Efficient generation and representation of failure lists out of an information flux model for modeling safety critical systems

Michael Pock, Hicham Belhadaoui, Olaf Malassé, Max Walter

Efficient Generation and Representation of Failure Lists out of an Information Flux Model for Modeling Safety Critical Systems

Michael Pock[1,2,4], Hicham Belhadaoui[1,3,4], Olaf Malassé[1] & Max Walter[2]
1) Arts et Métiers Paristech, Metz
2) Technische Universität München
3) ENSEM Casablanca
4) Centre de Recherche en Automatique de Nancy

ABSTRACT: This article presents a novel way to model safety critical systems hierarchically. An information flow diagram as high level and finite automatons as low level model are combined. With these models, scenarios leading to dangerous failures as well as spurious shutdowns can be generated. Furthermore, we will show how to extract the different scenarios out of the model in a very efficient way using different BDD-techniques. Finally, we will present some related work.

1 INTRODUCTION

A serious problem for the design and evaluation of safety critical systems, for example used in the automation domain, is to forecast the safety of the system. It's not possible to measure it for two reasons. At first, normally these accidents should occur very rarely, so that a system has to run for a very long time to get reasonable results. Furthermore, it is very questionable to run a safety-critical system which can cause a lot of harm without knowing anything about its dependability. So the only solution is to use a model to evaluate the safety of the system.
Another problem are unspecificated activations of safety functions, the so called spurious trips. They can lead to the unavailability of the system and cause extra costs. Furthermore, too many spurious trips can be dangerous as well. If there are too many false alarms or shutdowns, the operators and surrounding people could start to ignore the alarms or try to prevent the shutdowns, even if there is a dangerous situation this time.
There are in general two problems for the evaluation. The first one is to get all scenarios leading to failures of the System. Such a qualitive analysis is normally done with methods like FEMA. For complex systems, this can be quite difficult as there are a lot of failure propagations and common causes possible which have to be included in the result. Often these are not obvious, a very detailed analysis of the whole system is necessary to find them.
The second problem is the quantitive solution. Currently, mainly standard models like fault trees (FTs) (Lee et al. 1985) and Petri nets (PNs) (Leveson and Stolzy 1987) are used. Fault trees are easy to use, but they have several limitations. At first, components in safety critical systems often have more than one failure mode. They can create dangerous failures or spurious trips itself and propagate these to other components. They even can prevent specific failures of other components. For example, an unwanted shutdown will prevent any accident. But Fault Trees are just boolean models, every event or component can only have two modes. Furthermore they cannot model dependencies directly which will occur between different kind of failures. The only possibility to use fault trees for safety-critical systems is by defining several basic events for single components. As these events are stochastically dependent, extensions of fault trees like non-monotonous FTs have to be used. In general, the trees can get very complicated and very large for complex systems. There are also at least two unwanted events for safety-critical systems, whereas fault trees are just capable to model single events. So it would be necessary to create multiple models with a lot of common components and events.
Petri Nets in contrary are more powerful. There is no limitation to the number of states of a component, and it is possible to calculate several events in one PN. But they are not very intuitive. They also cannot be created hierarchically, as a small change in one com-

ponent will affect the whole PN.
This is the reason that we will present a powerful high-level model which can handle all these specific problems for safety-critical systems, based on the theoretical work of Karim Hamidi (Hamidi 2005) in section 3. With this model it is much easier to find all scenarios leading to a failure of the system. An efficient solution method will follow in section 4. At the moment, there is no implementation of the presented algorithms, though. So it is not possible to present measurements yet.

