(I)$). These leaves can be substituted using the following method:

```
   CreateSerialZBDD(Block finalBlock, Mode x)
1     ZBDD global = CreateZBDD(finalBlock.getExpression(x));
2     Queue inputNodes = global.getInputNodes();
3     While (!inputNodes.isEmpty())
4          Node inputNode = inputNodes.getNext();
5          ZBDD local = CreateZBDD(getExpression(inputNode));
6          for all incoming edges e of inputNode:
7               e.setTarget(local.getRoot());
8          inputNodes.add(local.getInputNodes());
9     return global;
```

CreateBDD(Expression e) creates a local z-BDD for the given expression *e* by decomposition. This method will use already existing equivalent nodes in the global z-BDD if possible. *getInputNodes(ZBDD zbdd)* returns all *Input*-nodes of the ZBDD *zbdd*, and *getExpression()* delivers the applicable expression.

An $\mathit{Input}(x)$-leaf is replaced by the root of the z-BDD for the expression x of the predecessor block. If there are multiple *Input*-leafs, the sub z-BDDs will also share equivalent nodes. After substituting the *Input*-nodes of one block, new *Input*-nodes can occur if the predecessor block is not a SRC block. So the

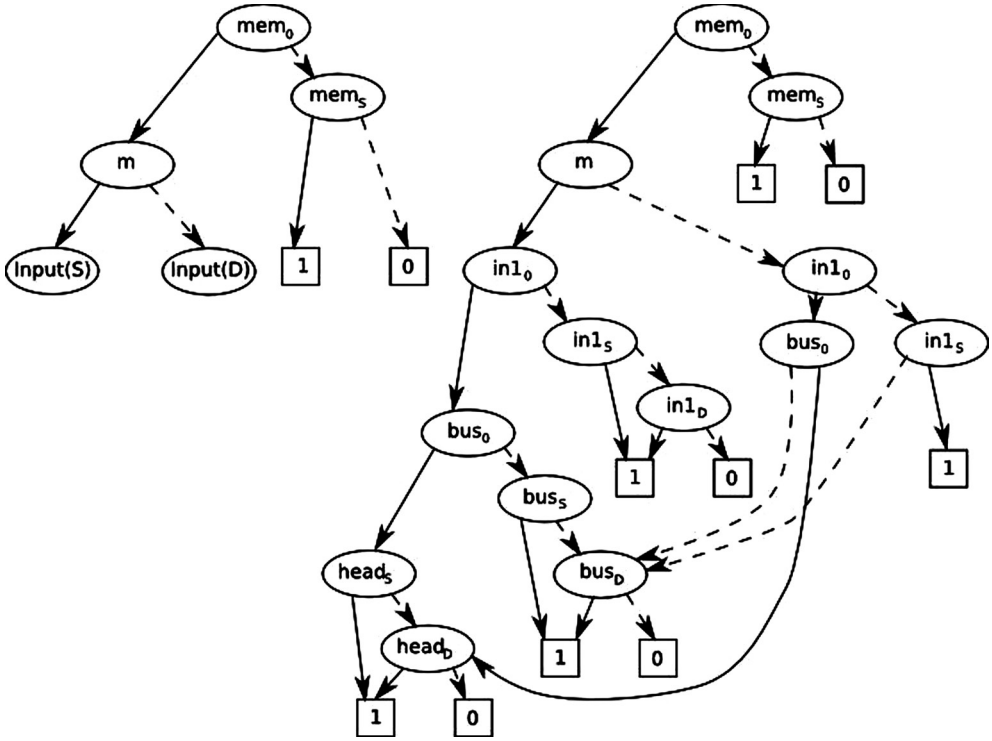

Fig. 4 A local z-BDD for the block mode S of *Store*1 (left) and the corresponding global z-BDD (right)

substitution has to be repeated recursively until it arrives at the SRC block. An example is shown in Fig. 4. As the z-BDD of the whole IFD contains 146 nodes, only the z-BDD of a small subsystem is shown in order to demonstrate the algorithm. Note, however, that BDDs of such a size are no problem for a computer.

The z-BDD for the mode S of the block *Store*1 from Fig. 1 is used as an example. On the left side the local z-BDD of this block is shown while the right side depicts the global z-BDD after replacing recursively all *Input*-nodes with the corresponding z-BDDs of the predecessor blocks.