2 AN EMERGENCY STOP SYSTEM

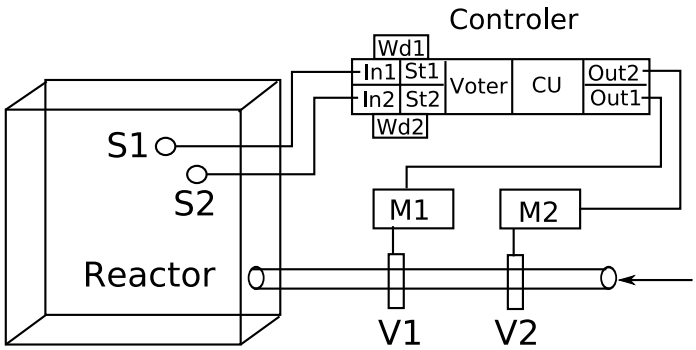

Figure 1: A emergency stop system of the chemical reactor

To illustrate the presented model, the emergency stop system of a chemical reactor (Figure 1) will be used which is described in this section. This system should stop the reaction if the temperature in the reactor is getting 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 his 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 $Wd2$ supervise the inputs. If an input is lost or arrives to 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 unneccessary unavailability of the whole reactor (spurious shutdown).
A failure of the whole emergency system can be caused by several different failures of its components. We will classify these failures as dangerous (D), non-dangerous (S) or omission (I), 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 (I). The input modules can either lose the data of the sensors (I) or change it to a higher (S) or lower (D) value. In the memory the stored data can be distorted by a bitflip 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 we will limit our explanation on the general outline of the system.
A fault tree for the dangerous failure is shown in Figure 2. The names of the basic events are composed of the failed component and the kind of failure. For a more detailed analysis, the basic events could be replaced with subtrees.
This fault tree has some problems though. At first, some basic events for the same components, but for different failure modes are stochastically dependent, for example $S1_D$ and $S1_I$. Furthermore, some component failures will prevent a dangerous failure of the whole system, for example a blocked valve in its closed state. These effects can not be included directly in the fault tree. Finally, creating several fault trees to model different disjunct failure modes of one system can lead to inconsestencies. If the system is large, it is very probable that there are combinations of faults which will lead to a failure in two or more fault trees.

3 THE INFORMATION FLOW MODEL

We are mainly interested in two different events of the system:

- Dangerous incidents which could lead to accidents
- Non-dangerous spurious trips

We want to extract all scenarios leading to one of these two undesired events of the whole system. Especially interessant are single point of failures, common cause failures and failure propagation. These kind 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. We use a directed block diagram representing the information flow through the system for high level and

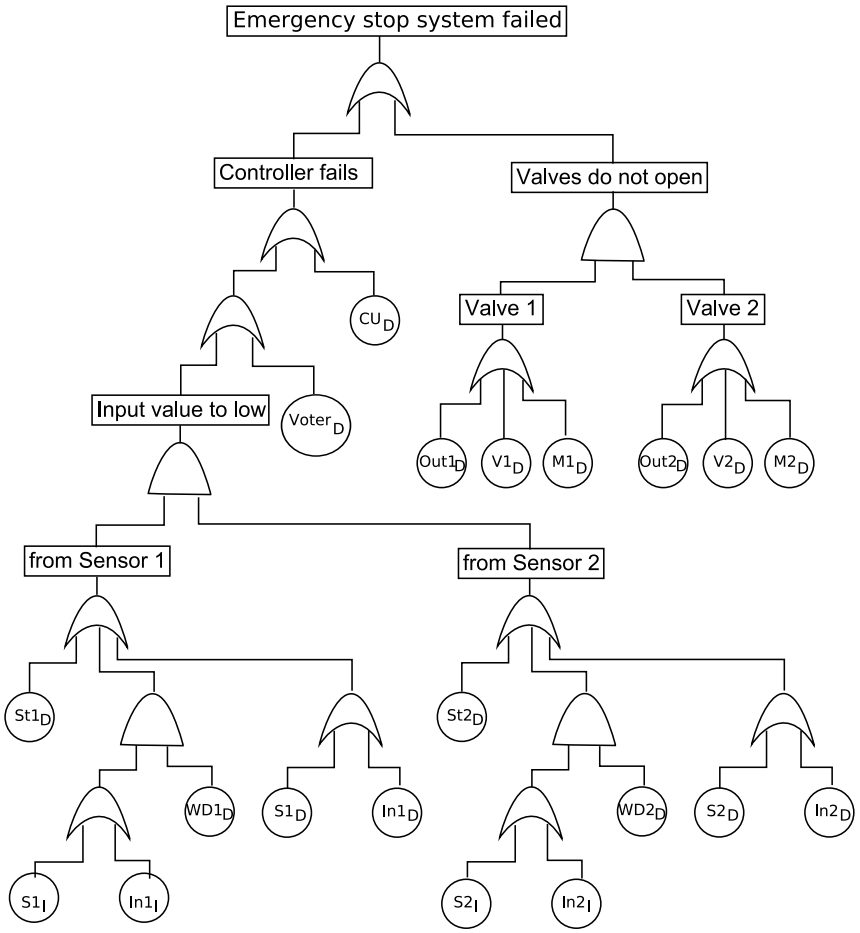

Figure 2: The fault tree for the dangerous failure of the emergency stop system

finite state automatons and rules for low level modelling. This information flow diagram (IFD) and the automatons are generalized versions of the diagrams and automatons presented in (Hamidi 2005).

3.1 Information Flow Diagram

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

- WD-blocks for watch dogs
- SRC-blocks as sources of information
- DEC-blocks for logical decisions
- ST-blocks for all other functions (storage of information, transformation of information, self-tests etc.)

Blocks of the type WD are used specially for control units with a watch dog. 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 in- 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 behavior of multiple interconnected sources of information. They do not describe any physical entities. The components which make this decision have to be included by adding a successing ST-block.

One block in the diagram, normally a ST- or DEC-block, can be marked as final block. This block has no output and is used to generate the failure scenarios which will be described in Section 3.4. An example of an IFD for the given example in section 2 is shown in Figure 3. There are blocks for the different modules of the system, and some extra decision blocks. $Lost1$ and $Lost2$ decide, if the signal of the sensor is lost, CD decides if the voter will get the necessary inputs for a correct voting and $Safe$ is used for the final decision if there is a dangerous failure or a spurious trip. 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 our diagram. This information proceeds to the successing blocks where it is processed and proceeded further. The exchange between the blocks al-

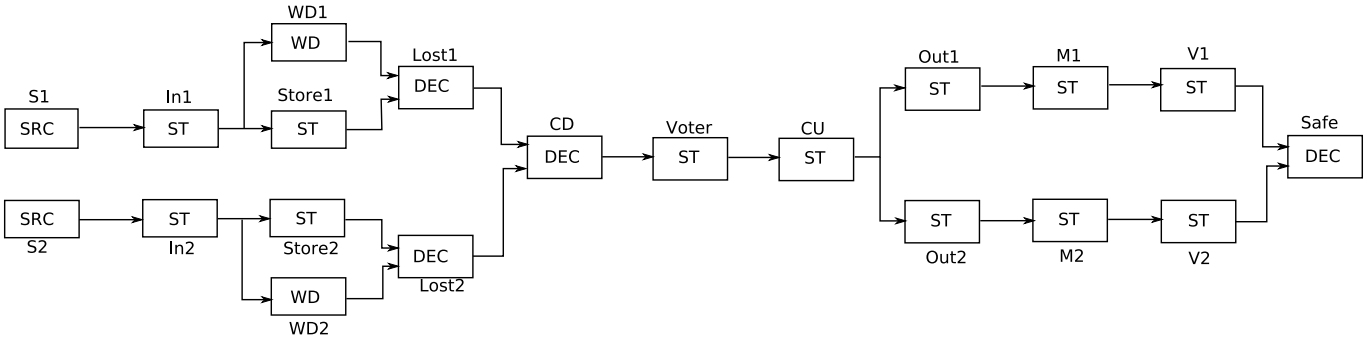

Figure 3: The IFD for the emergency stop system

ways works faultless. While processing the data in the blocks, faults can occur or be detected. This means, that the state of signal can change within a block. We distinguish three different erreonous states for the signals:

- A non-existent failure has been detected. (Safe failure state S)
- An existent failure has not been detected. (Dangerous failure state D)
- The signal is lost. (Inhabitant failure state I).

3.2 Finite automatons

After modeling the information flow it is necessary to specify the state changes of the information. For non-DEC-blocks finite, acyclic state automatons are used to represent a mapping function. A deterministic finite automaton (Brookshear 1989) is a quintuple $(S, \Sigma, \delta, x_0, F)$ with

- a finite set of states S
- an input alphabet Σ
- a transition function $\delta : S \times \Sigma \rightarrow S$
- an initial state $x_0 \in S$
- a set of final states $F \subset S$

For our example, we have one predefined initial state x_0 and three predefined final states x_S, x_D, and x_I which represent the three faulty states of information. The used alphabet in these automatons are symbols which represent different kind of failures. They are denoted as follows:

- $init(i)$ with $i \in \{S, D, I\}$ (For ST- and WD-blocks)
- $d(x, y)$ with x as a hardware resource and $y \in \{0, S, D, I\}$
- $bf(e)$ with e as bitflip resource
- $tf(f)$ with f as testing resource