For each of the two global failure modes a z-BDD is needed as these can only describe one kind of failure. But it is possible to create two different z-BDDs which share equivalent nodes if possible, leading to one z-BDD with two different roots.

The advantage of the method of modularly constructing the z-BDD is that globally it will only grow linearly in size compared with the number of blocks. After every block there are at most two non-trivial leaves. Locally, however, it is still possible that a sub z-BDD grows exponentially. But this should not cause any problems as the local z-BDDs are normally quite small and they can be created independently from the z-BDDs of the other blocks. Overall this approach can reduce the complexity enormously compared with creating the z-BDD for the whole system without splitting it into different blocks as the z-BDD will be reduced regularly to at most two nodes on the same level.

It was decided to choose z-BDDs rather than BDDs as they are better suited for this application. For a hardware resource x, three Boolean variables x_0, x_S, and x_D are created. There are eight different possibilities to set these variables as true or false. Four of them ($x_0 \wedge x_S \wedge x_D$, $x_0 \wedge x_S \wedge \overline{x_0}$, $x_0 \wedge \overline{x_S} \wedge x_D$, $\overline{x_0} \wedge x_S \wedge x_D$) are always invalid, however, as only at most one of the variables can be true. So these four combinations will always lead to N_0 which can be reduced in z-BDDs. Figure 5 shows the general structure of a z-BDD and a BDD for three generic variables x_0, x_S, and x_D. While a BDD needs up to seven nodes for these three variables, a z-BDD needs at most three. Besides, as the structure is known in advance, it is possible to optimize the decomposition process. If x_0 has been set to true, setting x_S or x_D to true will lead to the zero node followed by a reduction of the z-BDD. Instead of creating a node which will be removed immediately afterwards, x_S and x_D will be set to false while x_0 is set to true. In the end this will lead to exactly the same z-BDD with less effort required to create it.

4.3 DDs for DEC blocks

DEC blocks are specified by different expressions than ST, WD, and SRC blocks. In order to transform these expressions into z-BDDs ZBEDs are used. Similar to

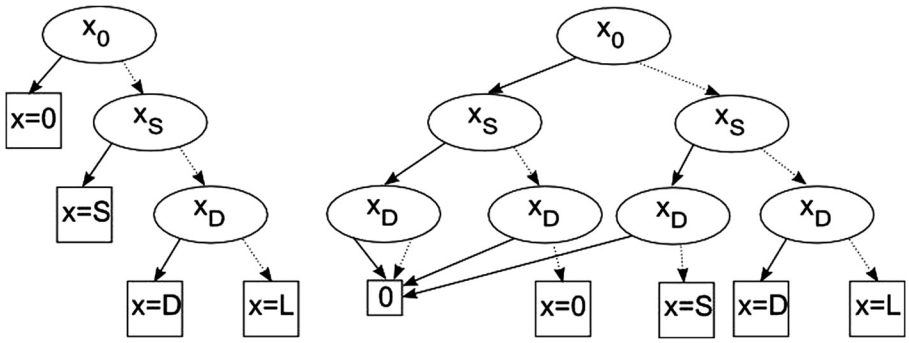

Fig. 5 The structure of a z-BDD and a BDD for a generic hardware resource

non-DEC blocks, up to three ZBEDs are created, one for every expression used in the block. The leaves of the ZBED represent the predecessor blocks in a certain state (S, D, I) while operator nodes are used to link the leaves according to the rule. The following algorithm is used:

```
   CreateZBED(Block decBlock, Mode x)
01     ZBED zbed = ExpressionToZBED(decBlock.getExpression(x));
02     Queue inputNodes = zbed.getInputNodes();
03     While (!inputNodes.isEmpty())
04          Node inputNode = inputNodes.getNext();
05          ZBDD local = CreateZBDD(getExpression(inputNode));
06          for all incoming edges e of inputNode:
07               e.setTarget(local.getRoot());
08          inputNodes.add(local.getInputNodes());
09          while minimisation is possible:
10               ApplyZBEDRules(zbed);
11     return zbed;
```

The linking process with the predecessors and the successor block is similar to the method described in section 4.2. The leaf representing the predecessor block B in a certain state X is substituted with the appropriate z-BDD for the expression $X(B)$, while the local ZBED for the expression for state X substitutes the $Input(x)$-node of the successor. The expressions of the DEC-block itself are transformed directly to a ZBED, in which leaves represent the roots of the local z-BDDs of the predecessor blocks, analogously to the *Input*-nodes in serial diagrams. An example ZBED for the expression $X = S \wedge (Y = D \vee Z = D)$ is shown in Fig. 6.

The main difference between serial IFDs and IFDs with DEC-blocks can be recognized in lines 9 and 10 of the algorithm: before continuing the substitution, the operator nodes are pushed down as far as possible in order to transform the ZBED into a z-BDD. For reducing the ZBED to a z-BDD, there are two possibilities in general:

(a) apply the reduction rules once after all local z-BDDs and ZBEDs have been linked;

(b) apply the reduction rules iteratively always after a new local z-BDD or ZBED has been added, and stop this reduction at *Input*-nodes.

The second approach is used as the reduction rules can already simplify the diagram significantly while constructing it as soon as operator nodes are linked to terminal nodes. Finally, all operator nodes will be removed leading to a standard z-BDD.

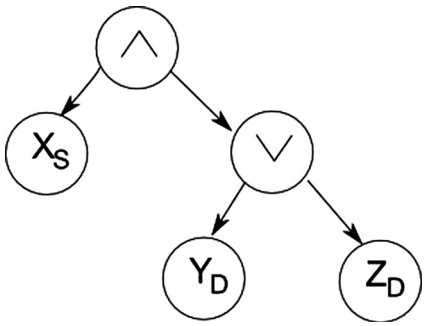

Fig. 6 A local ZBED for the expression $X = S \wedge (Y = D \vee Z = D)$

It is possible to apply Boolean operators directly to z-BDDs, so theoretically it would not be necessary to use the BED-technique. In this case, however, it has a major advantage. In order to apply Boolean operators directly, the whole sub z-BDD has to be known including all substitutions. In this case, it would not be possible to build the z-BDD block by block, which is a severe disadvantage. In contrast, the BED-technique allows efficient local substitutions to be made.

4.4 DDs for general IFDs

In section 4.2 it was assumed that no component will occur in more than one block. However, this assumption is quite unrealistic for many systems, so it is necessary to extend the presented algorithm in order to allow for multiple occurrences of components.

To be able to include such cases, the algorithm for creating the z-BDD is extended. Overall, two attributes are added to the z-BDD:

(a) an array *localLists* containing all local lists;
(b) a hash table *compMap* mapping components to blocks in which they appear.

The construction of the z-BDD begins as in the simple case. The decomposition of the local lists starts at the final block and ends with the source blocks. The only difference is that before every decomposition of a variable the *compMap* is checked to find other blocks which use it too. If there are other blocks with the same variable, the decomposition is also applied to copies of the local lists of these blocks, stored in *localLists*. All descendants of the current node will use these modified lists instead of the original ones as soon as the blocks using these lists are reached. In order to achieve this, every node stores an array with pointers to the lists which have to be used in the future. Children inherit these lists from their parents and alter them only if their variable will also occur in other blocks of the IFD. An example is shown in Fig. 7 which represents the list L_D for a small serial system in which a component a occurs in two blocks (C and A). The local expressions are defined as follows

$$D(C) = (c = D) \vee (input(D) \wedge (a = S) \wedge (c = S)) \vee (input(D) \wedge (a = 0))$$
$$D(B) = (b = D) \vee (input(D) \wedge (b = 0))$$
$$D(A) = (a = D)$$

After setting the value of a to a specific value in block C, the list for block a is modified too. The following nodes will use these modifications by linking to it using their pointer arrays.