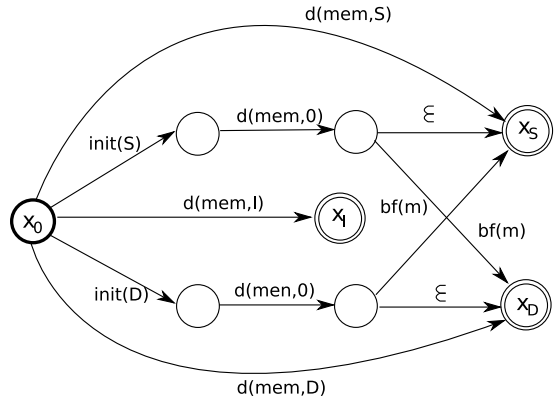

Figure 4: The Automaton for the block Store1

- The empty word ϵ

The advantage of the automatons for our purpose is that they are able to include different kinds of failures and several failure modes. Furthermore they are very intuitive and can be deduced quite easily for small subsystems.

An example, the automaton for the block $Store1$, is shown in Figure 4. While storing the results, different failures can occur. At first, the memory mem can be damaged physically, so that there are several bits which are locked to one (S), locked to zero (S) or the memory is not available at all (I). There is also the possibility of a bitflip in the memory, changing the value of the stored data.

$init(i)$ is used for the fault propagation of the predecessor block. It only occurs at the initial state x_0. The possible values for i depend on the type of the current block. In ST-blocks, i can only be S or D as ST-blocks aren't capable of handling lost information. By contrast, WD-blocks only contain $init(I)$.

For hardware failures, the symbol $d(x, y)$ is used. y indicates the state of the hardware resource x: working correctly (0), non-dangerous failure (S), potential dangerous failure (D) or no output (I). $bf(e)$ represents environmental errors like bit flips of a resource e. $ft(f)$ indicates that a testing resource f has not detected an error. To simplify the calculation, we assume that the state of a hardware-, bit flip- or fault test-resource is always the same in one time step.

Note that all three final states are not always needed. Only final states which can be detected by the inits of the successor blocks have an influence on the result and are necessary. This will be explained in more detail in the next subsection.
The automatons define a language which describes all scenarios leading to the different failures of the block. We can distinguish three different sub languages for non-dangerous failures, potential dangerous failures and failures with no output. These languages can be extracted and are saved in the three lists $L_S(B)$, $L_D(B)$, and $L_I(B)$ for the block B. The automaton in Figure 4 defines the following languages:
$L_S(\text{Store1}) = \{d(mem, S); init(S)d(mem, 0);$
$init(D)d(mem, 0)bf(m)\}$
$L_D(\text{Store1}) = \{d(mem, D); init(D)d(mem, 0);$
$init(S)d(mem, 0)bf(m)\}$
$L_I(\text{Store1}) = \{d(mem, I)\}$

3.3 Rules of DEC-blocks

Decision blocks use another low level model to describe their behavior. For this purpose, boolean rules are introduced. These rules use the state of the signals (either S, I or D) of each input. There are three rules, they represents the lists L_S, L_D and L_I. If we take a look at the final block of the IFD shown in 3, the following rules are chosen:
$S : V1 = S \vee V2 = S$
$D : V1 = D \wedge V2 = D$
$I : false$
So, the final block will create a spurious trip if at least one of the two valves will create one. A dangerous failure will only occur if both valves will fail dangerously. As we are searching a complete list, we have to define how to extract it. Disjunctions in the rules will be handled by unifying two lists, Conjunctions will be handled by set products. For the given example we can conclude:
$L_S(\text{Safe}) = L_S(V1) \cup L_S(V2)$
$L_D(\text{Safe}) = L_D(V1) \times L_D(V2)$
$L_I(\text{Safe}) = \{\}$

3.4 Generation of the global lists

The main interest is to generate all scenarios for dangerous failures and spurious trips of a final block. They will be stored in the lists L_D and L_S. To get these lists, the lists $L_D(B_f)$ and $L_S(B_f)$ of the final block B_f are created in order to connect them with the local lists of the other blocks. It's necessary to distinguish two cases: DEC-blocks and non-DEC-blocks. For DEC-blocks, the method presented in the previous subsection to connect the blocks is used. For non-DEC-blocks, all $init(i)$ in the list are substituted recursively. The sequence after an $init(i)$ is combined with all sequences of a local list $L_i(B)$ of the predecessing block B by a set product. To illustrate this, the