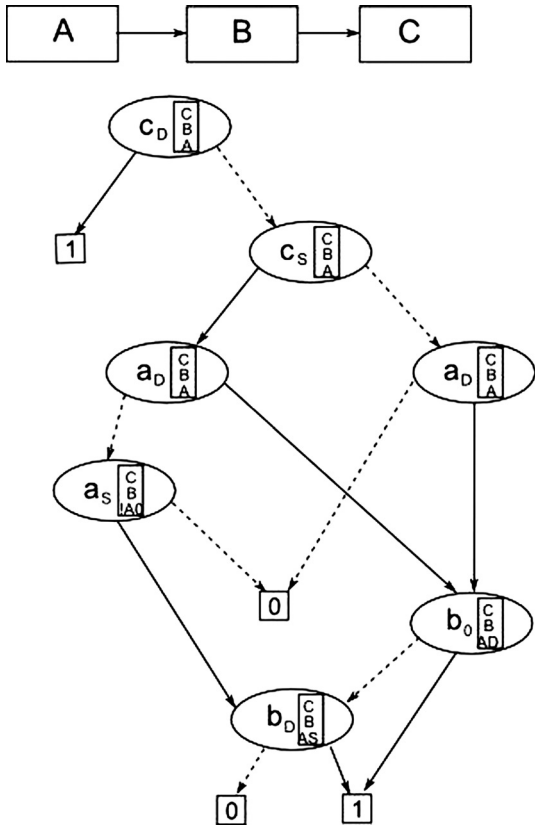

Fig. 7 A z-BDD for a serial system with multiple occurrences of one variable

4.5 Quantitative evaluation

A basic assumption for the quantitative evaluation of z-BDDs in order to estimate the overall reliability is that all variables used in the z-BDD are stochastically independent. In the case of variables for hardware failure, however, this is not true. For each HW component c there are three variables: c_0, c_S, and c_D. Obviously, at most one of them can be true, so dependencies have to be taken into account.

The solution for this problem is to use conditional dependencies. Let c_t be the state of the variable at the time t. The following probabilities are used for the variables c_0, c_S, and c_D.

(a) $P(c_t = 0)$ for c_0;
(b) $P(c_t = S | c_t \neq 0) = \frac{P(c_t=S)}{P(c_t=S)+P(c_t=D)+P(c_t=L)}$ for c_S;
(c) $P(c_t = D | c_t \neq 0 \wedge c_t \neq S) = \frac{P(c_t=D)}{P(c_t=D)+P(c_t=L)}$ for c_D.

To calculate the probabilities $P(c_t = s)$ with $s \in \{0, S, D, L\}$, a model for the component c and its different failure modes has to be included. At the moment, exponential distribution is assumed for component failures. For each failure mode s a failure rate λ_s is defined. With these given data, a Markov chain as in Fig. 8 can be created. The probabilities $P(c_t = s) = q_s(t)$ can be calculated by solving the following

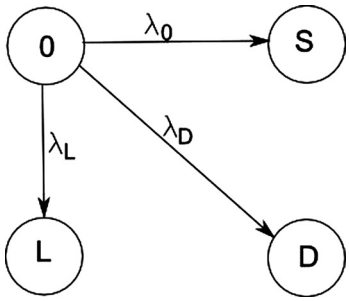

Fig. 8 A Markov chain modelling the component failures

differential equation system which can be attained from the Markov chain

$$\frac{\mathrm{d}}{\mathrm{d}t}q_0(t) = -\lambda_S q_0(t) - \lambda_D q_0(t) - \lambda_I q_0(t)$$

$$\frac{\mathrm{d}}{\mathrm{d}t}q_S(t) = q_0(t)\lambda_S$$

$$\frac{\mathrm{d}}{\mathrm{d}t}q_D(t) = q_0(t)\lambda_D$$

$$\frac{\mathrm{d}}{\mathrm{d}t}q_I(t) = q_0(t)\lambda_I$$

Note that it is possible to alter the Markov chain, leading to a different differential equation system. It is also possible to use other distributions such as Weibull. In this case only the estimation of $P(c_t = s)$ has to be altered, the rest of the algorithm does not need to be changed.

5 RELATED WORK

Papadopoulos *et al.* [**10**] present a model also using the information flow and several failure modes of basic components such as omission faults or detected faults. It is especially well suited to describe the communication process in safety critical environments. It generates several fault trees in order to solve the model, which also can be extended in order to describe inter-component dependencies. However, it supports only one global failure mode.

One of the present authors' works [**11**] describes SafeMe, a model based on reliability block diagrams (RBDs) but capable of representing safety critical systems. It supports repair of failed components, but it also takes into account delayed repairs or a total destruction of the system owing to a catastrophic failure. Furthermore, dependencies between single components can be handled, too. It is solved by automatically creating a single state-based model equivalent to the high-level model.

Prescott *et al.* [**12–14**] present a technique for creating large BDDs by adding small local BDDs, which is used for phased-mission systems. For each mission phase a BDD is created in advance. These BDDs can be changed while the mission is carried out, e.g. in case of a component failure, leading to an on-the-fly estimation of the overall mission success chance. For such an application the BDD-creation process is easier than for the current case, although, as whole branches of the BDD can be cut off if they are impossible owing to a component failure which had already happened. In contrast, in the current application it is assumed that all components are working at the beginning of the analysis.

6 CONCLUSIONS AND OUTLOOK

In this paper, a model had been presented which can represent several failure modes occurring in fail-safe systems. Furthermore, a very efficient algorithm is discussed for evaluating IFDs based on different BDD techniques. By combining BEDs with z-BDDs, it is possible to create z-BDDs even for general diagrams in a very efficient way. Especially local properties are used to improve the performance of the algorithms.

The presented approach is already implemented and was used for modelling a remote redundancy system [**15**]. In the future, further case studies will follow in order to test the performance and the usability of IFDs and their evaluation algorithms.

ACKNOWLEDGEMENTS

The authors would like to thank the French-Bavarian centre for cooperation of universities (BFHZ-CCUFB) and the Région Lorraine for their support of this work.

REFERENCES

1 **Pock, M., Belhadaoui, H., Malassé, O.,** and **Walter, M.** Efficient generation and representation of failure lists out of an information flux model for modeling safety critical systems. In *Proceedings of the European Safety and Reliability Conference (ESREL 2008)*, 2008. pp. 1829–1838 (Taylor & Francis Ltd).

2 **Brace, K., Rudell, R.,** and **Bryant, R.** Efficient implementation of a BDD package. In *27th ACM/IEEE Design Automation Conference*, 1990, pp. 40–45 (IEEE, New York).

3 **Zang, X., Sung, N.,** and **Triverdi, K. S.** A BDD-based algorithm for reliability analysis of phased-mission systems. *IEEE Trans. Reliability*, 1999, **48**(1), 50–60.

4 **Ibanez-Llano, C., Melendez, E.,** and **Nieto, F.** Variable ordering schemes to apply to the binary decisions diagram methodology for event tree sequences assessment. *J. Risk Reliability*, 2008, **222**(1/2008), 7–16.

5 **Rauzy, A. B.** Some disturbing facts about depth-first left-most variable ordering heuristics for binary decision diagrams. *J. Risk Reliability*, 2008, **222**(4/2008), 573–582.

6 **Contini, S.** Quantification of fault trees containing mutually exclusive events. *J. Risk Reliability*, 2008, **222**(4/2008), 623–634.

7 **Ibanez-Llano, C., Rauzy, A., Melandez, E.,** and **Nieto,F.** Minimal cutsets-based reduction approach for the use of binary decision diagrams on probabilistic safety assessment fault tree models. *J. Risk Reliability*, 2009, **223**(4/2009), 301–311.

8 **Minato, S.** Zero-supressed BDDs for set manipulation in combinatorical problems. In *30th ACM/IEEE Design Automation Conference*, 1993, pp. 272–277 (ACM/IEEE).

9 **Andersen, H.** and **Hulgaard, H.** Boolean expression diagrams. In 12th Annual IEEE Symposium on *Logic in computer science*, 1997, pp. 88–111 (IEEE).

10 **Papadopoulos, Y., Grante, C., Grunske, L.,** and **Kaiser, B.** Continuous assessment of designs and re-use in model-based safety analysis. In *16th IFAC World Congress*, 2005, Prague.

11 **Walter, M.** and **Trinitis, C.** Automatic generation of state based dependability models: from availability to safety. In Workshop Proceedings of the 20th International Conference on *Architecture of computing systems (ARCS 2007)*, 2007 pp. 47–54 (VDE-Verlag Berlin).

12 **Presscott, D.** and **Andrews, J.** A cause consequence analysis approach to modelling multi-platform phased missions. In *Advances in Risk and Reliability Technology Symposium*, 2009, Loughborough, UK, pp. 419–437.

13 **Prescott, D. R., Remenythe-Prescott, R., Reed, S., Andrews, J. D.,** and **Downes, C. G.** A reliability analysis method using binary decision diagrams in phased mission planning. *J. Risk Reliability*, 2009, **223**(2/2009), 133–143.

14 **Prescott, D. R., Andrews, J. D.,** and **Downes, C. G.** Multiplatform phased mission reliability modelling for mission planning. *J. Risk Reliability*, 2009, **223**(1/2009), 27–39.

15 **Echtle, K., Kimmeskamp, T., Jaquet, S., Malassé, O., Pock, M.,** and **Walter, M.** Reliability analysis of a control systems built using remote redundancy. In *Advances in Risk and Reliability Technology Symposium*, 2009, pp. 335–346.