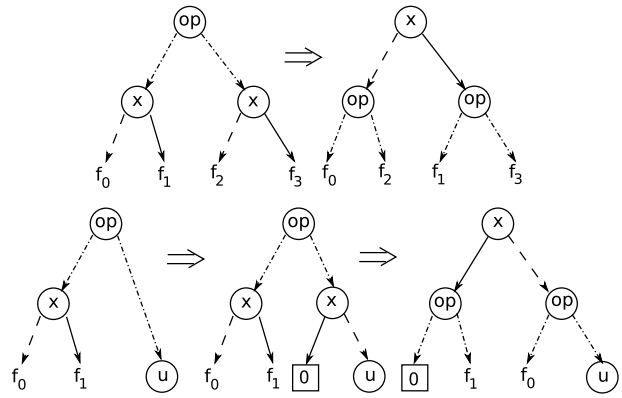

Figure 5: The reduction rules for ZBEDs

three following lists are used:
$L_S(B_f) = \{init(S)d(x, S)d(y, D); init(D)d(y, 0)\}$
$L_S(B) = \{init(S)d(v, 0); init(D)d(v, S)d(w, D)\}$
$L_D(B) = \{init(D)d(v, D); init(S)d(w, D)\}$
If $init(S)$ and $init(D)$ is substituted with $L_S(B)$ and $L_D(B)$, we obtain:
$L_S(B_f) = \{init(S)d(v, 0)d(x, S)d(y, D);$
$init(D)d(v, S)d(w, D)d(x, S)d(y, D);$
$init(D)d(v, D)d(y, 0);$
$init(S)d(w, D)d(y, 0)\}$
It is quite obvious that using this method directly will lead to an exponential growth of the list. This is a severe problem as it will limit the usability of the proposed model. Therefore the size of the created list has to be reduced. In order to reach this aim *Binary Decision Diagrams* (BDDs) are used to control the combinatorial explosion.

4 THE BINARY DECISION DIAGRAM

In this section we will show how BDD-techniques can be adapted to this problem. BDDs in general were introduced by Bryant. (Brace et al. 1990) For this special problem, two extensions of BDDs are used: *Zero-suppressed Binary Decision Diagrams* (ZBDDs) (Minato 1993) will be combined with *Binary Expression Diagrams* (BEDs) (Andersen and Hulgaard 1997). The combination will be called *Zero-surpressed Binary Expression Diagrams* (ZBEDs). These ZBEDs can be reduced to normal ZBDDs by applying the reduction rules of BEDs.
In zero-suppressed BDDs, all 1-edges leading to the terminal 0-node will be deleted while nodes with the same 1- and 0-successor will remain. For this application, the usage of ZBDDs will reduce the size of the diagram significantly as there are many edges like this.
BEDs are similar to BDDs, but they also contain nodes for the boolean operators $\vee$ and $\wedge$. They can be reduced to a regular BDDs with a complexity equivalent to the creation of a normal BDD by pushing down the operator nodes until they reach the leaves of the BED. The two reduction rules specially for ZBEDs are illustrated in Figure 5. While the first rule is the

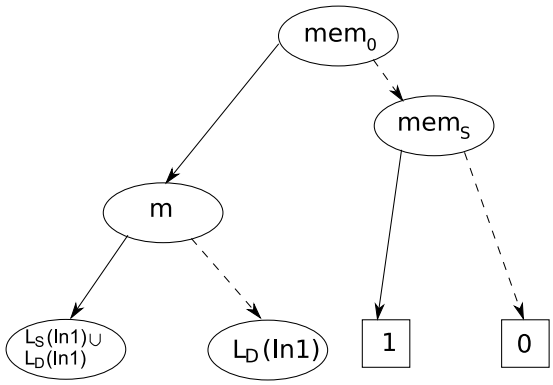

Figure 6: The ZBED for the list $L_S(Store1)$

same as for standard BEDs, the second one had to be modified to be compatible with ZBDDs.
But first, a boolean interpretation of the lists is given which is necessary to use BDDs in general. Afterwards we will explain how to create a ZBED for single blocks, simple serial IFDs, and general IFDs.

4.1 Boolean interpretation of the lists

As BDDs are an alternative representation of boolean expressions, the lists have to be interpreted as the latter. Bit flip- and fault test-resources can have two states, so it is no problem to see them as simple boolean variables. Hardware resources have four states, though. But this is not such a big problem as 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_I is not necessary. As x can only be in one state at one time, the value of x_I can be deduced from the values of the other three variables.
Sequences can be interpreted as conjunctions of their comprised resources. For example, $d(x,S)bf(e)$ is interpreted as $x_S \wedge e$. A whole list is seen as a disjunction of all sequences. $\{d(x,S)bf(e); d(x,D)d(y,0)\}$ is interpreted as $(x_S \wedge e) \vee (x_D \wedge y_0)$.
$Init(i)$ is used as abbreviation to the whole expression of the list L_i of the predecessor block and will be used like any other variable.

4.2 ZBEDs for non-DEC-blocks

At first, we have to create a ZBED for each local list $L_S(B)$, $L_D(B)$ and $L_I(B)$ of every non-DEC-block B. The ZBED is created by a simple decomposition of the list. As example the list L_S(Store1) is used. The original list is:
L_S(Store1) $= \{init(S)d(mem,0); d(mem,S);$
$init(D)d(mem,0)bf(m)\}$
This is interpreted as:
$(init(S) \wedge mem_0) \vee mem_S \vee (init(D) \wedge mem_0 \wedge m)$
Now we can take one of the variables and set its value. We will begin with mem. For example, its value is set to 0 which means mem_0 is set to true and mem_D and mem_S are set to false. This leads to the following reduced list:
$init(S) \vee (init(D) \wedge m)$
There are similar results for setting mem to $0, I$ or D. The decomposition process can be continued recursively with the other variables until only $init$-values or boolean constants remain. We can represent this decomposition process graphically. For this example, we receive the diagram in Figure 6. Equivalent subexpressions are shared to reduce the combinatorial explosion. Furthermore, there are at most three non-trivial leaves ($init(S), init(D), init(S) \wedge init(D)$) for standard blocks and one non-trivial ($init(I)$) leaf for WD-blocks possible. These leaves are also called temporal leaves as they will be replaced later on.

4.3 ZBEDs for DEC-blocks

As DEC-blocks are defined by boolean expressions, it would be possible to use a normal decomposition to create the ZBED. The only problem is that the expressions use the lists of several predecessor blocks and that these lists can be combined with set products, too. This leads too much more possible permutations of $init$-values than for serial diagrams.
The solution to this problem is to use ZBEDs as they can represent the rules directly. The creation of a ZBED for a DEC-block is demonstrated with the list L_D(Safe) of the final block of the example. It is defined by the following expression: $(V_1 = D) \wedge (V_2 = D)$. The lists $L_D(V_i)$ with $i \in \{1,2\}$ are defined as $\{init(D)d(vi,0); d(vi,D)\}$. Figure 7 shows the BED before and after the first reduction. The reduction stops if the or- and and-nodes disappear or if they have one non-trivial leaf as a child. In the last case, the reduction of the ZBED to a ZBDD can be continued recursively by replacing all temporal leafs with the ZBED of the corresponding list and applying the reduction rules to the extended diagram until the source nodes are reached.

4.4 ZBEDs for serial IFDs

In the next step, we assume a simple serial IFD. No hardware-, bit flip-, and fault test-resource occurs in more than one block. With these assumption, it is easy to create a ZBED for an accumulated list. The $init$-nodes are replaced successively with the ZBEDs of the lists represented by these nodes. If these sub-ZBEDs share equivalent nodes, these nodes will be unified. This process can be continued recursively until the source blocks without any $init$-values are reached.
Let us have a closer look at the given example. The following three blocks will be aggregated: $Store1$, $In1$ and $S1$ with $L_S(In1) = \{init(S)d(in1,0)d(bus,0); d(in1,S); d(bus,S)\}$, $L_D(In1) = \{init(D)d(in1,0)d(bus,0); d(in1,D); d(bus,D)\}$, $L_S(S1) = \{d(head_1,S)\}$ and $L_D(S1) =$

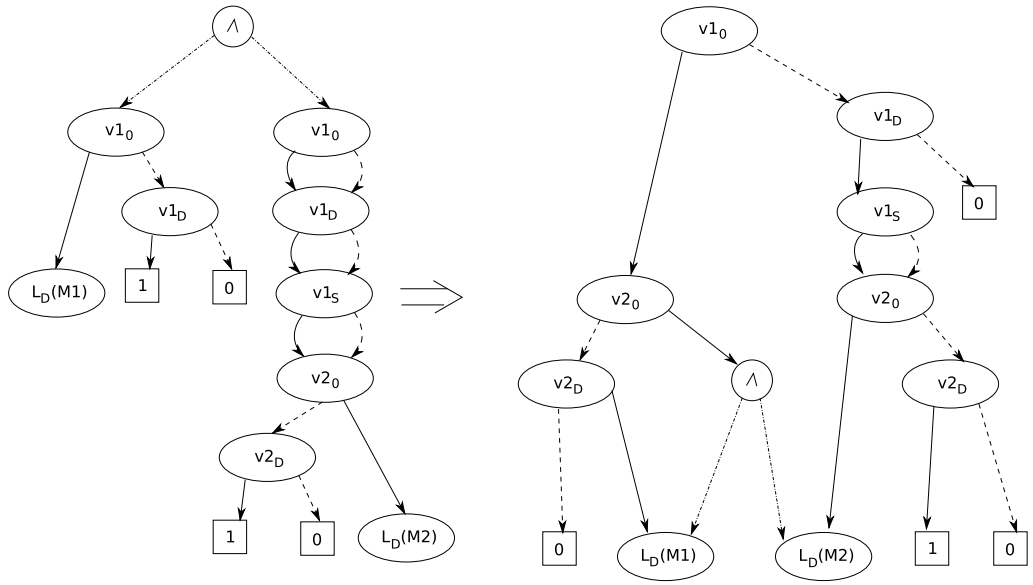

Figure 7: The ZBED for the list $L_D(\text{Safe})$ in its original (left) and in its reduced (right) stage

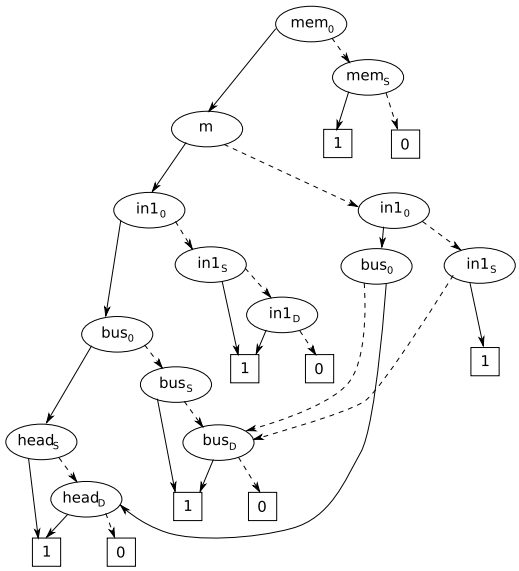

Figure 8: The ZBED for the aggregated list $L_S(Store1)$

$\{d(head_1, D)\}$, a ZBED for all three blocks can be created by replacing the $init$-nodes with the roots of the ZBEDs of $In1$, where the $init$-nodes are substituted with the ZBEDs of $S1$. The result is shown in Figure 8.
An interesting result is that after replacing the temporal leaves of one block, there are still at most three new temporal leaves from the predecessor block.

4.5 ZBEDs for general IFDs

In the previous section we assumed that each hardware-, bit flip- and fault test-resources occurs only in one block. This was necessary as the algorithm just looks at one block at a time. With shared resources, this could lead to inconsistent lists in which one resource can be in two or more states at the same time. But to forbid shared resources is a quite strong limitation for the presented model, so a solution for more general IFDs is presented in this section.
The first step is to create the local lists L_S, L_D, and L_I of all non-DEC-blocks. Furthermore a look-up-table will be established which maps resources to blocks, in which they appear. After that, the creation of the whole ZBED can start. In general, we use the same aggregation rules for the blocks like in section 4.3 and 4.4. There are only two differences. At first, every node gets another attribute, a list of pointers to the local lists which have to be used later. Furthermore, the decomposition of a variable is started in every local list in which it occurs. This can be easily checked by using the look-up-table. If there are other blocks which use the same resource, the pointers to the three local lists are changed to the modified ones. So it will be ensured that there are no inconsistent sequences without having to create the whole ZBDD in one single go.
There is an important change to the structure of the ZBED for the case in 4.4, though. For every shared resource, the number of non-trivial leaves can quadruple (for HW-resources) or double (for bitflip- or faulttest-resources) as there can be four respective two different settings for the state of the resource which have to be taken into account in each of the leaves.

5 RELATED WORK

There are already several approaches for the automatic generation of boolean models, especially fault trees with the help of a high level model. In this section, some of them will be presented.

5.1 Generation of fault trees with Little-JIL models

In (Chen, Avrunin, Clarke, and Osterweil 2006) the authors present a method for automatic fault tree generation based on Little-JIL models. These models are based on the single steps of a process, which is quite similar to the idea of the information flow. This method was tested in the domain of medical treatment which includes also saftey critical aspects. But this approach does not support several failure modes for single components or steps, though.

5.2 Generation of fault trees out of FBDs

In (Oha, Yoob, Chab, and Son 2005), a method to create fault trees out of function block diagrams (FBDs) is discussed. FBDs are used for programming programmable logic controllers (PLCs) especially in the domain of safety critical systems. This method has the advantage that the fault trees can be generated without creating extra models which minmizes potential error sources in the process of the fault tree generation. It is limited on the software of PLCs, though and can not be used for more general systems.

5.3 The FSAP/NuSMV-SA platform

This powerful platform described in (Bozzano and Villaforiata 2003) is capable to create monoton or non-monoton fault trees out of a formal system model. There are plenty of possibilities like fault injection and different failure modes. The system and its requirements are modelled with a formal language. This approach is very powerful and can be used in different domains, although it is not as intuitive as the other approaches.

6 CONCLUSIONS AND OUTLOOK

The main advantage of the presented model is that it is capable to model two undesired events in one diagram. With other high-level models like fault-trees it would be necessary to create two different models with a lot of similarities. So our approach can save a much time for the modeler. Another big advantage is that the model is hierarchic. The system in general can be modeled with the IFD, the smaller entities of it with finite automatons. This gives the modeller the possibility to create a detailed but understandable model. Furthermore, we have a very efficient algorithm for creating the scenarios based on BDD-techniques. By using local properties, we can create a ZBED even for very general Diagrams in a very efficient way, which can be used as compact representation for the failure scenarios as well as a base for a quantitative solution.

Currently, this approach does not take real-time properties into account. As these can be very important in many safety critical systems, we have to do further research here. Furthermore, we plan to add inter-component dependencies and the event ordering into the model to make it more powerful. We also will implement the presented algorithm and a modeling environment to make measurements in order to check our theoretical results and to offer a powerful modeling tool.

7 ACKNOWLEDGMENT

We want to thank the French-Bavarian center for cooperation of universities (BFHZ-CCUFB) and the region Lorraine for their support of this work.

REFERENCES

Andersen, H. and H. Hulgaard (1997). Boolean Expression Diagrams. In *12th Annual IEEE Symposium on Logic in Computer Science*, pp. 88 – 111. IEEE.

Bozzano, M. and A. Villaforiata (2003). Improving System Reliability via Model Checking: The FSAP/NuSMV-SA Safety Analysis Platform. Technical report, Istituto Trentino di Cultura.

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

Brookshear, J. (1989). *Theory of Computation: Formal Languages, Automata, and Complexity*. Benjamin/Cummings Publish Company, Inc.

Chen, B., G. Avrunin, L. Clarke, and L. Osterweil (2006). Automatic Fault Tree Derivation from Little-JIL Process Definitions. In *SPW/ProSim 2006*, pp. 150–158. Springer.

Hamidi, K. (2005). *Contribution à un modèle dévaluation quantitative des performances fiabilistes de fonctions électroniques et programmables dédiées à la sécurité*. Ph. D. thesis, Institut National Polytechnique de Lorraine.

Lee, W., D. Grosh, F. Tillmann, and C. Lie (1985, August). Fault tree analysis, methods, and applications - A review. *IEEE Transactions on Reliability R-34*, 194 – 203.

Leveson, N. and J. Stolzy (1987, March). Safety Analysis Using Petri Nets. *IEEE Transactions on Software Engineering SE-13*, 386 – 397.

Minato, S. (1993). Zero-Surpressed BDDs for Set Manipulation in Combinatorical Problems. In *30th ACM/IEEE Design Automation Conference*, pp. 272 – 277. ACM/IEEE.

Oha, Y., J. Yoob, S. Chab, and H. Son (2005). Software safety analysis of function block diagrams using fault trees. *Reliability Engineering & System Safety 88*, 215 – 228.